

在 SAM7S 上运行 FreeRTOS

文档编号	MAN2003A_CH				
文档版本	Rev. A				
文档摘要	以 SAM7S 为例，说明 FreeRTOS 在 SAM7 上的运行，开发环境是 Keil MDK				
关键词	SAM7S FreeRTOS Keil MDK				
创建日期	2009-12-10	创建人员	Dracula	审核人员	Hotislandn
文档类型	公开发布/开发板配套文件				
版权信息	Mcuzone 原创文档，转载请注明出处				

更新历史

版本	时间	更新	作者
Rev. A	2009-12-10	初始创建	Dracula

微控电子 乐微电子
杭州市登云路 639 号 2B143
销售 TEL: +86-571-88908193
支持 TEL: 18913989166 13770507096
FAX: +86-571-88908193
www.mcuzone.com www.atarm.com

1.概述

[FreeRTOS](#) 是一个免费的 [RTOS](#)。其体积很小，核心只有 3 个 c 文件，目前已经移植到了各种体系架构上，包括 ARM7。

使用 RTOS 可以简化软件系统的设计，分解系统中的各种任务，并且代码便于移植与重用。

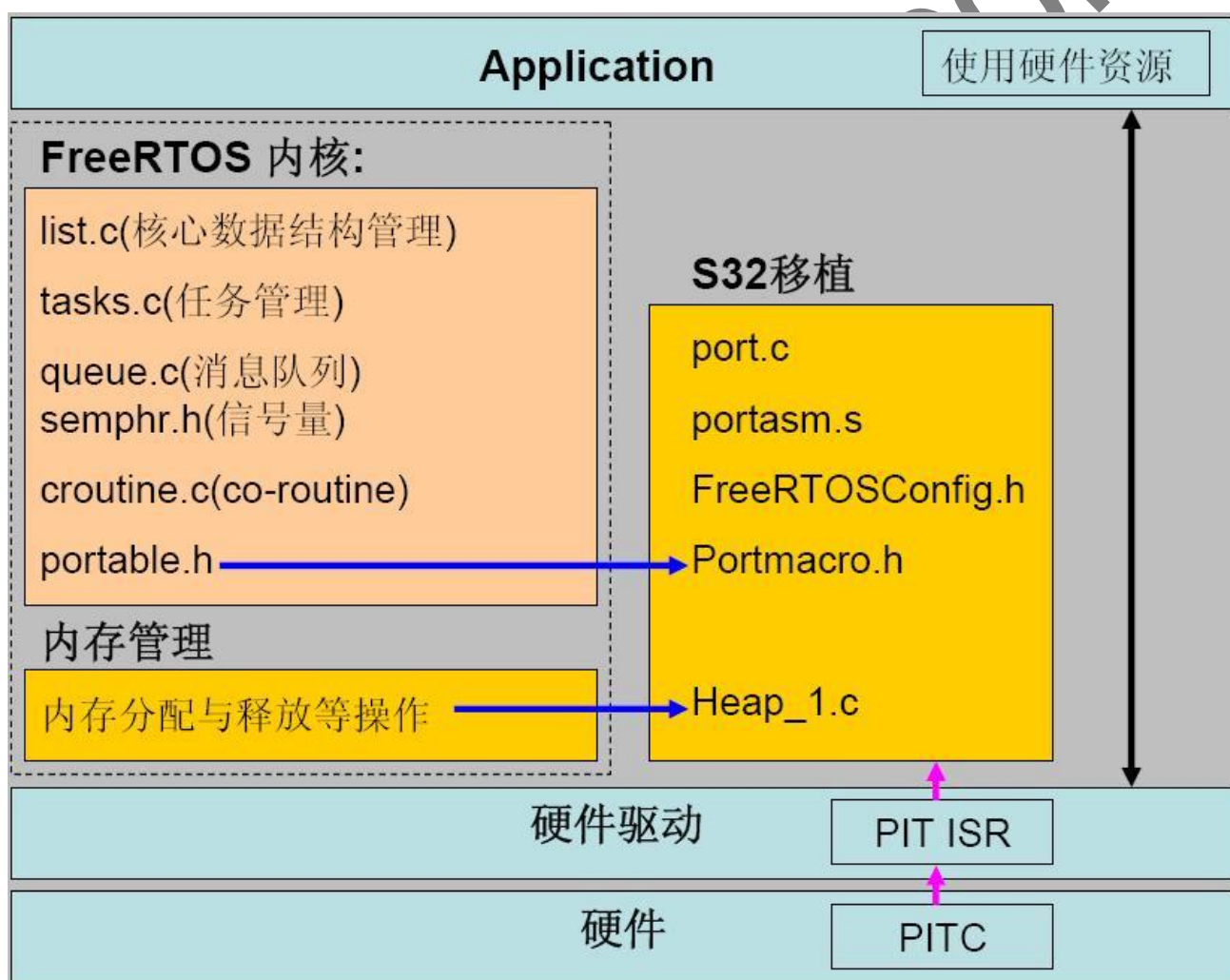
使用 RTOS 会带来额外的性能开销，包括处理能力，ROM(FLASH), RAM 的占用。

[ATMEL](#) 的 [SAM7S](#) MCU 基于 ARM7TDMI 架构，有足够的空间运行 RTOS，以简化软件设计。基于开源与易用的考虑，选择 FreeRTOS。

开发环境选择 [Keil MDK](#)，界面直观，且使用 ARM 官方的工具链，性能有保证。

本站的 wiki 中有一篇讲解 FreeRTOS 移植的文章，请参考 [《FreeRTOS 在 S32 上的移植》](#)。

FreeRTOS 的结构图如下：



2. 设置环境

2.1 安装 Keil MDK

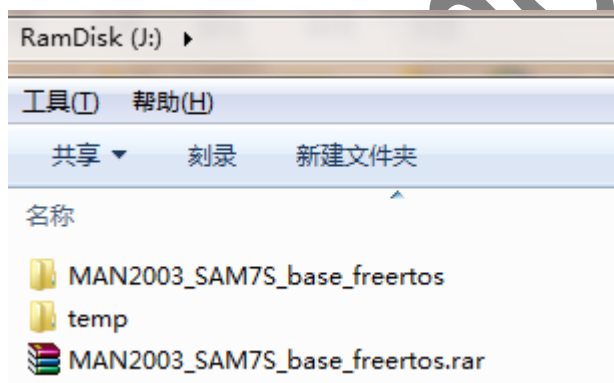
下载 Keil MDK 并安装。

本文档涉及的项目工程基于 MDK 3.80a 创建。

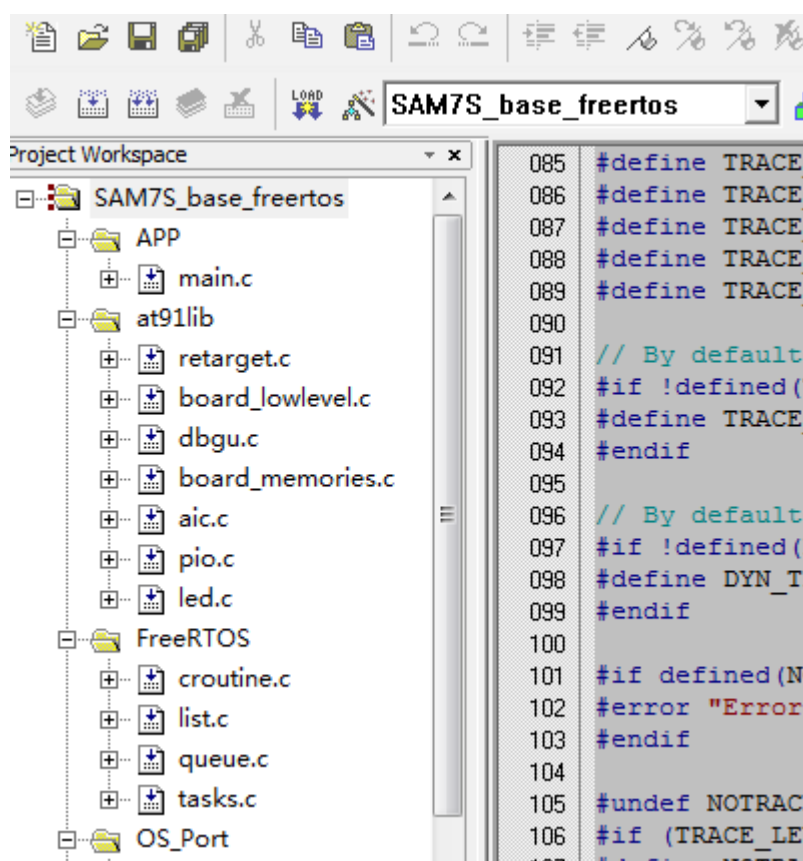


2.2 打开工程

下载项目文件 MAN2003_SAM7S_base_freertos.rar, 并展开:



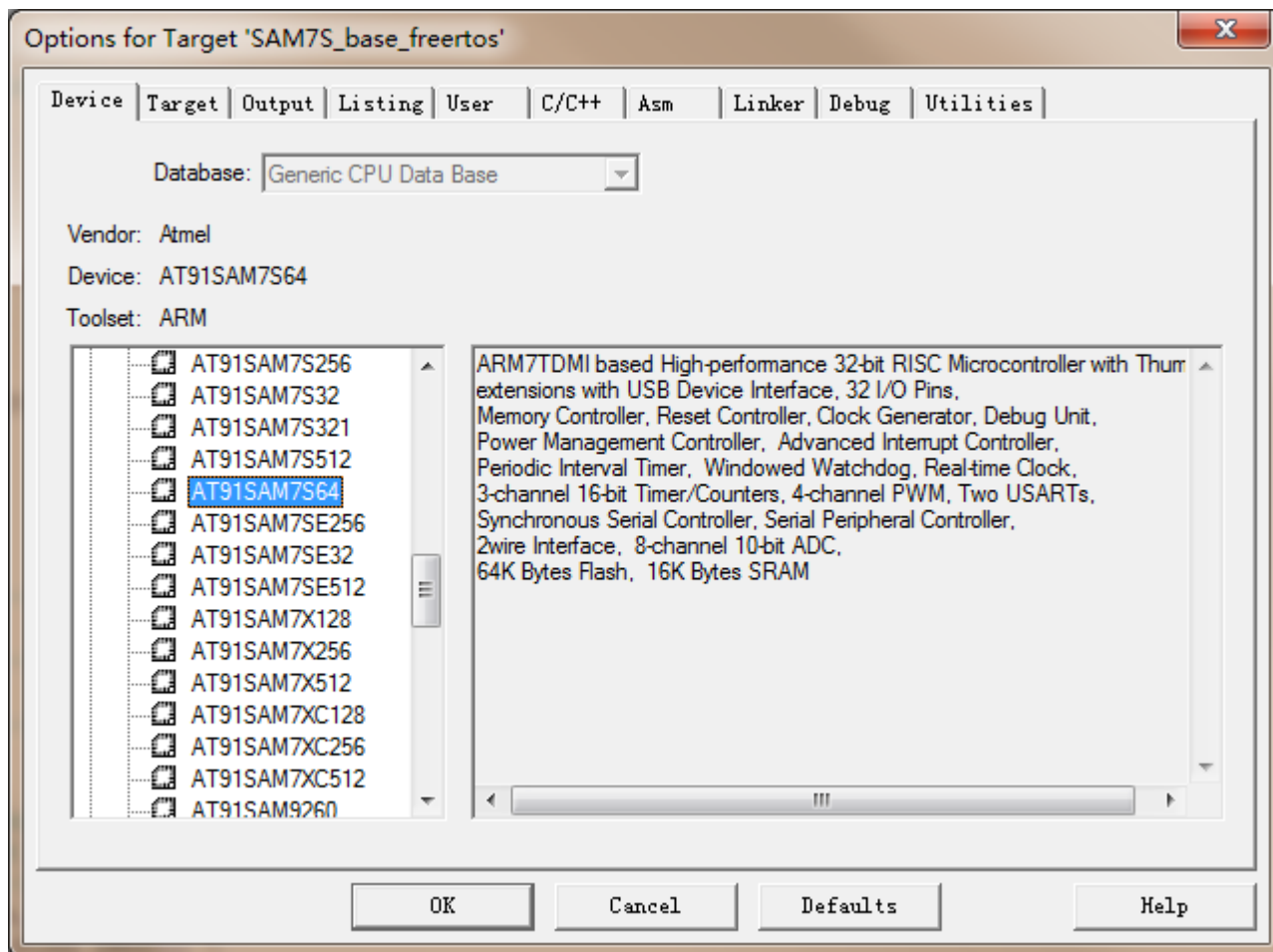
使用 keil 打开工程 sam7s_base_freertos.Uv2:



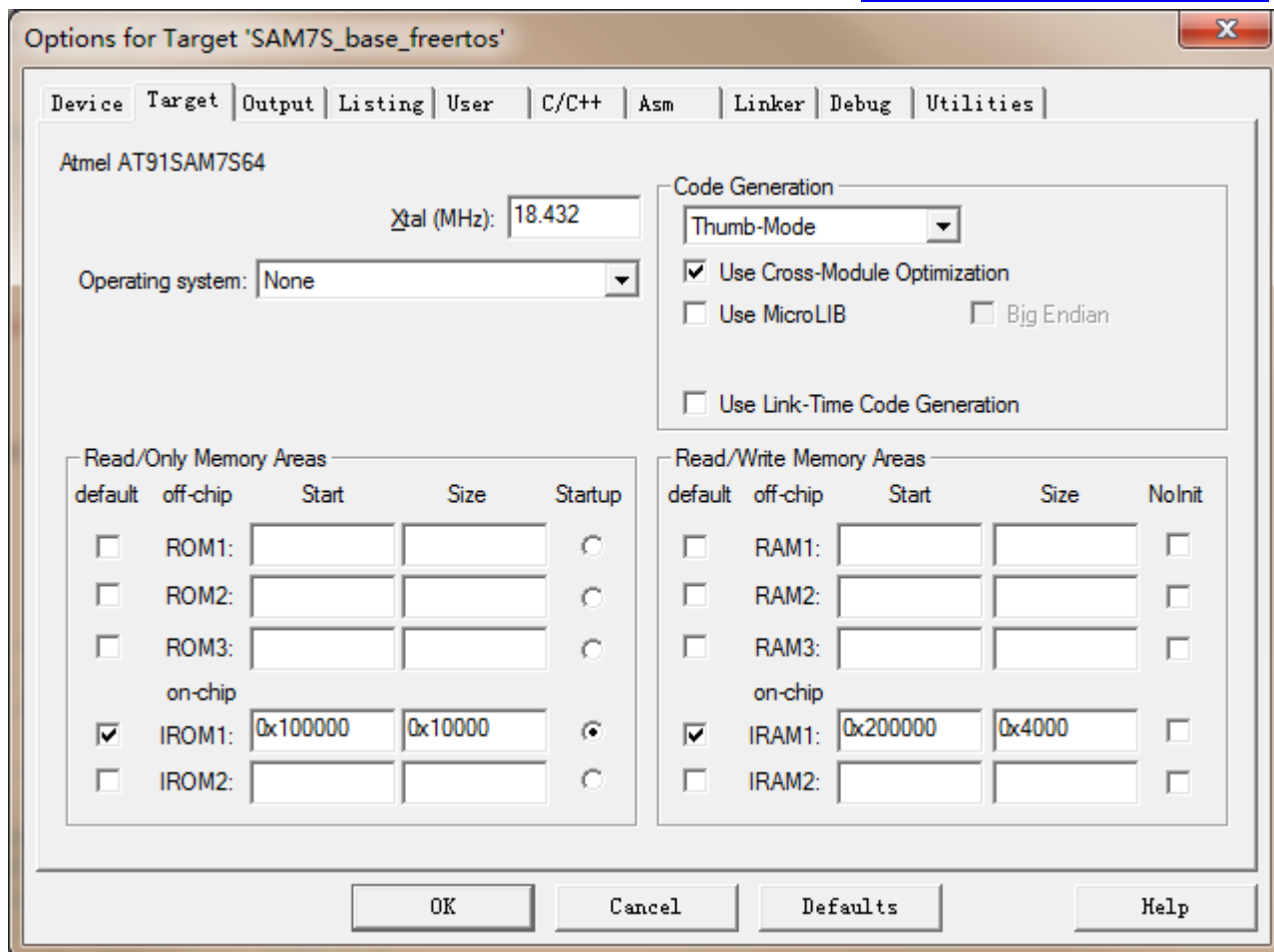
2.3 配置工程

为了保证工程能正确编译，需要核对工程的相关设置。

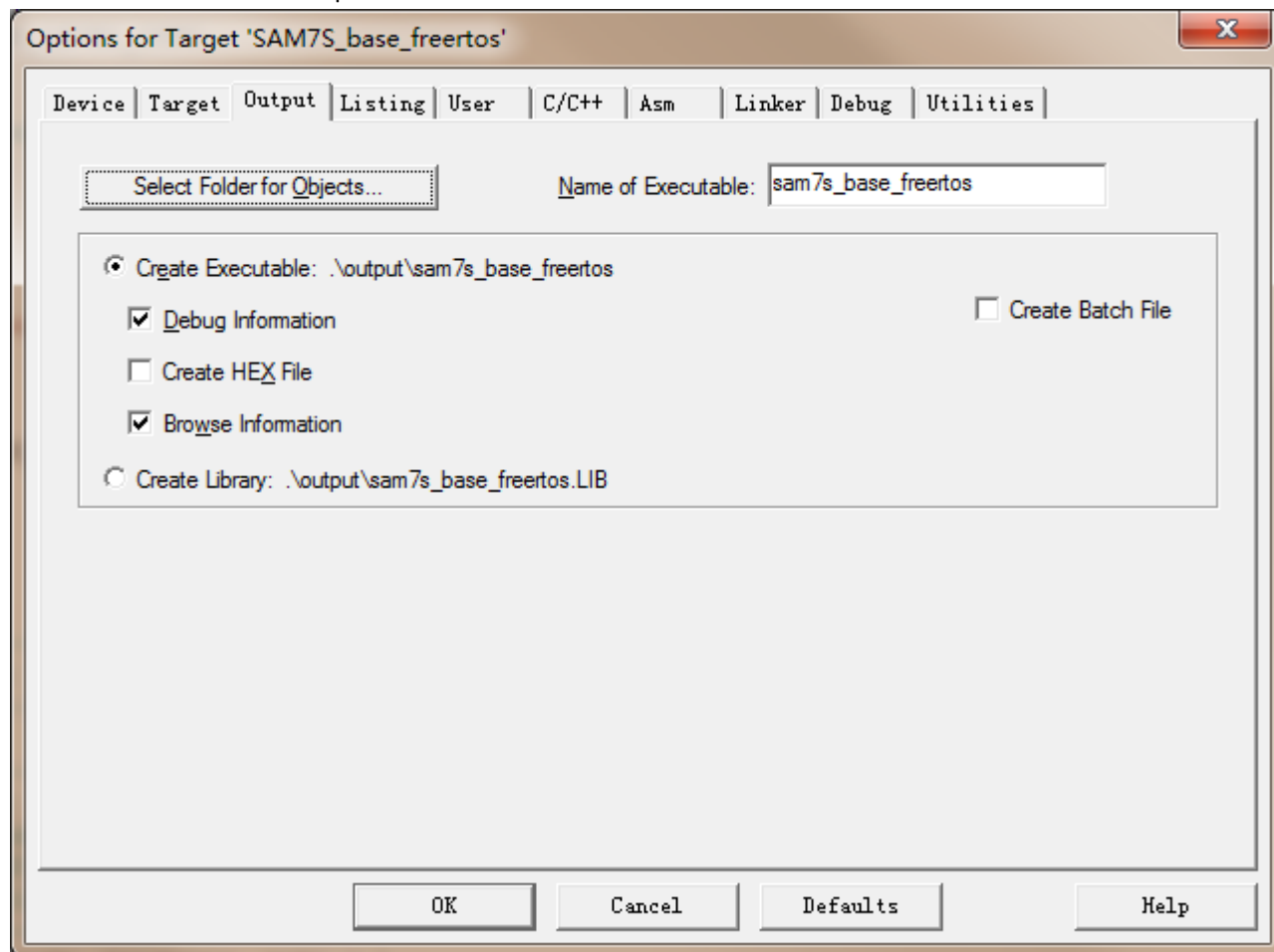
首先检查芯片设置：



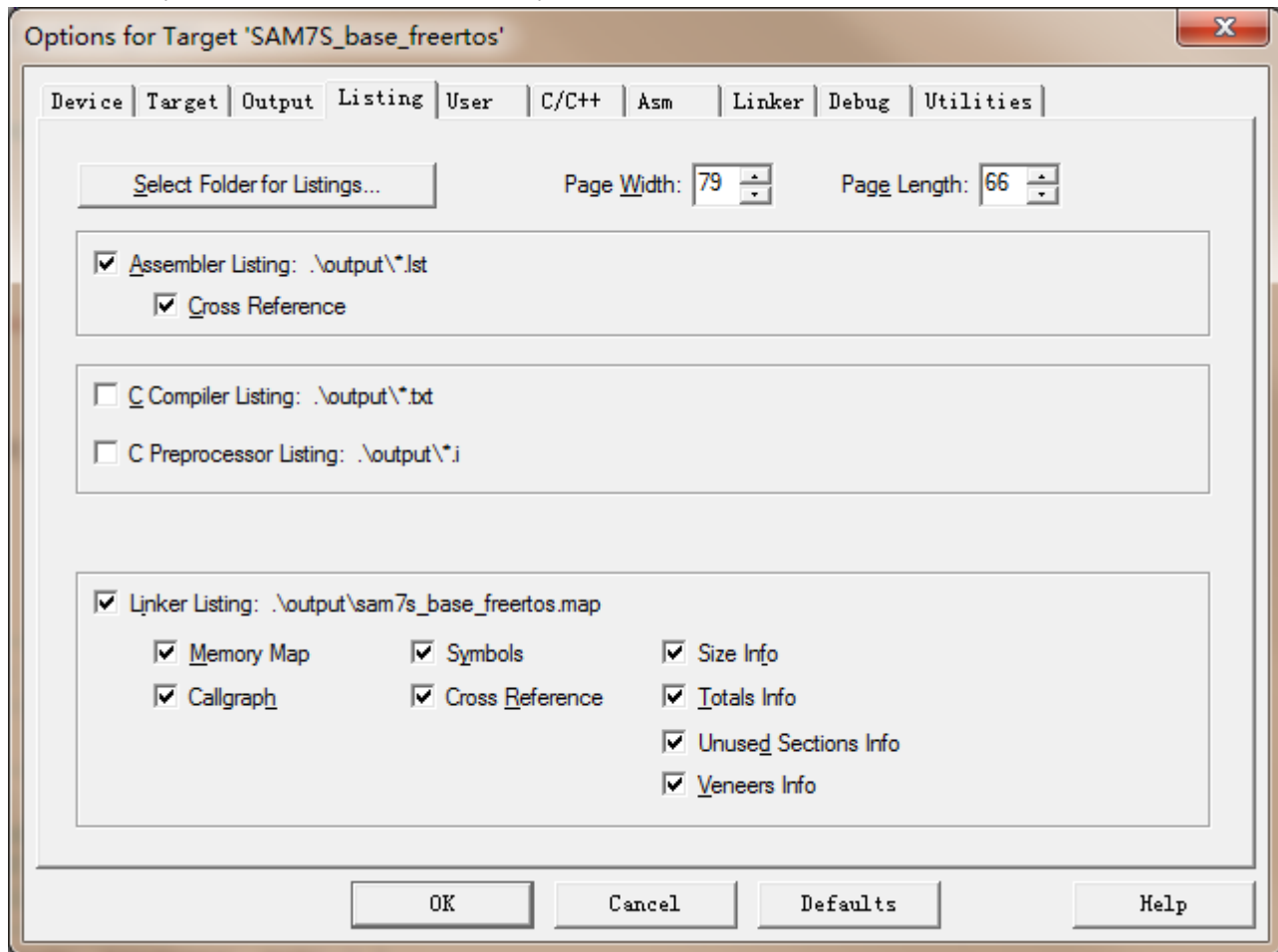
然后是时钟及处理器模式，时钟设置为 18.432MHz，只是 ATEML 默认的时钟配置。使用 Thumb 来生成代码，这样处理特别声明的代码，都将编译为 Thumb，可以节省 flash 空间，[并提高代码在 Flash 中运行的速度](#)。



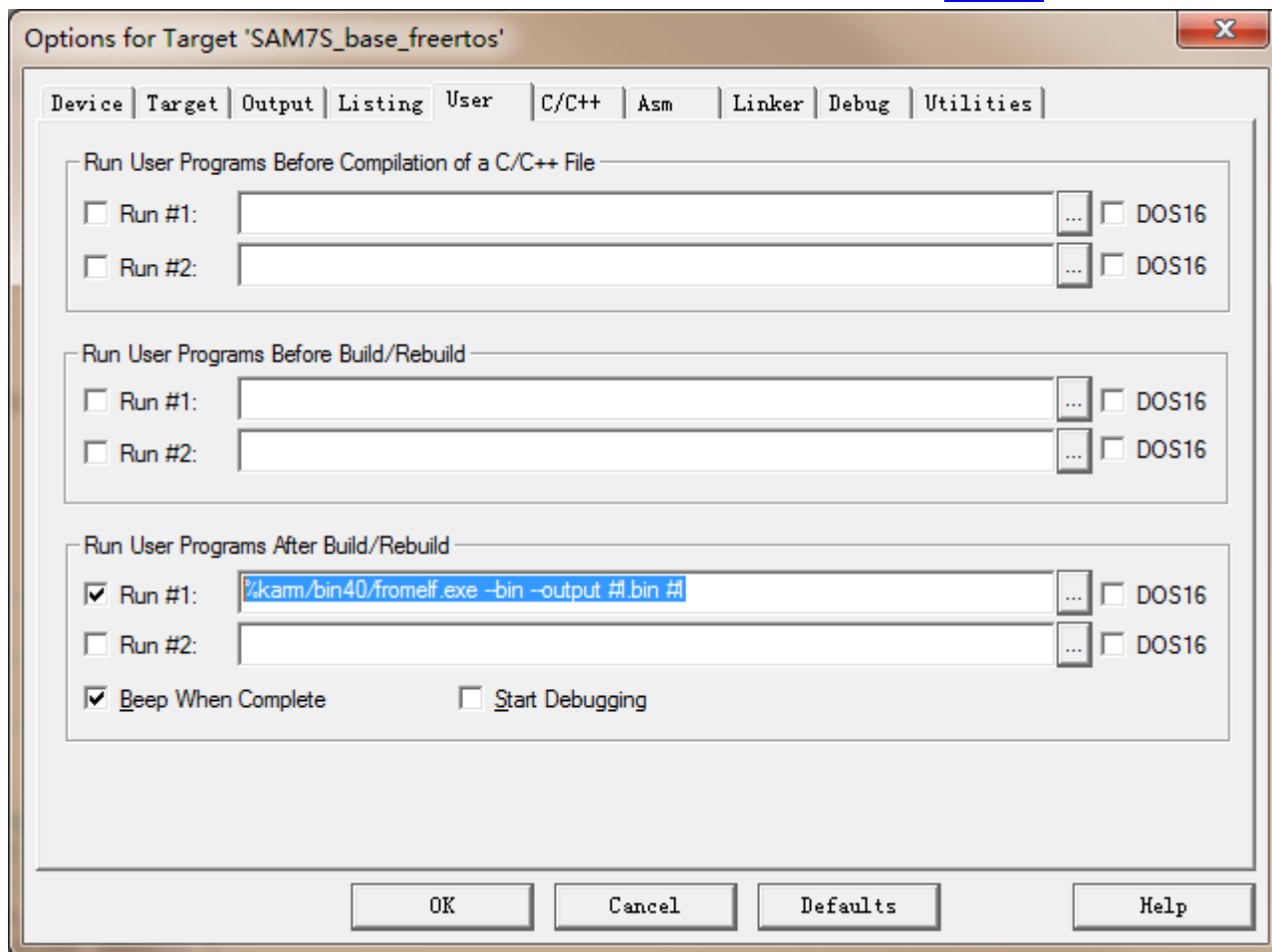
配置输出为工程目录的 output 文件夹，可以避免在工程的根目录生成中间文件，使得工程比较整洁：



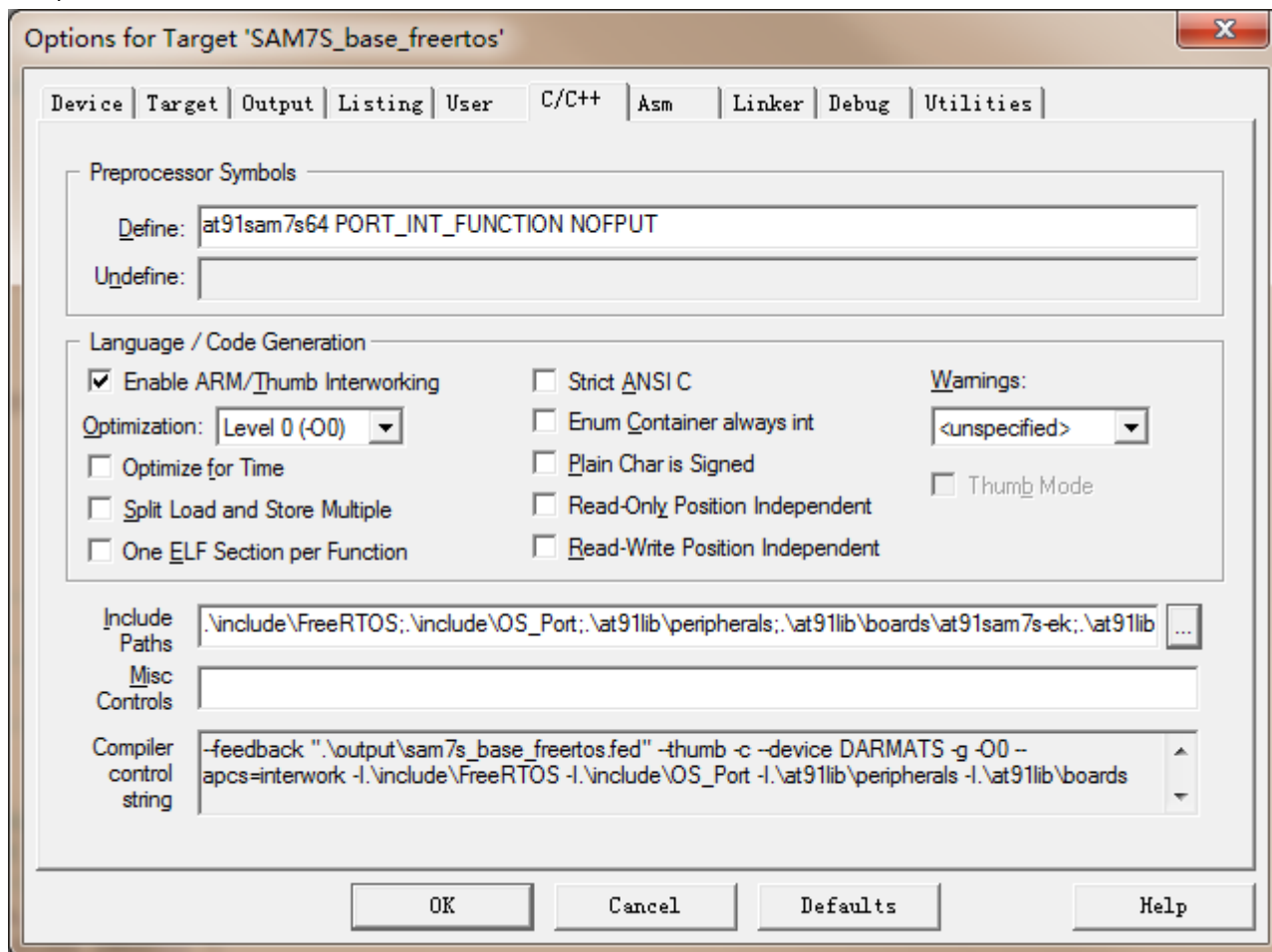
选择输出 map 文件，便于对代码的 memmap 的分析：



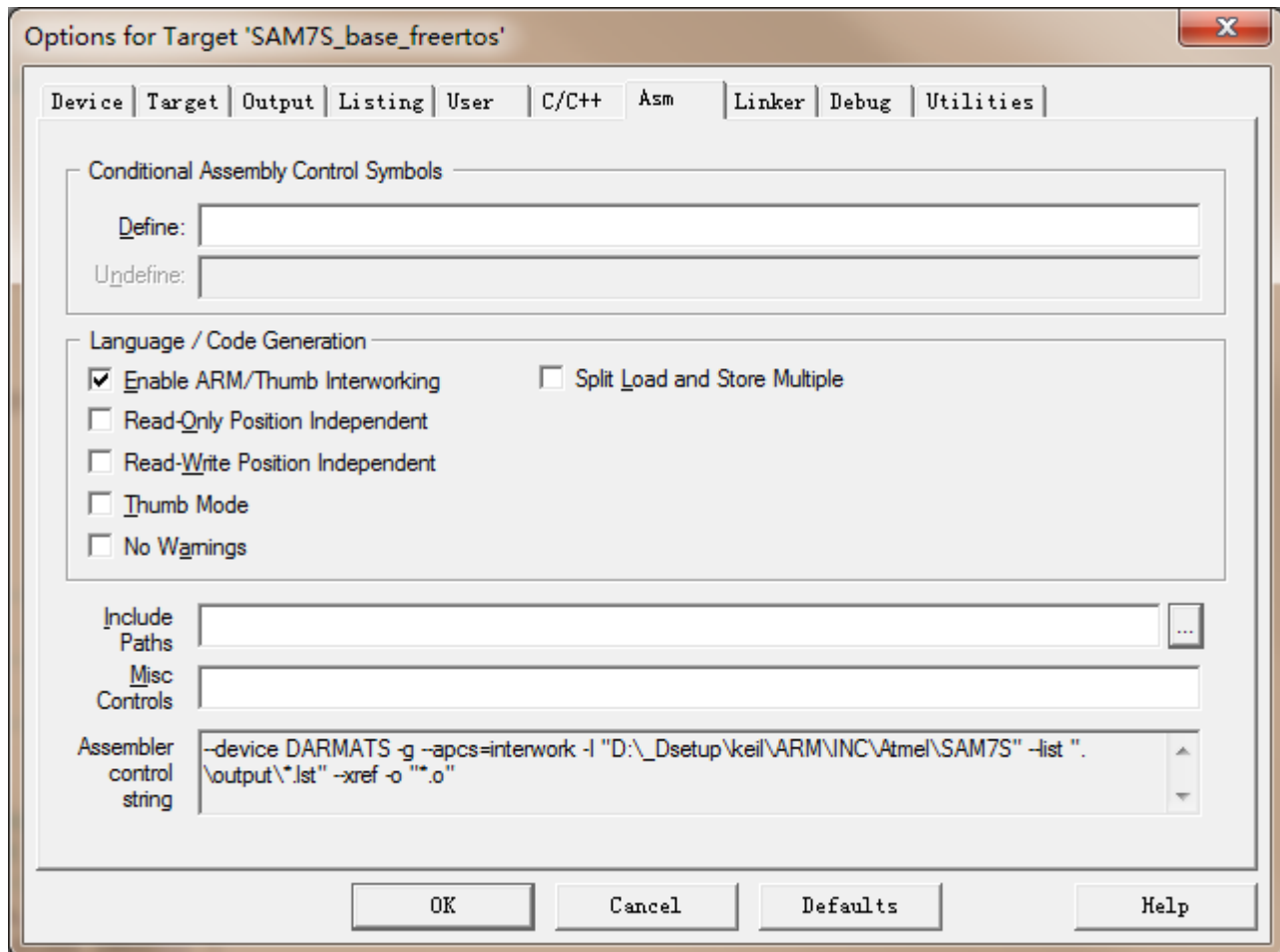
配置在编译完成后运行一个用户命令，使用 `fromelf` 工具输出 `bin` 文件，便于 [SAM-BA](#) 烧写：



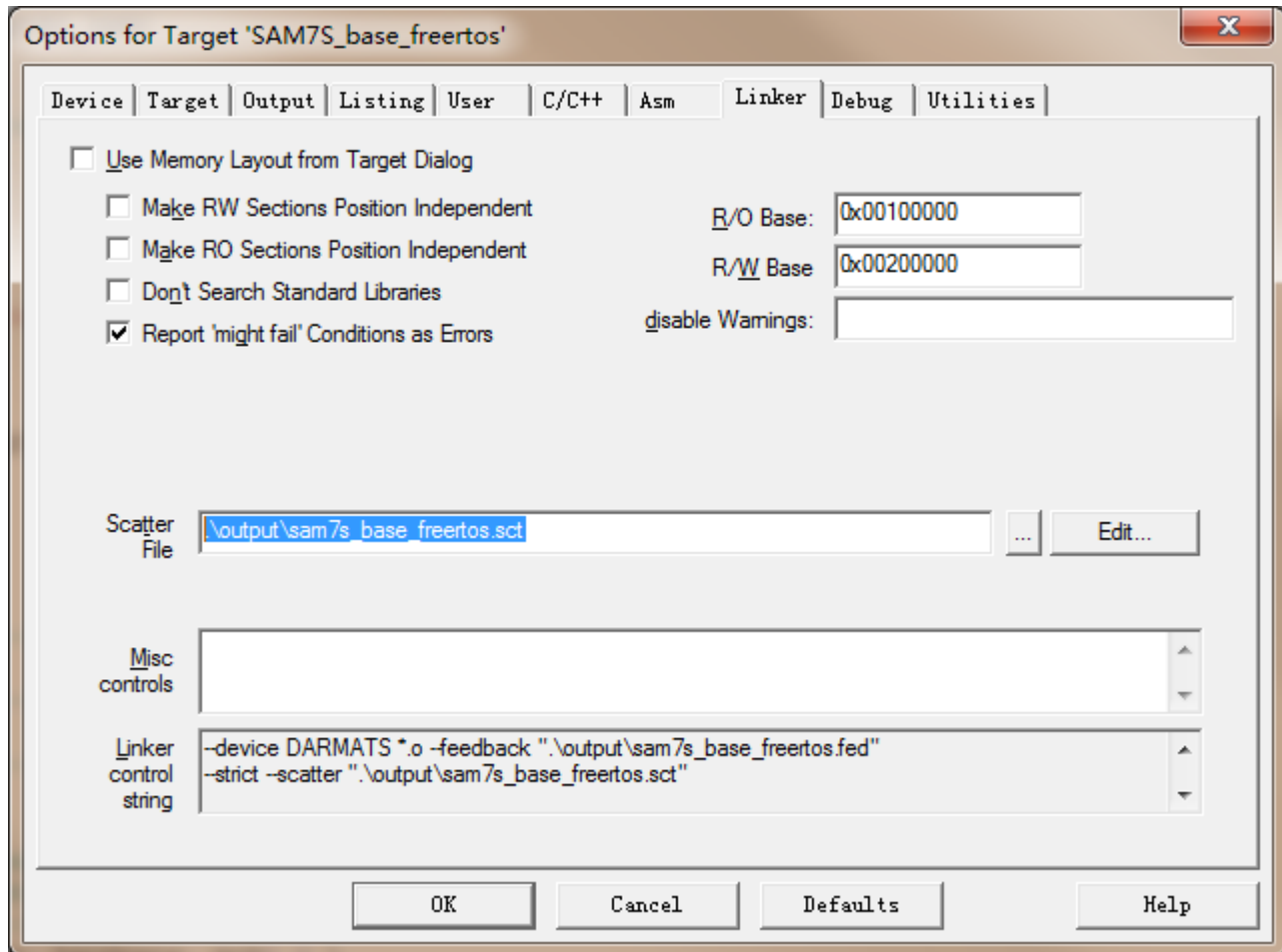
在 C/C++选项卡中可以定义一些宏，控制优化，头文件位置及特殊的编译选项：



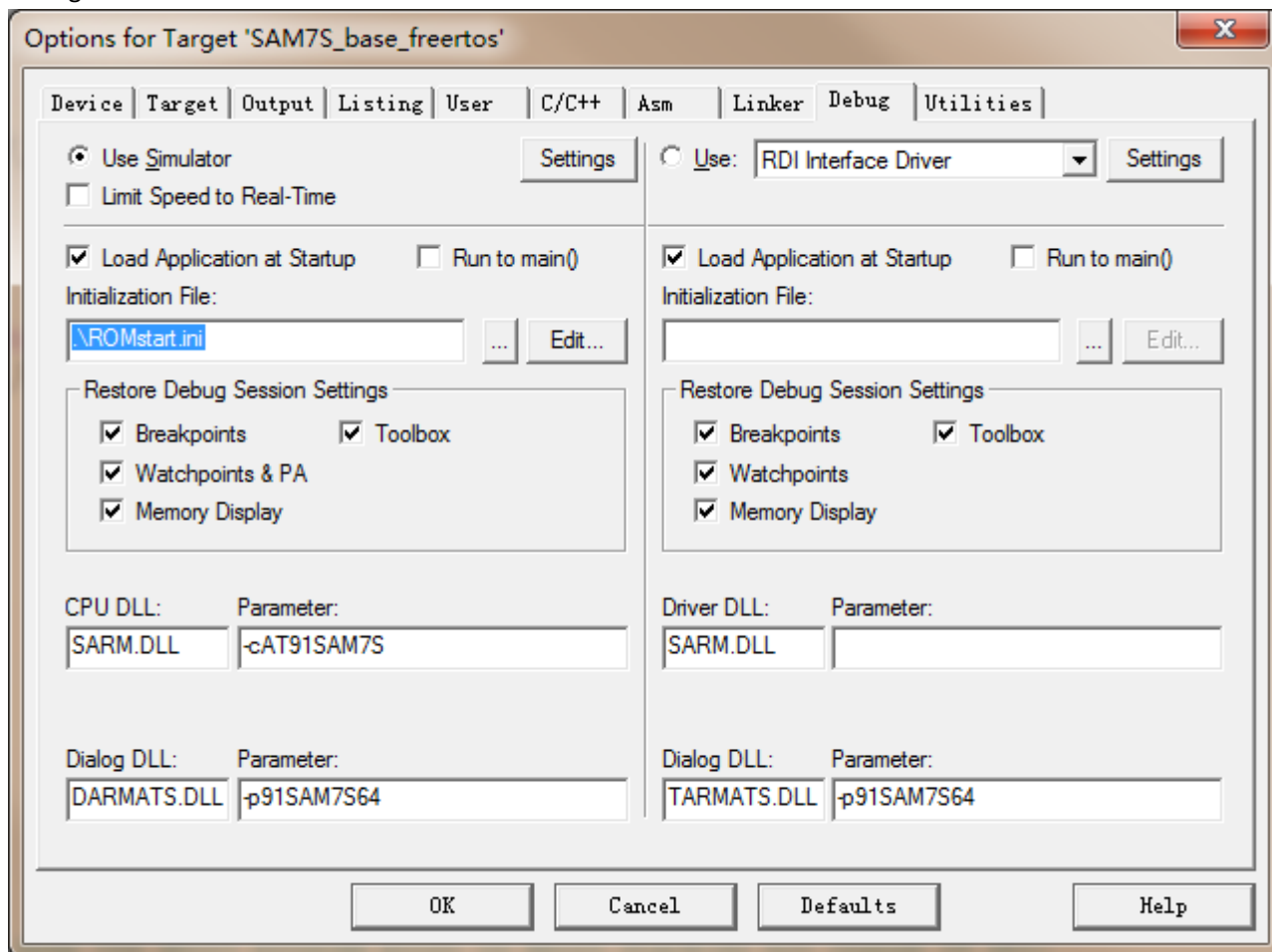
ASM 选项卡与 C/C++ 的类似，但是是为控制汇编文件准备的：



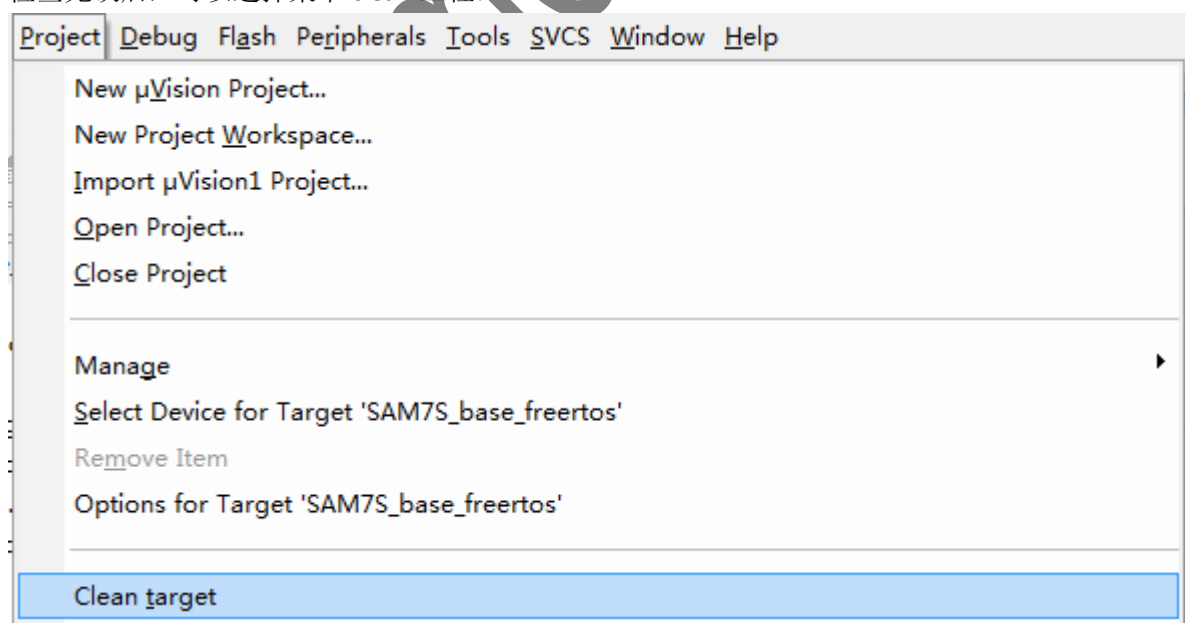
Linker 选项卡控制代码的连接，比较重要。本工程中需要比较复杂的 memmap(部分代码位于 RAM)，所以使用 scatter loader file 来控制：



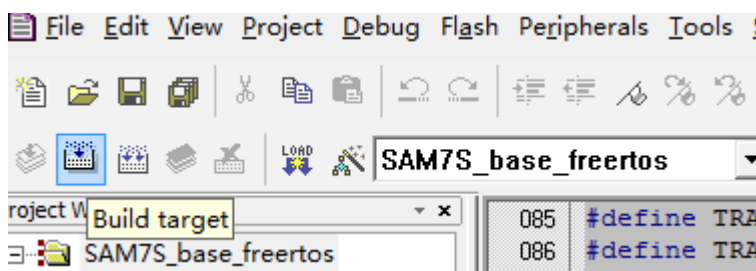
Debug 选项卡中配置使用软件仿真：



检查无误后，可以选择菜单 clean 工程：



然后重新编译工程：



无误的话将生成 axf 文件及 bin 文件：

```

.\output\sam7s_base_freertos.sct(36): warning: L6314W: No section matches pattern rtt.o(R0).
.\output\sam7s_base_freertos.sct(37): warning: L6314W: No section matches pattern adc.o(R0).
.\output\sam7s_base_freertos.sct(43): warning: L6329W: Pattern queue.o(R0) only matches removed unused sections.
Program Size: Code=7680 RO-data=124 RW-data=76 ZI-data=9320
User command #1: D:\Dsetup\keil\arm\bin40\fromelf.exe --bin --output J:\MAN2003 SAM7S base freertos\output\sam7s_base_freertos.axf
".\output\sam7s_base_freertos.axf" - 0 Error(s), 8 Warning(s).

```

注意编译后的 memmap：

tasks.c	Execution Region RAM_VECT (Base: 0x00200000, Size: 0x000000c0, Max: 0x00000300, ABSOLUTE)						
	Base Addr	Size	Type	Attr	Idx	E Section Name	Object
OS_Port	0x00200000	0x000000c0	Code	RO	350	RAM_VECTOR	ram_vect.o
heap_1.c	Execution Region RAM_CODE (Base: 0x002000c0, Size: 0x00000128, Max: 0xffffffff, ABSOLUTE)						
port.c	0x002000c0	0x00000004	Ven	RO	343	FreeRTOS_PORT_ASM	portasm.o
portISR.c	0x002000c4	0x00000124	Code	RO	343	FreeRTOS_PORT_ASM	portasm.o
portasm.s	Execution Region ER_data_ram (Base: 0x002001e8, Size: 0x000020b4, Max: 0xffffffff, ABSOLUTE)						
SAM7S							
Cstartup_rv.S							
ram_vect.S							
other							
sam7s_base_freertos.sct							
sam7s_base_freertos.ma							
ROMstart.ini							

portasm.s 中的代码在 RAM 中运行。

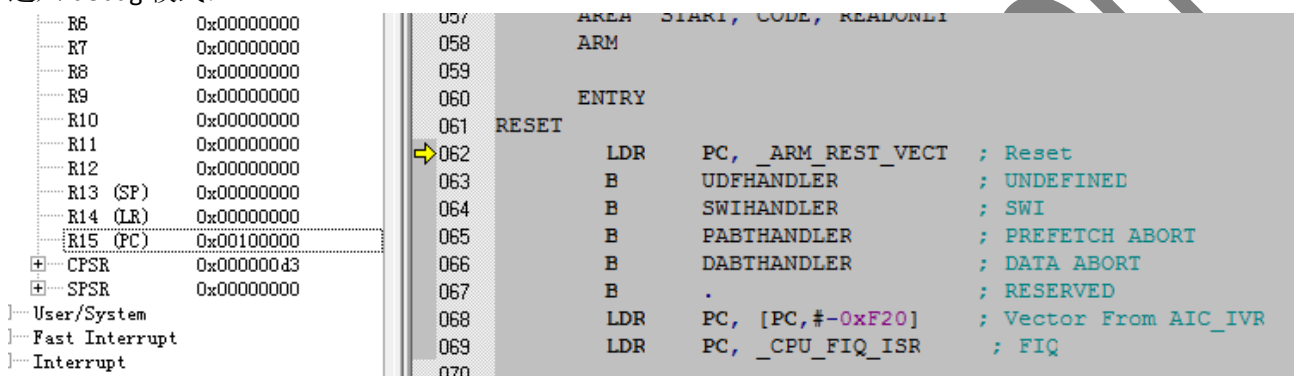
3. 模拟仿真

3.1 简单仿真

点击 debug 按钮:

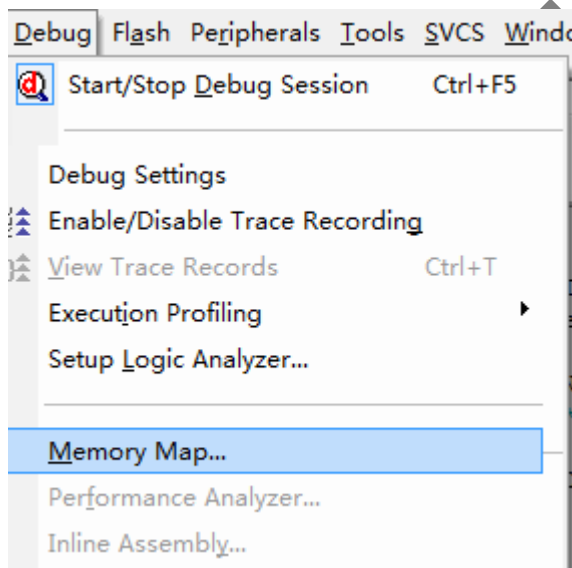


进入 debug 模式:

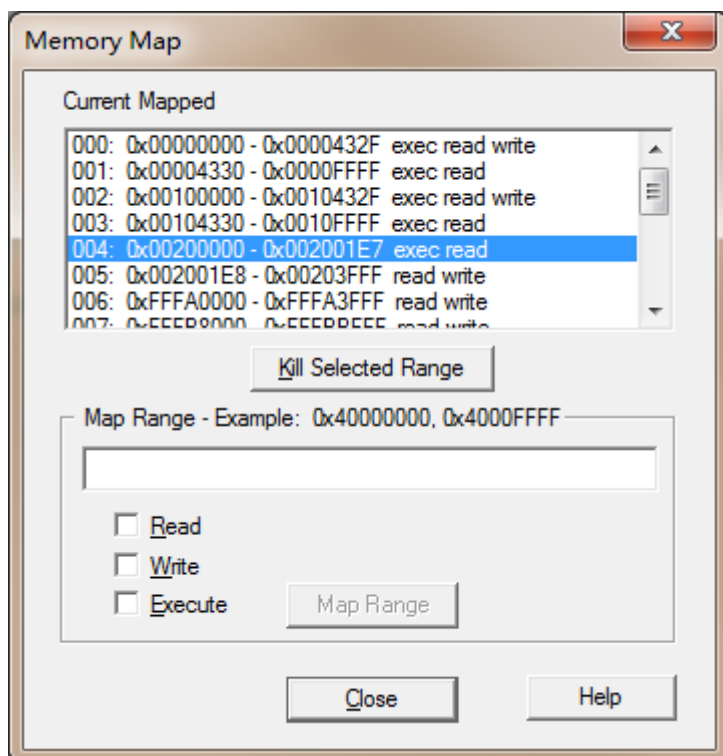


黄色箭头就是 PC 指针，也就是当前程序运行的位置。

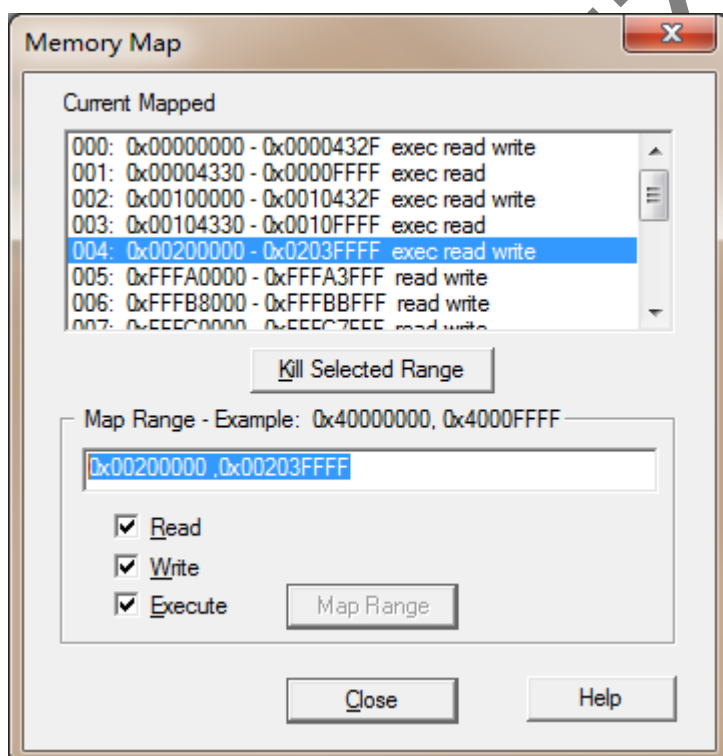
由于代码部分运行于 RAM，所以首先要修改 Keil 默认的 mem 权限:



首先删除内部 RAM:



为 RAM 新建一个属性，支持所有操作：



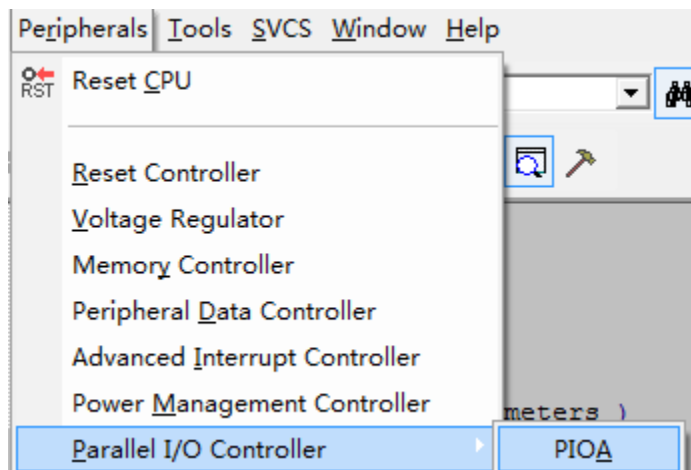
选择 Map Range 新建新的 memmap。

打开文件 main.c，并设置断点：

```

24     while(1)
25     {
26         LED_Set(0);
27         vTaskDelay(250);
28         LED_Set(1);
29         vTaskDelay(250);
30         LED_Set(2);
31         vTaskDelay(250);
32
33         LED_Clear(0);
34         LED_Clear(1);
35         LED_Clear(2);
36         vTaskDelay(248);
37     }
38 }
    
```

打开外设窗口的 PIO 窗口：



模拟的 PIO 出现:

PIOA: Parallel I/O Controller A

PIO / Output / Input Filter / Output Data / Multi Driver / Pull-up / AB Select / Output Write

PIOA_PSR: 0x00000000

PIOA_OSR: 0x00000000

PIOA_IFSR: 0x00000000

PIOA_ODSR: 0x00000000

PIOA_MDSR: 0x00000000

PIOA_PUSR: 0x00000000

PIOA_ABSR: 0x00000000

PIOA_OWSR: 0x00000000

Pin Data Status

PIOA_PDSR: 0x09000080

I/O Pins

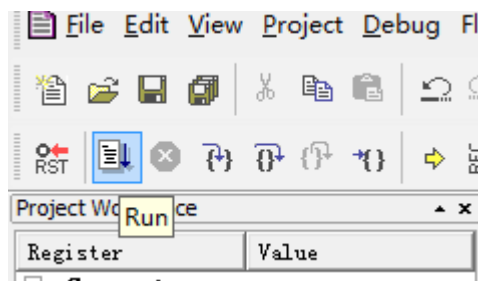
Pins: 0x09000080

Interrupt Mask & Status

PIOA_IMR: 0x00000000

PIOA_ISR: 0x00000000

点击运行按钮:

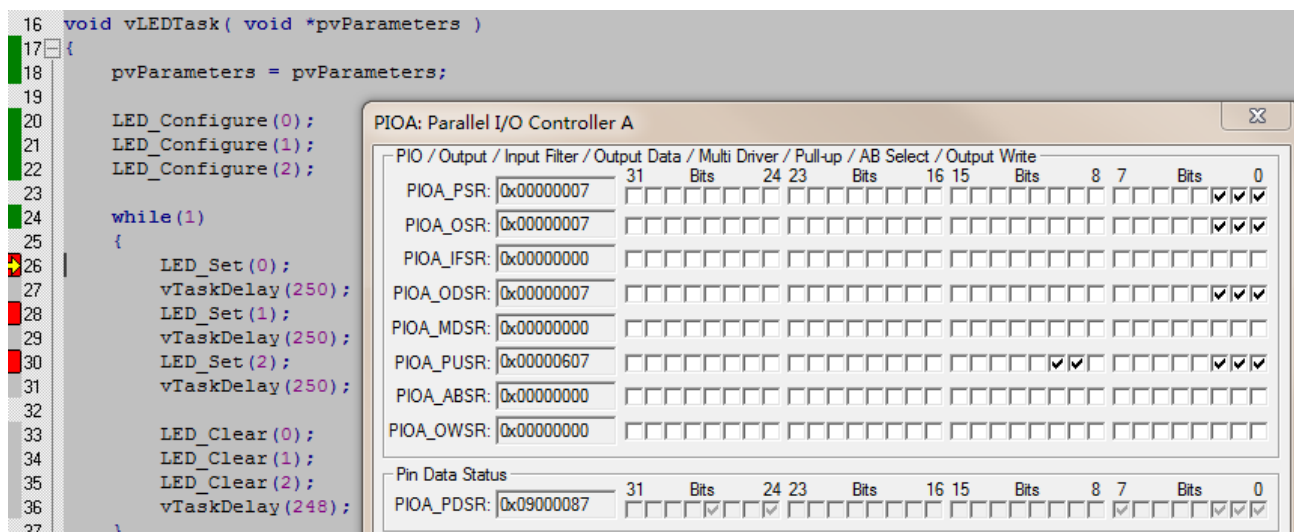


代码运行到第一个断点处:

```

61 int main(void)
62 {
63     /* Setup the ports. */
64     prvSetupHardware();
65
66     /* LED task */
67     xTaskCreate( vLEDTask,      ( signed portCHAR * )
68 
```

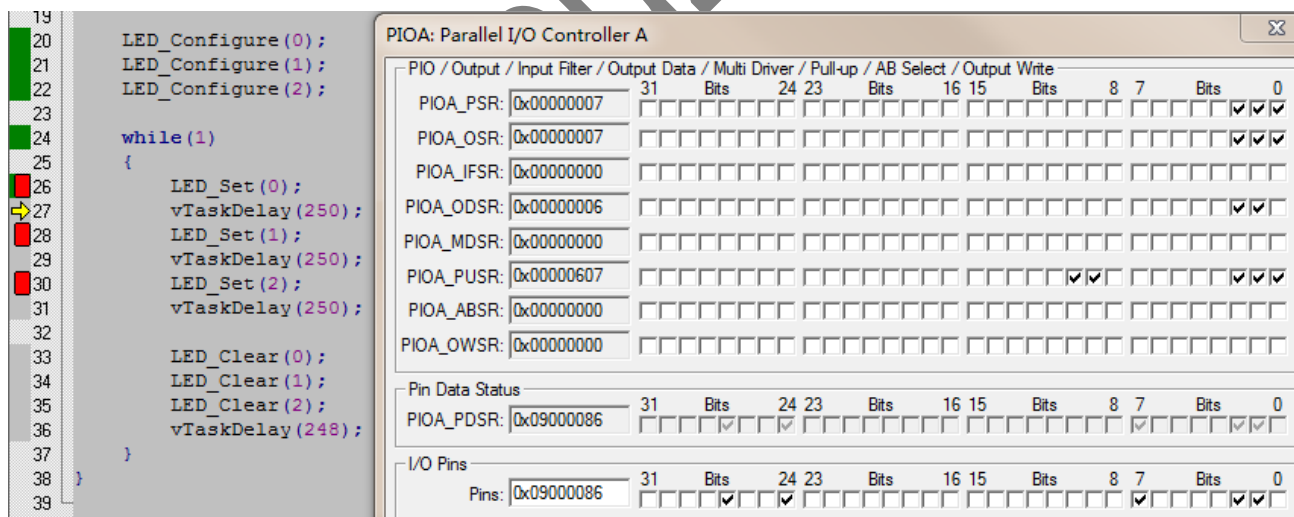
再次点击运行，停在 LED 任务中：



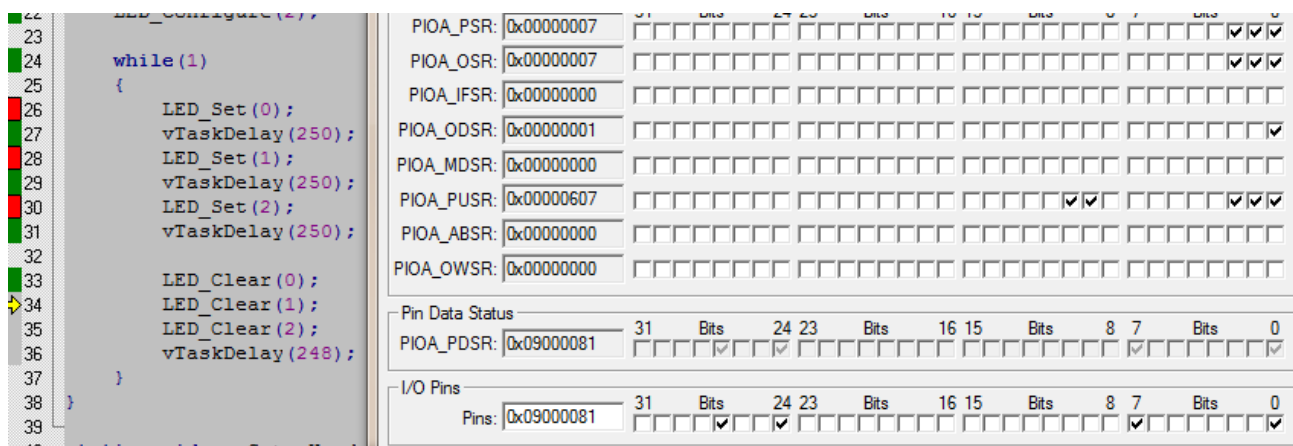
注意此时 LED 对应的 PIO 状态：

```
/// LED #0 pin definition (PA0).
#define PIN_LED_DS1 {1 << 0, AT91C_BASE_PIOA, AT91C_ID_PIOA, PIO_OUTPUT_1, PIO_DEFAULT}
/// LED #1 pin definition (PA1).
#define PIN_LED_DS2 {1 << 1, AT91C_BASE_PIOA, AT91C_ID_PIOA, PIO_OUTPUT_1, PIO_DEFAULT}
/// LED #2 pin definition (PA2).
#define PIN_LED_DS3 {1 << 2, AT91C_BASE_PIOA, AT91C_ID_PIOA, PIO_OUTPUT_1, PIO_DEFAULT}
/// LED #3 pin definition (PA3).
#define PIN_LED_DS4 {1 << 3, AT91C_BASE_PIOA, AT91C_ID_PIOA, PIO_OUTPUT_1, PIO_DEFAULT}
/// List of the four LED pin definitions (PA0, PA1, PA2 & PA3)
#define PINS_LEDS PIN_LED_DS1, PIN_LED_DS2, PIN_LED_DS3, PIN_LED_DS4
```

按下 F10，运行过 LED，可以看到 PIOA 的状态改变：



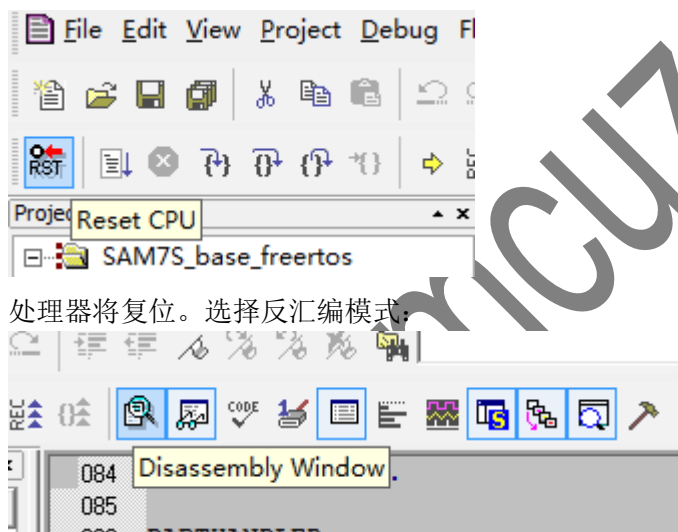
PIOA0 变成了低电平。再次运行可以看到对应引脚的变化:



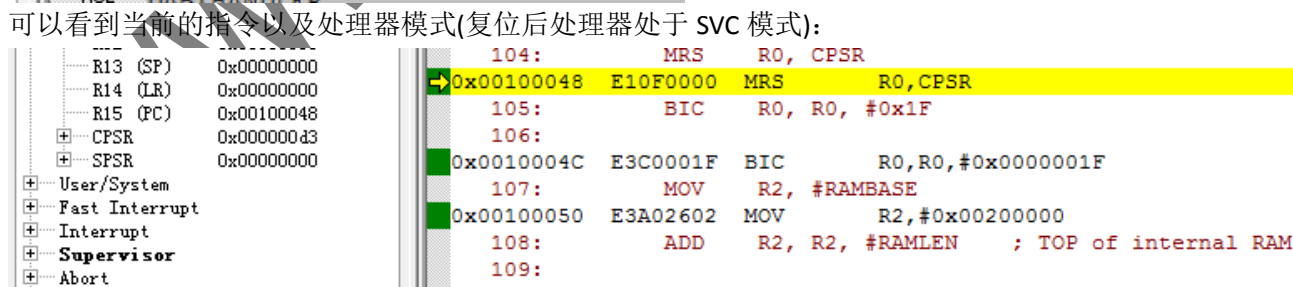
LED 任务就是控制 3 个 LED 闪烁。

3.2 调试中断

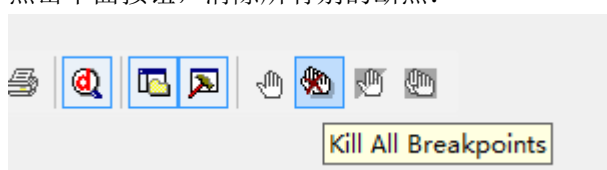
点击复位:



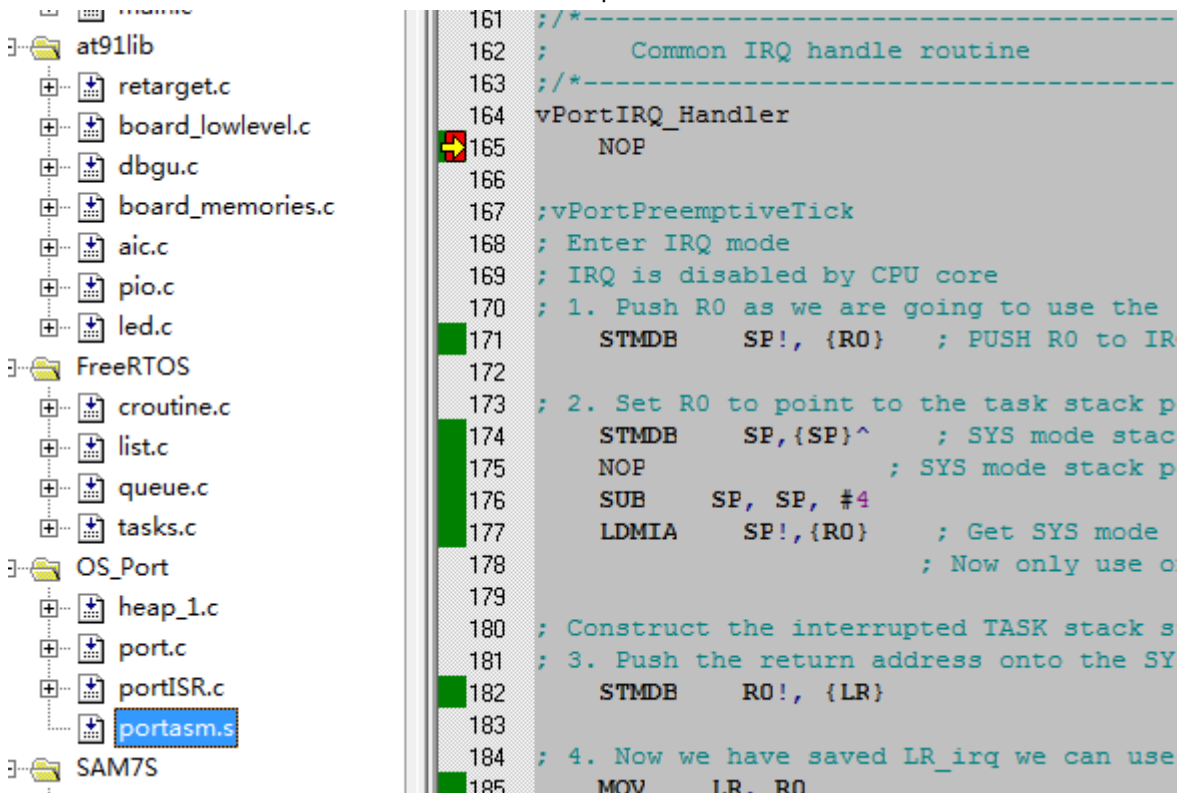
处理器将复位。选择反汇编模式:



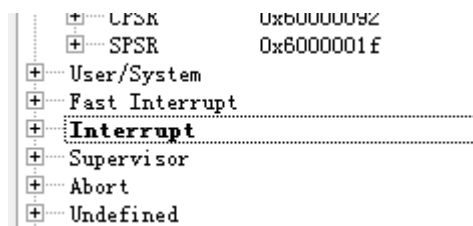
可以看到当前的指令以及处理器模式(复位后处理器处于 SVC 模式):



FreeRTOS 中有 PIT 中断，为了调试中断，可以在 portasm.S 中的 IRQ 异常服务程序处打断点：



点击运行，程序将停在该处。此时的处理器模式为：



AIC 控制器状态为：

Source	Name	Index	Vector	Type	Pending	Mask	Fast	Priority
Fast External Interrupt	FIQ	0	001002F5H	Low Level	1	0		
System Interrupt	SYSIRQ	1	00100568H	Level Sensitive	1	1	0	7
Parallel I/O Controller A	PIOAIRQ	2	001002F9H	Level Sensitive	0	0	0	0
A/D Converter	ADCIRQ	4	001002F9H	Level Sensitive	0	0	0	0
SPI	SPIIRQ	5	001002F9H	Level Sensitive	1	0	0	0
USART Channel 0	US0IRQ	6	001002F9H	Level Sensitive	0	0	0	0
USART Channel 1	US1IRQ	7	001002F9H	Level Sensitive	0	0	0	0
SSC	SSCIRQ	8	001002F9H	Level Sensitive	0	0	0	0
TWI	TWIIIRQ	9	001002F9H	Level Sensitive	0	0	0	0
PWM	PWMIRQ	10	001002F9H	Level Sensitive	0	0	0	0
Timer Channel 0	TC0IRQ	12	001002F9H	Level Sensitive	0	0	0	0
Timer Channel 1	TC1IRQ	13	001002F9H	Level Sensitive	0	0	0	0

Selected Interrupt			
AIC_SVR1: 0x00100568	Type: Level Sensitive	<input checked="" type="checkbox"/> Pending	<input checked="" type="checkbox"/> Mask <input type="checkbox"/> Fast Priority: 7

AIC_IVR: 0x00100568	AIC_IPR: 0xC0000023	AIC_CISR: 0x00000002	<input type="checkbox"/> NFIQ <input checked="" type="checkbox"/> NIRQ
AIC_FVR: 0x001002F1	AIC_IMR: 0x00000002	AIC_SPU: 0x001002F1	
AIC_ISR: 0x00000000	AIC_FFSR: 0x00000000	AIC_DCR: 0x00000001	<input checked="" type="checkbox"/> PROT <input type="checkbox"/> GMSK

在 SAM7S 上运行 FreeRTOS

MAN2003A

PIT 状态为:

PIT: Periodic Interval Timer

Mode

PIT_MR: 0x03000BB3 PIV: 0x000BB3 ☒ PITEN ☒ PITIEN

Status

PIT_SR: 0x00000001 ☒ PITS

Periodic Interval Value

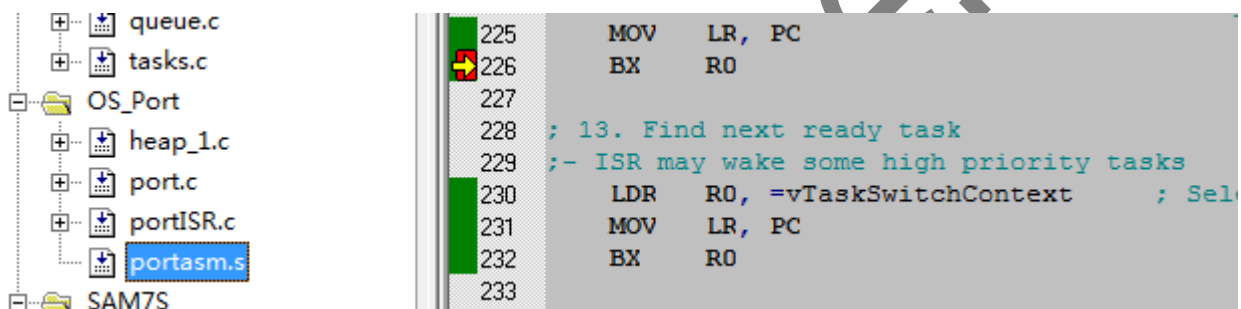
PIT_PIVR: 0x00100000 CPIV: 0x000000 PICNT: 0x0001

Periodic Interval Image

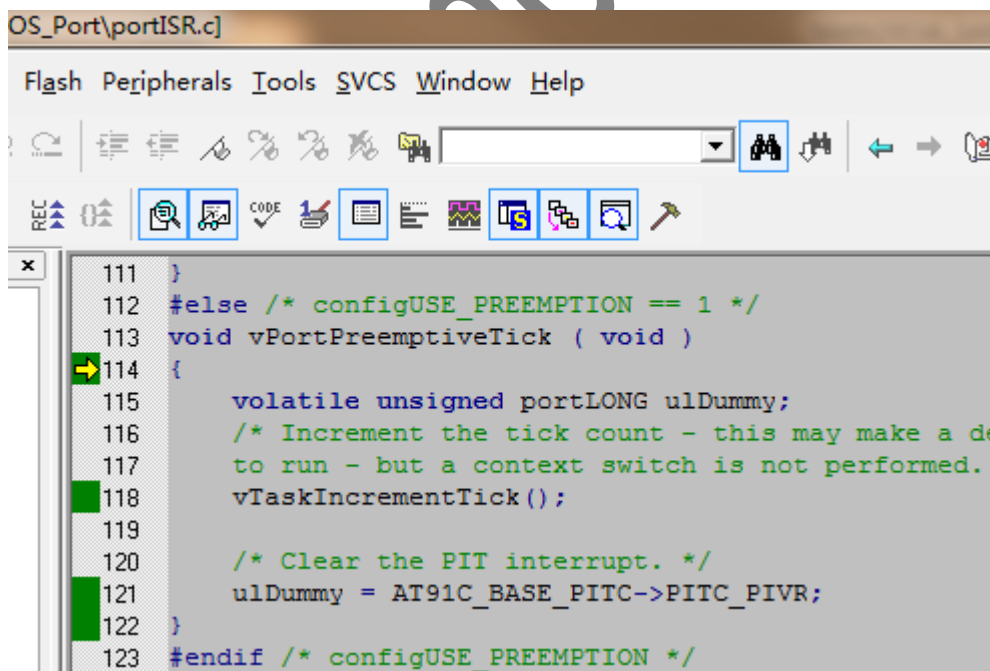
PIT_PIIR: 0x00100000 CPIV: 0x000000 PICNT: 0x0001

PITS 已经置位。

按 F11 可以单步跟踪代码的运行, 注意寄存器和栈的变化, 结合代码上的注释理解 RTOS IRQ 运行的过程。在下面的位置打个断点, 并全速运行到断点位置:



按下 F11, 程序跳转到真正的 C 语言写的对应的 ISR:



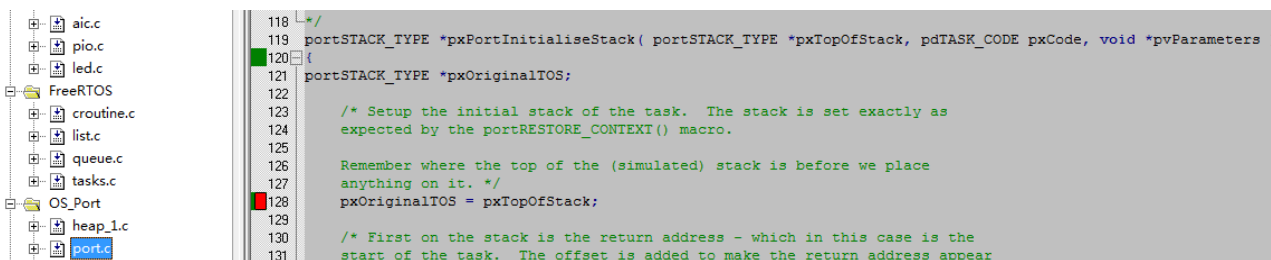
3.3 调试任务

任务(task)可以是 RTOS 的一个独立执行单元，一个任务(task)认为它自己独享了系统资源。但实际的任务被调度器调度运行。

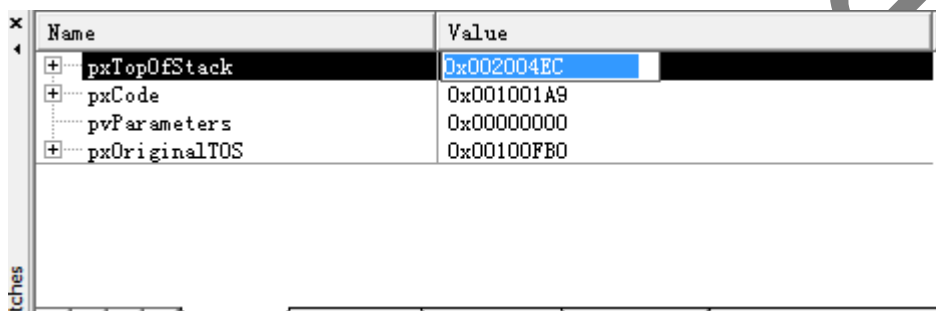
复位处理器，并清除前面的断点。

任务的创建可以跟踪如下代码来了解。

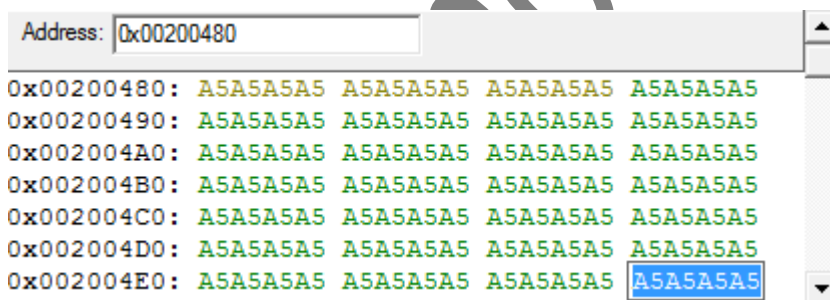
在 port.c 的如下位置打个断点：



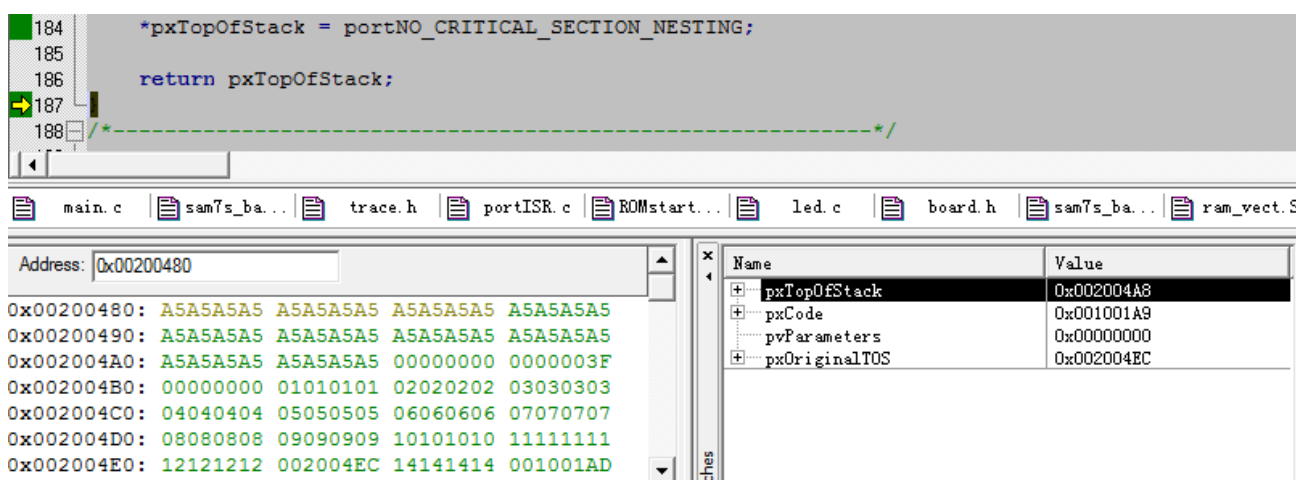
全速执行到断点位置，从变量窗口可以看到当前的值：



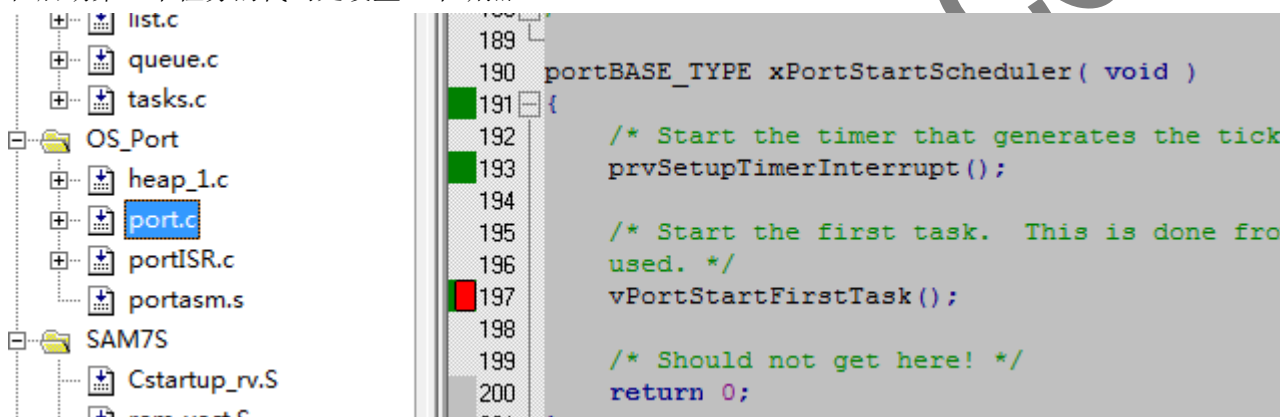
打开 mem 窗口，并查看此位置：



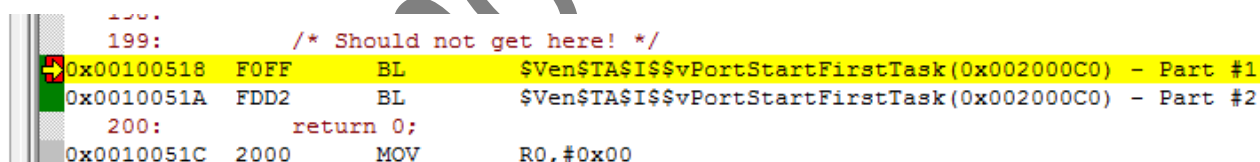
按 F11 单步执行到函数结束，可以看到系统构造的任务栈及新的栈位置：



在启动第一个任务的代码处设置一个断点：



直接运行到此处并打开反汇编模式：



可以看到该文件被编译为 thumb 代码。

按 F11 单步跟入，系统进入启动第一个任务的代码：

```

MAN2003_SAM7S_base_freertos\OS_Port\portasm.s

ash Peripherals Tools SVCS Window Help

063
064 /*-----
065 ;      vPortStartFirstTask
066 ;      Start the first ready task
067 /*-----
068 portRESTORE_CONTEXT
069 vPortStartFirstTask
070 RUN_NEXT_TASK
071 ; make sure this function is called in SVC
072 ; 1. Set the LR to the task stack
073     LDR    R0, =pxCurrentTCB
074     LDR    R0, [R0]
075     LDR    LR, [R0] ; First Item is pxTopOfStack
076
077 ;      The critical nesting depth is the first item on the stack.
078 ; 2. Load it into the ulCriticalNesting variable.
079     LDR    R0, =ulCriticalNesting
080     LDMFD  LR!, {R1}
081     STR    R1, [R0]
082
083 ; 3. Get the SPSR from the stack.
084     LDMFD  LR!, {R0}
085     MSR    SPSR_cxsf, R0

```

单步运行到如下位置：

R3	0x00100519	068 portRESTORE_CONTEXT
R4	0x00000001	069 vPortStartFirstTask
R5	0x0020223c	070 RUN_NEXT_TASK
R6	0x00000000	071 ; make sure this function is called in SVC mode
R7	0x00000000	072 ; 1. Set the LR to the task stack
R8	0x00000000	073 LDR R0, =pxCurrentTCB
R9	0x00000000	074 LDR R0, [R0]
R10	0x00000000	075 LDR LR, [R0] ; First Item is pxTopOfStack
R11	0x00000000	076
R12	0x00100733	077 ; The critical nesting depth is the first item on the stack.
R13 (SP)	0x00203fc8	078 ; 2. Load it into the ulCriticalNesting variable.
R14 (LR)	0x002004a8	079 LDR R0, =ulCriticalNesting
R15 (PC)	0x002000d0	080 LDMFD LR!, {R1}
CPSR	0x400000d3	081 STR R1, [R0]
SPSR	0x00000000	

注意此时的处理器模式为 SVC，而 LR 的值，正是前面构建的任务的 stack 值。

单步运行到函数末，可以看到寄存器的恢复情况：

Register	Value
R0	0x00000000
R1	0x01010101
R2	0x02020202
R3	0x03030303
R4	0x04040404
R5	0x05050505
R6	0x06060606
R7	0x07070707
R8	0x08080808
R9	0x09090909
R10	0x10101010
R11	0x11111111
R12	0x12121212
R13 (SP)	0x00203fc8
R14 (LR)	0x001001ad
R15 (PC)	0x002000f0
CPSR	0x400000d3
SPSR	0x0000003f

```

077 ; The critical nesting depth is the first item
078 ; 2. Load it into the ulCriticalNesting variable.
079 LDR R0, =ulCriticalNesting
080 LDMFD LR!, {R1}
081 STR R1, [R0]
082
083 ; 3. Get the SPSR from the stack.
084 LDMFD LR!, {R0}
085 MSR SPSR_cxsf, R0
086
087 ; 4. Restore all system mode registers for the task
088 LDMFD LR, {R0-R14}^ ; in SVC mode, ^ means
089 NOP
090
091 ; 5. Restore the return address (the task start address)
092 LDR LR, [LR, #60]
093
094 ; 6. Start the task
095 ; LR - 4 is the real address
096 ; SUB[S] means store SPSR to CPSR, from SVC to
097 SUBS PC, LR, #4
098
099 /*-----
100 ; vPortYieldProcessor
101 ; TASK level task switch function
102 ; Implement via ARM SWI (task run in SYS mode)
103 */

```

再次按下 F11，程序进入第一个就绪的任务运行：

Register	Value
R0	0x00000000
R1	0x01010101
R2	0x02020202
R3	0x03030303
R4	0x04040404
R5	0x05050505
R6	0x06060606
R7	0x07070707
R8	0x08080808
R9	0x09090909
R10	0x10101010
R11	0x11111111
R12	0x12121212
R13 (SP)	0x002004ec
R14 (LR)	0x14141414
R15 (PC)	0x001001a8
CPSR	0x0000003f
SPSR	0x0000003f

```

11 #include "sempr.h"
12
13 #define ledtask_PRIORITY
14 #define ledtask_TASK_STACK
15
16 void vLEDTask( void *pvParameters )
17 {
18     pvParameters = pvParameters;
19
20     LED_Configure(0);
21     LED_Configure(1);
22     LED_Configure(2);
23
24     while(1)
25     {
26         LED_Set(0);
27         vTaskDelay(250);
28         LED_Set(1);
29         vTaskDelay(250);
30         LED_Set(2);
31         vTaskDelay(250);
32
33         LED_Clear(0);
34         LED_Clear(1);
35         LED_Clear(2);
36         vTaskDelay(248);
37     }

```

此时的处理模式为 system，所以的寄存器均为任务创建时的状态。