---

title: 一种Cortex-M内核中的精确延时方法 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-07 21:58:46 img: coverImg: password: summary: tags:

- Cortex-M
- STM32
- DWT categories: STM32

本文介绍一种Cortex-M内核中的精确延时方法

前言

为什么要学习这种延时的方法？

1. 很多时候我们跑操作系统，就一般会占用一个硬件定时器——SysTick，而我们一般操作系统的时钟节拍一般是设置100-1000HZ，也就是1ms——10ms产生一次中断。很多裸机教程使用延时函数又是基于SysTick的，这样一来又难免产生冲突。
2. 很多人会说，不是还有定时器吗，定时器的计时是超级精确的。这点我不否认，但是假设，如果一个系统，总是进入定时器中断（10us一次/1us一次/0.5us一次），那整个系统就会经常被打断，线程的进行就没办法很好运行啊。此外还消耗一个硬件定时器资源，一个硬件定时器可能做其他事情呢！
3. 对应ST HAL库的修改，其实杰杰个人觉得吧，ST的东西什么都好，就是出的HAL库太恶心了，没办法，而HAL库中有一个HAL_Delay()，他也是采用SysTick延时的，在移植操作系统的时候，会有诸多不便，不过好在，HAL_Delay()是一个弱定义的，我们可以重写这个函数的实现，那么，采用内核延时当然是最好的办法啦（个人是这么觉得的）当然你有能力完全用for循环写个简单的延时还是可以的。
4. 可能我说的话没啥权威，那我就引用Cortex-M3权威指南中的一句话——“DWT 中有剩余的计数器，它们典型地用于程序代码的“性能速写”（profiling）。通过编程它们，就可以让它们在计数器溢出时发出事件（以跟踪数据包的形式）。最典型地，就是使用 CYCCNT寄存器来测量执行某个任务所花的周期数，这也可以用作时间基准相关的目的（操作系统中统计 CPU使用率可以用到它）。”

Cortex-M中的DWT

在Cortex-M里面有一个外设叫DWT(Data Watchpoint and Trace)，是用于系统调试及跟踪，

6.2 详细的框图

CM3 处理器其实是个大礼包，里面除了处理核心外，还有了好多其它组件，以用于系统管理和调试支持。

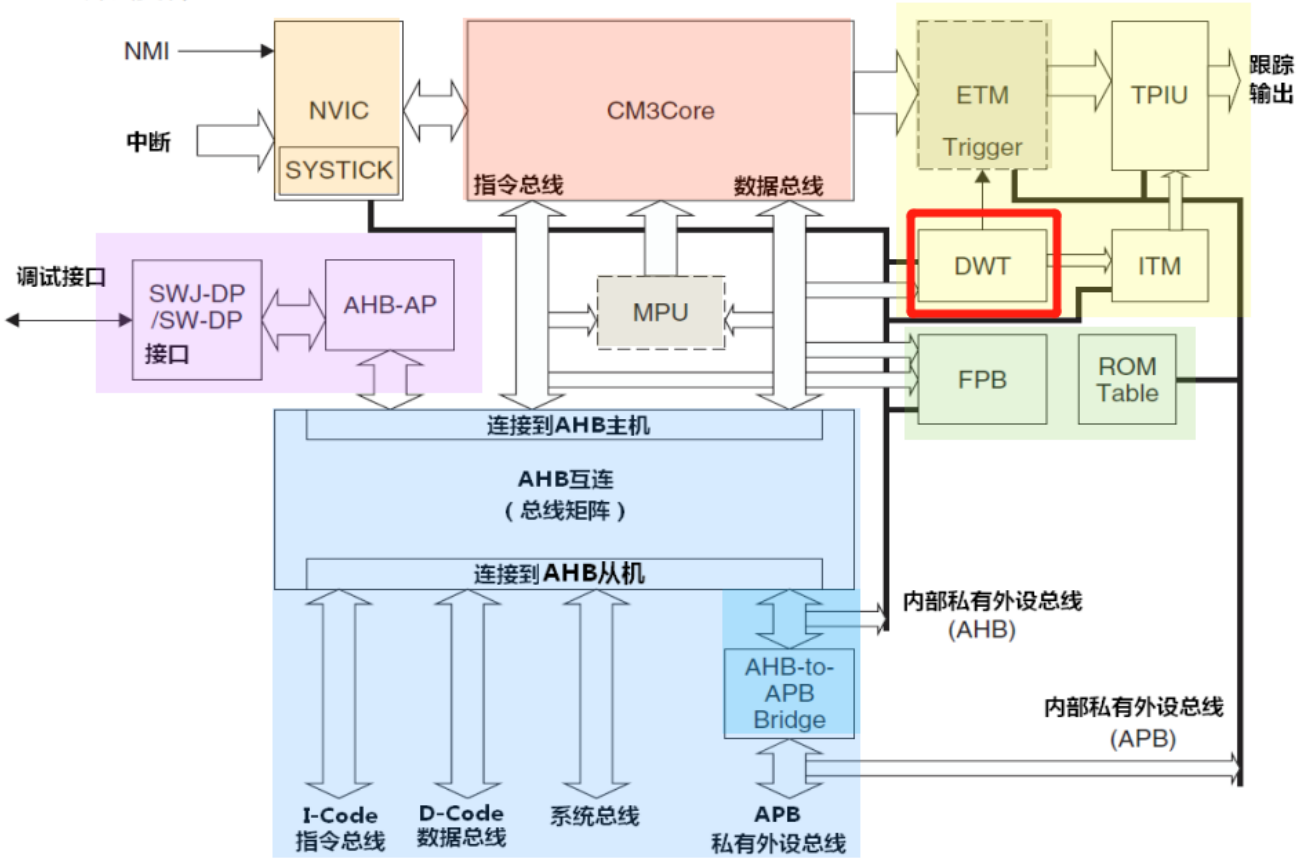

图 6.3 Cortex-M3 处理器系统方框图

它有一个32位的寄存器叫CYCCNT，它是一个向上的计数器，记录的是内核时钟运行的个数，内核时钟跳动一次，该计数器就加1，精度非常高，决定内核的频率是多少，如果是F103系列，内核时钟是72M，那精度就是1/72M = 14ns，而程序的运行时间都是微秒级别的，所以14ns的精度是远远够的。最长能记录的时间为：60s=2的32次方/72000000(假设内核频率为72M，内核跳一次的时间大概为1/72M=14ns)，而如果是H7这种400M主频的芯片，那它的计时精度高达2.5ns（1/400000000 = 2.5），而如果是 i.MX RT1052这种比较牛逼的处理器，最长能记录的时间为： 8.13s=2的32次方/528000000 (假设内核频率为528M，内核跳一次的时间大概

为1/528M=1.9ns) 。当CYCCNT溢出之后，会清0重新开始向上计数。

16.2 跟踪组件：数据观察点与跟踪(DWT)

本节的主角是 DWT，它提供的调试功能包括：

1. 它包含了 4 个比较器，可以配置成在发生比较匹配时，执行如下动作：
   a) 硬件观察点（产生一个观察点调试事件，并且用它来调用调试模式，包括停机模式和调试监视器模式
   b) ETM 触发，可以触发 ETM 发出一个数据包，并汇入指令跟踪数据流中
   c) 程序计数器（PC）采样器事件触发
   d) 数据地址采样器触发
   e) 第一个比较器还能用于比较时钟周期计数器（CYCCNT），用于取代对数据地址的比较
2. 作为计数器，DWT 可以对下列项目进行计数：
   a) 时钟周期（CYCCNT）
   b) 被折叠（Folded）的指令
   c) 对加载/存储单元（LSU）的操作
   d) 睡眠的时钟周期
   e) 每指令周期数（CPI）
   f) 中断的额外开销（overhead）
3. 以固定的周期采样 PC 的值
4. 中断事件跟踪

当用于硬件观察点或 ETM 触发时，比较器既可以比较数据地址，也可以比较程序计数器 PC。当用于其它功能时，比较器则只能比较数据地址。

每一个比较器都有 3 个寄存器

- COMP 寄存器
- MASK 寄存器
- FUNCTION 控制寄存器

其中，COMP 寄存器是一个 32 位寄存器，用于存储要比较的值。MASK 寄存器可以用于掩蔽数据地址的一些位，被掩蔽的位不参与比较。如表 16.1 所示：

> m3、m4、m7杰杰实测可用（m0不可用）。 精度：1/内核频率(s)。

要实现延时的功能，总共涉及到三个寄存器：DEMCR 、DWT_CTRL、DWT_CYCCNT，分别用于开启DWT功能、开启CYCCNT及获得系统时钟计数值。

DEMCR

想要使能DWT外设，需要由另外的内核调试寄存器DEMCR的位24控制，写1使能（划重点啦，要考试！！）。DEMCR的地址是**0xE000 EDFC**

表 15.2 调试及监视器控制寄存器 DEMCR （地址：0xE000_EDFC）

位段	名称	类型	复位值	描述
24	TRCENA	RW	0*	跟踪系统使能位。在使用 DWT,ETM,ITM 和 TPIU 前，必须先设置此位
23:20	保留			
19	MON_REQ	RW	0	1=调试监视器异常不是由硬件调试事件触发，而是由软件手工悬起的
18	MON_STEP	RW	0	让处理器单步执行，在 MON_EN=1 时有效
17	MON_PEND	RW	0	悬起监视器异常请求，内核将在优先级允许时响应
16	MON_EN	RW	0	使能调试监视器异常
15:11	保留			
10	VC_HARDERR	RW	0*	发生硬 fault 时停机调试
9	VC_INTERR	RW	0*	指令/异常服务错误时停机调试
8	VC_BUSERR	RW	0*	发生总线 fault 时停机调试
7	VC_STATERR	RW	0*	发生用法 fault 时停机调试
6	VC_CHKERR	RW	0*	发生用法 fault 使能的检查错误时停机调试（如未对齐，除数为零）
5	VC_NOCPERR	RW	0*	发生用法 fault 之无处理器错误时停机调试
4	VC_MMERR	RW	0*	发生存储器管理 fault 时停机调试
3:1	保留			

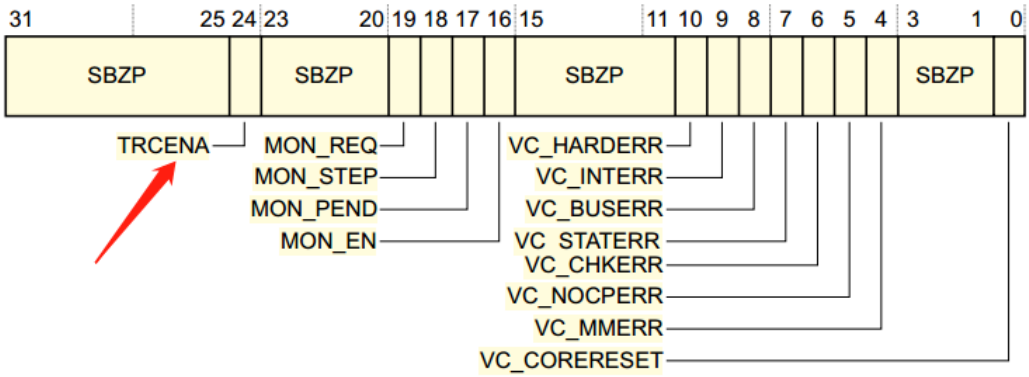

Figure 10-3 Debug Exception and Monitor Control Register bit assignments

Table 10-4 shows the bit functions of the Debug Exception and Monitor Control Register.

Table 10-4 Debug Exception and Monitor Control Register

Bits	Type	Field	Function
[31:25]	-	-	Reserved, SBZP
[24]	Read/write	**TRCENA**	This bit must be set to 1 to enable use of the trace and debug blocks: • *Data Watchpoint and Trace* (DWT) • *Instrumentation Trace Macrocell* (ITM) • *Embedded Trace Macrocell* (ETM) • *Trace Port Interface Unit* (TPIU). This enables control of power usage unless tracing is required. The application can enable this, for ITM use, or use by a debugger. **Note** If no debug or trace components are present in the implementation then it is not possible to set TRCENA.

关于**DWT_CYCCNT**

使能DWT_CYCCNT寄存器之前，先清0。 让我们看看DWT_CYCCNT的基地址，从ARM-Cortex-M手册中可以看到其基地址是**0xE000 1004**，复位默认值是0，而且它的类型是可读可写的，我们往**0xE000 1004**这个地址写0

就将DWT_CYCCNT清0了。

Table 3-4 DWT register summary

Name	Type	Address	Reset value	Description
DWT_CTRL	Read/write	0xE0001000	0x40000000	DWT Control Register
DWT_CYCCNT	Read/write	0xE0001004	0x00000000	DWT Current PC Sampler Cycle Count Register
DWT_CPICNT	Read/write	0xE0001008	-	DWT Current CPI Count Register
DWT_EXCCNT	Read/write	0xE000100C	-	DWT Current Interrupt Overhead Count Register
DWT_SLEEPCNT	Read/write	0xE0001010	-	DWT Current Sleep Count Register
DWT_LSUCNT	Read/write	0xE0001014	-	DWT Current LSU Count Register
DWT_FOLDCNT	Read/write	0xE0001018	-	DWT Current Fold Count Register
DWT_PCSR	Read-only	0xE000101C	-	DWT PC Sample Register
DWT_COMP0	Read/write	0xE0001020	-	DWT Comparator Register
DWT_MASK0	Read/write	0xE0001024	-	DWT Mask Registers
DWT_FUNCTION0	Read/write	0xE0001028	0x00000000	DWT Function Registers
DWT_COMP1	Read/write	0xE0001030	-	DWT Comparator Register

关于**CYCCNTENA**

CYCCNTENA Enable the CYCCNT counter. If not enabled, the counter does not count and no event is generated for PS sampling or CYCCNTENA. In normal use, the debugger must initialize the CYCCNT counter to 0. 它是DWT控制寄存器的第一位，写1使能，则启用CYCCNT计数器，否则CYCCNT计数器将不会工作。

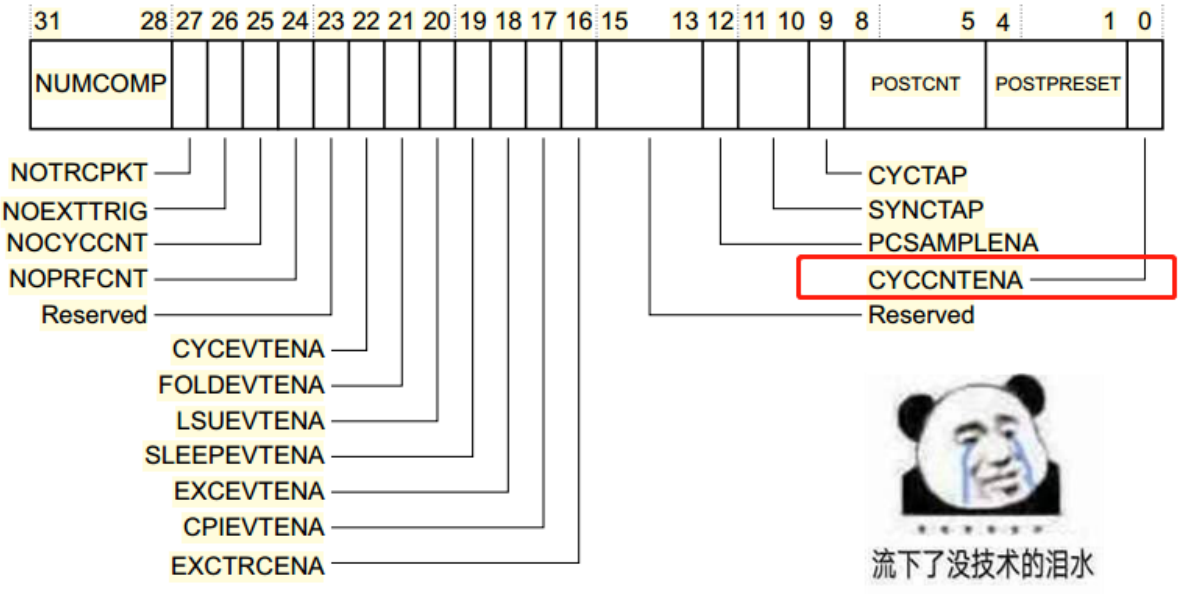

Figure 11-5 DWT Control Register bit assignments

Table 11-7 describes the bit assignments of the DWT Control Register.

综上所述

想要使用**DWT**的**CYCCNT**步骤：

> 1. 先使能DWT外设，这个由另外内核调试寄存器DEMCR的位24控制，写1使能
> 2. 使能CYCCNT寄存器之前，先清0。
> 3. 使能CYCCNT寄存器，这个由DWT的CYCCNTENA 控制，也就是DWT控制寄存器的位0控制，写1使能

代码实现

```
/**
  ******************************************************************
  * @file    core_delay.c
  * @author  fire
  * @version V1.0
  * @date    2018-xx-xx
  * @brief   使用内核寄存器精确延时
  ******************************************************************
  * @attention
  *
  * 实验平台:野火 STM32开发板
  * 论坛     :http://www.firebbs.cn
  * 淘宝     :https://fire-stm32.taobao.com
  *
  ******************************************************************
  */

#include "./delay/core_delay.h"

/*
*************************************************************************
*         时间戳相关寄存器定义
*************************************************************************
*/
/*
 在Cortex-M里面有一个外设叫DWT(Data Watchpoint and Trace)，
 该外设有一个32位的寄存器叫CYCCNT，它是一个向上的计数器，
 记录的是内核时钟运行的个数，最长能记录的时间为：
 10.74s=2的32次方/400000000
 (假设内核频率为400M，内核跳一次的时间大概为1/400M=2.5ns)
 当CYCCNT溢出之后，会清0重新开始向上计数。
 使能CYCCNT计数的操作步骤：
 1、先使能DWT外设，这个由另外内核调试寄存器DEMCR的位24控制，写1使能
 2、使能CYCCNT寄存器之前，先清0
 3、使能CYCCNT寄存器，这个由DWT_CTRL(代码上宏定义为DWT_CR)的位0控制，写1使能
 */


#define  DWT_CR      *(__IO uint32_t *)0xE0001000
#define  DWT_CYCCNT  *(__IO uint32_t *)0xE0001004
#define  DEM_CR      *(__IO uint32_t *)0xE000EDFC
```

```
#define  DEM_CR_TRCENA                   (1 << 24)
#define  DWT_CR_CYCCNTENA                (1 <<  0)


/**
  * @brief  初始化时间戳
  * @param  无
  * @retval 无
  * @note   使用延时函数前，必须调用本函数
  */
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /* 使能DWT外设 */
    DEM_CR |= (uint32_t)DEM_CR_TRCENA;

    /* DWT CYCCNT寄存器计数清0 */
    DWT_CYCCNT = (uint32_t)0u;

    /* 使能Cortex-M DWT CYCCNT寄存器 */
    DWT_CR |= (uint32_t)DWT_CR_CYCCNTENA;

    return HAL_OK;
}

/**
  * @brief  读取当前时间戳
  * @param  无
  * @retval 当前时间戳，即DWT_CYCCNT寄存器的值
  */
uint32_t CPU_TS_TmrRd(void)
{
  return ((uint32_t)DWT_CYCCNT);
}

/**
  * @brief  读取当前时间戳
  * @param  无
  * @retval 当前时间戳，即DWT_CYCCNT寄存器的值
  */
uint32_t HAL_GetTick(void)
{
  return ((uint32_t)DWT_CYCCNT/SysClockFreq*1000);
}


/**
  * @brief  采用CPU的内部计数实现精确延时，32位计数器
  * @param  us : 延迟长度，单位1 us
  * @retval 无
  * @note   使用本函数前必须先调用CPU_TS_TmrInit函数使能计数器，
            或使能宏CPU_TS_INIT_IN_DELAY_FUNCTION
            最大延时值为8秒，即8*1000*1000
  */
void CPU_TS_Tmr_Delay_US(uint32_t us)
```

```
{
  uint32_t ticks;
  uint32_t told,tnow,tcnt=0;

  /* 在函数内部初始化时间戳寄存器， */
#if (CPU_TS_INIT_IN_DELAY_FUNCTION)
  /* 初始化时间戳并清零 */
  HAL_InitTick(5);
#endif

  ticks = us * (GET_CPU_ClkFreq() / 1000000);  /* 需要的节拍数 */
  tcnt = 0;
  told = (uint32_t)CPU_TS_TmrRd();         /* 刚进入时的计数器值 */

  while(1)
  {
    tnow = (uint32_t)CPU_TS_TmrRd();
    if(tnow != told)
    {
        /* 32位计数器是递增计数器 */
      if(tnow > told)
      {
        tcnt += tnow - told;
      }
      /* 重新装载 */
      else
      {
        tcnt += UINT32_MAX - told + tnow;
      }

      told = tnow;

      /*时间超过/等于要延迟的时间,则退出 */
      if(tcnt >= ticks)break;
    }
  }
}

/*********************************************END OF FILE**********************/
```

```
#ifndef __CORE_DELAY_H
#define __CORE_DELAY_H

#include "stm32h7xx.h"

/* 获取内核时钟频率 */
#define GET_CPU_ClkFreq()       HAL_RCC_GetSysClockFreq()
#define SysClockFreq            (218000000)
/* 为方便使用，在延时函数内部调用CPU_TS_TmrInit函数初始化时间戳寄存器，
   这样每次调用函数都会初始化一遍。
```

```
    把本宏值设置为0，然后在main函数刚运行时调用CPU_TS_TmrInit可避免每次都初始化 */

#define CPU_TS_INIT_IN_DELAY_FUNCTION   0


/*******************************************************************************
 * 函数声明
 *******************************************************************************/
uint32_t CPU_TS_TmrRd(void);
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority);

//使用以下函数前必须先调用CPU_TS_TmrInit函数使能计数器，或使能宏
CPU_TS_INIT_IN_DELAY_FUNCTION
//最大延时值为8秒
void CPU_TS_Tmr_Delay_US(uint32_t us);
#define HAL_Delay(ms)     CPU_TS_Tmr_Delay_US(ms*1000)
#define CPU_TS_Tmr_Delay_S(s)       CPU_TS_Tmr_Delay_MS(s*1000)


#endif /* __CORE_DELAY_H */
```

注意事项：

使用者如果不是在HAL库中使用，注释掉：

```
uint32_t HAL_GetTick(void)
{
  return ((uint32_t)DWT_CYCCNT/SysClockFreq*1000);
}
```

同时建议重新命名`HAL_InitTick()`函数。

按照自己的平台重写以下宏定义：

```
/* 获取内核时钟频率 */
#define GET_CPU_ClkFreq()       HAL_RCC_GetSysClockFreq()
#define SysClockFreq            (218000000)
```

后记

其实在ucos-iii 源码中，有一个功能是测量关中断时间的功能，就是使用STM32的时间戳，即记录程序运行的某个时刻，如果记录下程序前后的两个时刻点，即可以算出这段程序的运行时间。 但是有关内核寄存器的描述的资料非常少，还好找到一个（arm手册），里面有这些内核寄存器的详细描述，其中时间戳相关的寄存器在第10章和11章有详细的描述。关于资料想看的可以后台找我拿。

喜欢就关注我吧！

相关代码可以在公众号后台回复“ **DWT** ”获取。