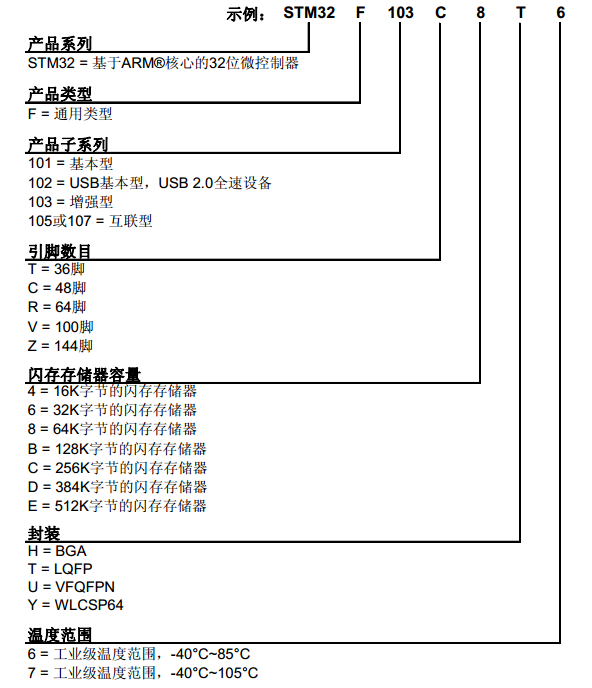
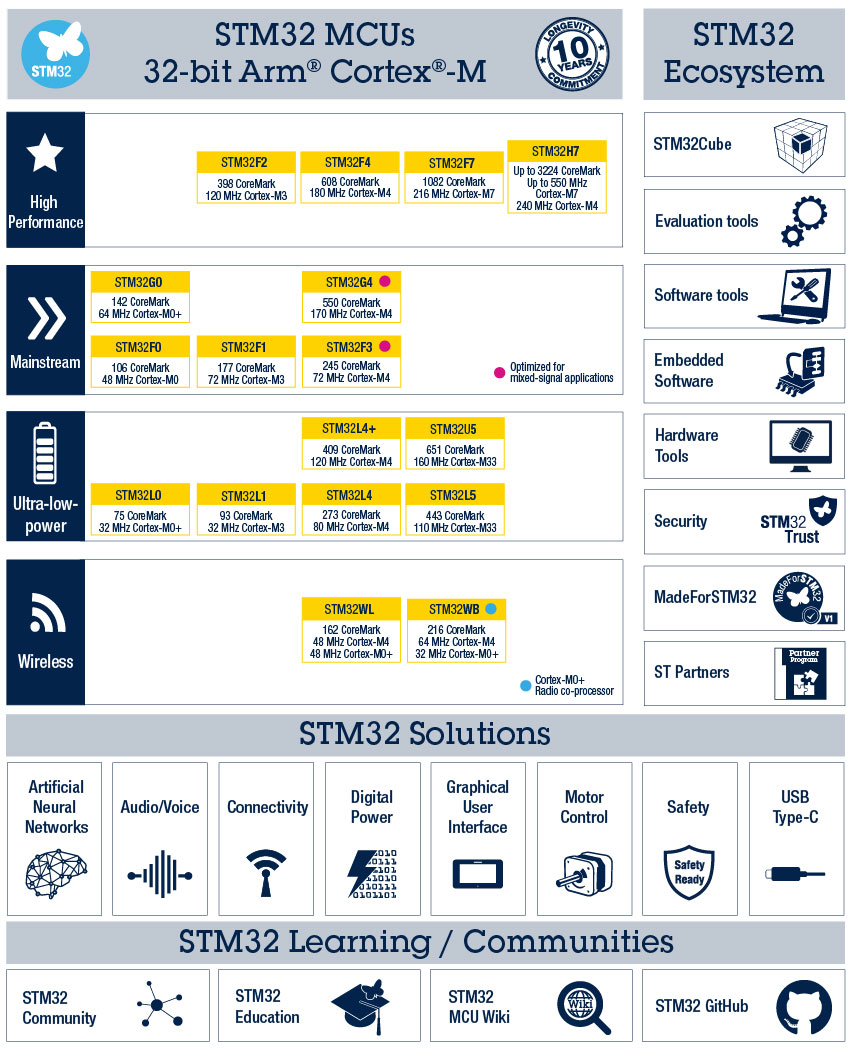
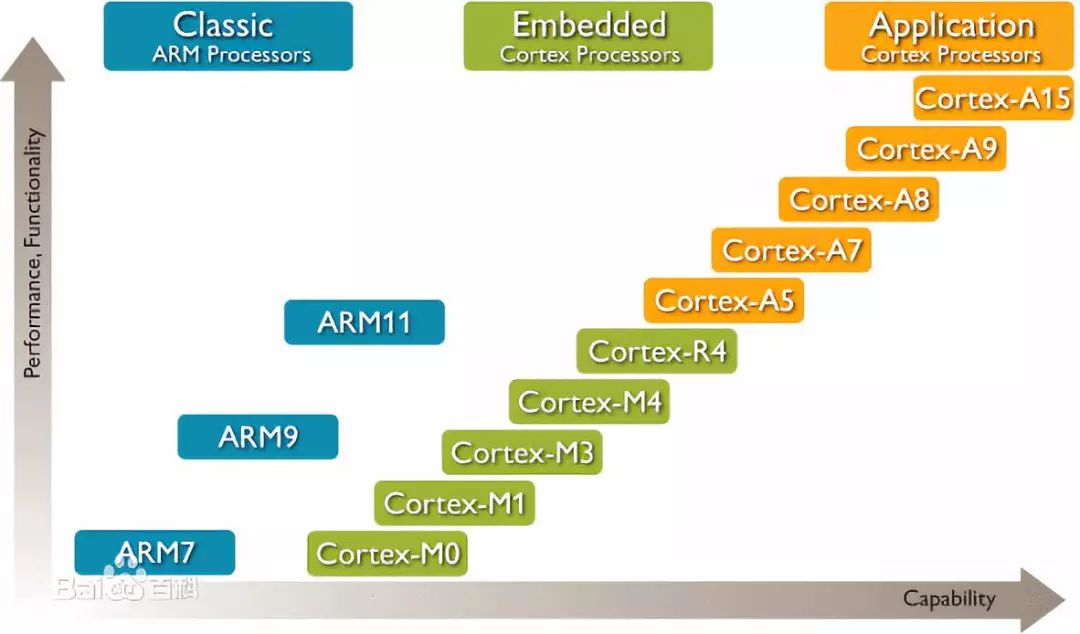
# 1. STM32简介

## 1.1 STM32命名规范







ARM既指ARM公司，也指ARM处理器内核。

A：Application，主要偏向于应用

R：Realtime，偏向于实时性

M：Microcontroller，微控制器

## 1.2 STM32F103C8T6特性

系列：主流系列STM32F1

内核：ARM Cortex-M3

主频：72MHz

RAM：20K（SRAM）

ROM：64K（Flash）

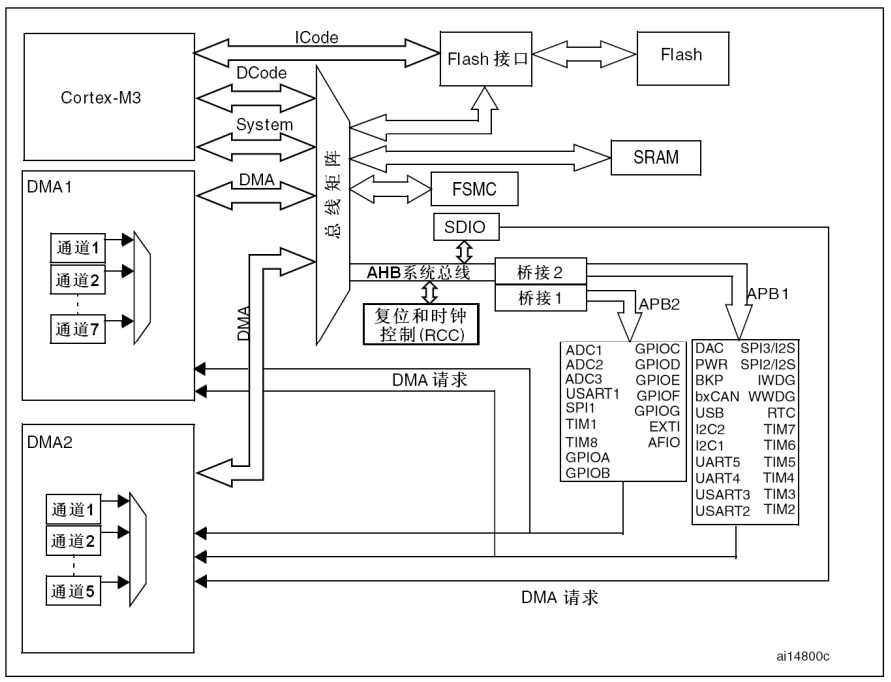
供电：2.0~3.6V（标准3.3V）

封装：LQFP48

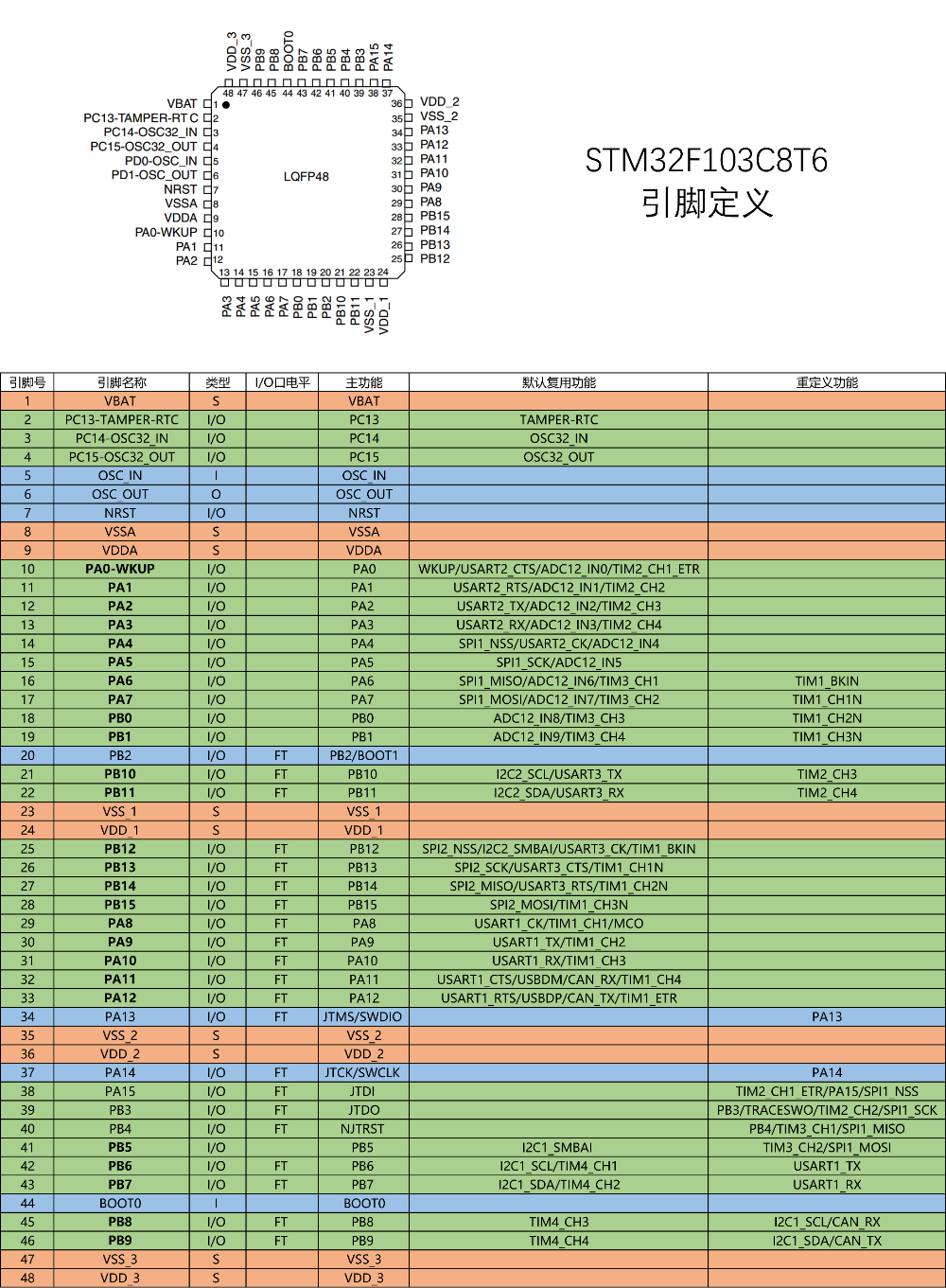
## 1.3 片上资源/外设



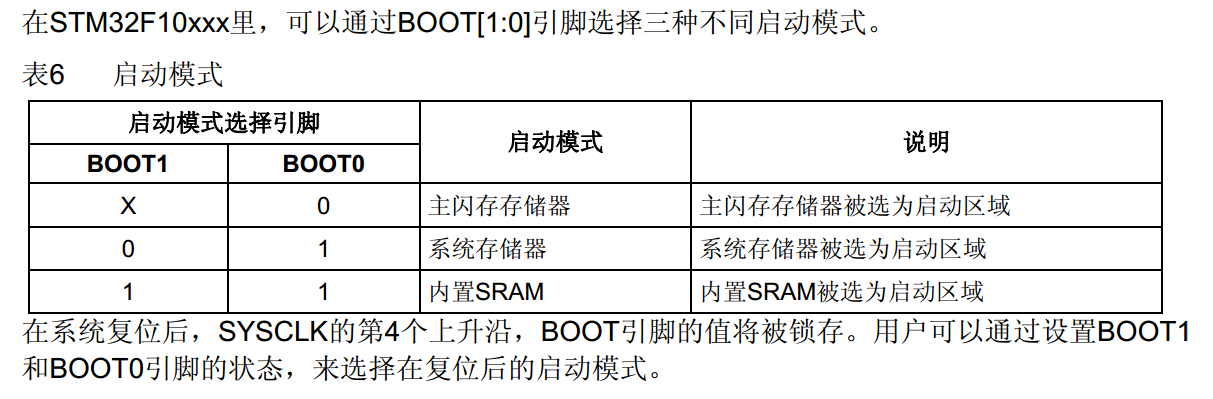
## 1.4 系统架构



## 1.5 引脚定义



## 1.6 启动配置

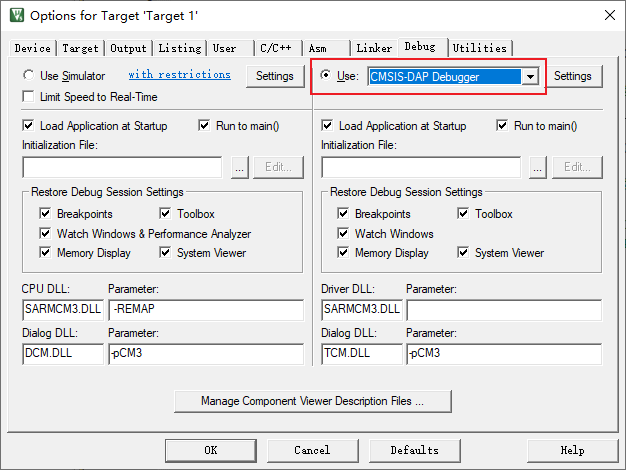


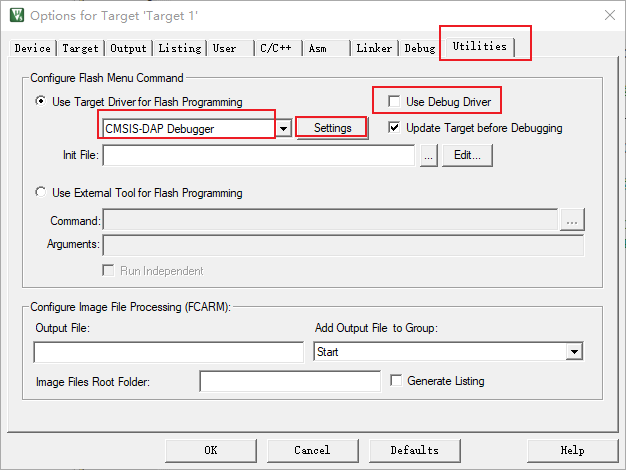
(1)最常用配置：BOOT0设为0，正常的执行Flash闪存里面的程序，使用STLINK、DAPlink进行下载调试

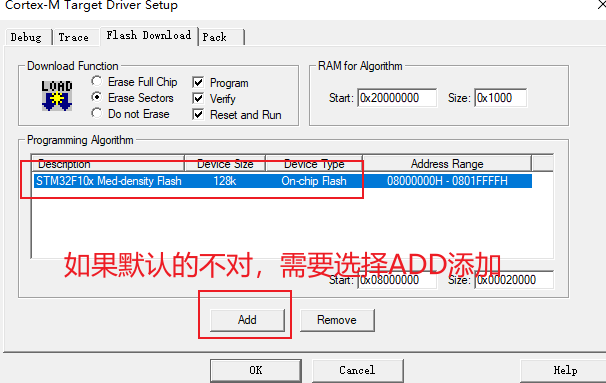
(2)串口下载模式：BOOT0设为1，BOOT1设为0，系统存储器(存放Bootloader程序，接收串口的数据，然后刷新到主闪存中)

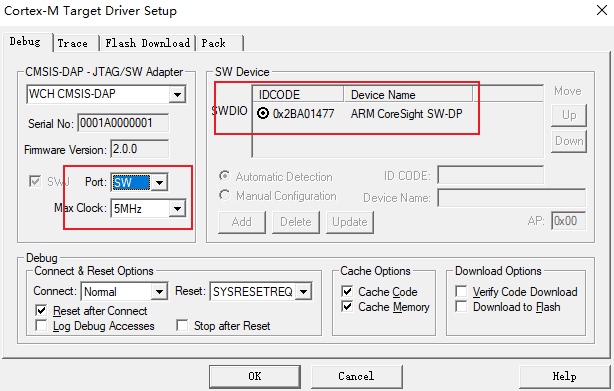
(3)最后一种模式，主要用于程序调试，用的比较少

## 1.7 DAPLink烧录

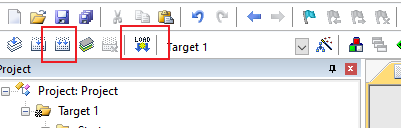








每次下载程序，需要先rebuild编译，在下载！



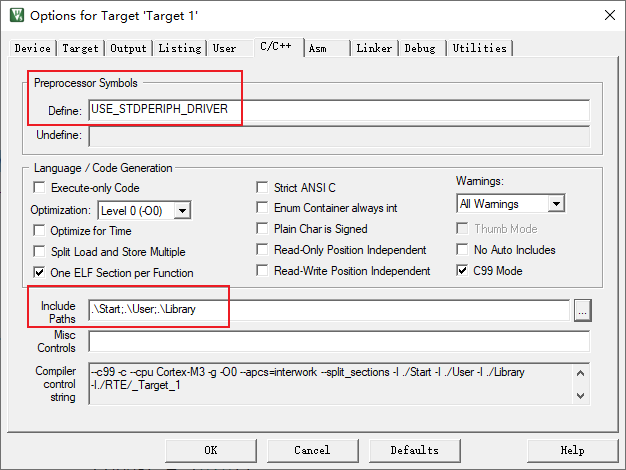
# 2. STM32工程模板

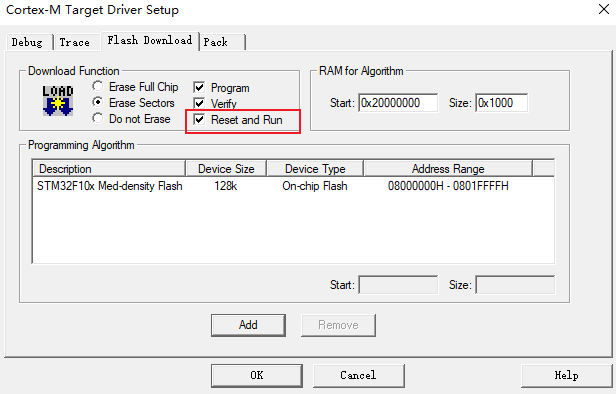
## 2.1 Keil配置工作模板

使用ST库函数的开发方式进行寄存器配置。

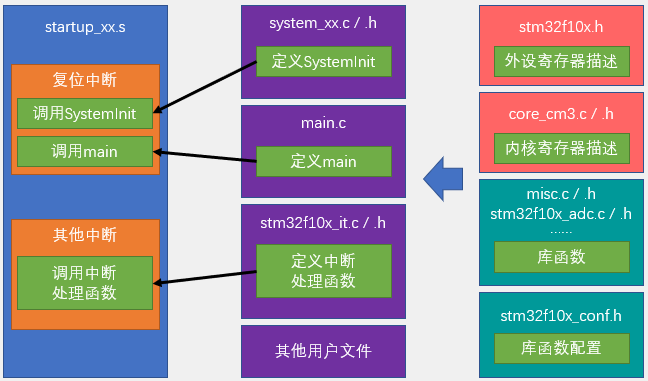
新建工程步骤：

* 建立工程文件夹，Keil中新建工程，选择型号
* 工程文件夹里建立Start、Library、User等文件夹，复制固件库里面的文件到工程文件夹
* 工程里对应建立Start、Library、User等同名称的分组，然后将文件夹内的文件添加到工程分组里
* 工程选项，C/C++，Include Paths内声明所有包含头文件的文件夹
* 工程选项，C/C++，Define内定义USE\_STDPERIPH\_DRIVER (标准外设驱动字符串，使用库函数开发时，这个是必须添加的)
* 工程选项，Debug，下拉列表选择对应调试器，Settings，Flash Download里勾选Reset and Run

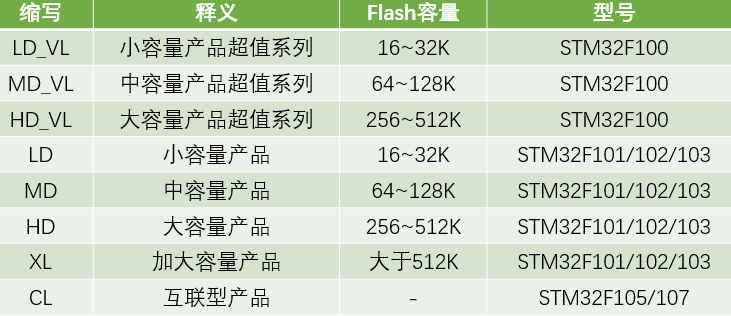


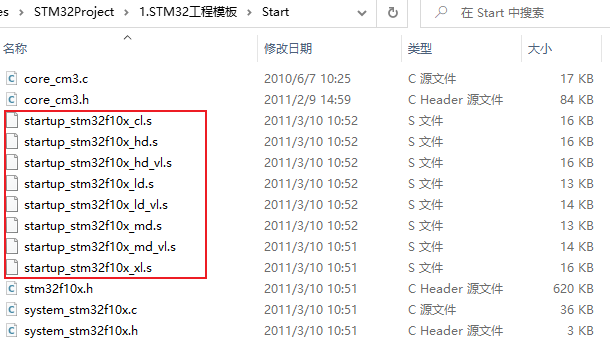


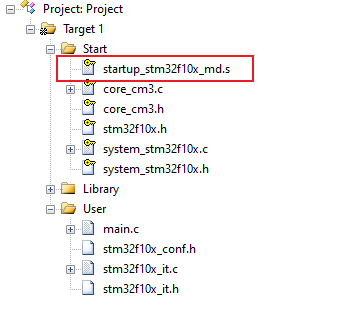
工程架构：

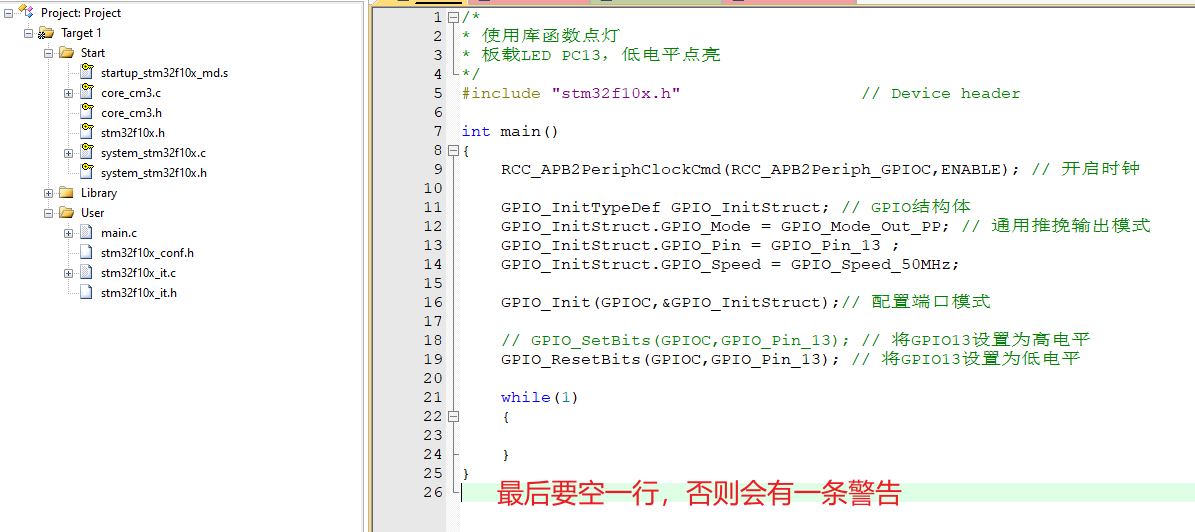


根据不同的产品型号，选择不同的文件



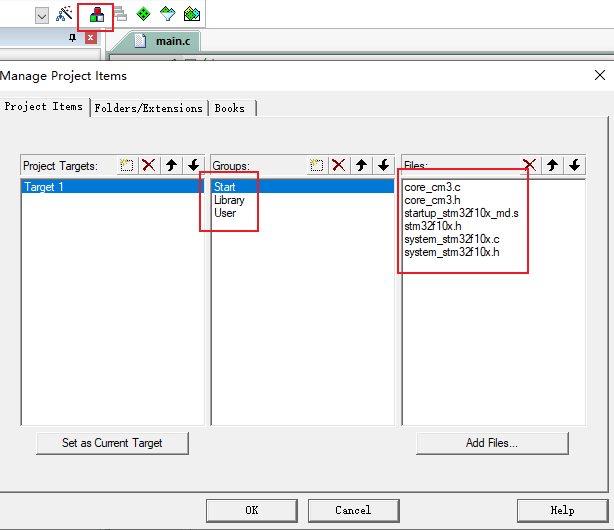




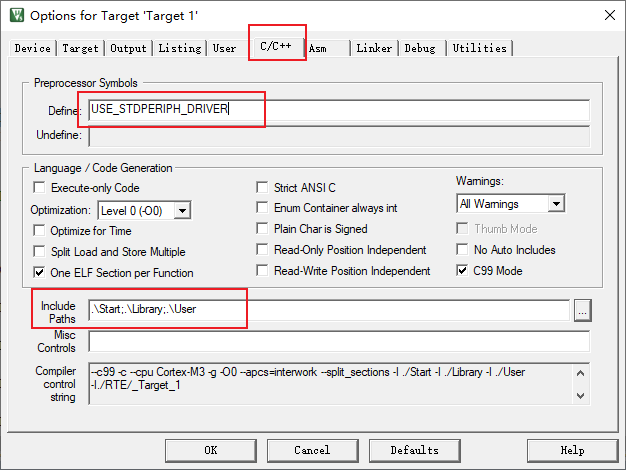


## 2.2 创建工程步骤总结

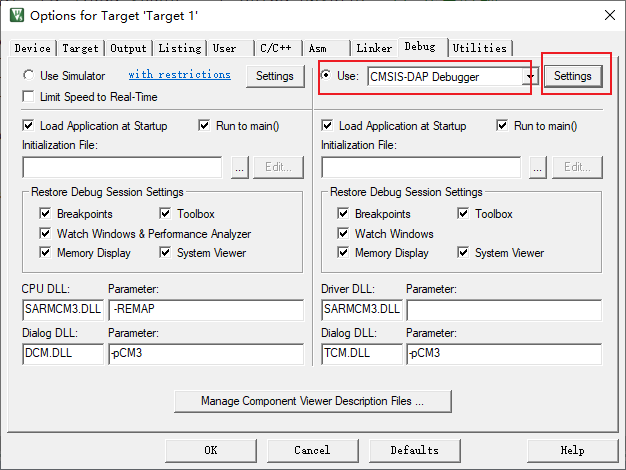
1. 快速创建Start、Library、User组，添加相关的文件

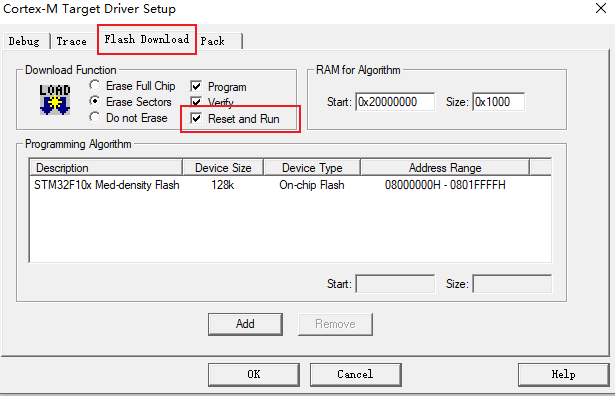


2.点击魔术棒按钮，添加第一步文件的路径，并添加这句话USE\_STDPERIPH\_DRIVER



3. 选择对应的调试器(DAPLink)

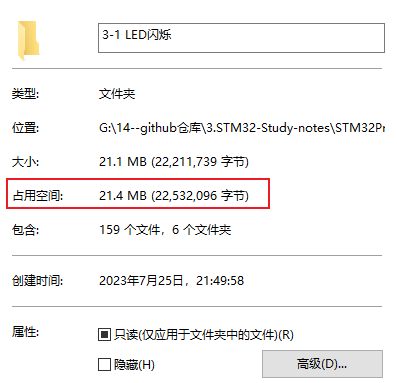
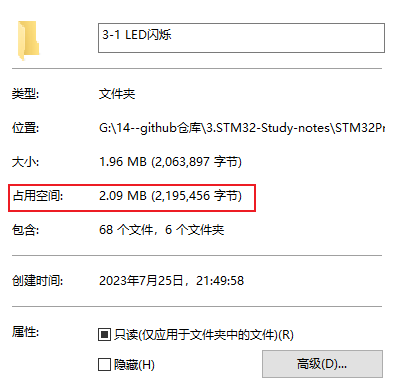




4.小工具：keilkill，是一个批处理文件，可以把工程编译产生的中间文件都删掉



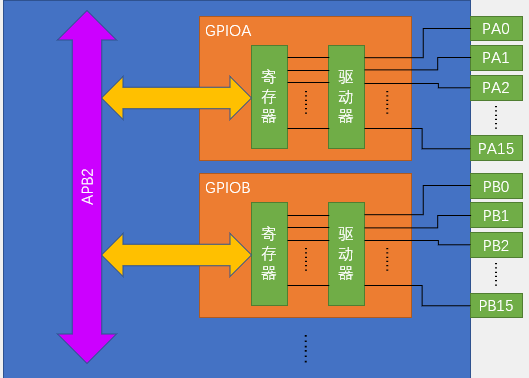
使用前后的对比：减小工程体积，之后可以添加到压缩包分享发送给别人

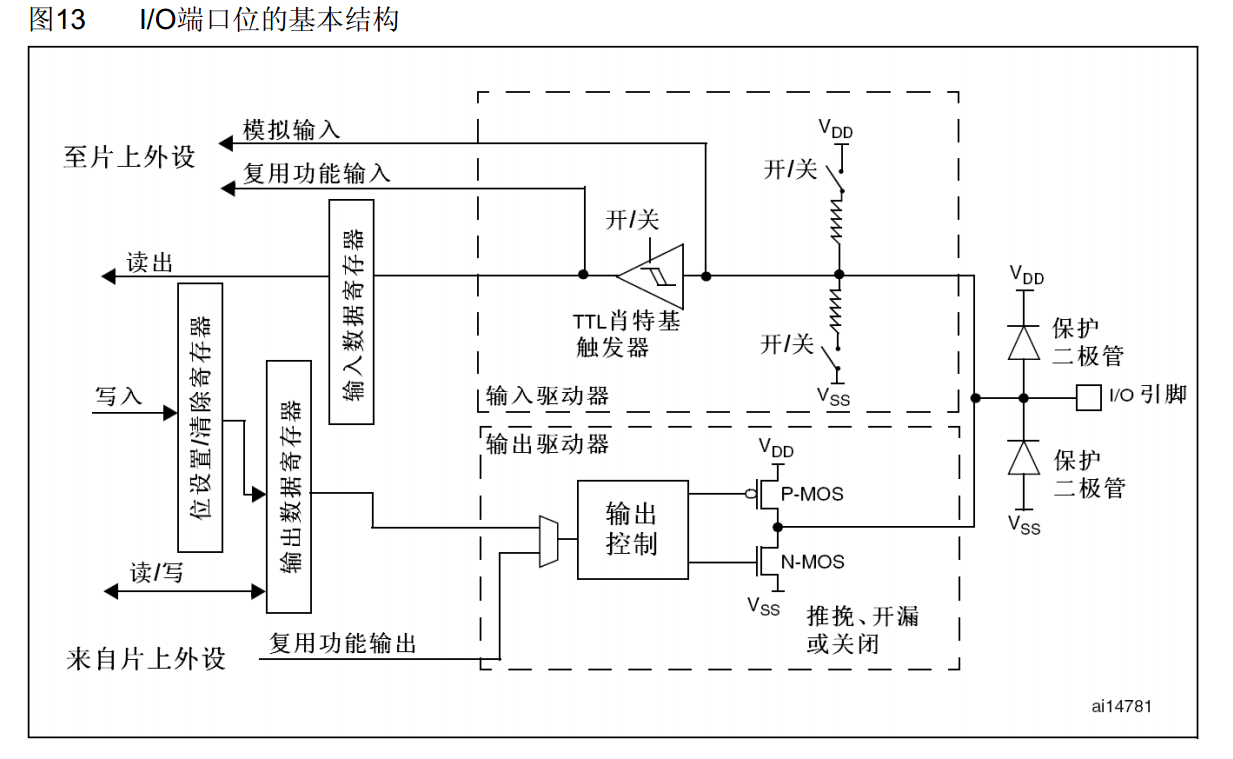
# 3. GPIO配置及应用

## 3.1 GPIO工作模式介绍

GPIO基本结构：

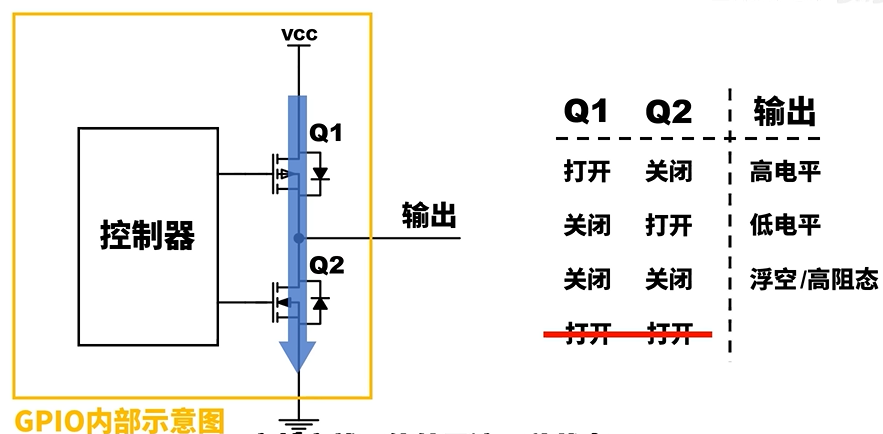


GPIO位结构：





单片机的GPIO配置为输出模式时，会有两种模式：推挽模式、开漏模式。



上下MOS同时导通时，发生短路，这种状态需要避免。

推完输出模式理解：GPIO输出高低电平时，电流的一个动作而已

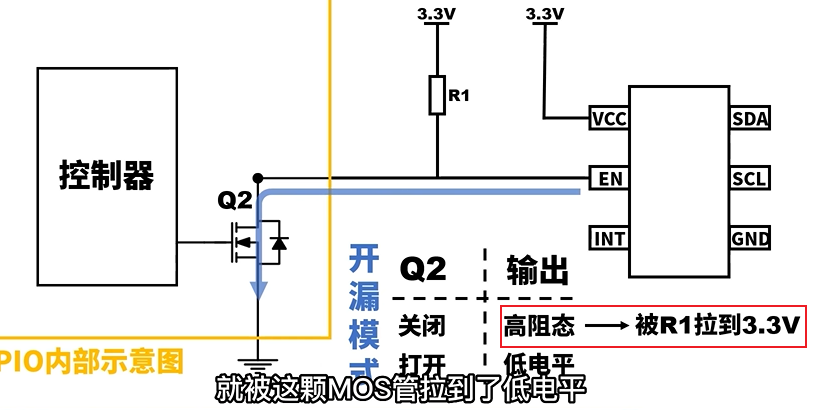
推：IO输出高电平时，把电流推出去

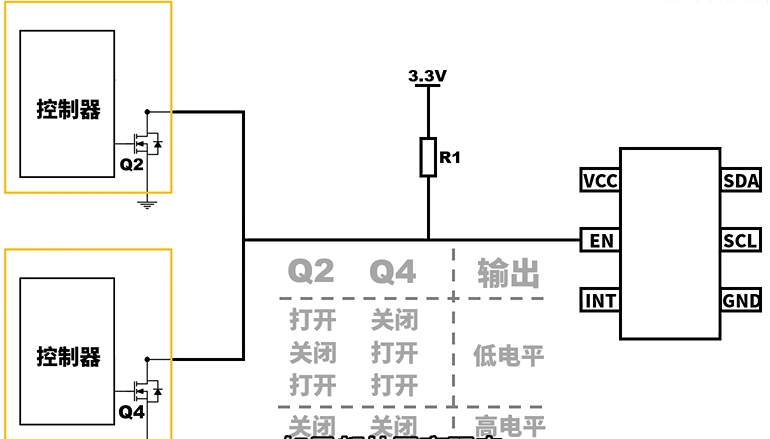
挽：IO输出低电平时，外界电流流进来，把电流挽回来。

开漏模式：低电平和高阻态(被外接上拉电阻拉到高电平)的组合，称为开漏输出。

需要在外面接一个上拉电阻，常用于几个GPIO控制一个输入。

开漏模式输出高电平相当于高阻态，没有驱动能力；低电平有驱动能力。





只要有任意一个GPIO输出低电平，enable就是低电平

如果都处于高阻态，enable就是高电平

## 3.2 LED闪烁



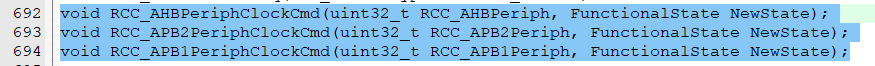
操作STM32的GPIO共需要3个步骤：

1.使用RCC开启GPIO的时钟

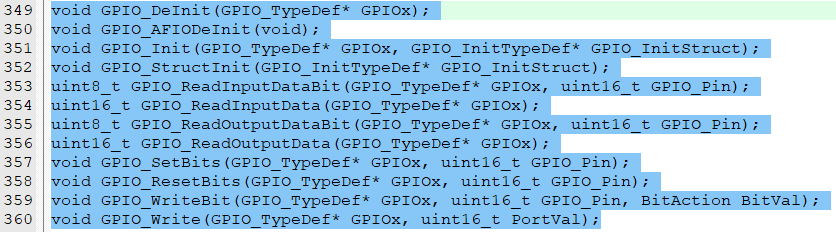
2.使用GPIO\_Init函数初始化GPIO

3.使用输出或者输入的函数控制GPIO口

1.RCC(Reset Clock Controller)，复位与时钟控制器，常用RCC AHB外设时钟控制、RCC APB2外设时钟控制、RCC APB1外设时钟控制

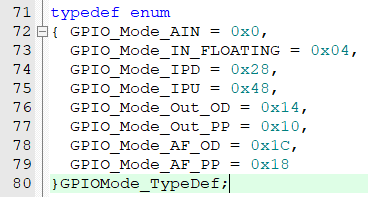


2.GPIO常用库函数



GPIO\_Init()函数：用结构体的参数来初始化GPIO口，需要先定义一个结构体变量，然后给结构体赋值，最后调用这个函数。

GPIO工作模式：



typedef enum

{ GPIO\_Mode\_AIN = 0x0, // Analog IN，模拟输入

GPIO\_Mode\_IN\_FLOATING = 0x04, // 浮空输入

GPIO\_Mode\_IPD = 0x28, // In Pull Down，下拉输入

GPIO\_Mode\_IPU = 0x48, // In Pull Up，上拉输入

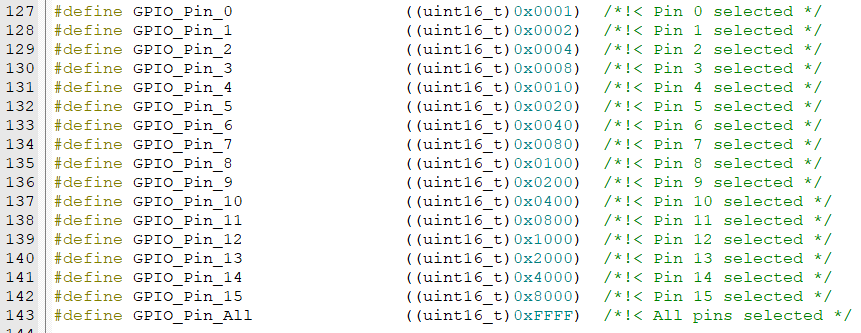
GPIO\_Mode\_Out\_OD = 0x14, // Out Open Drain，开漏输出

GPIO\_Mode\_Out\_PP = 0x10, // Out Push pull，推挽输出

GPIO\_Mode\_AF\_OD = 0x1C, // Atl Open Drain，复用开漏

GPIO\_Mode\_AF\_PP = 0x18 // Atl Push Pull，复用推挽

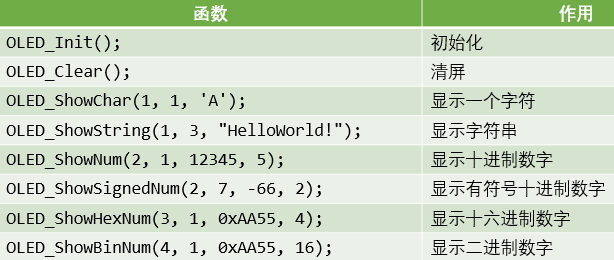
}GPIOMode\_TypeDef;



void GPIO\_SetBits(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin); // 把指定的端口设置为高电平

void GPIO\_ResetBits(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin); 设置GPIO输出低电平

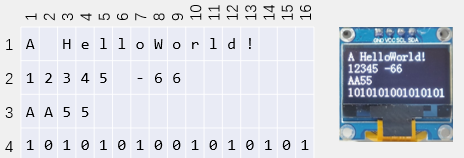
# 4. OLED显示屏使用



参数1：指定起始行；

参数2：指定起始列

0.96寸OLED，分辨率：128\*64，把分辨率设置为4行16列，字符大小8\*8



STM32的引脚上电后，如果不初始化，默认为浮空输入的模式。

SCL --PB8

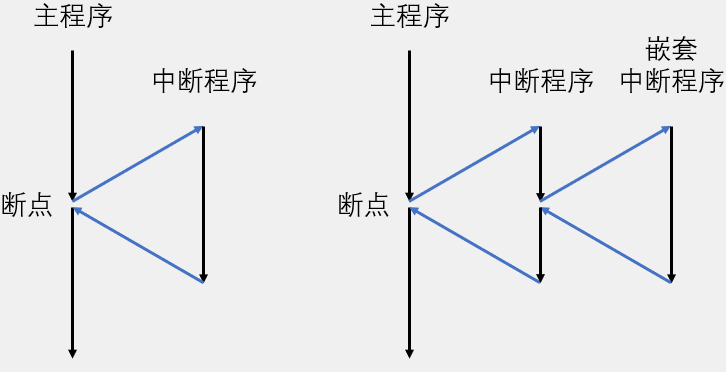
SDA --PB9

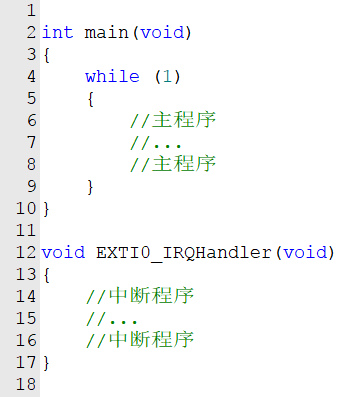
# 5. EXIT外部中断

使用中断系统可以加大的提高程序的运行效率，如果没有中断系统，为了防止外部中断被忽略或者串口数据被覆盖，主程序就只能不断地查询是否有这些事件发生。

比如如果没有定时器中断，主程序只能使用delay函数，才能实现定时的功能。

有了中断系统后，主程序就可以放心执行其他事情，有中断的时候再去处理。





抢占优先级高的可以中断嵌套，响应优先级高的可以优先排队。

抢占优先级和响应优先级均相同的按**中断号**排队。

中断触发方式：上升沿/下降沿/双边沿(上升沿和下降沿都可以触发)/软件触发(如何进行软件触发设计？程序里执行一句代码，就可以触发中断)

触发响应方式：中断响应/事件响应

通道数：16个GPIO\_PIN、外加PVD输出、RTC闹钟、USB唤醒、以太网唤醒

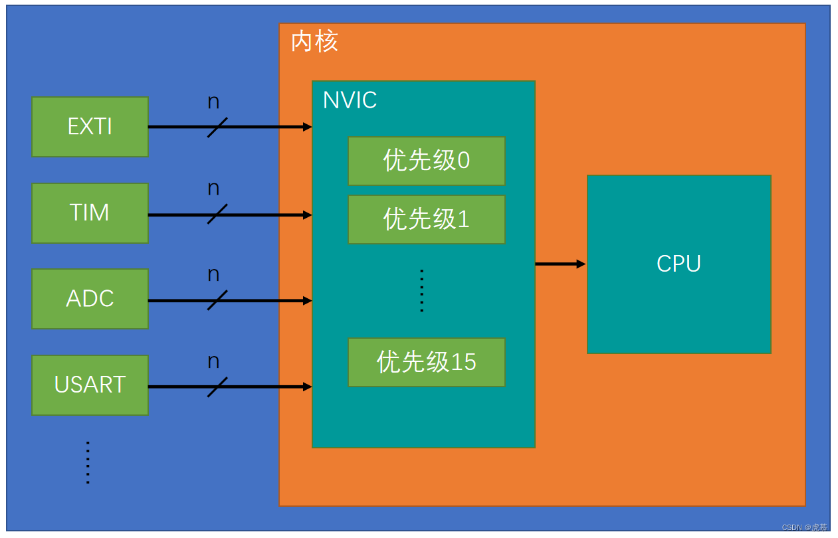
中断响应：引脚电平变化触发中断

事件响应：不会触发中断，而是触发别的外设操作，属于外设之间的联合工作，比如触发ADC转换、DMA

注意：

1. 进入中断函数的时候要检查中断标志位，退出的时候清零中断标志位。

2. 中断函数一般执行简短快速的代码，如操作中断标志位。



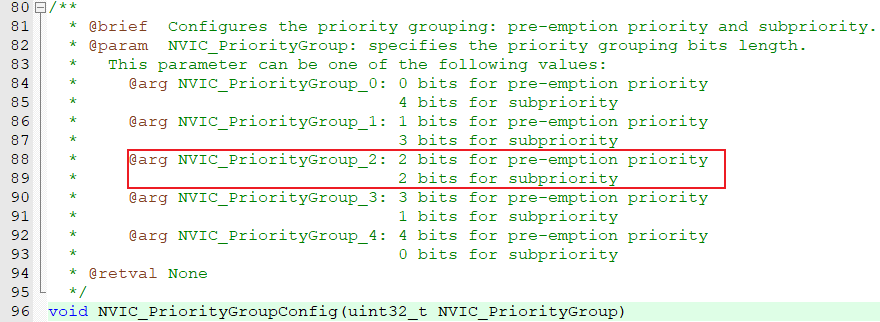
NVIC用于统一管理中断和分配中断优先级，属于内核外设，是CPU的小助手，可以让CPU专注于运算。相当于医院里的叫号系统(给病人进行排号，CPU相当于医生，只负责看病)。

## 5.1 NVIC优先级分组

优先级：抢占优先级、响应优先级。

* NVIC的中断优先级由优先级寄存器的4位（0~15）决定，这4位可以进行切分，分为高n位的抢占优先级和低4-n位的响应优先级(插队)
* 抢占优先级高的可以中断嵌套，响应优先级高的可以优先排队，抢占优先级和响应优先级均相同的按中断号排队





如果中断不多，很难导致中断冲突时，可以随意选择优先级分组，那个都行。

## 5.2 EXIT(Extern Interrupt)外部中断

EXIT的基本功能：检测引脚的电平变化，申请中断。

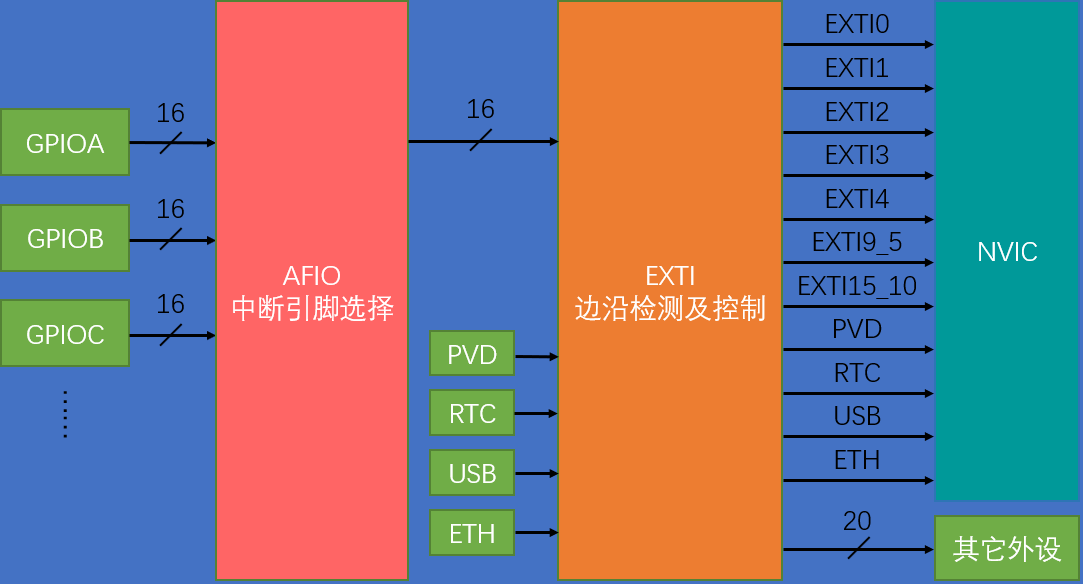
支持的GPIO口：所有的GPIO口，但相同的Pin不能同时触发中断。

比如PA0和PB0不能同时用，端口GPIO\_Pin\*只能选择一个作为中断引脚。

PA0和PA1可以同时触发中断。

通道数：16个GPIO\_pin(也就是只能有16个引脚可以触发中断)，外加PVD输出(电源电压监测)、RTC闹钟、USB唤醒、以太网唤醒。

EXIT基本结构：



AFIO中断引脚选择：在GPIO(A~G)外设的16个GPIO\_Pin中选择一个连接到后面的EXIT的通道里。

## 5.3 常用的触发外部中断的硬件模块

对于STM32来说，想要获取的信号是外部驱动的很快的突发信号，但是又不需要一直进行检测。

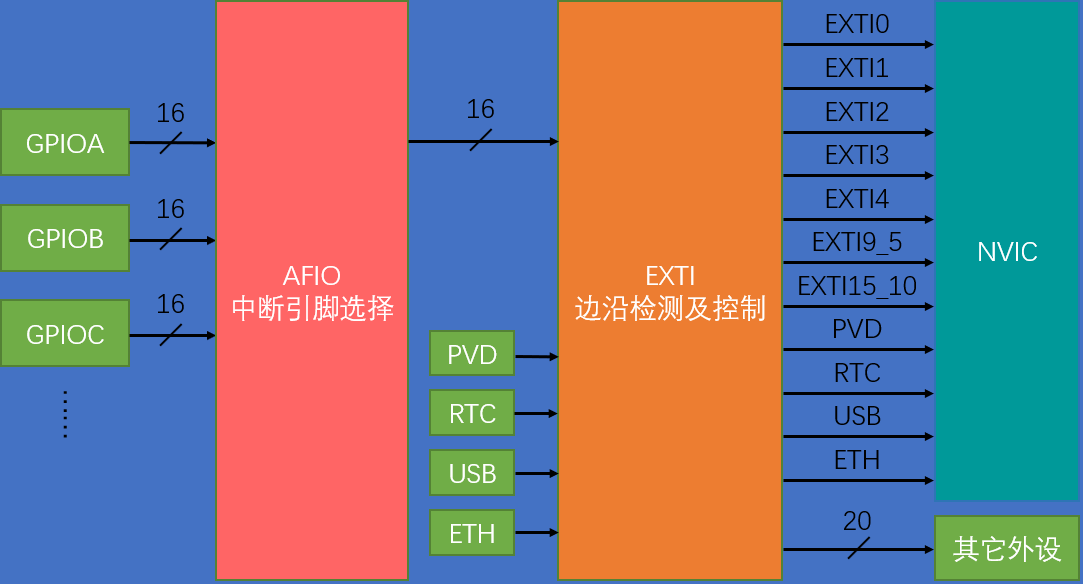
不需要经常使用、不需要一直检测的模块：

1. 按键(注意抖动问题)，不推荐，可以使用定时器中断

2. 旋转编码器

3. 红外遥控接收器

## 5.4 外部中断程序设计步骤



涉及到的外设：RCC、GPIO、AFIO、EXTI、NVIC

1. 打开RCC时钟，不打开时钟，外设是无法进行工作的

2. 配置GPIO，选择需要的端口，设为输入模式(推荐浮空、上拉、下拉)

3. 配置AFIO，选择用的这一路GPIO，连接到后面的EXTI

4. 配置EXTI，选择外部中断触发方式，选择触发响应方式(一般选择中断响应)

5. 配置NVIC，给中断选择一个合适的优先级

配置完之后，写中断执行函数：

在STM32中，中断函数的名字都是固定的，每个中断通道都对应一个中断函数。

中断函数的名字可以参考启动文件中的“stratup\_stm32f10x\_md文件

配置AFIO相关函数



void GPIO\_PinRemapConfig(uint32\_t GPIO\_Remap, FunctionalState NewState);

参数1：选择重映射方式

参数2：新的状态

void GPIO\_EXTILineConfig(uint8\_t GPIO\_PortSource, uint8\_t GPIO\_PinSource);

配置AFIO的数据选择器，来选择想要的中断引脚

注意：主函数和中断函数不要操控同一个硬件(比如OLED显示)，避免不必要的冲突，中断函数只执行突发的事件，需要快速的执行，如操作变量、中断标志位。

# 6. 定时器中断

## 6.1 TIM简介

定时器：可以对输入的时钟进行计数，并在计数值达到设定值时触发中断

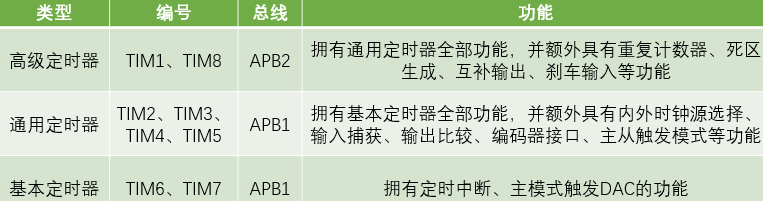
STM32在72MHz计数时钟下可以实现最大59.65s的定时。

1. 16位计数器：执行计数定时的一个寄存器，每来一个时钟，计数器加1

2. 预分频器：可以对计数器的时钟进行分频，让这个计数更加灵活

3. 自动重装寄存器：计数的目标值(计多少个时钟申请中断)，这些寄存器构成了最核心的部分。

这三部分组成了定时器的时基单元(计数器、自动重装寄存器、预分频器)

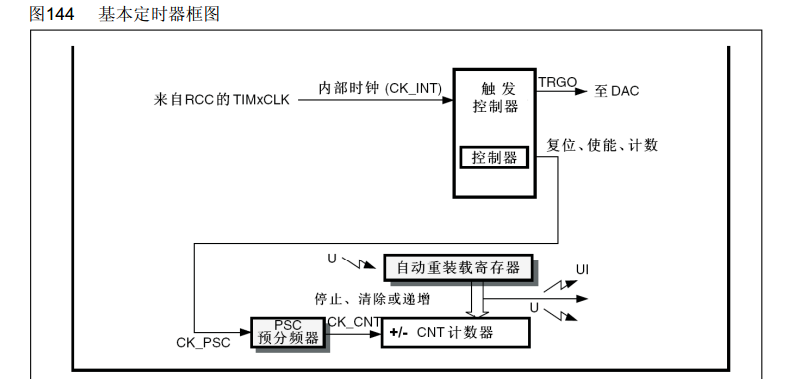


STM32F103C8T6定时器资源：TIM1、TIM2、TIM3、TIM4

## 6.2 基本定时器

可以完成定时中断，主模式触发DAC的功能。

DAC功能：数模转换



* 内部时钟（CK\_CNT）：一般就是系统的主频72MHz，通向时基单元的输入。
* 时基单元：16位预分频器 + 16位计数器 + 16位自动重装载寄存器。

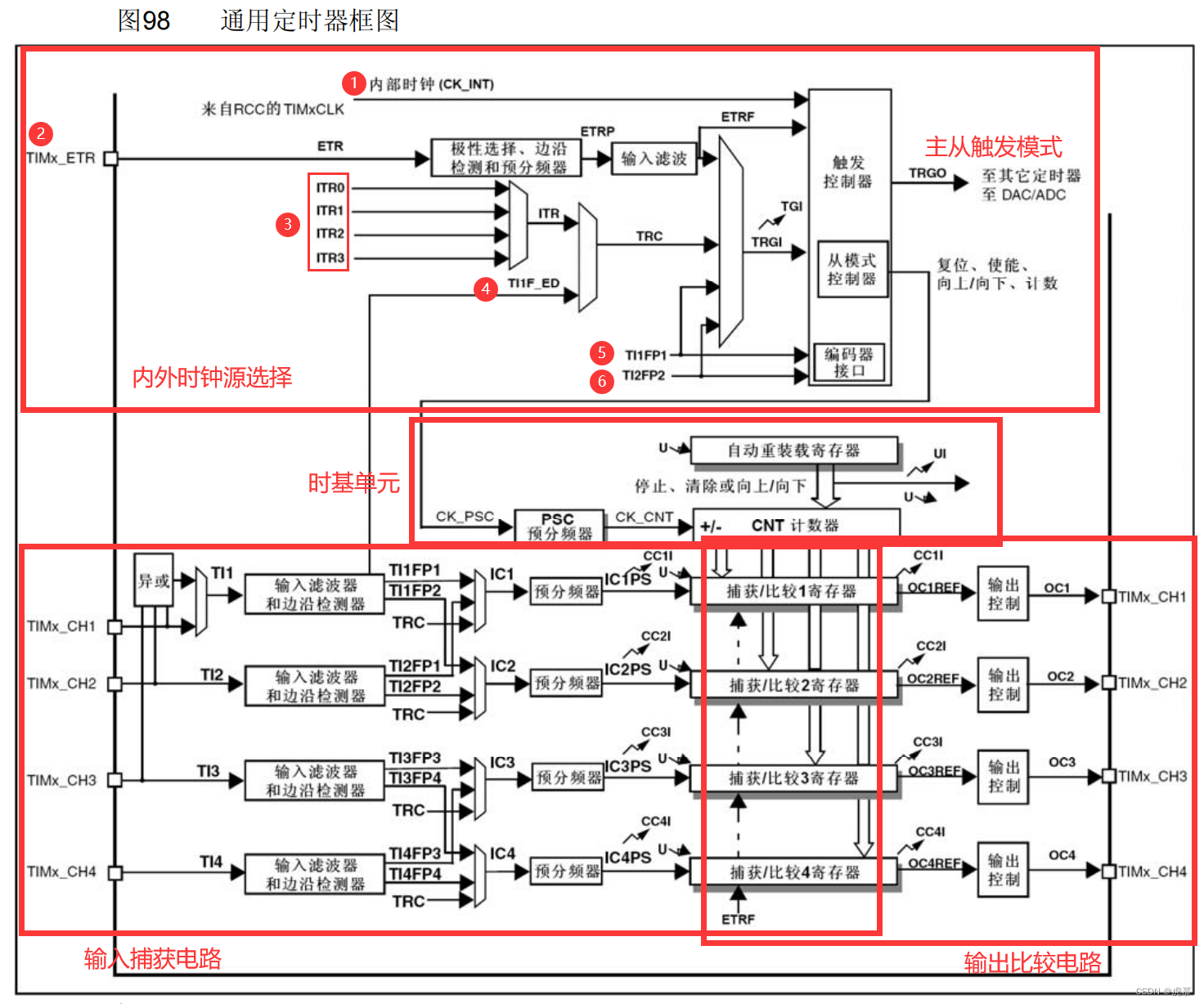
1. 预分频器：对输入的72MHz时钟进行预分频，寄存器内存储的值是实际的分频系数减一。写0就是不分频，写1就是2分频，写2就是3分频……
2. 计数器：对预分频后的计数时钟进行计数，每遇到上升沿就加一。
3. 自动重装载寄存器：存储计数的最大值，到达此值后触发中断并清零计数器。

* 折线UI：向上的折线箭头表示该位置会产生中断信号——“更新中断”（由计数值等于自动重装值产生的中断），这个中断信号会通向NVIC
* 折线U：向下的折线箭头表示该位置会产生事件——“更新事件”，这个更新事件不会触发中断，但可以触发内部其他电路的工作。比如将更新事件映射到触发输出TRGO，然后TRGO直接接到DAC的触发转换引脚上。

主模式触发DAC数模转换功能：

* stm32的一大特色就是主从触发模式，可以让内部的硬件在不受程序的控制下自动运行，可以极大地减轻CPU的负担。
* 驱动DAC正常思路及其问题：每隔一段时间就产生一个定时中断，手动更新DAC的值。但这样子会频繁的产生中断，会影响主程序的运行和其他中断的响应。
* 解决方法：定时器设计了一个主模式，使用主模式可以将定时器的“更新事件”映射到“触发输出TRGO”，然后TRGO直接接到DAC的触发转换引脚上，于是定时器的更新就不需要中断来实现了。整个过程无需软件参与，实现了硬件的自动化。

## 6.3 通用定时器



* 时基单元：PSC预分频器、CNT计数器、自动重装寄存器。通用定时器和高级定时器新增两个功能：向下计数模式、中央对齐计数模式。

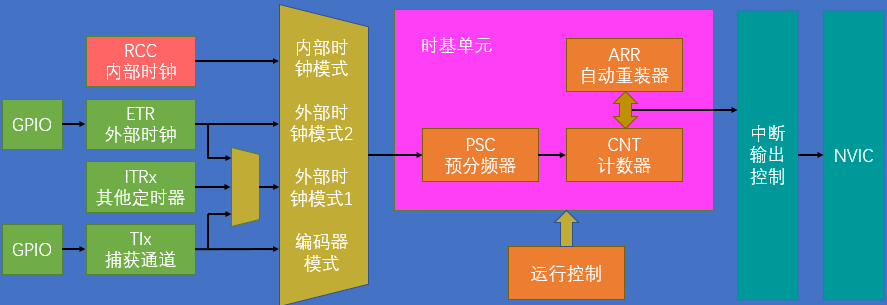
1. 向上计数模式(常用)：从0开始累加，到自动重装值触发中断。

2. 向下计数模式：从自动重装值递减，到0触发中断

3. 中央对齐计数模式：从0开始累加，到自动重装载值触发中断，然后递减，到0再次触发中断。常用于电机控制的SVPWM算法中。

内外时钟源选择和主从触发模式结构：上面的一大块。下面介绍各种各样的内外时钟源：

定时器中断基本结构：



* 时基单元：中间的粉色部分。
* 运行控制：控制寄存器的一些位，如启动停止、向上或向下计数等，操作这些寄存器就可以控制时基单元的运行了。
* 内部时钟模式、外部时钟模式2、外部时钟模式1：外部时钟源选择。这个选择器的输出就是为时基单元提供时钟。
* 编码器模式：编码器独有的模式，一般用不到。
* 中断申请控制：由于定时器内部有很多地方要申请中断，“中断申请控制”就用来使能控制这些中断是否使能。比如，中断信号会先在状态寄存器里置一个中断标志位，这个标志位会通过中断输出控制，到NVIC申请中断。

## 6.4 定时器初始化步骤

1. 开启RCC内部时钟
2. 选择时基单元的时钟源，对于定时中断，选择内部时钟源
3. 配置时基单元(预分频器、自动重装寄存器、计数模式)
4. 配置输出中断控制，允许更新中断输出到NVIC
5. 配置NVIC，在NVIC中打开定时器中断的通道，并分配一个优先级
6. 运行控制

## 6.5 TIM输出比较原理

* OC（Output Compare）输出比较
* 输出比较可以通过比较CNT与CCR寄存器值的关系，来对输出电平进行置1、置0或翻转的操作，用于输出一定频率和占空比的PWM波形
* 每个高级定时器和通用定时器都拥有4个输出比较通道
* 高级定时器的前3个通道额外拥有死区生成和互补输出的功能

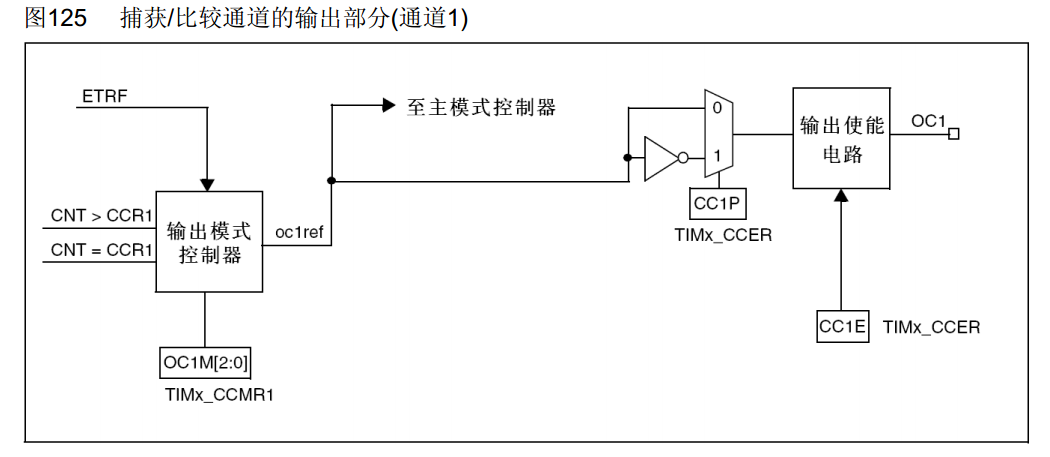
CCR：Capture Compare，捕获比较寄存器。

PWM（Pulse Width Modulation）脉冲宽度调制

* 频率 = 1 / TS，一般在 几kHz~几十kHz。
* 占空比 = TON / TS
* 分辨率 = 占空比变化步距，也就是占空比变化的精细程度。一般1%足够使用。

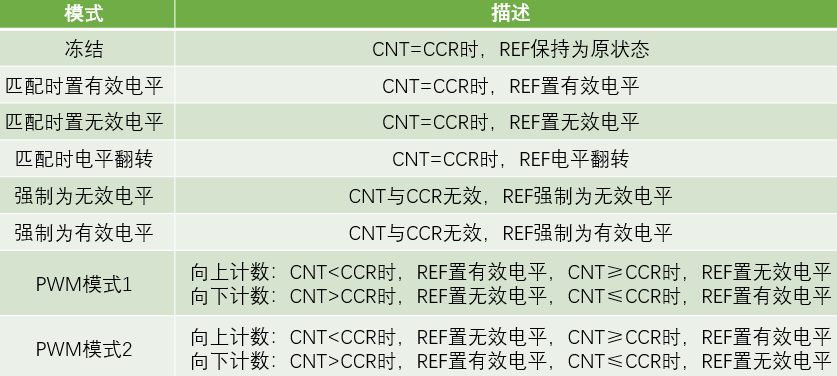
注：定时中断的频率就是PWM波的频率，只不过占空比的变化范围由自动重装载值决定。

## 6.6 通用定时器输出比较通道



REF：reference，参考信号

输出比较模式：

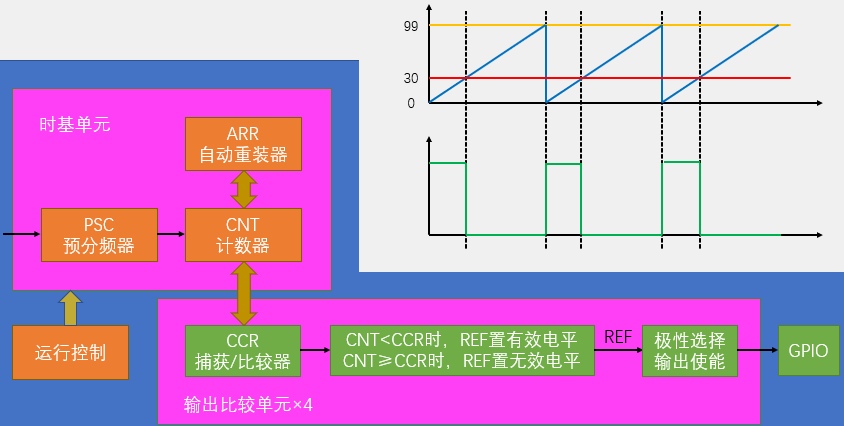


* 置有效电平：置高电平
* 置无效电平：置低电平

PWM模式1：常用向上计数的方式

## 6.7 PWM输出的基本结构

在时基单元部分，输出PWM不需要中断。



黄色线：ARR的值

蓝色线：CNT的值

红色线：CCR

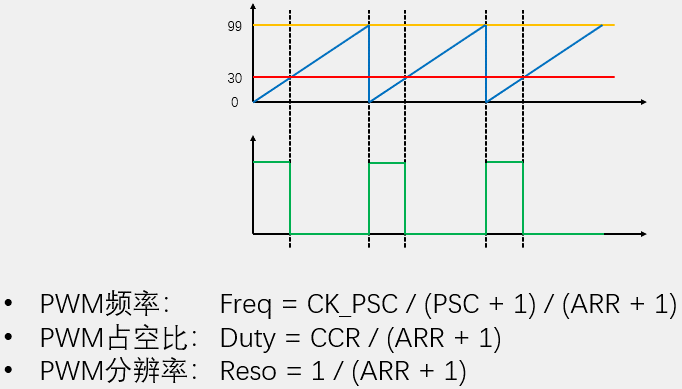
绿色线：输出PWM

CNT < CCR：输出高电平

CNT ≥ CCR：输出低电平

CCR设置的高一些，输出的占空比就变大；CCR设置的低一些，输出的占空比就变小。

参数计算：



分辨率：占空比变化的步距离。

PWM频率：一秒钟PWM有多少个周期。

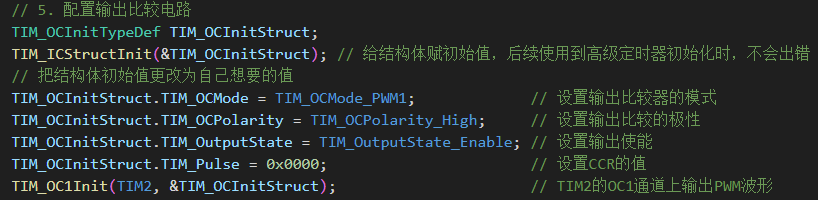
问题：如何输出一个占空比频率为1Khz，占空比可任意调节，且分辨率为1%的PWM波形，相关参数应如何进行设置。

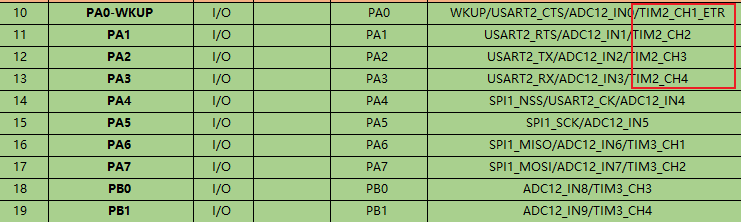
ARR = 99

PSC = 719

设计PWM输出的基本步骤：

1. 开启RCC时钟，把需要的TIM外设和GPIO外设的时钟打开
2. 配置时基单元，包括前面的时钟源选择
3. 配置输出比较单元：CCR的值，输出比较模式、极性选择、输出使能这些参数
4. 配置GPIO，把PWM对应的GPIO口，初始化为复用推挽输出的配置。





如果要使用TIM2的OC1也就是CH1通道输出PWM，只能使用PA0输出PWM

TIM2的OC2只能使用PA1输出PWM

TIM2的OC3使用PA2

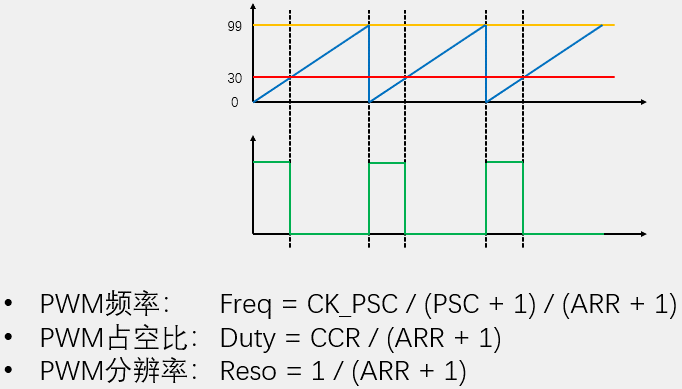
TIM2的OC4使用PA3

引脚重定义(重映射)：当一个引脚有多个功能时，如果可以进行重定义功能，则可以通过重映射的功能把原来的引脚映射到新的引脚上。这样就可以避免两个外设引脚的冲突。

如果想要定时器来控制输出引脚，需要使用复用推挽输出的模式

## 6.8 PWM驱动舵机

舵机是一种根据输入PWM信号占空比来控制输出角度的装置，根据输入PWM信号的要求，周期为20ms，高电平宽度为0.5ms~2.5ms。



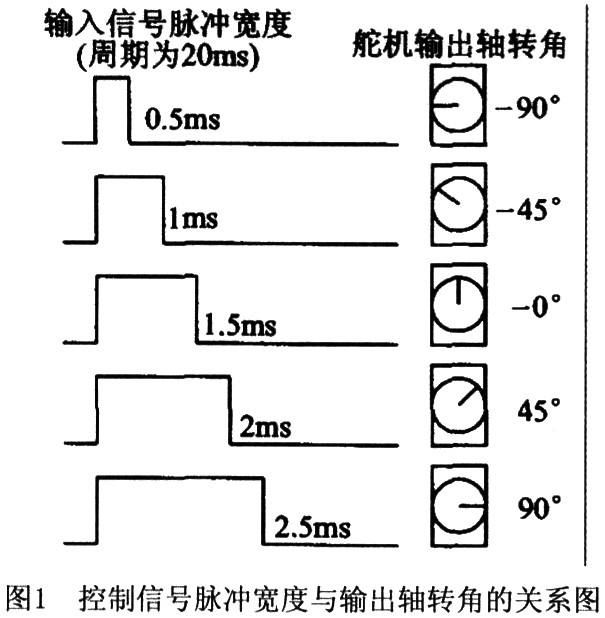
PWM参数计算：

PWM频率：50hz

ARR = 20k - 1

PSC = 72 -1

注：20K对应20ms，那么CCR设置为500，就是0.5ms，参数很直观

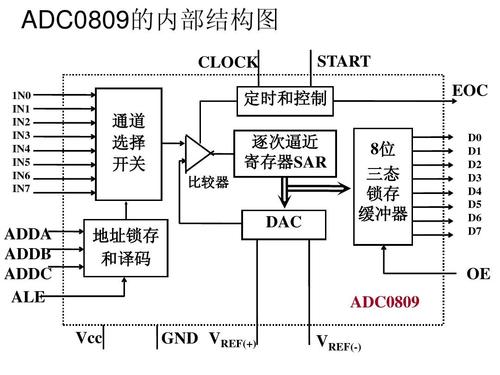


注意PWM的VCC接到5V。

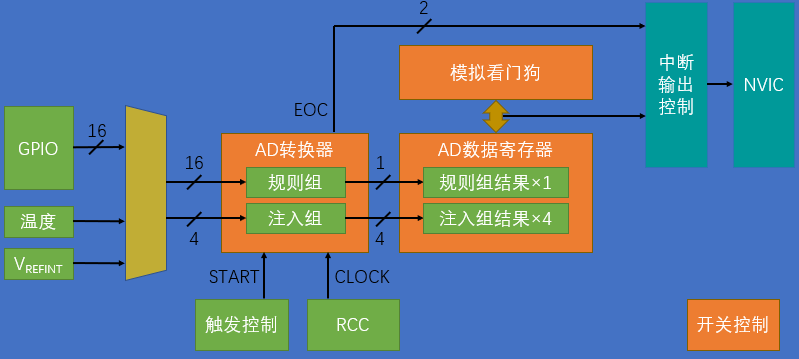
# 7. ADC数模转换器

* STM32的ADC是12位的，AD结果最大值是4095，也就是2^12-1 = 4095，对应的电压为3.3V
* STM32F103C8T6的ADC资源：ADC1、ADC2，10个外部输入通道。
* 12位逐次逼近型ADC，最大1us转换时间(1MHz)
* 18个输入通道，可测量16个外部和2个内部信号源
* 规则组和注入组两个转换单元
* 模拟看门狗自动监测输入电压范围

逐次逼近型ADC：



ADC基本结构：

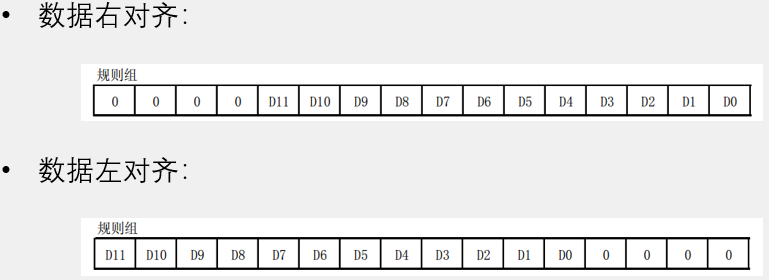


双ADC模式：ADC1和ADC2一起工作，它俩可以配合组成同步模式、交叉模式等。

转换模式：

* 单次转换，非扫描模式
* 单次转换，扫描模式
* 连续转换，非扫描模式
* 连续转换，扫描模式

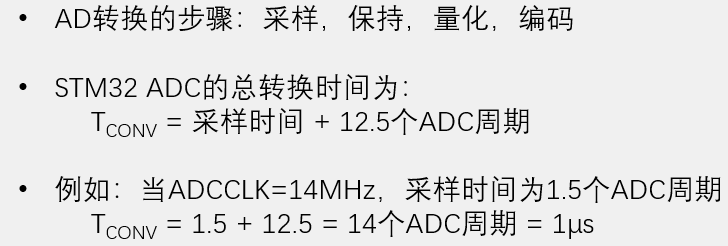
数据对齐：



STM32的ADC是12位的，它的转换结果就是一个12位的数据。但是这个数据寄存器是16位的，所以就存在一个数据对齐的问题。

一般使用右对齐，这样读取这个16位寄存器，直接就是转换结果。

转换时间：

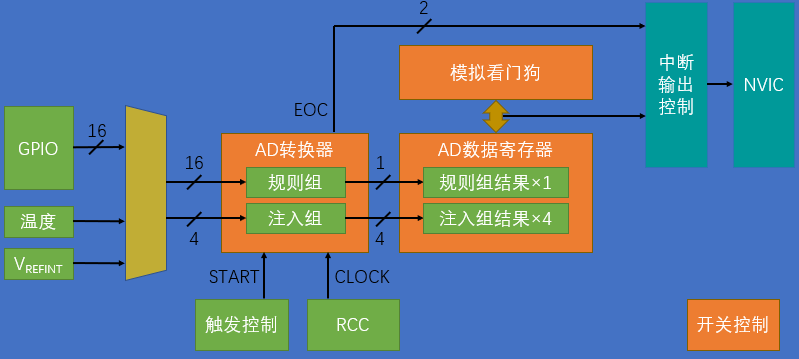


采样时间越大，越能避免一些毛刺信号的干扰。不过转换时间也会延长。

校准：

* ADC有一个内置自校准模式。校准可大幅减小因内部电容器组的变化而造成的准精度误差。校准期间，在每个电容器上都会计算出一个误差修正码(数字值)，这个码用于消除在随后的转换中每个电容器上产生的误差
* 建议在每次上电后执行一次校准
* 启动校准前， ADC必须处于关电状态超过至少两个ADC时钟周期

## 7.1 AD单通道



1. 开启RCC时钟，包括ADC和GPIO的时钟
2. 配置ADC的分频器
3. 配置GPIO，把需要用的GPIO配置成模拟输入的模式
4. 配置多路开关，把左边的通道接入到右边的规则组列表里
5. 配置ADC转换器，在库函数里使用结构体进行配置

消除ADC返回结果的方法：

1. 迟滞比较：设置两个阈值，低于下阈值开灯，高于上阈值时关灯。
2. 滤波：平滑数据(均值滤波)
3. 裁剪分辨率：直接把数据的尾数去掉，也可以减小数据的波动

## 7.2 AD多通道

扫描模式：如果想要使用扫描模式实现多通道，最好要配合DMA来实现。因为：

* 问题1：在扫描模式下，启动列表之后，每一个单独的通道转换完成之后，不会产生任何的标志位，也不会触发中断，不知道某一个通道是不是转换完了。
* 问题2：AD的速度非常快，转换一个通道大概只有几us，如果不能在几us的时间内把数据转运走，那数据就会丢失。

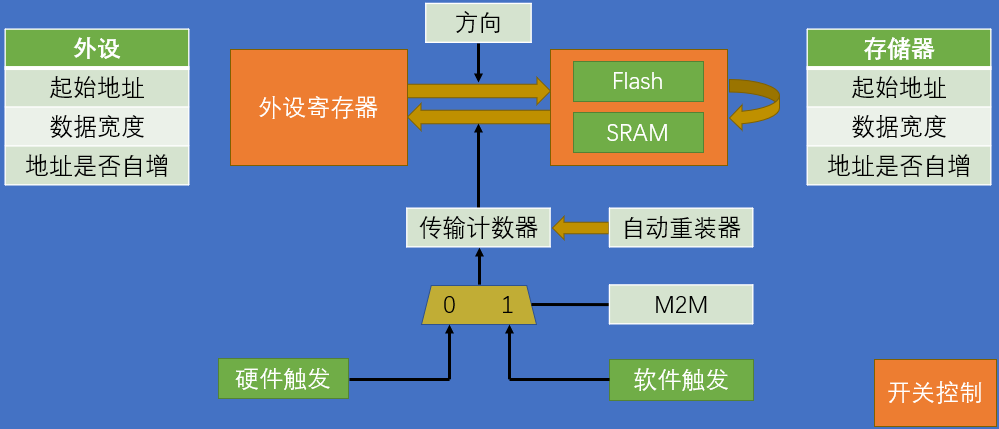
如果不使用扫描模式，可以使用单次转换、非扫描的方式来实现多通道。只需要在每次触发转换之前，手动更改一下列表第一个位置的通道就行了。

扫描模式+DMA转运数据：DMA是转运多通道数据的最优解。

# 8. DMA直接存取器存取

* DMA（Direct Memory Access）直接存储器存取
* DMA可以提供外设和存储器或者存储器和存储器之间的高速数据传输，无须CPU干预，节省了CPU的资源
* 12个独立可配置的通道： DMA1（7个通道）， DMA2（5个通道）
* 每个通道都支持软件触发和特定的硬件触发
* STM32F103C8T6 DMA资源：DMA1（7个通道）

DMA基本结构：



DMA最常见的用途：配合ADC的扫描模式。