ThreadX系列 | 最新v6.1.6版本在MDK中的移植方法

zhuanlan.zhihu.com/p/378901531

去年在threadx刚开源的时候移植体验了一波，并分享了移植文章，最近发现这一年threadx在不断的更新，目前更新至v6.1.6版本，所以更新最新版本的移植方法，顺便吐槽一下！

1. 前言

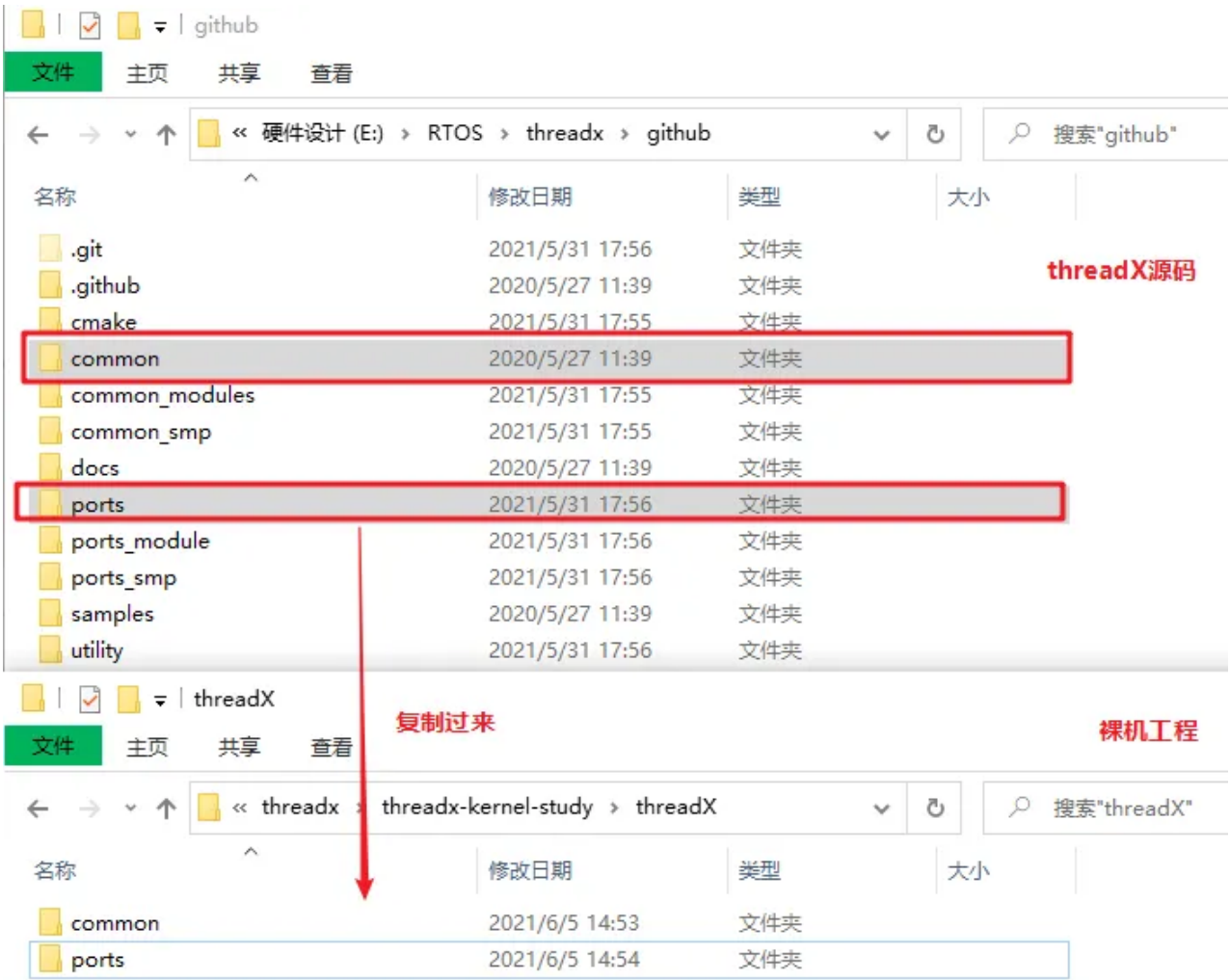
本文中使用的开发板为小熊派IoT开发板，主控为STM32L431RCT6：

请准备一份可以「正常使用**printf**串口输出的裸机工程」，本文中我使用cubemx生成。

2. 复制ThreadX源码

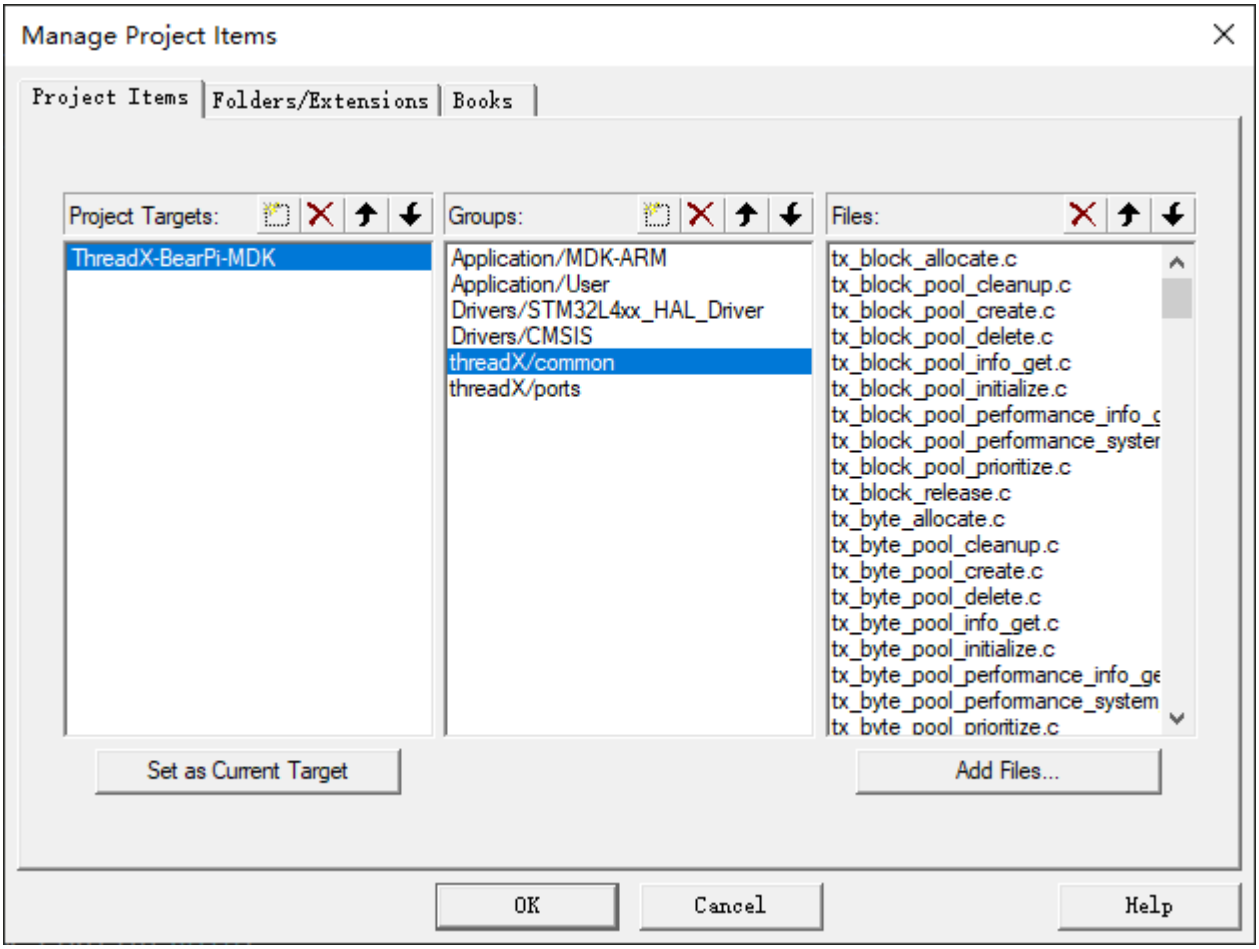
ThreadX源码请访问开源仓库获取：

github.com/azure-rtos/t



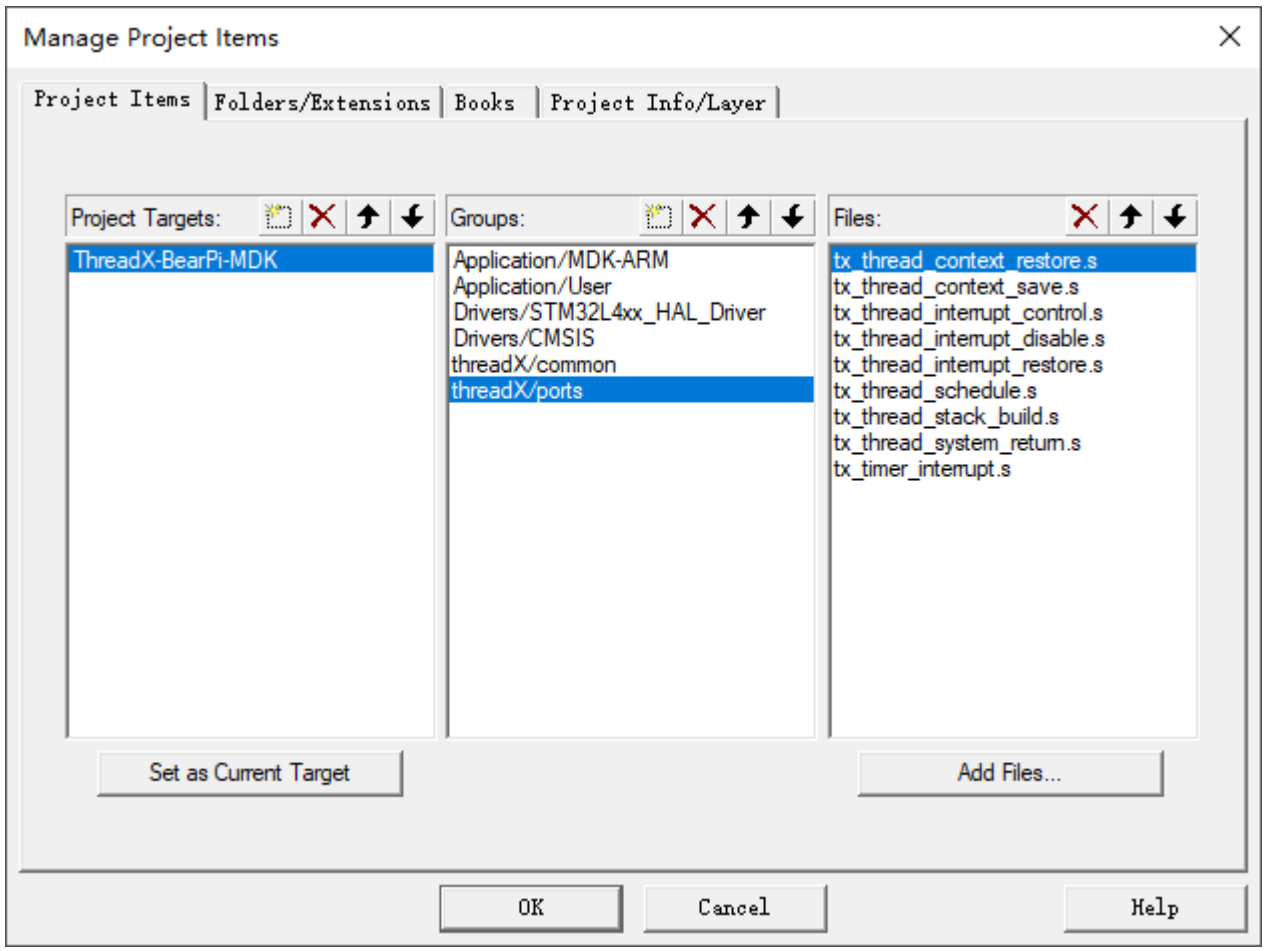
3. 添加源码到MDK工程

新建 `threadX/common` 分组，添加threadX/common/src下的所有c文件：

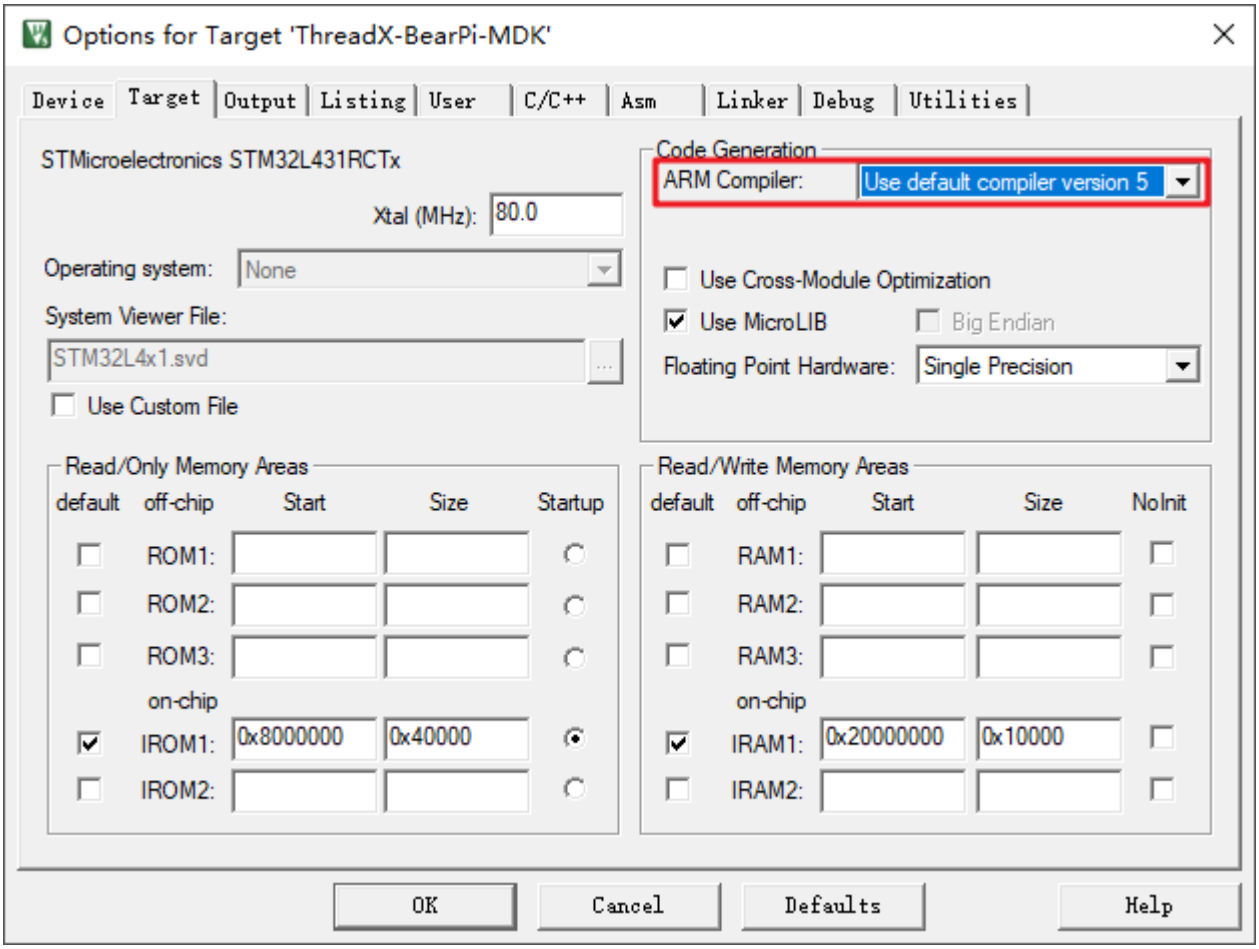


「新建 `threadX/ports` 分组，此时需要根据编译环境来选择」。

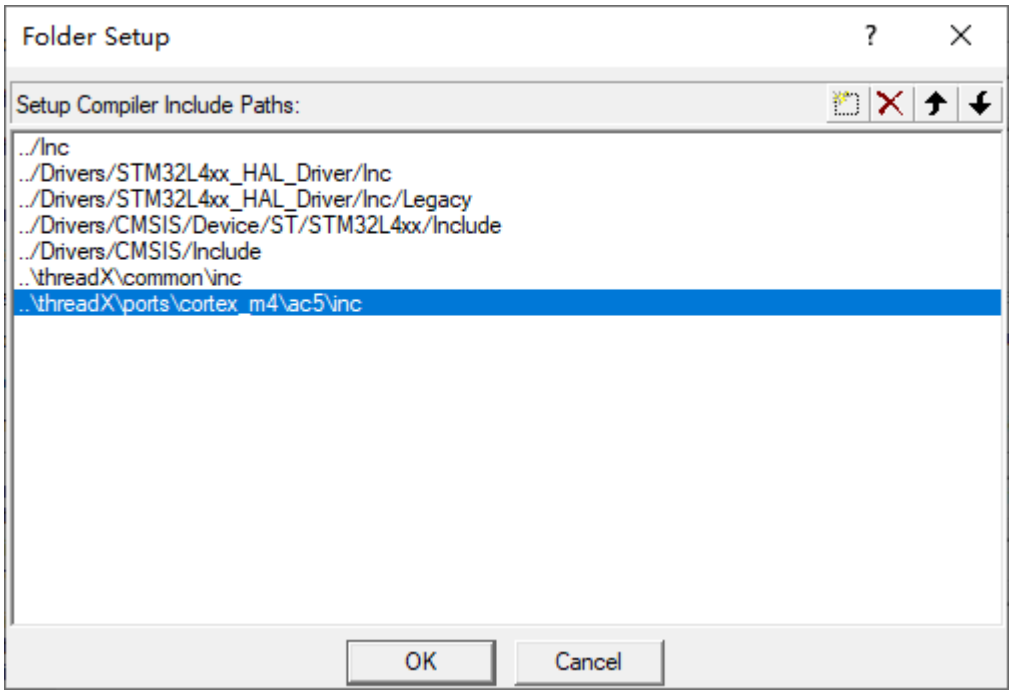
此处我们使用的是AC5编译器，则添加 threadX\ports\cortex_m4\ac5\src 下的所有 .s 文件：



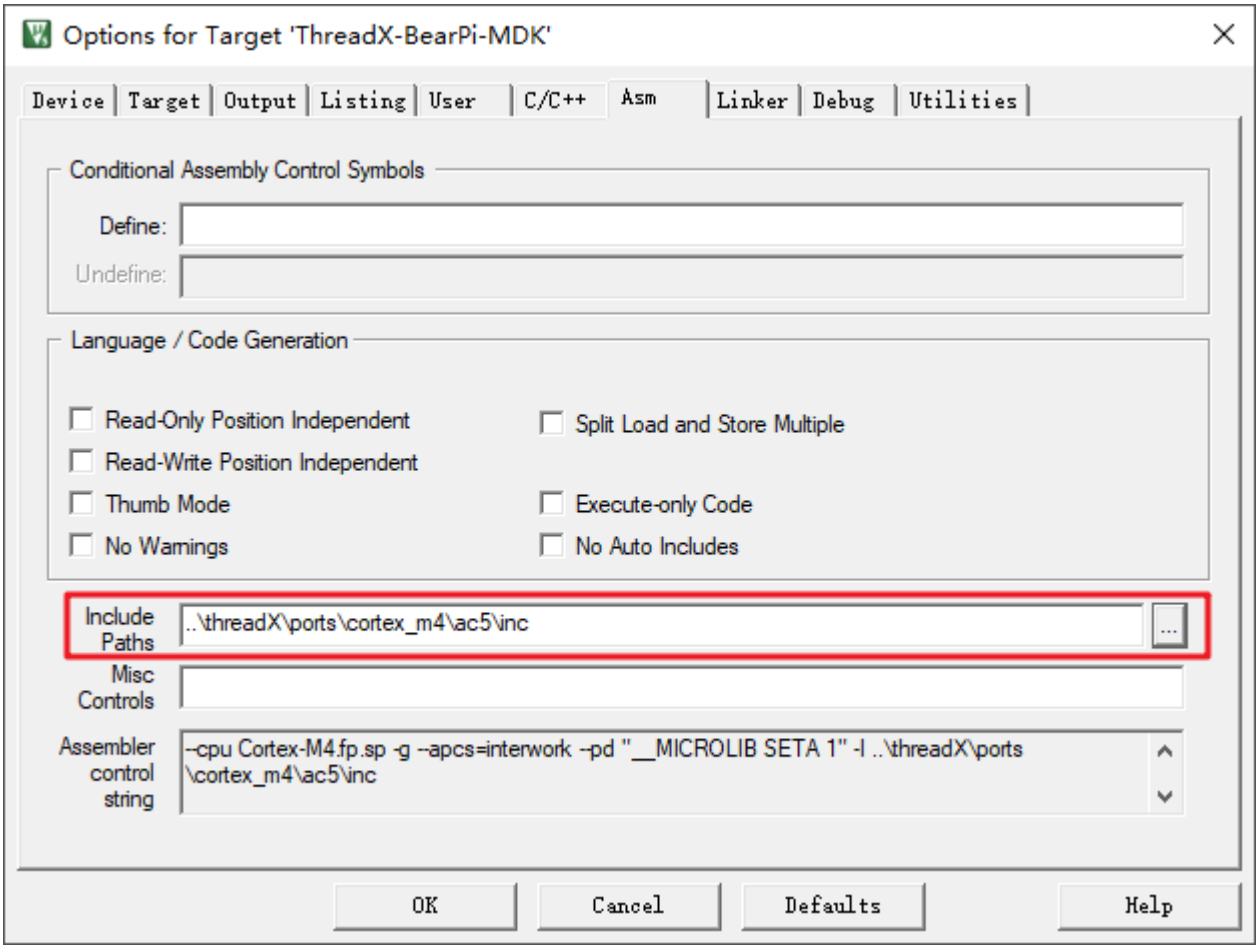
设置使用AC5编译器：



添加头文件路径：



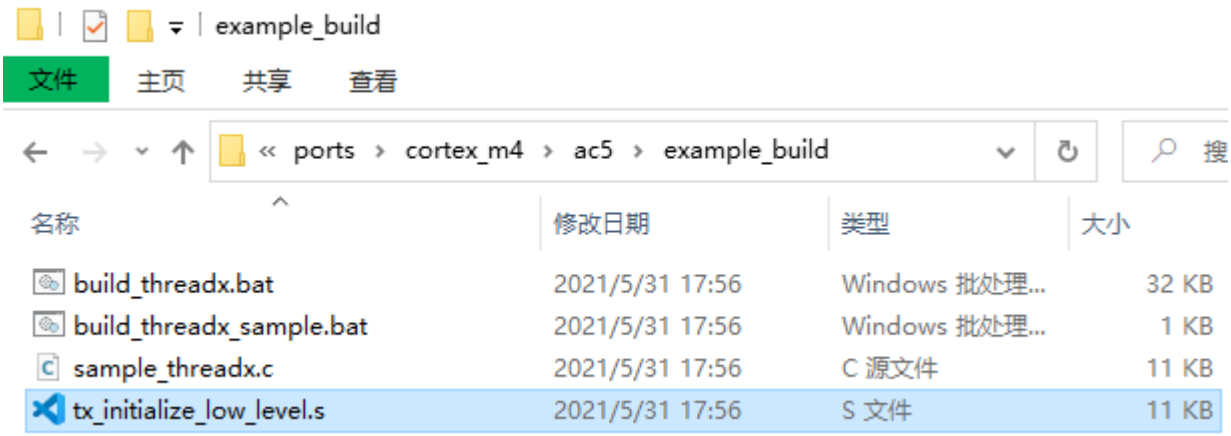
设置ASM汇编头文件路径：



4. 添加并修改适配底层文件

4.1. tx_initialize_low_level.s

threadX官方提供了一个底层适配文件`tx_initialize_low_level.s`，所在位置如图：



「这里我就不得不吐槽一下了！」

本来这个文件中实现了`_tx_initialize_low_level()`函数，该函数用于完成处理器的底层初始化，包括：

> 寻找RAM中首块可用地址传入tx_application_define函数供使用，也就是first_unused_memory指针的值

「但是**threadx**在**v6**版本及以后，竟然想在这个文件中接管原有的处理器启动文件」，证据如下。

设置堆栈环境的证据：

tx_initialize_low_level_bearpi.S

```
37  ;
38  ;/* Setup the stack and heap areas.  */
39  ;
40  ;STACK_SIZE          EQU     0x00000400
41  ;HEAP_SIZE           EQU     0x00000000
42
43  ;    AREA    STACK, NOINIT, READWRITE, ALIGN=3
44  ;StackMem
45  ;    SPACE   STACK_SIZE
46  ;__initial_sp
47
48
49  ;    AREA    HEAP, NOINIT, READWRITE, ALIGN=3
50  ;__heap_base
51  ;HeapMem
52  ;    SPACE   HEAP_SIZE
53  ;__heap_limit
```

重新定义向量表的证据：

tx_initialize_low_level_bearpi.S

```
55
56  ;    AREA    RESET, CODE, READONLY
57  ;
58  ;    EXPORT  __tx_vectors
59  ;__tx_vectors
60  ;    DCD     __initial_sp                    ; Reset and system stack ptr
61  ;    DCD     Reset_Handler                   ; Reset goes to startup function
62  ;    DCD     __tx_NMIHandler                 ; NMI
63  ;    DCD     __tx_BadHandler                 ; HardFault
64  ;    DCD     0                               ; MemManage
65  ;    DCD     0                               ; BusFault
66  ;    DCD     0                               ; UsageFault
67  ;    DCD     0                               ; 7
68  ;    DCD     0                               ; 8
69  ;    DCD     0                               ; 9
70  ;    DCD     0                               ; 10
71  ;    DCD     __tx_SVCallHandler              ; SVCall
72  ;    DCD     __tx_DBGHandler                 ; Monitor
73  ;    DCD     0                               ; 13
74  ;    DCD     __tx_PendSVHandler              ; PendSV
75  ;    DCD     __tx_SysTickHandler             ; SysTick
76  ;    DCD     __tx_IntHandler                 ; Int 0
77  ;    DCD     __tx_IntHandler                 ; Int 1
78  ;    DCD     __tx_IntHandler                 ; Int 2
79  ;    DCD     __tx_IntHandler                 ; Int 3
```

接管复位程序的证据：

tx_initialize_low_level_bearpi.S

```
82      AREA ||.text||, CODE, READONLY
83  ;    EXPORT  Reset_Handler
84  ;Reset_Handler
85  ;    CPSID   i
86  ;    IF  {TARGET_FPU_VFP} = {TRUE}
87  ;    LDR     r0, =0xE000ED88                 ; Pickup address of CPACR
88  ;    LDR     r1, [r0]                        ; Pickup CPACR
89  ;    MOV32   r2, 0x00F00000                  ; Build enable value
90  ;    ORR     r1, r1, r2                      ; Or in enable value
91  ;    STR     r1, [r0]                        ; Setup CPACR
92  ;    ENDIF
93  ;    LDR     r0, =__main
94  ;    BX      r0
```

作为一个用来提供调度能力的RTOS，仅仅接管pendSV中断和Systick中断就够了，甚至Systick中断还需要给HAL库用，不能直接接管走，竟然想把系统所有中断都接管了.....

是该说野心勃勃呢？还是该说画蛇添足呢？

退一步海阔天空，把系统所有中断直接都接管了总得干点正事吧~

tx_initialize_low_level_bearpi.S

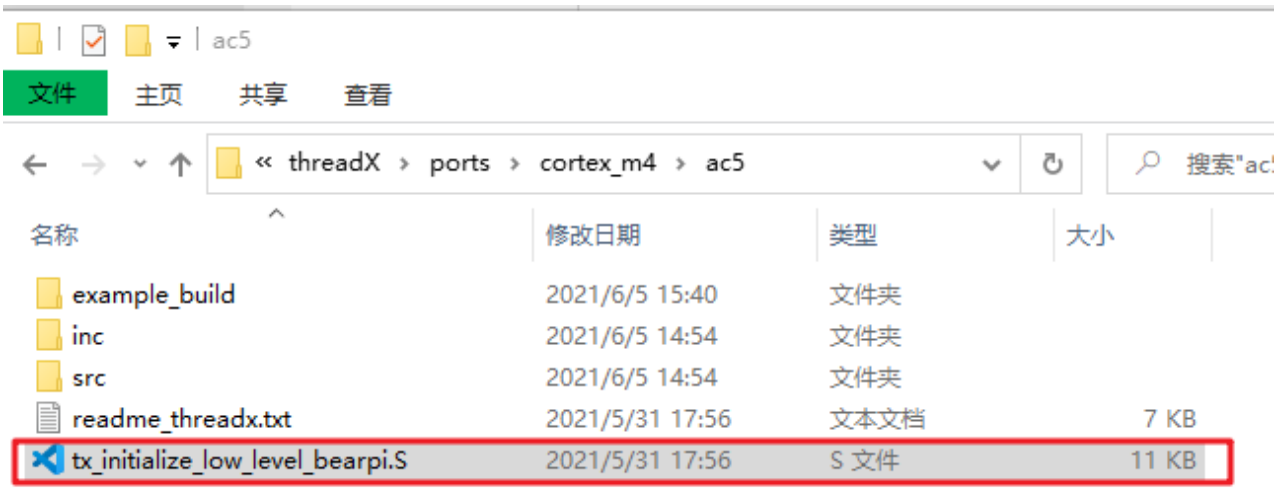
```
224  __tx_IntHandler
225  ; VOID InterruptHandler (VOID)
226  ; {
227      PUSH    {r0, lr}
228
229  ;    /* Do interrupt handler work here */
230  ;    /* .... */
231
232      POP     {r0, lr}
233      BX      LR
234  ; }
```

接管中断了就写个这？？？

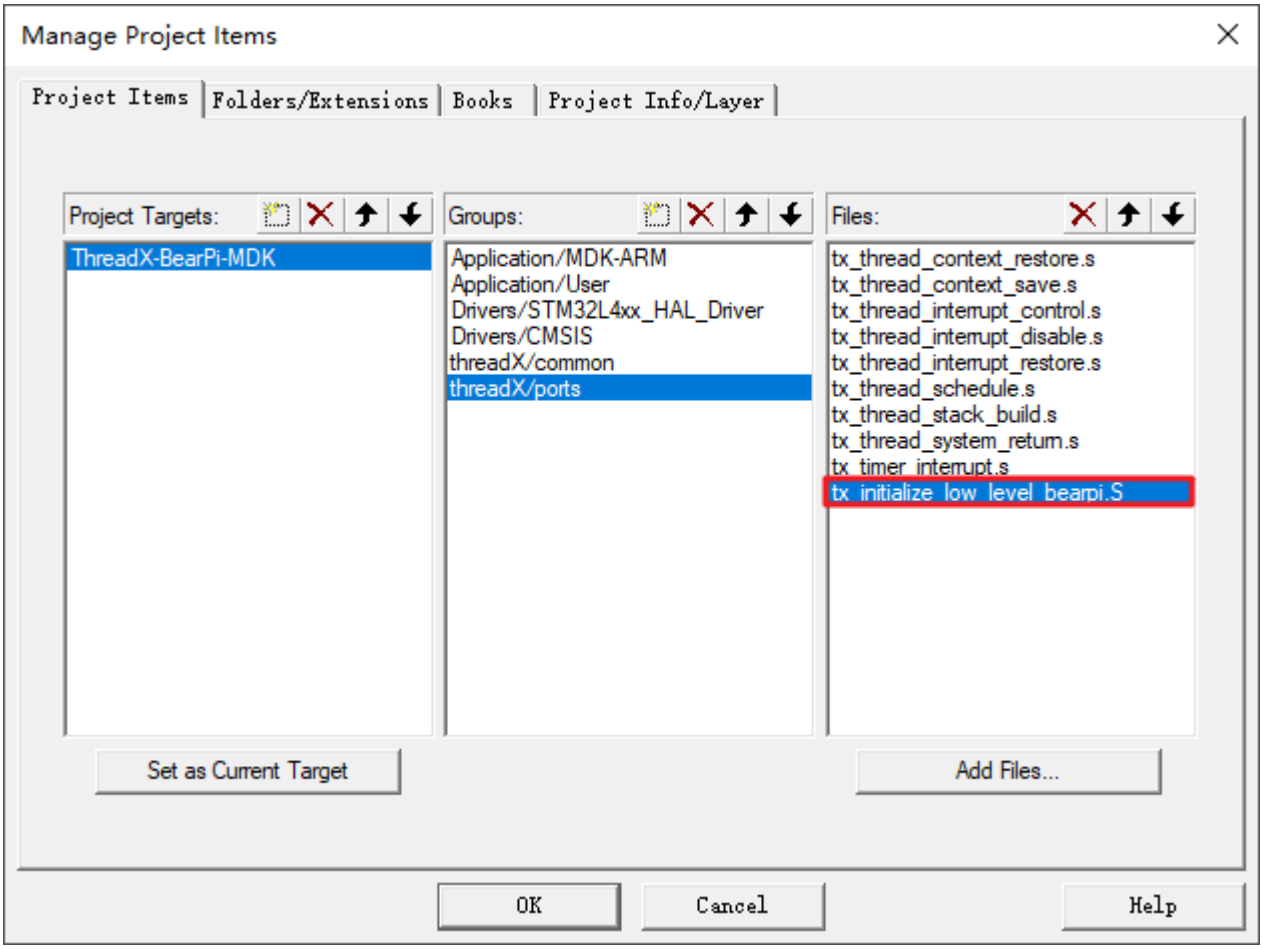
吐槽归吐槽，接着干活！移植threadx之后玩起来还是很舒服的！

4.2. 添加适配文件

将 tx_initialize_low_level_sample.S 文件复制出来一份，改名为 `tx_initialize_low_level_bearpi.S`，作为本项目的适配文件：



将该文件添加到工程中：



4.3. 修改适配文件

① 将没有用到的标号注释，手动添加 `_Vectors` 和 `__initial_sp` 标号，分别是STM32启动文件中导出的中断向量表和栈顶指针初始值：

tx_initialize_low_level_bearpi.S

```
18  ;
19      IMPORT  _tx_thread_system_stack_ptr
20      IMPORT  _tx_initialize_unused_memory
21  ;    IMPORT  _tx_thread_context_save
22  ;    IMPORT  _tx_thread_context_restore
23      IMPORT  _tx_timer_interrupt
24  ;    IMPORT  __main
25  ;    IMPORT  |Image$$RO$$Limit|
26  ;    IMPORT  |Image$$RW$$Base|
27  ;    IMPORT  |Image$$ZI$$Base|
28  ;    IMPORT  |Image$$ZI$$Limit|
29  ;    IMPORT  __tx_PendSVHandler
30      IMPORT __Vectors
31      IMPORT __initial_sp
32  ;
```

② 设置时钟频率（80Mhz）和时钟节拍（1ms），该值用来初始化Systick定时器：

tx_initialize_low_level_bearpi.S

```
33  ;
34  SYSTEM_CLOCK        EQU     80000000
35  SYSTICK_CYCLES      EQU     ((SYSTEM_CLOCK / 1000) -1)
36  ;
```

③ 将设置堆栈的代码全部注释（堆栈环境已经在STM32启动文件中设置了）

tx_initialize_low_level_bearpi.S

```
36  ;
37  ;
38  ;/* Setup the stack and heap areas.  */
39  ;
40  ;STACK_SIZE           EQU     0x00000400
41  ;HEAP_SIZE            EQU     0x00000000
42
43  ;    AREA    STACK, NOINIT, READWRITE, ALIGN=3
44  ;StackMem
45  ;    SPACE   STACK_SIZE
46  ;__initial_sp
47
48
49  ;    AREA    HEAP, NOINIT, READWRITE, ALIGN=3
50  ;__heap_base
51  ;HeapMem
52  ;    SPACE   HEAP_SIZE
53  ;__heap_limit
```

④ 将 threadx 定义的中断向量表全部注释（使用STM32启动文件中定义的向量表）：

tx_initialize_low_level_bearpi.S

```
54
55
56  ;    AREA    RESET, CODE, READONLY
57  ;
58  ;    EXPORT  __tx_vectors
59  ;__tx_vectors
60  ;    DCD     __initial_sp                    ; Reset and system stack ptr
61  ;    DCD     Reset_Handler                   ; Reset goes to startup function
62  ;    DCD     __tx_NMIHandler                 ; NMI
63  ;    DCD     __tx_BadHandler                 ; HardFault
64  ;    DCD     0                               ; MemManage
65  ;    DCD     0                               ; BusFault
66  ;    DCD     0                               ; UsageFault
67  ;    DCD     0                               ; 7
68  ;    DCD     0                               ; 8
69  ;    DCD     0                               ; 9
70  ;    DCD     0                               ; 10
71  ;    DCD     __tx_SVCallHandler              ; SVCall
72  ;    DCD     __tx_DBGHandler                 ; Monitor
73  ;    DCD     0                               ; 13
74  ;    DCD     __tx_PendSVHandler              ; PendSV
75  ;    DCD     __tx_SysTickHandler             ; SysTick
76  ;    DCD     __tx_IntHandler                 ; Int 0
77  ;    DCD     __tx_IntHandler                 ; Int 1
78  ;    DCD     __tx_IntHandler                 ; Int 2
79  ;    DCD     __tx_IntHandler                 ; Int 3
80  ;
```

⑤ 注释threadx定义的复位处理程序（使用STM32启动文件中的复位程序）：

tx_initialize_low_level_bearpi.S

```
81  ;
82      AREA ||.text||, CODE, READONLY          后面还有其它子程序，该行保留
83  ;    EXPORT  Reset_Handler
84  ;Reset_Handler
85  ;    CPSID   i
86  ;    IF  {TARGET_FPU_VFP} = {TRUE}
87  ;    LDR     r0, =0xE000ED88                 ; Pickup address of CPACR
88  ;    LDR     r1, [r0]                        ; Pickup CPACR
89  ;    MOV32   r2, 0x00F00000                  ; Build enable value
90  ;    ORR     r1, r1, r2                      ; Or in enable value
91  ;    STR     r1, [r0]                        ; Setup CPACR
92  ;    ENDIF
93  ;    LDR     r0, =__main
94  ;    BX      r0
```

⑥ 修改threadx底层初始化函数：

tx_initialize_low_level_bearpi.S

```
138 ;VOID   _tx_initialize_low_level(VOID)
139 ;{
140     EXPORT  _tx_initialize_low_level
141 _tx_initialize_low_level
142 ;
143 ;    /* Disable interrupts during ThreadX initialization.  */
144 ;
145     CPSID   i
146 ;
147 ;    /* Set base of available memory to end of non-initialised RAM area.  */
148 ;
149     LDR     r0, = tx_initialize_unused_memory       ; Build address of unused memory pointer
150     LDR     r1, =  initial_sp                       ; Build first free address
151     ADD     r1, r1, #4                              ;
152     STR     r1, [r0]                                ; Setup first unused memory pointer
153 ;
154 ;    /* Setup Vector Table Offset Register.  */
155 ;
156     MOV     r0, #0xE000E000                         ; Build address of NVIC registers
157     LDR     r1, =  Vectors                        ; Pickup address of vector table
158     STR     r1, [r0, #0xD08]                        ; Set vector table address
159 ;
160 ;    /* Enable the cycle count register.  */
161 ;
162 ;    LDR     r0, =0xE0001000                         ; Build address of DWT register
163 ;    LDR     r1, [r0]                                ; Pickup the current value
164 ;    ORR     r1, r1, #1                              ; Set the CYCCNTENA bit
165 ;    STR     r1, [r0]                                ; Enable the cycle count register
166 ;
167 ;    /* Set system stack pointer from vector value.  */
168 ;
169     LDR     r0, = _tx_thread_system_stack_ptr       ; Build address of system stack pointer
170     LDR     r1, =  Vectors                        ; Pickup address of vector table
171     LDR     r1, [r1]                                ; Pickup reset stack pointer
172     STR     r1, [r0]                                ; Save system stack pointer
173 ;
```

栈是向下增长的，栈顶指针开始向上是可用的RAM空间

设置中断向量表寄存器

中断向量表第一项就是栈顶指针sp，将其读取出来加载到th_thread_system_stack_ptr中

⑦ 注释用不到的函数：

tx_initialize_low_level_bearpi.S

```
198 ;}
199 ;
200 ;
201 ;/* Define initial heap/stack routine for the ARM RVCT startup code.
202 ;   This routine will set the initial stack and heap locations */
203 ;
204 ;    EXPORT  __user_initial_stackheap
205 ;__user_initial_stackheap
206 ;    LDR     r0, =Heap_Mem
207 ;    LDR     r1, =(Stack_Mem + Stack_Size)
208 ;    LDR     r2, =(Heap_Mem + Heap_Size)
209 ;    LDR     r3, =Stack_Mem
210 ;    BX      lr
211 ;
212 ;
213 ;/* Define shells for each of the unused vectors.  */
214 ;
215 ;    EXPORT  __tx_BadHandler
216 ;__tx_BadHandler
217 ;    B       __tx_BadHandler
218 ;
219 ;    EXPORT  __tx_SVCallHandler
220 ;__tx_SVCallHandler
221 ;    B       __tx_SVCallHandler
222
223 ;    EXPORT  __tx_IntHandler
224 ;__tx_IntHandler
225 ; VOID InterruptHandler (VOID)
226 ; {
227 ;    PUSH    {r0, lr}
228 ;
229 ;    /* Do interrupt handler work here */
230 ;    /* .... */
231 ;
232 ;    POP     {r0, lr}
233 ;    BX      LR
234 ; }
```

⑧ 处理Systick中断函数：

```
235 ;
236 ;         EXPORT    __tx_SysTickHandler
237           EXPORT  SysTick_Handler
238           IMPORT  HAL_IncTick
239 ;  __tx_SysTickHandler
240   SysTick_Handler
241   ; VOID TimerInterruptHandler (VOID)
242   ; {
243   ;
244       PUSH    {r0, lr}
245       BL      __tx_timer_interrupt
246       BL      HAL_IncTick
247       POP     {r0, lr}
248       BX      LR
249   ; }
250
251   ;     EXPORT   __tx_NMIHandler
252   ;__tx_NMIHandler
253   ;     B        __tx_NMIHandler
254   ;
255   ;     EXPORT   __tx_DBGHandler
256   ;__tx_DBGHandler
257   ;     B        __tx_DBGHandler
258
259       ALIGN
260       LTORG
261       END
```

该段子程序名字设置为Systick_Handler，
用来和STM32启动文件中定义的弱函数相对应

添加并调用HAL_IncTick，
使HAL库的时钟节拍正常工作

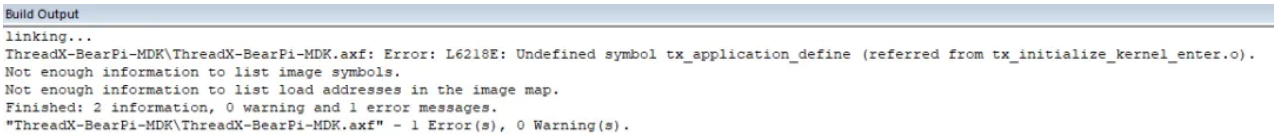
4.4. 注释HAL库提供的中断函数

去除原有stm32l4xx_it.c中的 PendSV 和 Systick 中断服务函数：

main.c | usart.c | stm32l4xx_it.c*

```
169 //void PendSV_Handler(void)
170 //{
171 //  /* USER CODE BEGIN PendSV_IRQn 0 */
172
173 //  /* USER CODE END PendSV_IRQn 0 */
174 //  /* USER CODE BEGIN PendSV_IRQn 1 */
175
176 //  /* USER CODE END PendSV_IRQn 1 */
177 //}
178
179 ///**
180 //  * @brief This function handles System tick timer.
181 //  */
182 //void SysTick_Handler(void)
183 //{
184 //  /* USER CODE BEGIN SysTick_IRQn 0 */
185
186 //  /* USER CODE END SysTick_IRQn 0 */
187 //  HAL_IncTick();
188 //  /* USER CODE BEGIN SysTick_IRQn 1 */
189
190 //  /* USER CODE END SysTick_IRQn 1 */
191 //}
```

至此，移植完成，编译会提示有一个错误：



这个函数是留给用户自己来定义应用程序入口的，接下来会创建。

5. 编写应用代码

新建一个 `application_entry.c` 文件并加入到工程中，在其中编写两个任务，然后在 `tx_application_define` 中创建这两个任务。

5.1. 编写示例代码

```
#include <stdio.h>
#include "tx_api.h"
#include "main.h"

#define THREAD1_PRIO         3
#define THREAD1_STACK_SIZE   1024
static  TX_THREAD thread1;
uint8_t thread1_stack[THREAD1_STACK_SIZE];

#define THREAD2_PRIO         2
#define THREAD2_STACK_SIZE   1024
static  TX_THREAD thread2;
uint8_t thread2_stack[THREAD2_STACK_SIZE];

void my_thread1_entry(ULONG thread_input)
{
  /* Enter into a forever loop. */
  while(1)
  {
    printf("threadx 1 application running...\r\n");
    /* Sleep for 1000 tick. */
    tx_thread_sleep(1000);
  }
}

void my_thread2_entry(ULONG thread_input)
{
  /* Enter into a forever loop. */
  while(1)
  {
    printf("threadx 2 application running...\r\n");
    /* Sleep for 1000 tick. */
    tx_thread_sleep(1000);
  }
}

void tx_application_define(void *first_unused_memory)
{
  /* Create thread */
  tx_thread_create(&thread1, "thread 1", my_thread1_entry, 0, &thread1_stack[0],
THREAD1_STACK_SIZE, THREAD1_PRIO, THREAD1_PRIO, TX_NO_TIME_SLICE, TX_AUTO_START);

  tx_thread_create(&thread2, "thread 2", my_thread2_entry, 0, &thread2_stack[0],
THREAD2_STACK_SIZE, THREAD2_PRIO, THREAD2_PRIO, TX_NO_TIME_SLICE, TX_AUTO_START);
}
```

5.2. 启动内核

在main.c中包含threadx头文件：

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "tx_api.h"
/* USER CODE END Includes */
```

然后在main函数中初始化部分之后启动内核：

```
/* USER CODE BEGIN 2 */

printf("threadX RTOS on BearPi IoT Board\r\n");

/* Enter the ThreadX kernel. */
tx_kernel_enter( );

/* USER CODE END 2 */
```

编译，下载，在串口终端查看系统运行结果：