# 节拍定时器与HAL\_Delay函数

## 系统节拍定时器

ARM Cortex-M4处理器，提供了一个系统节拍定时器:SysTick

* 24位递减计数，使用系统时钟
* 用于实时操作系统的节拍计数器
* 用于精确延时
* 用于测量一段程序的执行时间

系统节拍定时器由三个寄存器控制

节拍控制和状态寄存器STK\_CTRL：配置时钟、使能、使能中断、查看状态

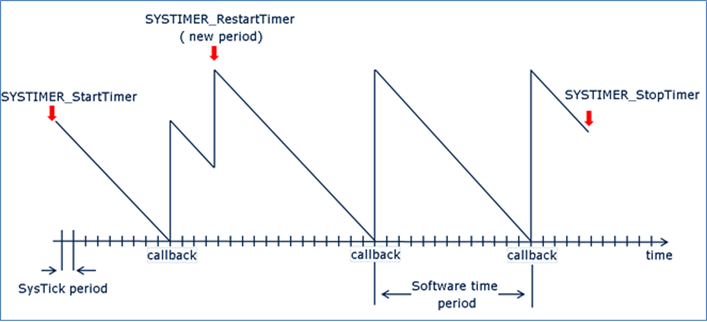
节拍计数初值寄存器STK\_LOAD: 计数器从这个值开始计数，向下计数，计到零后，又重新从这个值开始计数

节拍当前值寄存器STK\_VAL：计数器的当前值

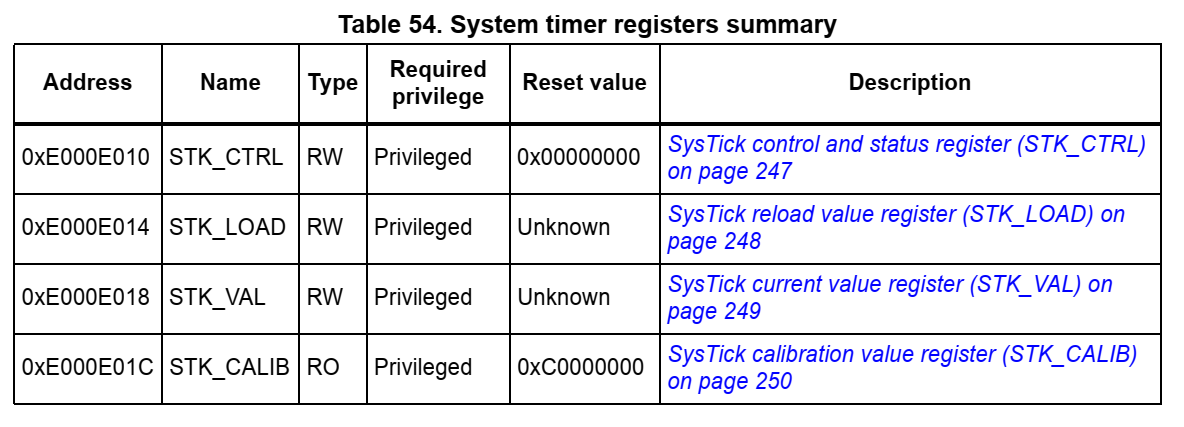
## 系统节拍定时器的工作原理

使能后，系统节拍定时器的计数器开始向下递减计数，每过一个系统时钟周期，计数值减一。

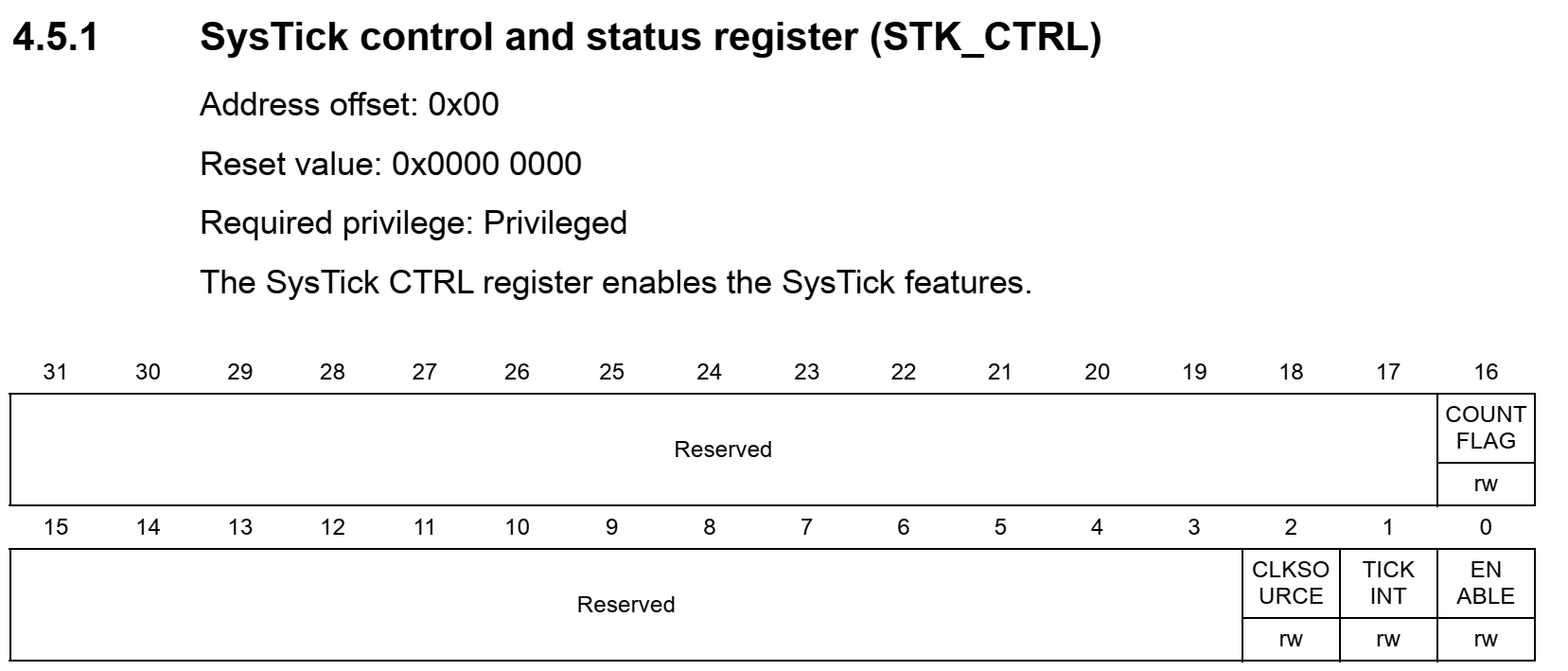
读取STK\_VAL寄存器可以获知当前的计数值。



## 系统节拍定时器的寄存器



**控制和状态寄存器STC\_CTRL**



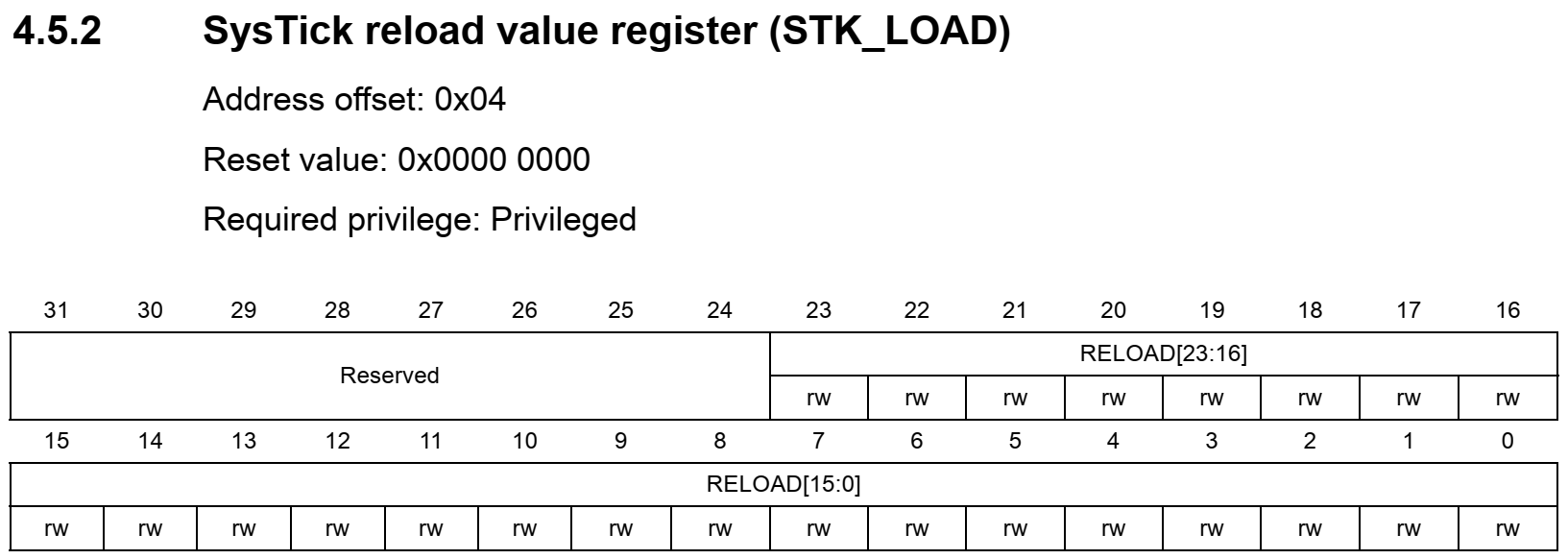
COUNTFLAG：计数器计数到零后，COUNTFLAG置1。读取该寄存器，使COUNTFLAG清零。写STK\_VAL寄存器，也可以使COUNTFLAG清零。

CLKSOURCE：时钟源选择。0：AHB/8。1：AHB（系统时钟）

TICKINT：中断使能控制。0：关闭SysTick中断。1：计数到0后，产生SysTick中断到NVIC。（在从1变到0的时刻产生中断）

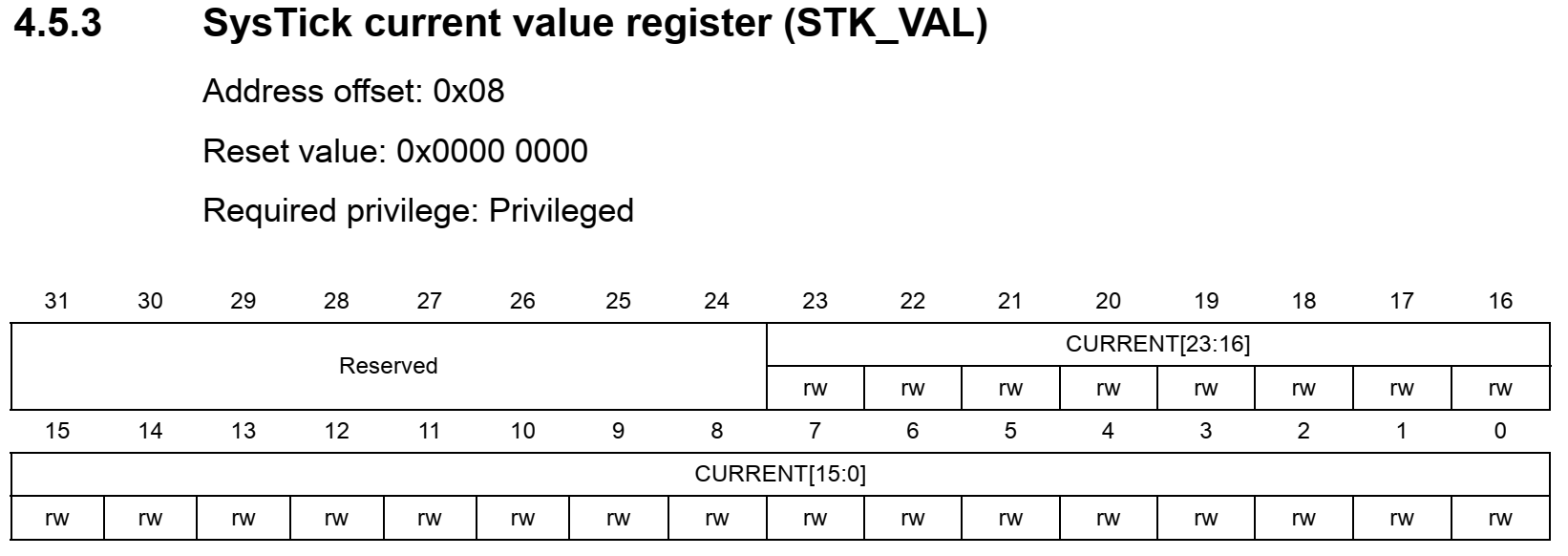
ENABLE：计数器使能控制。0：关闭计数器。1：开启计数器，计数器从STK\_LOAD开始计数。

**节拍计数初值寄存器STK\_LOAD**



RELOAD ：24位计数初值，可以是1到0x00FF FFFF之间的任何数。如果写0的话，会关闭计数器，使计数器不工作。但不会产生中断。计数器从1数到0的过程中，产生中断。

**节拍当前值寄存器STK\_VAL**



CURRENT：24位当前计数值

写入任何数据，都会使CURRENT清零，并清除STC\_CTRL的COUNTFLAG位，然后从24位计数初值STK\_LOAD重新开始计数。

## 系统节拍定时器的操作方法

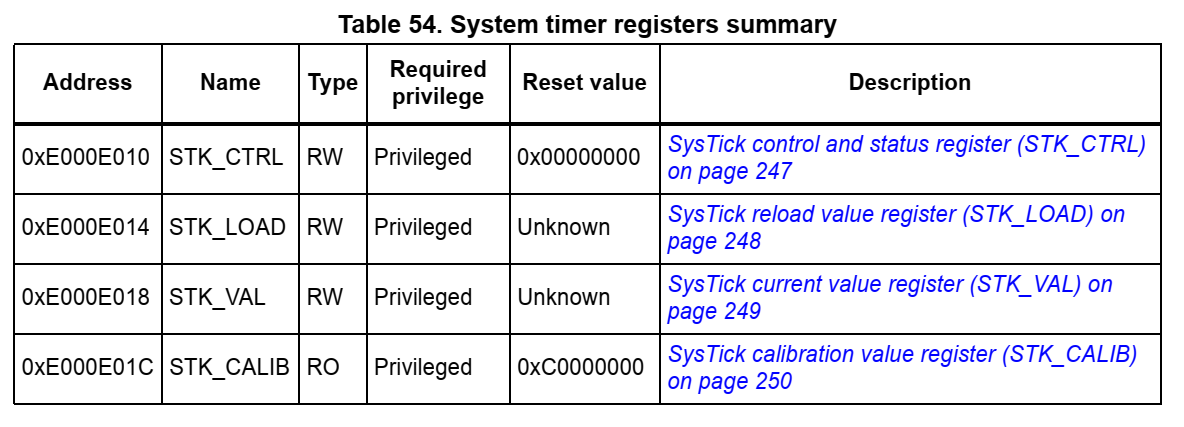
与节拍定时器有关的函数和宏定义在core\_cm4.h中。

这里面首先定义了节拍定时器模块的基地址：

**#define** SCS\_BASE (0xE000E000UL) /\*!< System Control Space Base Address \*/

**#define** SysTick\_BASE (SCS\_BASE + 0x0010UL) /\*!< SysTick Base Address \*/

与手册中节拍定时器的基地址一致：



然后定义了一个结构体类型SysTick\_Type，用于方便的操作节拍定时器模块中的各个寄存器

**typedef** **struct**

{

\_\_IOM uint32\_t CTRL; /\*!< Offset: 0x000 (R/W) SysTick Control and Status Register \*/

\_\_IOM uint32\_t LOAD; /\*!< Offset: 0x004 (R/W) SysTick Reload Value Register \*/

\_\_IOM uint32\_t VAL; /\*!< Offset: 0x008 (R/W) SysTick Current Value Register \*/

\_\_IM uint32\_t CALIB; /\*!< Offset: 0x00C (R/ ) SysTick Calibration Register \*/

} SysTick\_Type;

然后将SysTick\_BASE这个宏强制类型转换为SysTick\_Type，在用SysTick代替这个指针

**#define** SysTick ((SysTick\_Type \*) SysTick\_BASE ) /\*!< SysTick configuration struct \*/

这样，使用SysTick就可以方便的操作节拍定时器模块里的各个寄存器了，如：

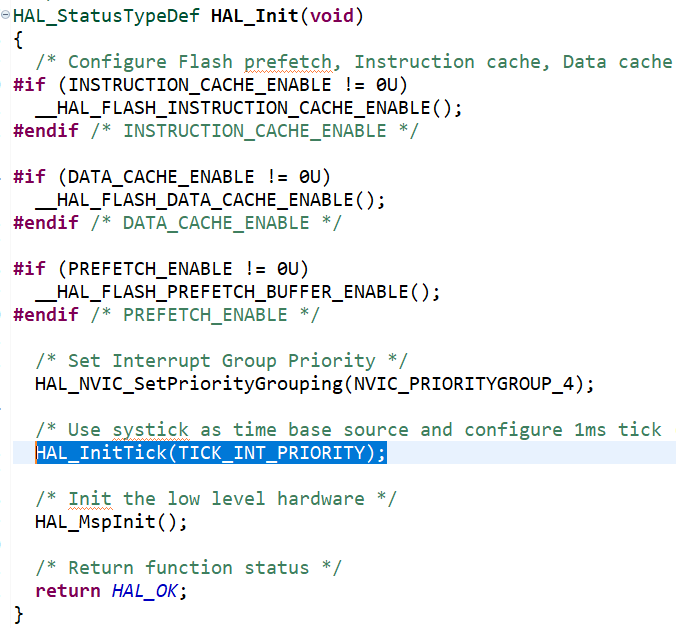
SysTick->VAL = 0UL;

## 系统节拍定时器在HAL库中的应用

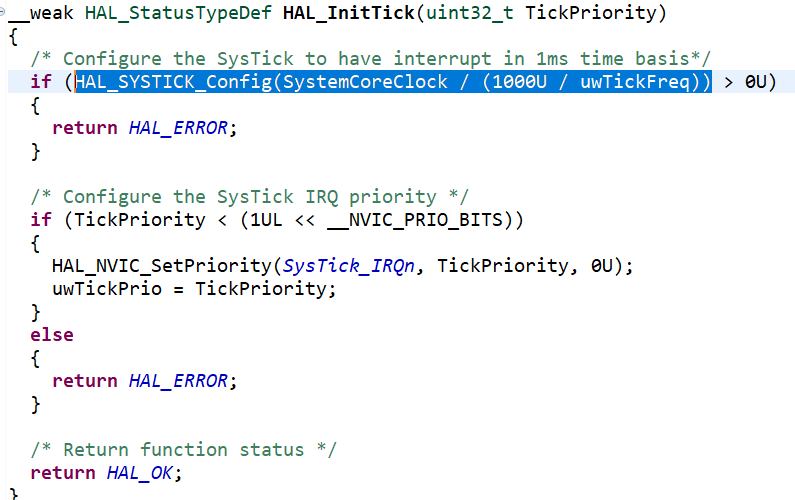
在HAL库中，强制将节拍定时器的中断频率设置为1kHz。此功能在自动生成的初始化函数HAL\_Init()函数和时钟初始化函数 SystemClock\_Config()中都有体现。

运行HAL\_Init()时，系统的时钟频率是16MHz，使用内部高速RC振荡器作为系统时钟。而在运行SystemClock\_Config()后，系统时钟变为锁相环输出的84MHz，因此前后两种情况系统时钟不同，要保证节拍定时器的中断频率一致，计数初始值需要重新设置。

在HAL\_Init();中，使用HAL\_InitTick(TICK\_INT\_PRIORITY);函数初始化节拍定时器，其中TICK\_INT\_PRIORITY=15；



HAL\_InitTick函数中，又调用了HAL\_SYSTICK\_Config(SystemCoreClock / (1000U / uwTickFreq))函数



SystemCoreClock为当前系统时钟频率，初值在system\_stm32f4xx.c中定义：

uint32\_t SystemCoreClock = 16000000;

uwTickFreq为节拍定时器的中断频率，其值为1，

HAL\_TickFreqTypeDef uwTickFreq = *HAL\_TICK\_FREQ\_DEFAULT*; /\* 1KHz \*/

**typedef** **enum**

{

*HAL\_TICK\_FREQ\_10HZ* = 100U,

*HAL\_TICK\_FREQ\_100HZ* = 10U,

*HAL\_TICK\_FREQ\_1KHZ* = 1U,

*HAL\_TICK\_FREQ\_DEFAULT* = *HAL\_TICK\_FREQ\_1KHZ*

} HAL\_TickFreqTypeDef;

因此可以看出HAL\_SYSTICK\_Config函数的输入参数为节拍定时器的计数初值。HAL\_SYSTICK\_Config又调用了

SysTick\_Config函数：

uint32\_t **HAL\_SYSTICK\_Config**(uint32\_t TicksNumb)

{

**return** SysTick\_Config(TicksNumb);

}

SysTick\_Config函数的定义为：

\_\_STATIC\_INLINE uint32\_t **SysTick\_Config**(uint32\_t ticks)

{

**if** ((ticks - 1UL) > SysTick\_LOAD\_RELOAD\_Msk)

{

**return** (1UL); /\* Reload value impossible \*/

}

SysTick->LOAD = (uint32\_t)(ticks - 1UL); /\* set reload register \*/

NVIC\_SetPriority (*SysTick\_IRQn*, (1UL << \_\_NVIC\_PRIO\_BITS) - 1UL); /\* set Priority for Systick Interrupt \*/

SysTick->VAL = 0UL; /\* Load the SysTick Counter Value \*/

SysTick->CTRL = SysTick\_CTRL\_CLKSOURCE\_Msk |

SysTick\_CTRL\_TICKINT\_Msk |

SysTick\_CTRL\_ENABLE\_Msk; /\* Enable SysTick IRQ and SysTick Timer \*/

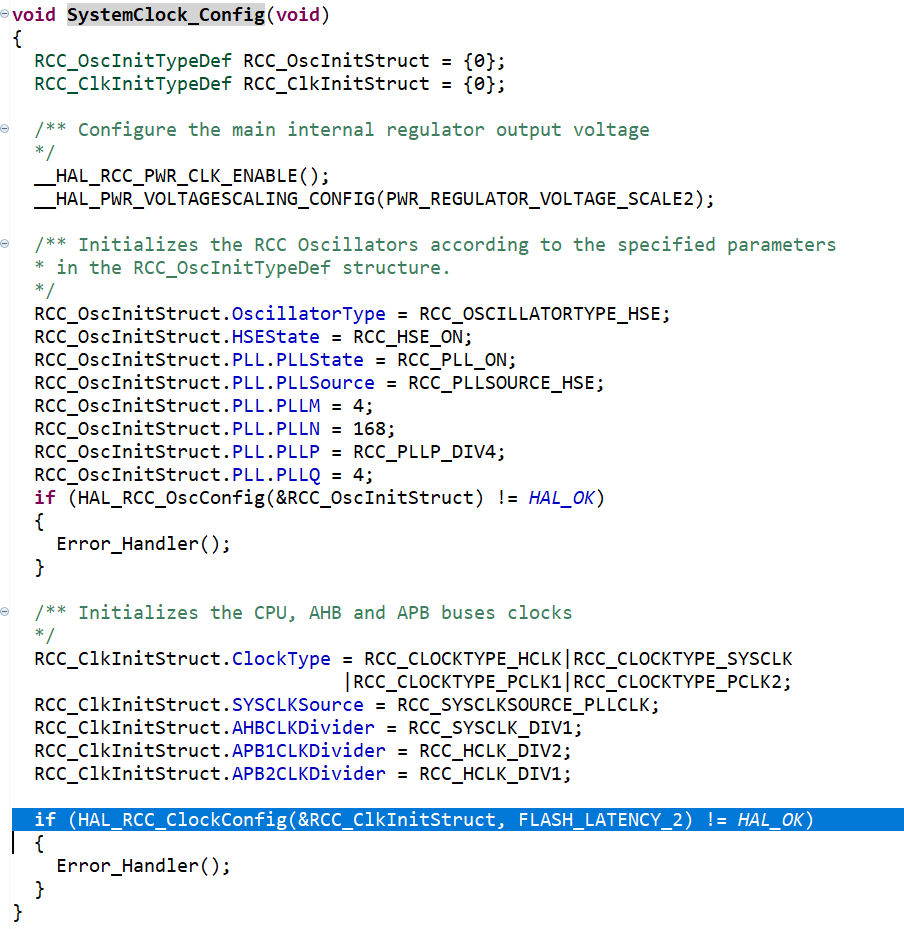
**return** (0UL); /\* Function successful \*/

}

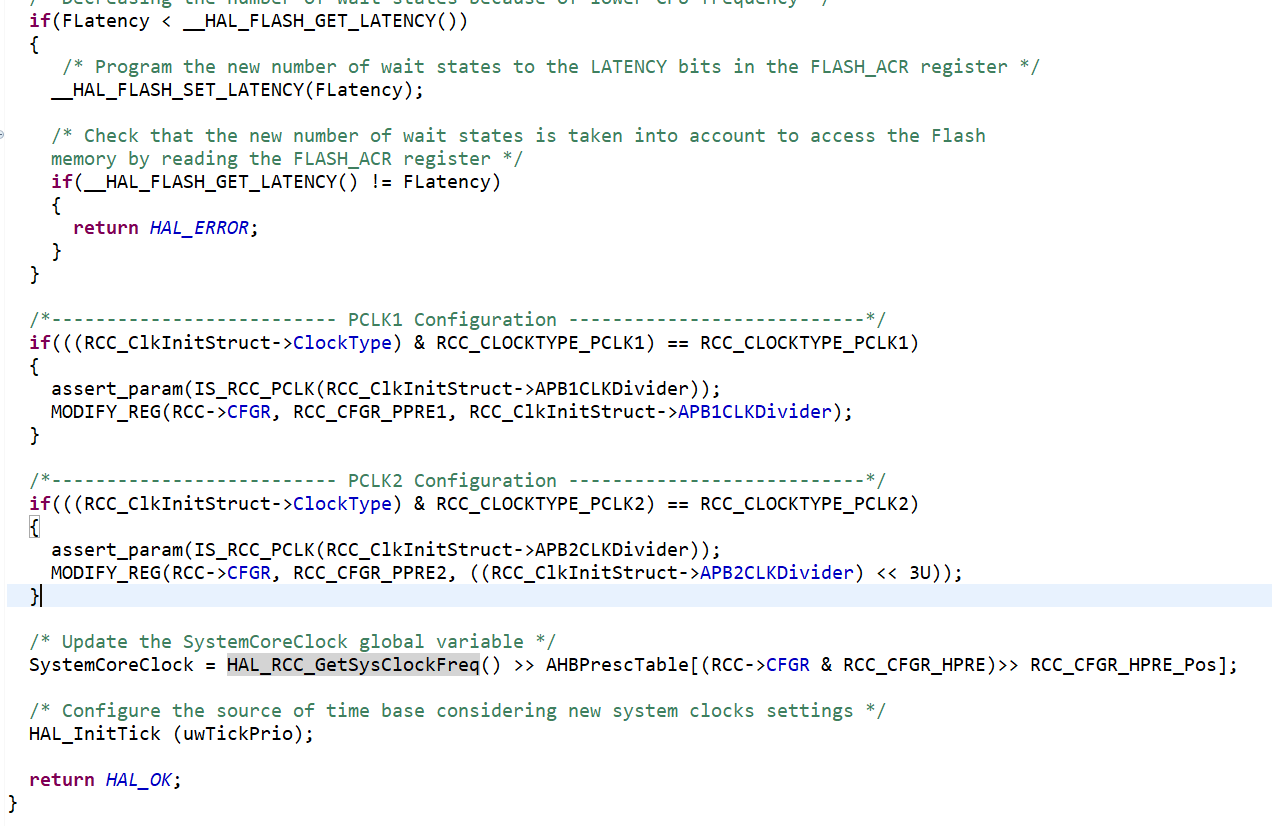
可见，SysTick\_Config函数将计数初始赋给了STK\_LOAD寄存器，然后写入STK\_VAL寄存器使其重置。最后设置STC\_CTRL寄存器，使用系统时钟计数，开启中断，开始计数。

SystemClock\_Config()函数前的代码，运行频率都为16MHz。

在SystemClock\_Config函数中，调用了HAL\_RCC\_ClockConfig函数，设置系统时钟为PLL的输出。



HAL\_RCC\_ClockConfig函数中，将变量SystemCoreClock设置为84000000。



最后再次调用 HAL\_InitTick (uwTickPrio);函数，对节拍定时器模块进行了重新设置，注意时重新设计STK\_LOAD寄存器，保证更改系统时钟后，节拍定时器的中断频率还是1000Hz。

在stm32f4xx\_it.c中，定义了中断服务函数SysTick\_Handler。每隔1ms，该函数自动运行一次。

**void** **SysTick\_Handler**(**void**)

{

/\* USER CODE BEGIN SysTick\_IRQn 0 \*/

/\* USER CODE END SysTick\_IRQn 0 \*/

HAL\_IncTick();

/\* USER CODE BEGIN SysTick\_IRQn 1 \*/

/\* USER CODE END SysTick\_IRQn 1 \*/

}

中断服务函数中，运行了HAL\_IncTick();函数。

这个函数每运行一次，就把一个32位的无符号变量uwTick加1。其中uwTickFreq为1。

\_\_weak **void** **HAL\_IncTick**(**void**)

{

uwTick += uwTickFreq;

}

## HAL\_Delay函数的实现

编程中常用的HAL\_Delay函数，使怎样实现的呢？我们看一下他的实现代码。

\_\_weak **void** **HAL\_Delay**(uint32\_t Delay)

{

uint32\_t tickstart = HAL\_GetTick();

uint32\_t wait = Delay;

/\* Add a freq to guarantee minimum wait \*/

**if** (wait < HAL\_MAX\_DELAY)

{

wait += (uint32\_t)(uwTickFreq);

}

**while**((HAL\_GetTick() - tickstart) < wait)

{

}

}

首先，他使用HAL\_GetTick函数获取uwTick变量，保存到tickstart变量中。然后将要延时的时间赋给了wait变量。然后将wait变量加1。

然后不断的调用HAL\_GetTick函数获取uwTick变量，如果uwTick变量与tickstart的差值比wait的值小，就一直等待。直到uwTick变量与tickstart的差值与wait相等，就退出函数。

由于uwTick变量的值，每隔1ms就自动加1，所以wait的值的大小可以决定延时时间的长短。

但是，有两点要注意：

**第一**，如果HAL\_Delay的输入参数Delay很小，比如Delay=1。那么wait为2，实际的延时时间并不是1ms，而是1ms-2ms。这取决于运行HAL\_Delay函数的时候STK\_VAL的值是多少。因此默认生成的HAL\_Delay函数，并不能提供精确的延时，而是保证了最小延时。

**第二**，由于uwTick变量的值，是在中断服务函数SysTick\_Handler中自加的。但是SysTick的中断优先级为15，因此在优先级更高的中断服务函数中，不能使用HAL\_Delay函数。运行其他中断服务函数时，由于SysTick\_Handler优先级低，无法运行，因此uwTick的值不会变化。这导致HAL\_Delay函数不能在中断服务函数中使用。

## HAL\_Delay函数的改进

有没有什么方法，可以克服HAL\_Delay函数计数不确定的问题？

可以看出，stm32f4xx\_hal.c自动生成的HAL\_Delay函数是一个弱函数。

\_\_weak **void** **HAL\_Delay**(uint32\_t Delay)

因此可以对他重新定义，并进行改进。

由于调用HAL\_Delay时，并不知道STK\_VAL的值是多少，因此延时的时间有不确定的因此，因此可以在进入HAL\_Delay后，立刻将STK\_VAL重置，从而确保延时的一致性。

在main.c的/\* USER CODE BEGIN 0 \*/和/\* USER CODE END 0 \*/之间重新定义HAL\_Delay函数:

**void** **HAL\_Delay**(uint32\_t Delay)

{

uint32\_t tickstart;

uint32\_t wait;

SysTick->VAL = 0UL;

wait=Delay;

**if** (wait > HAL\_MAX\_DELAY)

{

wait = HAL\_MAX\_DELAY;

}

tickstart = HAL\_GetTick();

**while**((HAL\_GetTick() - tickstart) < wait)

{

}

}

/\* USER CODE END 0 \*/

进入函数后，立刻使用SysTick->VAL = 0UL;将计数值重置。这时会立刻触发节拍定时器中断，将uwTick变量加一。然后使用HAL\_GetTick()函数获取当前的uwTick变量，保存到tickstart中。最后不断的获取uwTick变量，如果uwTick变量与tickstart的差值比wait小，就一直等待。直到uwTick变量与tickstart的差值与wait相等，就退出函数，延时结束。

这样改进后，就减弱了延时函数HAL\_Delay的不确定性。