stm32移植threadx rtos

原创 mcdx 2020-07-23 21:12:25 509 收藏 2 版权

分类专栏： threadx

文章目录

写在前面
准备移植
开始移植
验证
最后一点

点击下方阅读原文可访问文中超链接

写在前面

ThreadX 是由 Express Logic 公司开发的实时操作系统。目前已被微软收购，并且前不久开源了，当开源的时候很多论坛都第一时间发布了相关文章，可见其影响力还是很不错的，刚好最近有一个新项目，需要用到网络协议栈，而threadx有自己的网络协议栈组件，之前打算用freeRTOS加LWIP的方式，现在直接用threadx加netx的方式。
首先到github上下载threadx的源码，建议使用git，下载zip的话非常慢，而且容易失败，点击链接threadx下载。

准备移植

下载好源码后就准备开始移植了，我以stm32f407为例，keil版本5.31，threax版本6.0.1，其它系列都差不多，最近的一次更新增加了非常多的器件支持。看下源码目录：

.github	git相关的不管
cmake	用来管理编译的，使用IDE的话可以忽略
common	threadx的源代码
docs	存放文档的
ports	移植相关的
samples	使用例程

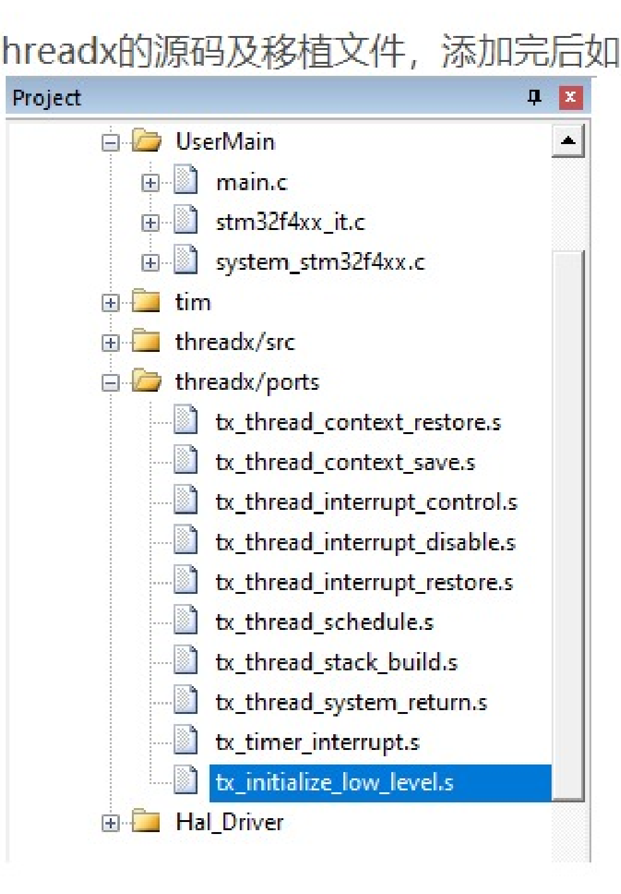
这次是移植，主要关心ports文件夹就行了，里面是针对各种内核写的移植文件，我这里对应的是cortex_m4这个文件夹，打开后里面有四个目录：

ac5	ac5编译器使用这个
gnu	gnu编译器使用这个
iar	用iar开发使用这个
keil	用keil开发使用这个

我用的是keil，但是我选择的是ac5这个文件夹，因为我用的也是ac5编译器，我对比了一下keil文件夹和ac5文件夹里面的内容不一样的地方只有一个，这个后面再说。

开始移植

首先搭建一个简单的裸机工程，然后添加threadx的源码及移植文件，添加完后如下：



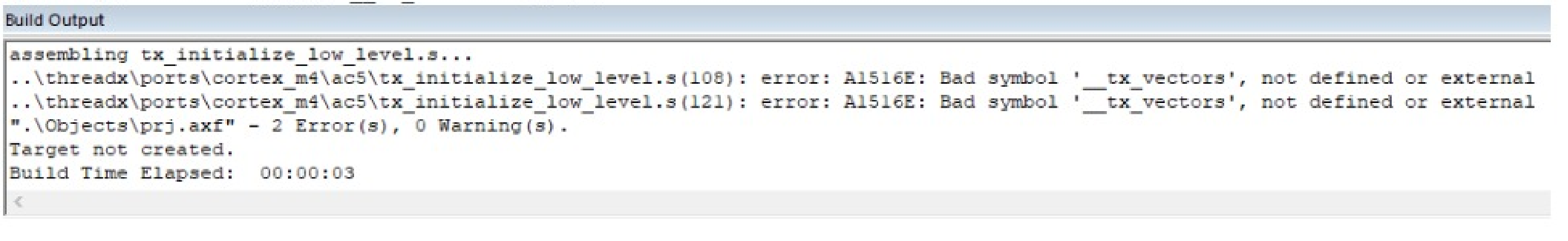
我们主要修改tx_initialize_low_level.s文件，其它几个文件都是可以不用动的，这个文件和st提供的启动文件冲突了，但是这个文件内容又不全。采用的方法类似freeRTOS的移植方法，不动st的启动文件。

- 首先更改时钟频率及给滴答定时器设置的重装载值

```
SYSTEM_CLOCK        EQU     168000000
SYSTICK_CYCLES      EQU     ((SYSTEM_CLOCK / 1000) -1)
```

- 删掉堆栈大小定义代码段（冲突）
- 删掉向量表的定义（冲突）
- 删掉Reset_Handler代码段（冲突）
- 删掉__user_initial_stackheap代码段（冲突）
- 删掉__tx_BadHandler代码段（没用）
- 删掉__tx_SVCallHandler代码段（没用）
- 删掉__tx_IntHandler代码段（没用）
- 删掉__tx_NMIHandler代码段（没用）
- 删掉__tx_DBGHandler代码段（没用）

编译一下，报了两个错说是__tx_vectors未定义

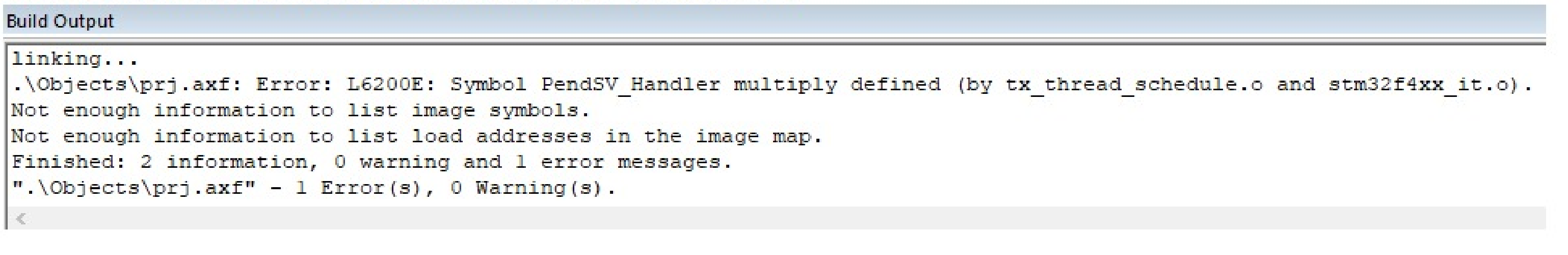


```
assembling tx_initialize_low_level.s...
..\threadx\ports\cortex_m4\ac5\tx_initialize_low_level.s(108): error: A1516E: Bad symbol '__tx_vectors', not defined or external
..\threadx\ports\cortex_m4\ac5\tx_initialize_low_level.s(121): error: A1516E: Bad symbol '__tx_vectors', not defined or external
".\Objects\prj.axf" - 2 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:03
```

这个是向量表的基地址，在st提供的启动文件中已经定义了，改成启动文件中的向量表__Vectors，注意要先在前面导入这个符号。

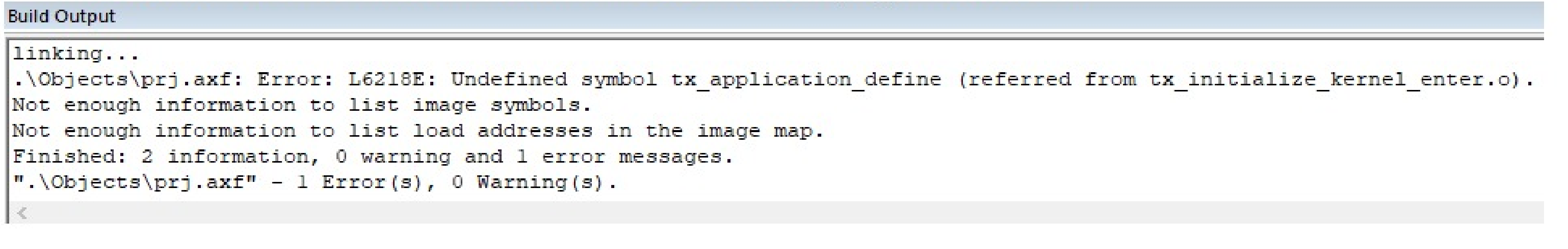
```
IMPORT __Vectors
```

再次编译提示PendSV_Handler重复定义，**但是却未提示SysTick_Handler重复定义，从以往移植其它rtos的经验来看，基本上都是使用滴答定时器作为系统心跳**，我们暂且继续往下看



```
linking...
.\Objects\prj.axf: Error: L6200E: Symbol PendSV_Handler multiply defined (by tx_thread_schedule.o and stm32f4xx_it.o).
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
".\Objects\prj.axf" - 1 Error(s), 0 Warning(s).
```

屏蔽掉stm32f4xx_it.c文件中的PendSV_Handler函数，threadx需要用来进行上下文切换。
再次编译提示tx_application_define未定义，这个是在threadx系统启动的时候会自动调用，我们如果需要创建任务或者信号量等资源的话都是在这个函数里面完成，参考实现可仿照sample_threadx.c文件。



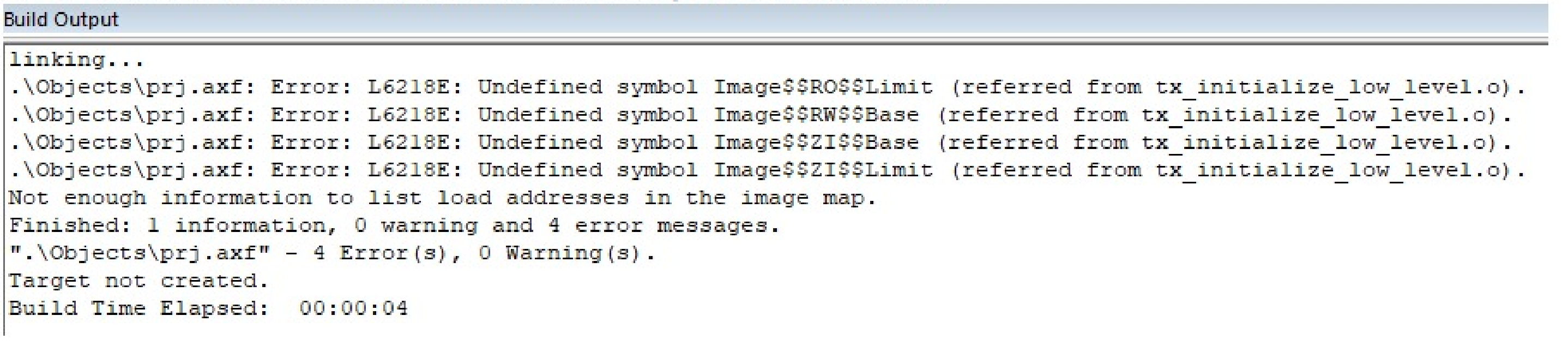
```
linking...
.\Objects\prj.axf: Error: L6218E: Undefined symbol tx_application_define (referred from tx_initialize_kernel_enter.o).
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
".\Objects\prj.axf" - 1 Error(s), 0 Warning(s).
```

在main函数中添加头文件并且添加此函数的实体

```
#include "tx_api.h"

VOID tx_application_define(VOID *first_unused_memory)
{
    /*暂时留空*/
}
```

这个问题解决后再次编译又报了几个错，也是未定义，猜测这几个符号是编译完代码后在某个中间文件生成的，在链接的时候这里再获取其值，**因为keil编译完程序生成的map文件有类似的符号**。



```
linking...
.\Objects\prj.axf: Error: L6218E: Undefined symbol Image$$RO$$Limit (referred from tx_initialize_low_level.o).
.\Objects\prj.axf: Error: L6218E: Undefined symbol Image$$RW$$Base (referred from tx_initialize_low_level.o).
.\Objects\prj.axf: Error: L6218E: Undefined symbol Image$$ZI$$Base (referred from tx_initialize_low_level.o).
.\Objects\prj.axf: Error: L6218E: Undefined symbol Image$$ZI$$Limit (referred from tx_initialize_low_level.o).
Not enough information to list load addresses in the image map.
Finished: 1 information, 0 warning and 4 error messages.
".\Objects\prj.axf" - 4 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:04
```

经过搜索，发现只有ImageZILimit有用（这里$显示不出来），其它都没有使用，从官方提供的其它移植的例程得知这里是用户可用内存（运行环境需要的除外，比如C运行环境的堆栈）的起始地址，设置为__initial_sp，同样的需要在前面IMPORT这个符号，__initial_sp有点特殊，需要勾选keil选项中的Use MicroLIB选项，不然的话提示找不到这个符号的定义，这是从启动文件得知的。

```
;*******************************************************************************
; User Stack and Heap initialization
;*******************************************************************************
                 IF      :DEF:__MICROLIB

                 EXPORT  __initial_sp
                 EXPORT  __heap_base
                 EXPORT  __heap_limit

                 ELSE

                 IMPORT  __use_two_region_memory
                 EXPORT  __user_initial_stackheap
                 ...
```

现在已经编译成功了，但是你会发现系统跑不起来，**这就是前面为什么说SysTick_Handler这个函数没有报重复定义的错**，因为系统的心跳还没有提供，看下PendSV_Handler是怎么实现的。

```
;    /* Generic context switching PendSV handler.  */
;
        EXPORT  __tx_PendSVHandler
        EXPORT  PendSV_Handler
__tx_PendSVHandler
PendSV_Handler
```

这里将两个标号写在一起，也就是相当于一个函数的两个名字，实际上是同一个东西，模仿一下，添加SysTick_Handler，然后屏蔽掉stm32f4xx_it.c文件中的SysTick_Handler函数。

```
        EXPORT  __tx_SysTickHandler
        EXPORT  SysTick_Handler
__tx_SysTickHandler
SysTick_Handler
```

编译成功，现在移植已经就大功告成了，添加两个线程测试下（参考sample_threadx.c文件）

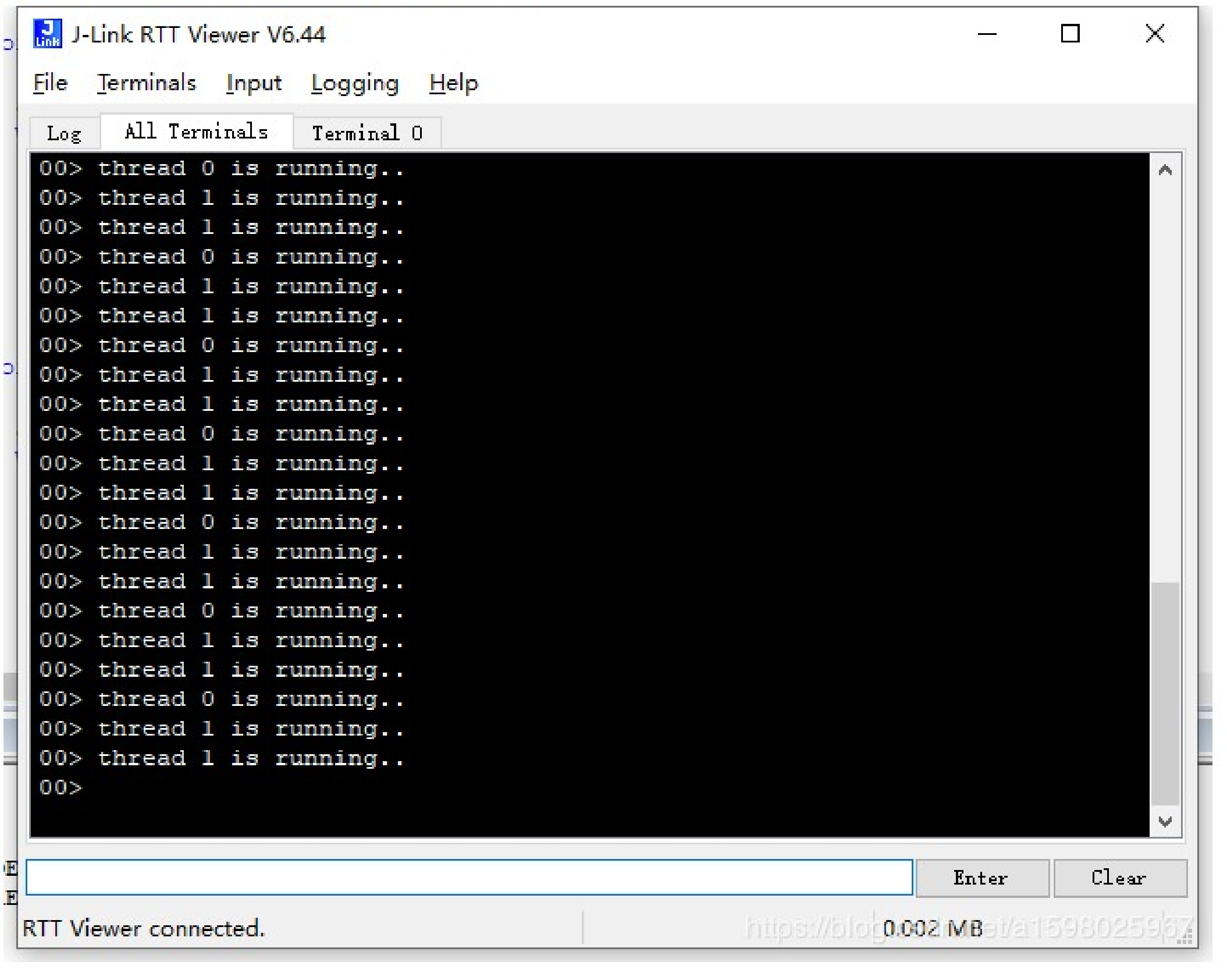
```
void    thread_0_entry(ULONG thread_input)
{
    /* This thread simply sits in while-forever-sleep loop.  */
    while(1)
    {
        PRINTF("thread 0 is running..\r\n");

        tx_thread_sleep(1000);
    }
}

void    thread_1_entry(ULONG thread_input)
{
    /* This thread simply sends messages to a queue shared by thread 2.  */
    while(1)
    {
        PRINTF("thread 1 is running..\r\n");

        tx_thread_sleep(500);
    }
}
```

测试结果，目前运行良好，暂时还不知道有没有其它问题



```
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00> thread 0 is running..
00> thread 1 is running..
00> thread 1 is running..
00>
```

验证

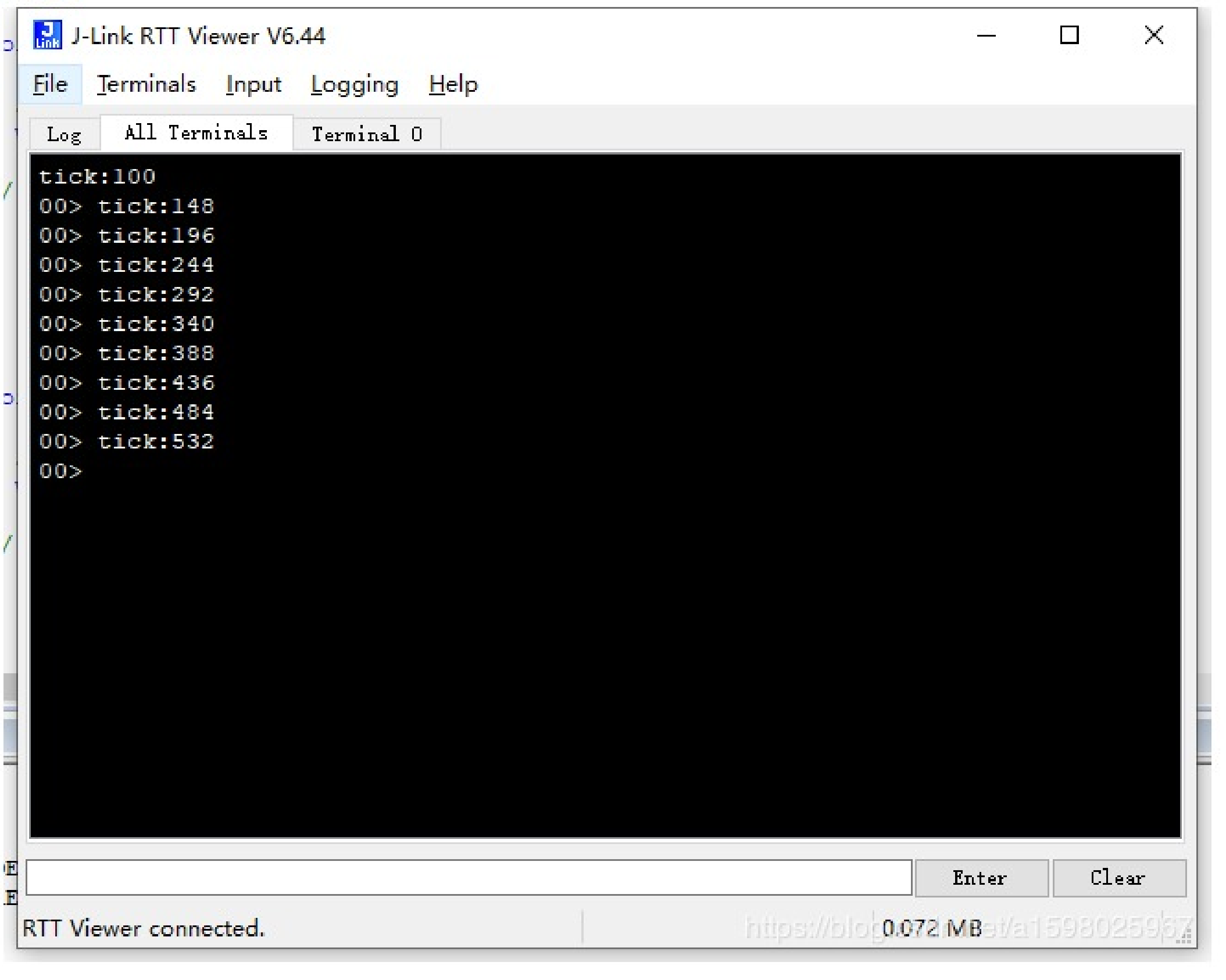
虽然系统移植好了，但是怎么才能验证呢，比如我这里延时的时间是准确的吗？我们可以将HAL_GetTick这个函数移植进来，然后测试下时间是否准确，当然也有其它方式。
将HAL库里面stm32f4xx_hal_timebase_tim_template.c文件拷贝出来，重命名为stm32f4xx_hal_timebase_tim.c，然后添加到工程中，本来裸机的时候使用滴答定时器来实现HAL_GetTick函数的功能，但是现在滴答定时器被用了，st提供了这个文件使用timer6来替代。注意HAL库F4的库版本1.25.0有个BUG在这里，我们还需要修改下stm32f4xx_hal_timebase_tim.c文件HAL_InitTick函数，否则会出现断言失败（如果开启了断言检测），因为这个函数在HAL_Init被调用了一次，传入的参数是TICK_INT_PRIORITY（0x0F），然后在SystemClock_Config函数又调用了一下，这次调用的时候入参是一个全局变量，而在老版本的库中，这里入参还是TICK_INT_PRIORITY，就是因为这里入参现在传了个全局变量才会导致断言失败，并且优先级设置的是个错误的值，修改如下：

```
/*略*/
    HAL_NVIC_SetPriority(TIM6_DAC_IRQn, TickPriority ,0U);

    /*新版BUG，这里参考stm32f4xx_hal.c文件的HAL_InitTick函数*/
    uwTickPrio = TickPriority;

    /* Enable the TIM6 global Interrupt */
    HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
/*略*/
```

但是测试结果却出乎意料



```
tick:100
00> tick:148
00> tick:196
00> tick:244
00> tick:292
00> tick:340
00> tick:388
00> tick:436
00> tick:484
00> tick:532
00>
```

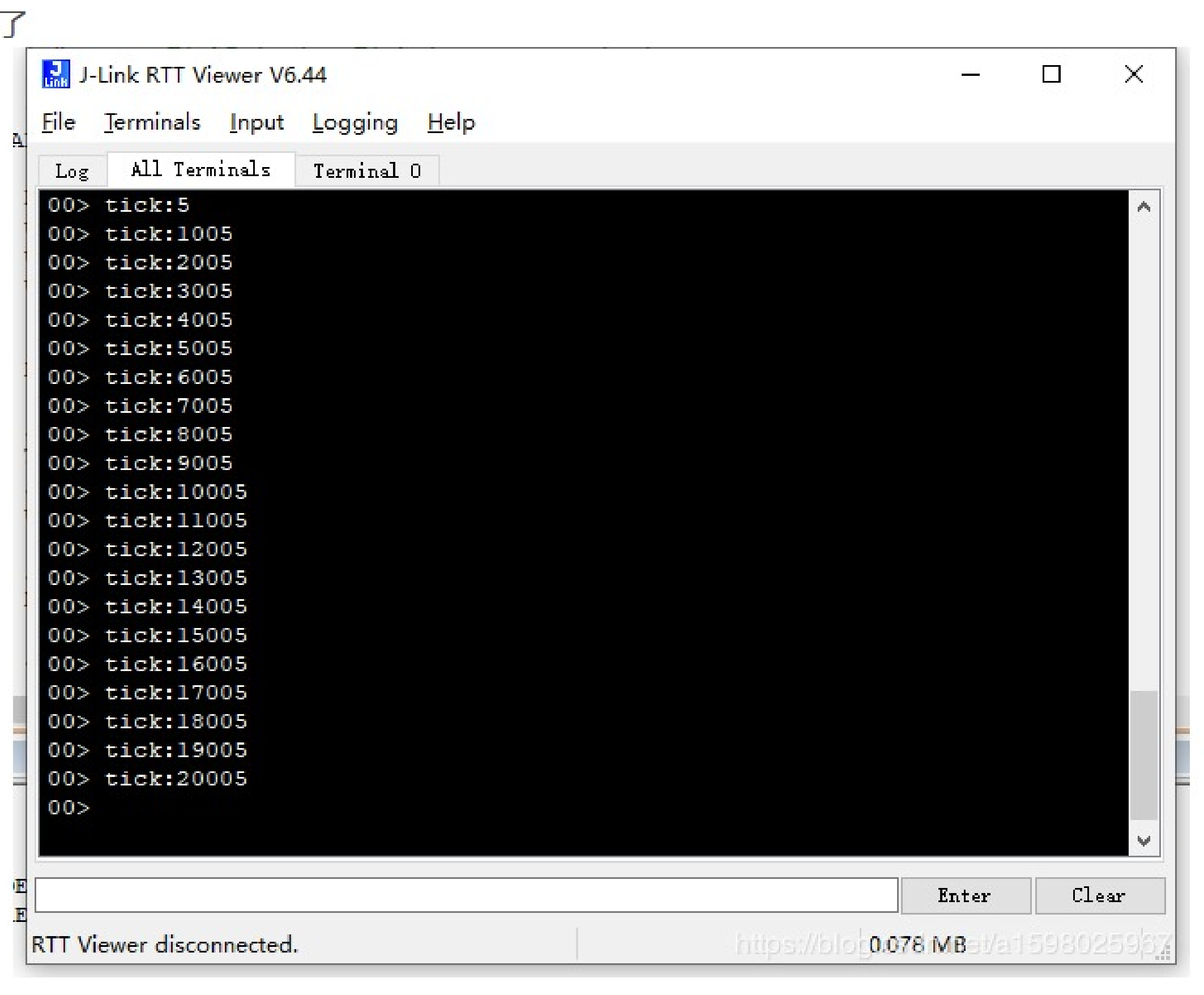
我明明延时的1秒，但是出来的两个值间隔时间根本就不是1秒，最开始怀疑系统移植哪里有错误，无奈只好看ports里面那几个汇编文件，看来看去也没有发现问题，各种测试都是一样的结果，最后尝试更改stm32f4xx_hal_timebase_tim.c文件的函数HAL_InitTick传入的优先级参数，试了一下，时间准确了，多次测试发现，只要优先级设置的不是0x0F就正确，又在tx_initialize_low_level.s文件的_tx_initialize_low_level代码段看到里面将PnSV和SVC设置的0x0F优先级，所以猜测这个中断优先级被threadx用了后，其它地方就不能用了，这个和以前用的uCOS和freeRTOS还有所不同。再次修改代码。

```
/*略*/
    HAL_NVIC_SetPriority(TIM6_DAC_IRQn, TickPriority ,0U);

    /*0x0f优先级已被threadx系统使用*/
    TickPriority -= 1;
    /*新版BUG，这里参考stm32f4xx_hal.c文件的HAL_InitTick函数*/
    uwTickPrio = TickPriority;

    /* Enable the TIM6 global Interrupt */
    HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
/*略*/
```

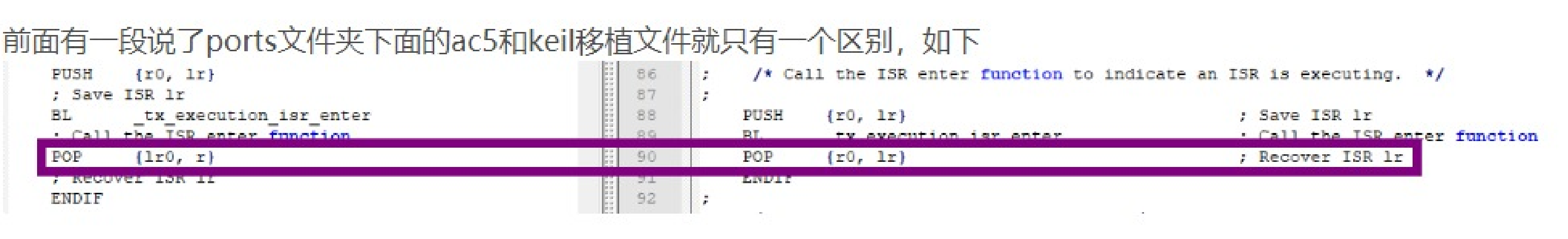
这次修改后，正确了



```
00> tick:5
00> tick:1005
00> tick:2005
00> tick:3005
00> tick:4005
00> tick:5005
00> tick:6005
00> tick:7005
00> tick:8005
00> tick:9005
00> tick:10005
00> tick:11005
00> tick:12005
00> tick:13005
00> tick:14005
00> tick:15005
00> tick:16005
00> tick:17005
00> tick:18005
00> tick:19005
00> tick:20005
00>
```

最后一点

前面有一段说了ports文件夹下面的ac5和keil移植文件就只有一个区别，如下



```
PUSH    {r0, lr}
; Save ISR lr
BL      _tx_execution_isr_enter
; Call the ISR enter function
POP     {lr0, r}
; Recover ISR lr
ENDIF
```

```
;    /* Call the ISR enter function to indicate an ISR is executing.  */
;
    PUSH    {r0, lr}                                    ; Save ISR lr
    BL      _tx_execution_isr_enter                     ; Call the ISR enter function
    POP     {r0, lr}                                    ; Recover ISR lr
    ENDIF
;
```

这里是出栈操作，按理说正确的汇编代码应该是右边的写法，但是左边的写法编译时居然没报错，而且运行也是正常的，两种写法我都测试了，都可以正常稳定运行，暂时不知道具体原因，只有看官方之后的更新到底是怎样的，可能这里本身就是个BUG，也可能左边写法有其它含义。
文章的整个测试源码可以在我的码云仓库获取。源码点这儿