

使用 Trace32 调试 SAM9 SDRAM 代码

文档编号	MAN3009A_CH				
文档版本	Rev. A				
文档摘要	描述了使用 Trace32 来调试运行于 SDRAM 的 SAM9 代码的方法和过程				
关键词	Trace32 SAM9 SDRAM 程序				
创建日期	2010-01-19	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	2010-01-19	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: +86-571-89908193
支持 TEL: 18913989166 13770507096
FAX: +86-571-89908193
www.mcuzone.com www.atarm.com

1.概述

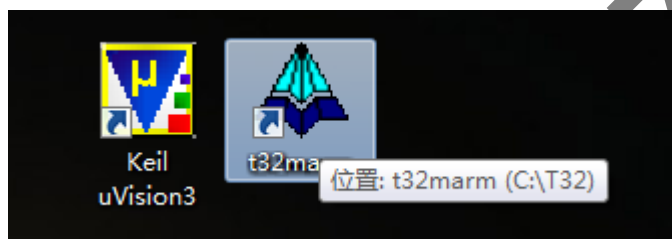
Trace32 ARM ICD(以下简称 T32)是一款高性能的 ARM 仿真器，自带调试软件环境，功能很强大。在裸奔的情况下，SAM9 大部分用户的代码都将运行于 SDRAM，因为内部 RAM 空间有限。而代码运行于 SDRAM 时，调试就有了其特殊性。但是如果拥有了 T32，那么一切就可以迎刃而解。本文介绍使用 T32 调试运行于 SAM9261S SDRAM 中的 ucos-II 的代码的基本过程。Ucos-II 使用 keil 编译，keil 版本是 3.80a。

2. 安装 T32 调试软件

2.1 安装软件

使用 T32 提供的安装光盘安装 T32 调试软件。
安装完成后连接上 T32 的硬件，然后安装驱动。

注意：如果是 vista 以上(含)的系统，需要安装特殊的驱动，与 XP 不同。
然后在桌面创建 T32 的快捷方式：



3. 编译 Bootstrap

3.1 修改代码

需要编译 Bootstrap 的原因有两点：一是需要使用编译生成的 bin 文件来加载正式的 SDRAM 代码，具体原因可以参考本站的 [MAN3008](#)，二是编译生成的 elf 文件可以用于 T32 初始化 SAM9261S 的外部 SDRAM，然后再加载 ucos-II 的 axf 文件，因为 SDRAM 在初始化之前是无法使用的。

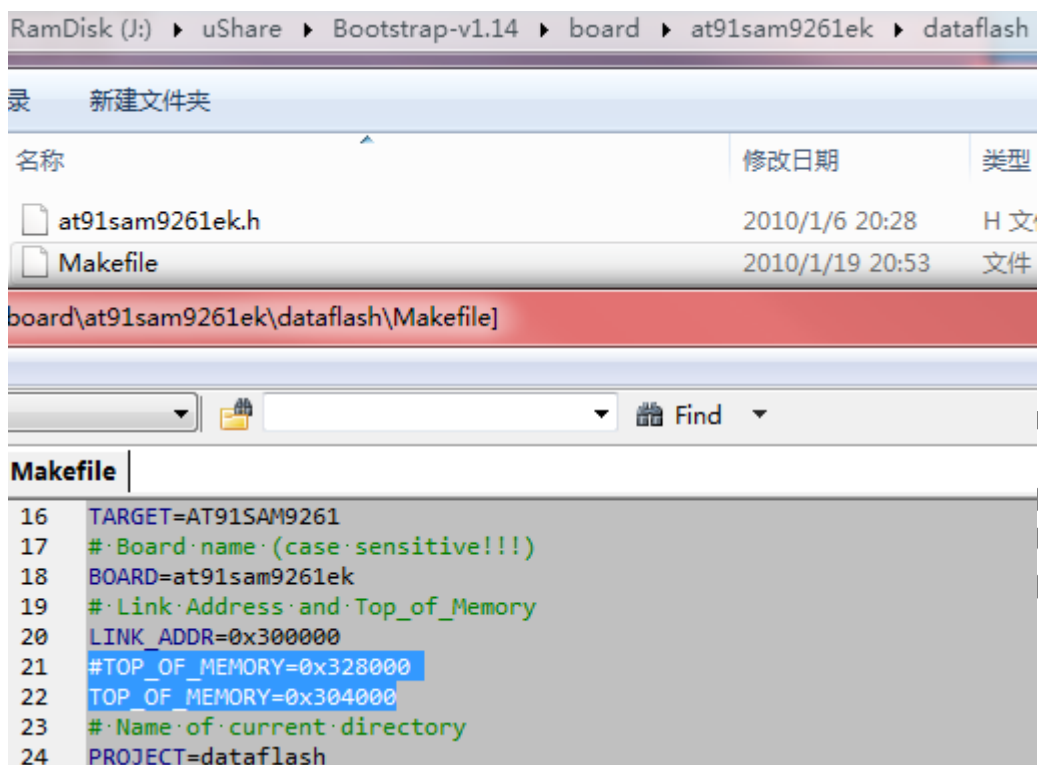
ATMEL 原始提供的 Bootstrap 源码是为 SAM9261 编写的，并不适合 SAM9261S，需要修改。

修改加载参数，ucos-II 的代码将被烧写到 data flash 的 0x8400 处，并被加载到 SDRAM 的 0x20000000 处运行：

```
#define IMG_ADDRESS → → 0x8400 → → /* Image Address in DataFlash */
#define IMG_SIZE → → 0x40000 → → /* Image Size in DataFlash */

#define MACH_TYPE → 0x350 → /* AT91SAM9261-EK */
#define JUMP_ADDR → 0x20000000 → /* Final Jump Address → */
```

修改 Makefile 中的栈顶的定义，因为 SAM9261S 只有 16KB 的内部 RAM：



也可以在头文件中打开 debug 的支持，添加一些用户定义的字符串。

3.2 编译代码

运行 make 即可：

```
$ make CROSS_COMPILE=arm-none-eabi-
```

这里使用的是 arm-none-eabi- 的工具链。

编译生成的文件：

```

4516 Jan 19 20:58 dataflash_at91sam9261ek.bin
25330 Jan 19 20:58 dataflash_at91sam9261ek.elf
15881 Jan 19 20:58 dataflash_at91sam9261ek.map
  
```

代码的 mem 消耗的信息：

```

$ arm-none-eabi-size dataflash_at91sam9261ek.elf
text    data    bss     dec     hex filename
4516      0      0    4516    11a4 dataflash_at91sam9261ek.elf
  
```

生成的 bin 文件用于烧写，elf 文件用于加载调试。



4. ucos-II 代码

4.1 编译代码

ucos-II 的代码使用 keil MDK 编译，代码的 link 地址必须在 0x2000-0000，与前面 Bootstrap 指定的一致。同时需要设定好 ram 中的异常向量。

对于 SAM9261S SDRAM 这样的应用，建议用 scatter loader file 来控制应用的 mem map。

编译完成后生成 axf 文件如下：

 ucos2_sdram.axf	AXF 文件	360 KB
 ucos2_sdram.bin	BIN 文件	27 KB

生成的 axf 文件用于调试，bin 文件用于烧写。

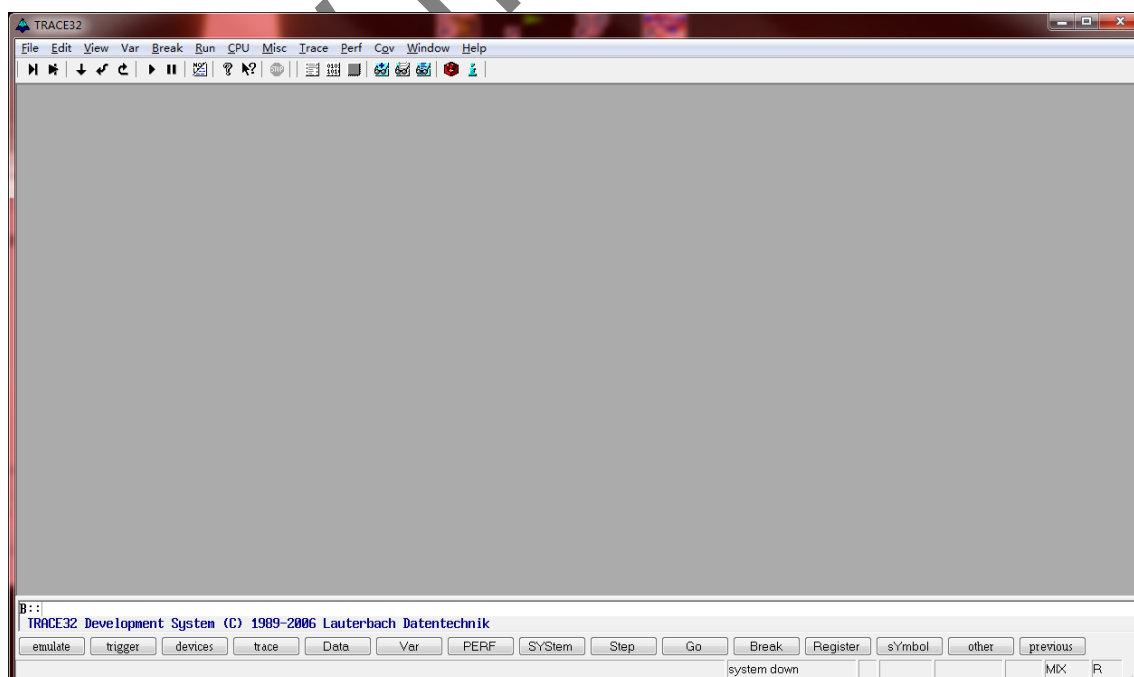
5. 调试代码

5.1 初始化环境

首先将 SAM9261S 板子上电，将 dbgu 连接到 PC，开启终端软件，并使板子进入 SAM boot 模式，终端收到如下信息：

```
RomBOOT
>
```

连接好 T32，包括板子侧的 20pin 电缆和 PC 侧的 USB 电缆，运行 t32 的 IDE：



在底端的命令行依次输入如下命令：

```
B::sys.cpu arm926ej  
TRACE32 Development System (C) 1989-2006 Lauterbach Datentechnik
```

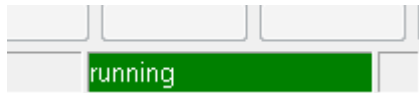
指定处理器。

```
B::sys.jtagclock 5MHz  
TRACE32 Development System (C)
```

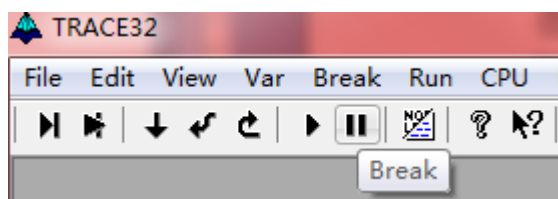
指定 jtag 时钟频率。

```
B::sys.mode attach
```

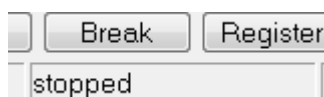
指定连接方式并回车后，会发现处理器处于运行状态：



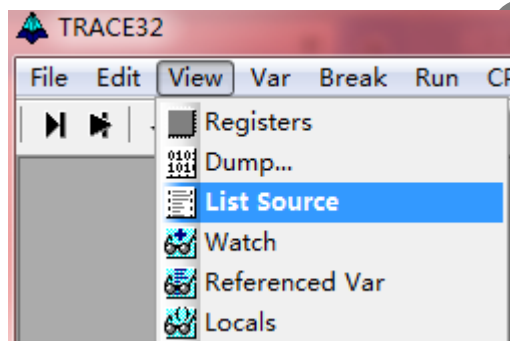
点击暂停按钮：



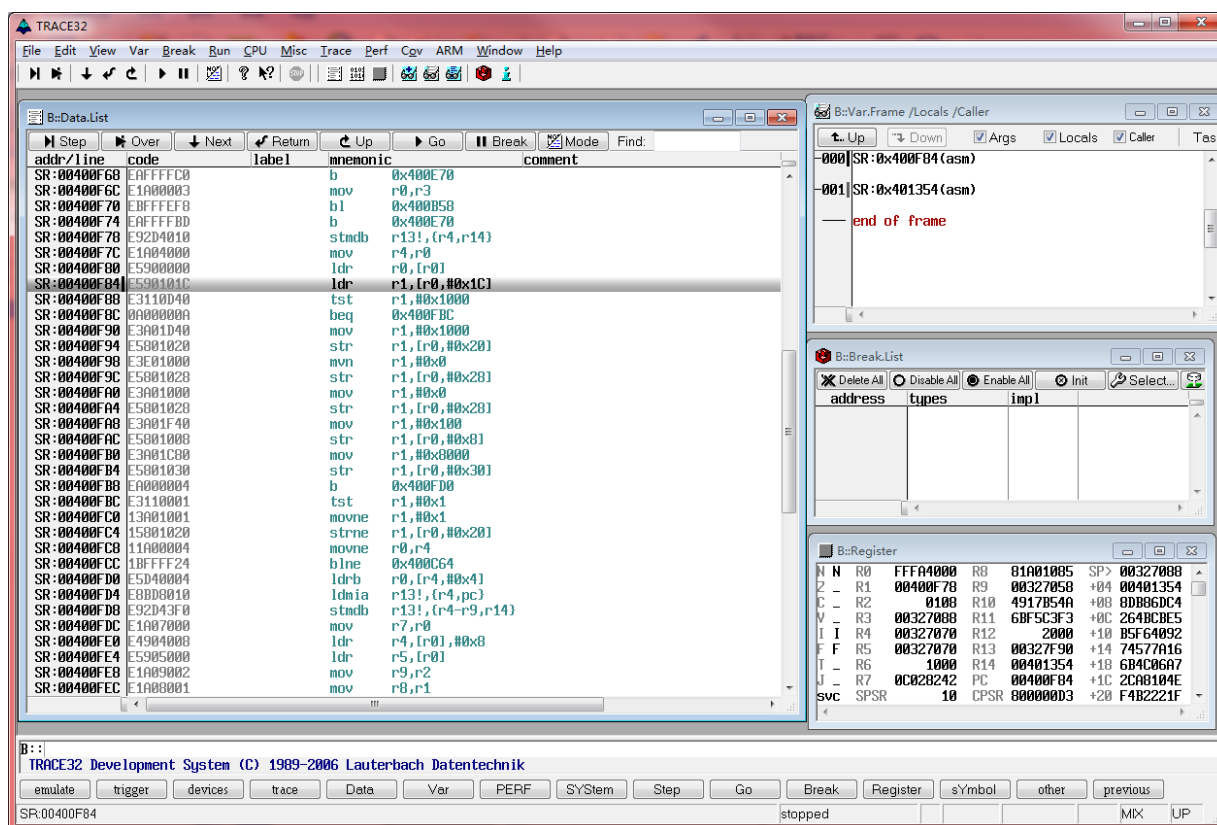
使得处理器暂停：



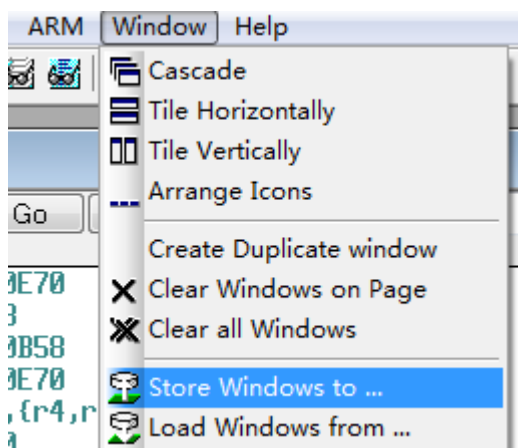
此时打开相关的窗口，比如 source， register：



并将其合理排列一下：

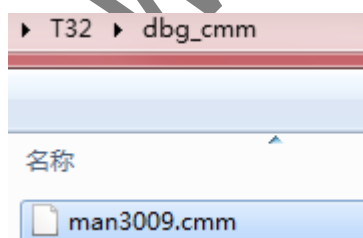


排列好后可以将当前的设置保存到文件：



下次可以直接 load 这个 window 的设置从而避免重复设置。

比如 save 到这个文件：



下次在处理器停止时，直接使用下面的命令就可以加载窗口设置：

```
B::do dbg_cmm\man3009.cmm
TRACE32 Development System (C)
```

使用 Trace32 调试 SAM9 SDRAM 代码

MAN3009A

5.2 加载 Bootstrap

使用命令加载 Bootstrap 的 elf 文件：

```
B::d.load.elf j:\uShare\Bootstrap-v1.14\board\at91sam9261ek\dataflash\dataflash_at91sam9261ek.elf
```

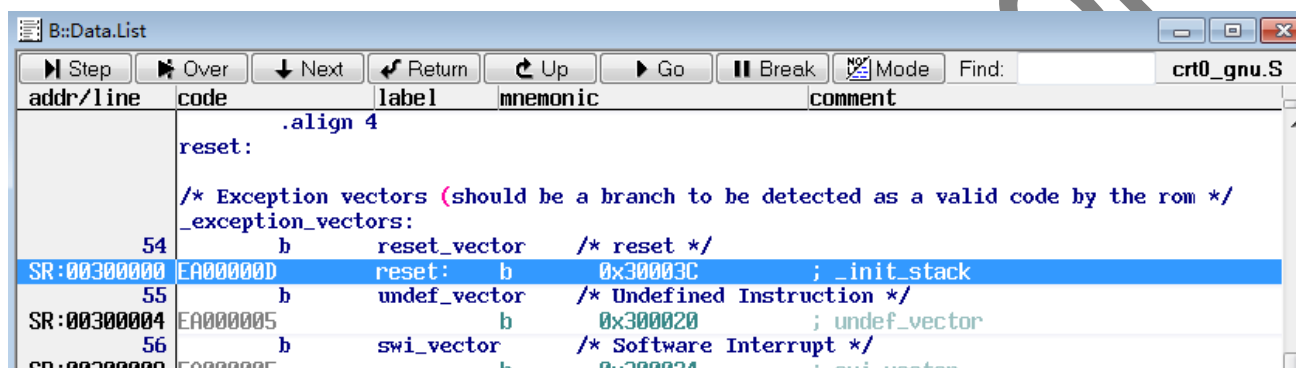
文件路径可以粘贴进去，也可以凭记忆输入，支持 tab 键自动补全。

敲回车可以看到 elf 被加载，并进入 debug 状态。

如果找不到 source 文件，可以用下面命令指定搜索路径：

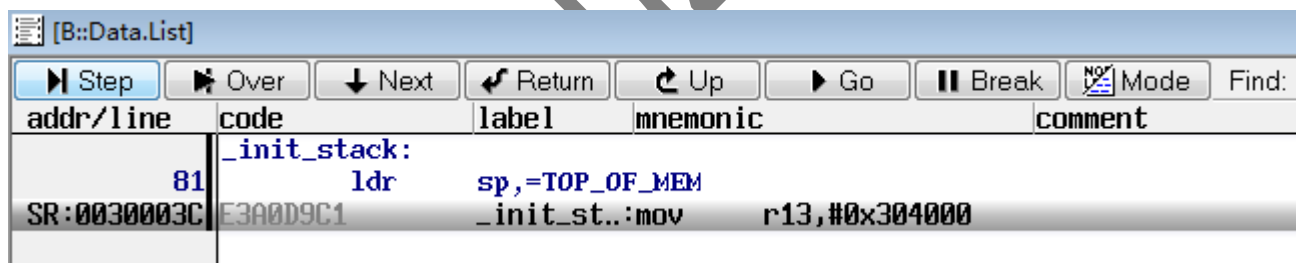
```
B::SYMBOL.SPATH j:\uShare\Bootstrap-v1.14\
```

代码合一的效果：

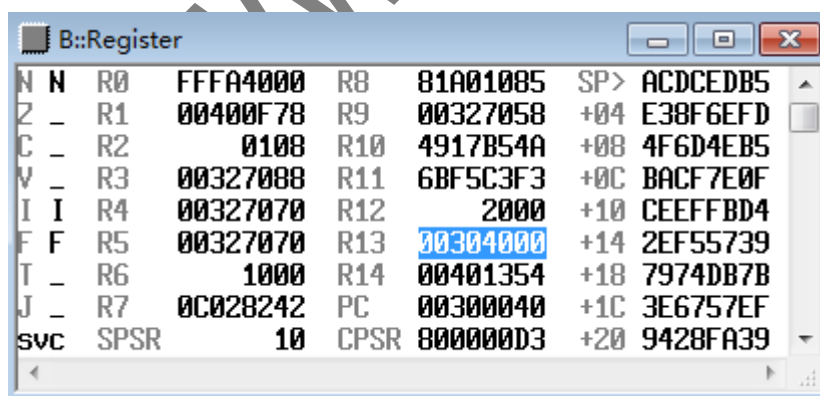


使用 mode 按钮可以转换为源文件和反汇编两种方式，上图显示的是反汇编方式，也可以方便的查看源码。

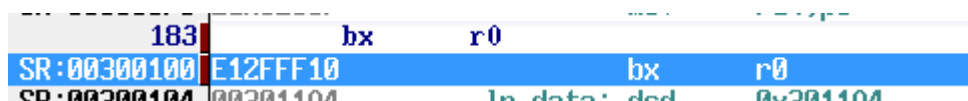
使用上方的按钮可以控制程序的运行：



同时可以观察寄存器窗口的变化：



由于加载 Bootstrap 的目的只是为了初始化 SDRAM，所以直接在程序最后设置一个断点：



使用 Trace32 调试 SAM9 SDRAM 代码

MAN3009A

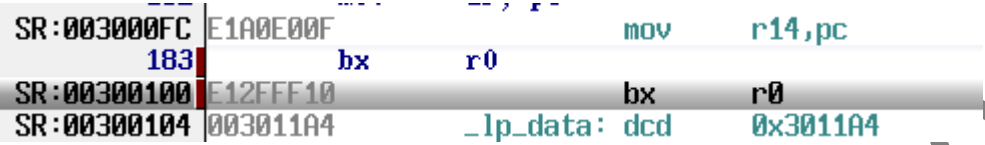
设置断点的方法是直接双击要设置的程序位置即可，设置完成后在断点窗口里也可以看见：



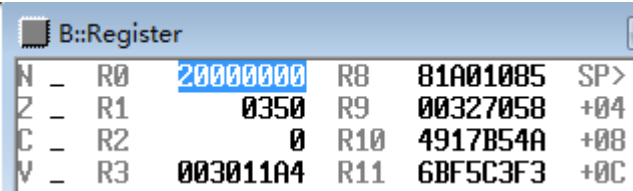
点击程序控制里的 Go 按钮，



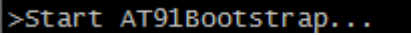
程序停在断点处：



灰条即是当前 PC 的位置，现在寄存器显示程序将跳转到 0x2000-0000 处运行：



PC 上的终端串口也会有 Bootstrap 的 debug 输出：



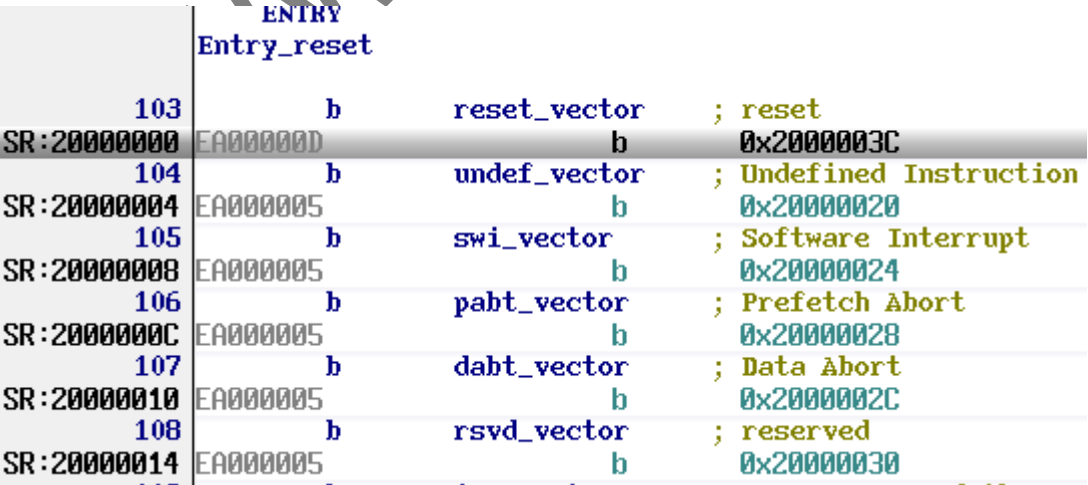
此时，SDRAM 已经初始化就绪。

5.3 加载 ucos-II

在前面的 Bootstrap 在断点处停下后，直接使用相同的命令再次加载 ucos-II 的 axf 文件：



加载的效果：



PC 指针位于 0x2000-0000 处。

源代码方式:

addr/line	source		
	Entry_reset		
103	b	reset_vector	; reset
104	b	undef_vector	; Undefined Instruction
105	b	swi_vector	; Software Interrupt
106	b	paht_vector	; Prefetch Abort
107	b	daht_vector	; Data Abort
108	b	rsvd_vector	; reserved
109	b	irq_vector	; IRQ : read the AIC
110	b	fiq_vector	; FIQ
	undef_vector		

此时可以使用程序控制命令控制程序的运行:

		; Init the stack	
		_init_stack	
130	ldr	r0,	=TOP_OF_STACK ; defined in Makefile
		; - Set up Fast Interrupt Mode and set FIQ Mode Stack	
133	msr	CPSR_c,	#(ARM_MODE_FIQ I_BIT F_BIT)
134	mov	r13,	r0 ; Init stack FIQ
135	sub	r0,	r0, #FIQ_STACK_SIZE
		; - Set up Interrupt Mode and set IRQ Mode Stack	
130	msr	CPSR_c,	#(ARM_MODE_IRQ I_BIT F_BIT)

5.4 设置断点

虽然可以直接找到代码所在，然后双击打断点。但是当工程中包含的文件很多时，找代码也是麻烦。在 T32 的调试环境下可以很方便的设置断点，只要知道公有函数的名字。

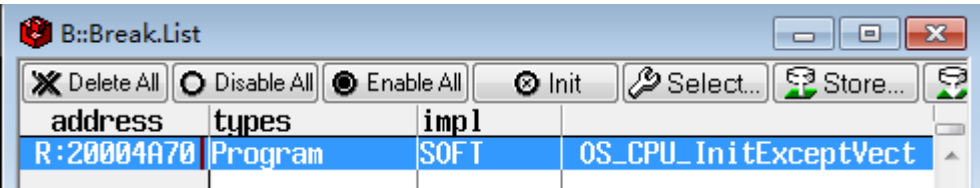
比如，ucos-II 的代码中有一个在 RAM 中安装异常向量的函数：

```
51 ; IRAM must remap to 0 for ARM execeptions
52 ER_IRAM_VECTOR 0x000000 EMPTY 0x100
53 {
54 ; OS_CPU_InitExceptVect() will install ISR here
55 }
56
```

直接在 T32 环境的命令行中输入：

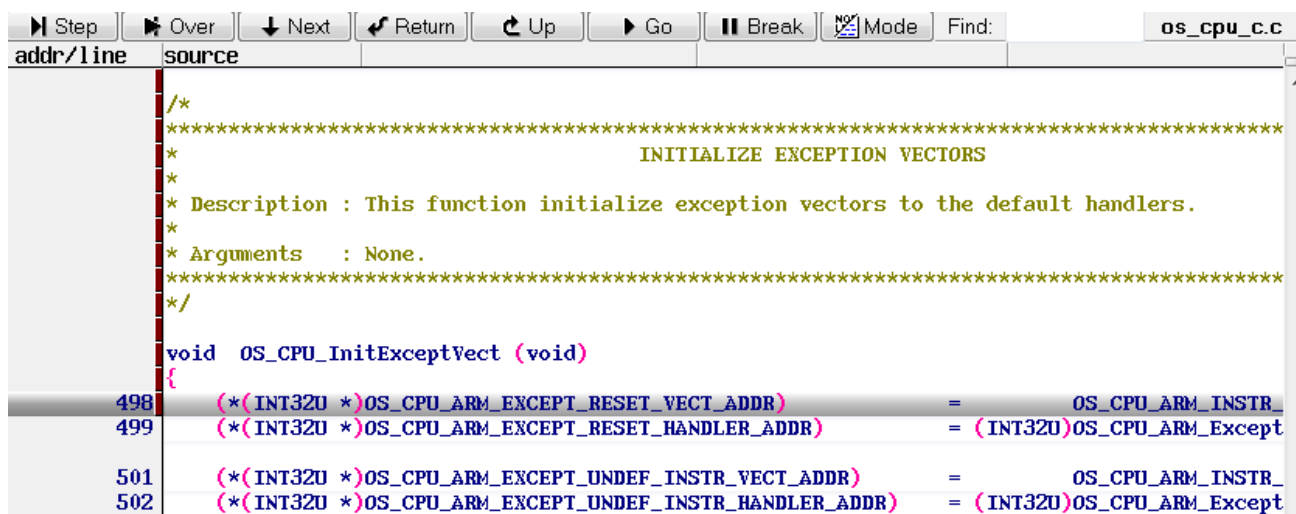
```
B::b.s OS_CPU_InitExceptVect
```

输入过程中可以使用 tab 键自动补全。输入完成后回车，即可在那个函数处设置一个断点：

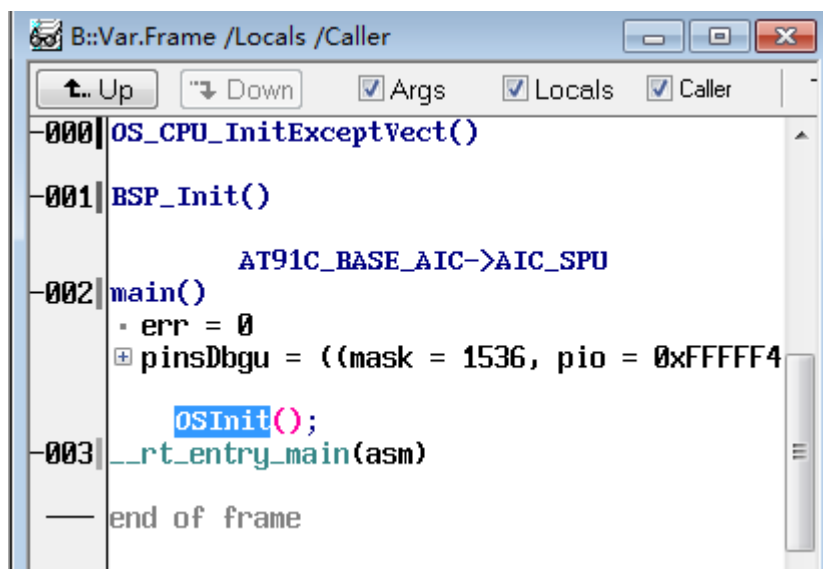


选择 Store...可以将当前的断点设置保存到文件，而使用 Load...可以加载以前的断点设置文件，可以直接使用之前的断点设置，继续调试。

按运行，看到代码停在断点处：



此时从 back trace 窗口可以看到代码执行的轨迹：



这个图中保留了当前函数的运行轨迹，可以看出当前代码调用过程(从下往上，与压栈顺序一致)。串口中也有相应的输出：

```

*****
AT91SAM9261 uC/OS-II Demo
uC/OS-II Version: 2.86
www.mcuzone.com
Compiled by ARM RealView Tool Chain.
-- AT91 LIB version: 1.5 --
-- AT91SAM9261-EK
-- Compiled: Jan 18 2010 20:13:37 --
*****

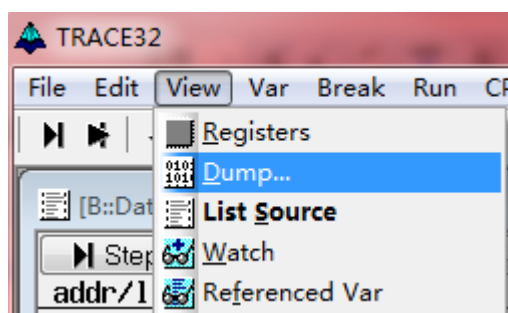
```

5.5 查看内存

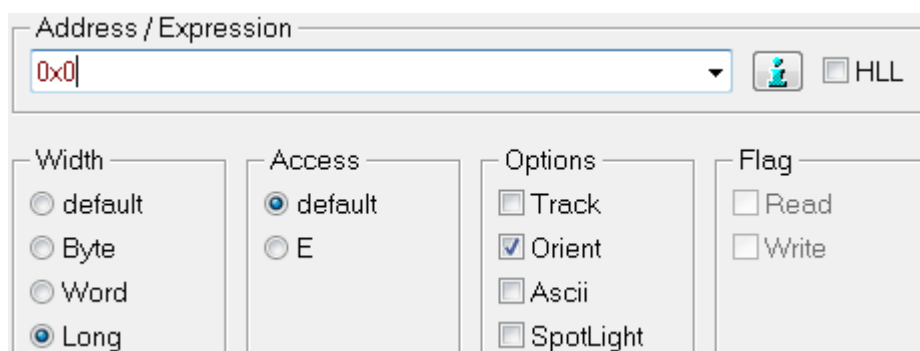
在前面的代码处停下来后，我们可以打开内存查看窗口，看看异常向量的安装。

```
B::d.dump 0x0 /NOASCII
```

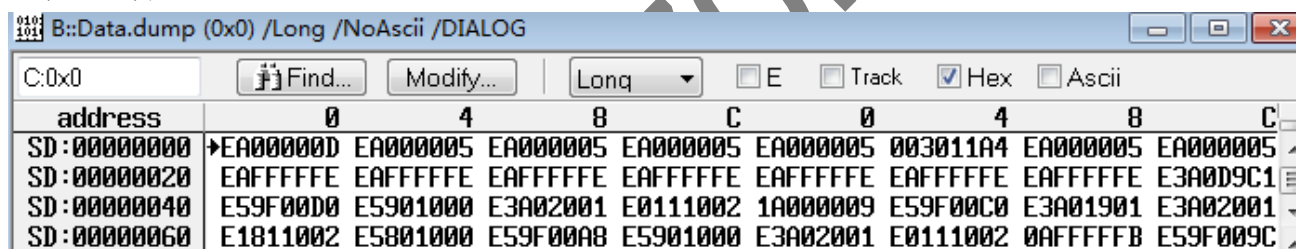
回车后会打开内存查看窗口。也可以使用菜单提供的功能：



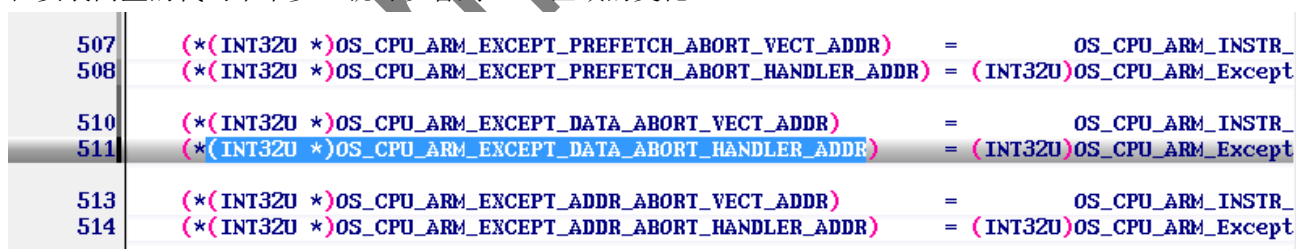
设置参看方式，由于是代码不需要 ascii 的显示，同时显示为 Long 类型，因为 ARM 指令是 32 位：



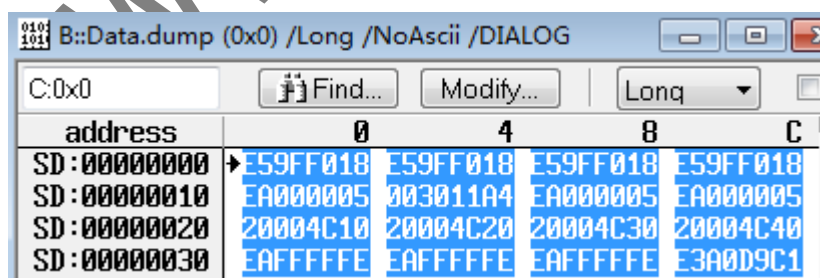
显示 data 窗口：



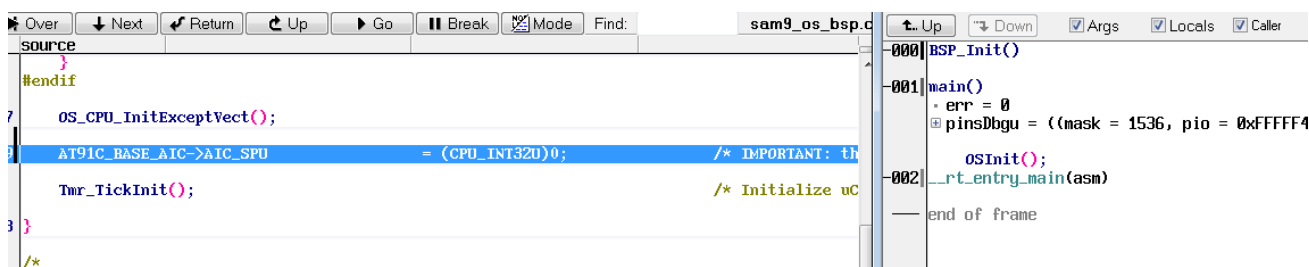
在安装向量的代码中单步，就可以看到 data 区域的变化：



安装的向量：



一直单步到该函数结束，可以看到代码的执行位置：



此时还处于系统初始化的阶段。

5.6 调试 OS 启动

在 ucos-II 中，多任务开始于 OSStart()调用之后：

```

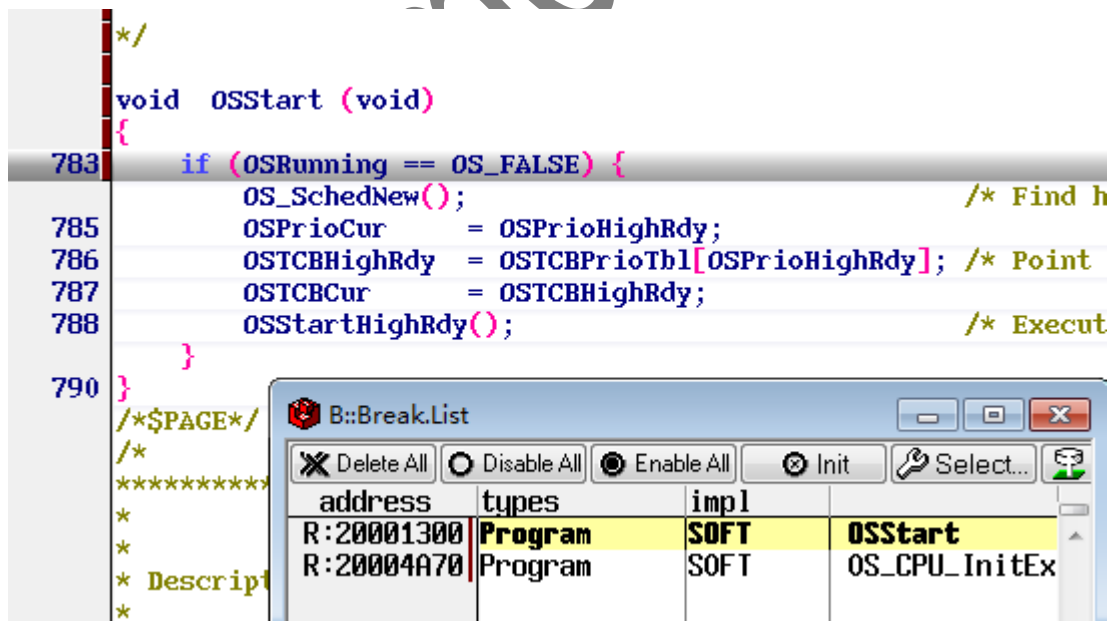
151 #if OS_TASK_NAME_SIZE > 11
152     OSTaskNameSet (APP_TASK_START_PRIO + 1, "US0 Task", &err);
153 #endif
154
155     OSStart();
156
157     return 0;
158 }

```

直接在该函数处打个断点：

B::b.s OSStart

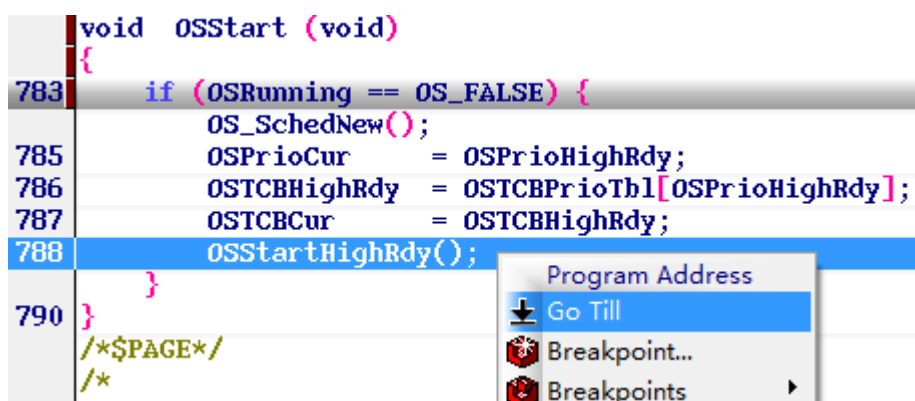
全速执行到此处：



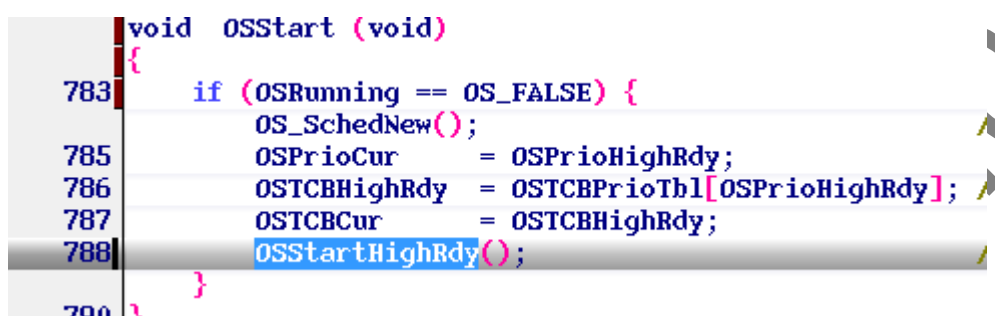
注意，断点窗口显示的断点命中状态。

这个函数计算当前系统最高优先级的任务，然后调用 OSStartHighRdy()真正的启动这个任务。

高亮调用 OSStartHighRdy()的那行，右键选择“Go Till”，



程序停在调用处：



打开反汇编模式：

SR:20001340	E5810028	str	r0,[r1,#0x28]	
SR:20001344	E5810020	str	r0,[r1,#0x20]	
788			OSStartHighRdy();	/* Execute target specific code to st
SR:20001348	EA000DF8	b	0x20004B30	; OSStartHighRdy
			/*\$PAGE*/	

单步跟入：

addr/line	code	label	mnemonic	comment
	***** ; ; START MULTITASKING ; void OSStartHighRdy(void) ; ; Note(s) : 1) OSStartHighRdy() MUST: ; a) Call OSTaskSwHook() then, ; b) Set OSRunning to TRUE, ; c) Switch to the highest priority task. ; ; *****			
				OSStartHighRdy
				; Change to SVC mode.
156	MSR	CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS OS_CPU_ARM_MODE_SVC)		
SR:20004B30	E321F0D3	OSStartHighRdy:	msr	cpsr_c,#0xD3
158	LDR	R0, _OS_TaskSwHook		; OSTaskSwHook();
SR:20004B34	E59F0258		ldr	r0,0x20004D94
159	MOV	LR, PC		
SR:20004B38	E1A0E00F		mov	r14,pc
160	BX	R0		
SR:20004B3C	F12FFF10		bx	r0

如果不习惯反汇编模式，可以直接看源码：

```

;*****
;
;                                START MULTITASKING
;                                void OSStartHighRdy(void)
;
; Note(s) : 1) OSStartHighRdy() MUST:
;             a) Call OSTaskSwHook() then,
;             b) Set OSRunning to TRUE,
;             c) Switch to the highest priority task.
;*****
OSStartHighRdy
; Change to SVC mode.
156  MSR    CPSR_c, #(OS_CPU_ARM_CONTROL_INT_DIS | OS_CPU_ARM_MODE_SVC)
; OSTaskSwHook();
158  LDR    R0, _OS_TaskSwHook
159  MOV    LR, PC
160  BX     R0
; OSRunning = TRUE;
162  LDR    R0, _OS_Running
163  MOV    R1, #1
164  STRB   R1, [R0]

```

单步跟踪这段代码，可以看到任务是如何被启动的。

在执行恢复寄存器之前停下：

```

171  LDR    R0, [SP], #4
172  MSR    SPSR_cxsf, R0
; LDMFD SP!, {R0-R12, LR, PC}^
175  LDMFD SP!, {R0-R12, LR}
176  LDMFD SP!, {PC}^

```

注意此时的寄存器状态：

B::Register									
N	-	R0	13	R8	0	SP>	00000000		
Z	Z	R1	1	R9	00327058	+04	01010101		
C	C	R2	208015D8	R10	1	+08	02020202		
V	-	R3	60	R11	0	+0C	03030303		
I	I	R4	20005FC4	R12	0	+10	04040404		
F	F	R5	208026BC	R13	20800834	+14	05050505		
T	-	R6	0	R14	20004B40	+18	06060606		
J	-	R7	0	PC	20004B60	+1C	07070707		
SVC		SPSR	13	CPSR	600000D3	+20	08080808		
Q	-					+24	09090909		
		USR:		FIQ:		+28	10101010		

特别要注意其中的 SP，栈中的数据就是马上要恢复的数据。

按下单步恢复寄存器：

```

; LDMFD SP!, {R0-R12, LR, PC}^
175  LDMFD SP!, {R0-R12, LR}
176  LDMFD SP!, {PC}^

```


恢复出的寄存器:

B::Register					
N	-	R0	0	R8	00000000 SP> 20000220
Z	Z	R1	01010101	R9	09090909 +04 00000000
C	C	R2	02020202	R10	10101010 +08 00000000
V	-	R3	03030303	R11	11111111 +0C 00000000
I	I	R4	04040404	R12	12121212 +10 00000000
F	F	R5	05050505	R13	2000086C +14 00000000
T	-	R6	06060606	R14	14141414 +18 00000000
J	-	R7	07070707	PC	20004B64 +1C 00000000
SVC		SPSR	13	CPSR	600000D3 +20 00000000
Q	-				+24 00000000
		USR:		FIQ:	+28 00000000

再执行一次单步，进入第一个任务:

```
ist]
[Over] [Next] [Return] [Up] [Go] [Break] [Mode] Find: main.c
Source
OS_STK stk_MainTask[APP_TASK_START_STK_SIZE] = {0};
OS_STK stk_US0Task[APP_TASK_START_STK_SIZE] = {0};

static void MainTask(void *p_arg)
24 {
    p_arg = p_arg;
27     printf("\n\rSYS info:\n\r");
29     printf("PCK = %10dHz\n\r", BOARD_MCK * 2);
31     printf("MCK = %10dHz\n\r", BOARD_MCK);
33     if (CP15_Is_I_CacheEnabled())
34         printf("--- I Cache Enabled --\n\r");
36     else
        printf("--- I Cache Disabled --\n\r");
```

单步运行一些代码:

```
41     printf("--- D Cache Disabled --\n\r");
43     if (CP15_Is_MMUEnabled())
44         printf("--- MMU Enabled --\n\r");
46     else
        printf("--- MMU Disabled --\n\r");
48     printf(BANNER);
50     LED_Configure(1);

#if OS_TASK_STAT_EN > 0
53     OSStatInit();
54     printf("OSIdleCtrMax = %10d\n\r", OSIdleCtrMax);
#endif
```


串口中的相应输出：

```
SYS info:
PCK = 198656000Hz
MCK = 99328000Hz
-- I Cache Enabled --
-- D Cache Disabled --
-- MMU Disabled --
*****
```

当前还在任务 MainTask()中，直接在另一个任务 US0Task()中打个断点：

```
static void US0Task(void *p_arg)
68 {
    // USART0 TXD pin definition.
    #define PIN_TXD_0    {1 << 8, AT91C_BASE_PIOC, AT91C_ID_PIOC, PIO_PERIPH_A, PIO_DEFAULT}
    // USART0 RXD pin definition.
    #define PIN_RXD_0    {1 << 9, AT91C_BASE_PIOC, AT91C_ID_PIOC, PIO_PERIPH_A, PIO_DEFAULT}

    static const Pin pinsUSART0[] = {PIN_TXD_0, PIN_RXD_0};

    p_arg = p_arg;

    // init srand
79    srand(RTT_GetTime(AT91C_BASE_RTTC));

81    LED_Configure(2);
```

直接 Go，停在了断点处，也就是第二个任务也被调度运行：

```
static void US0Task(void *p_arg)
68 {
    // USART0 TXD pin definition.
    #define PIN_TXD_0    {1 << 8, AT91C_BASE_PIOC, AT91C_ID_PIOC, PIO_PERIPH_A, PIO_DEFAULT}
    // USART0 RXD pin definition.
    #define PIN_RXD_0    {1 << 9, AT91C_BASE_PIOC, AT91C_ID_PIOC, PIO_PERIPH_A, PIO_DEFAULT}

    static const Pin pinsUSART0[] = {PIN_TXD_0, PIN_RXD_0};

    p_arg = p_arg;

    // init srand
79    srand(RTT_GetTime(AT91C_BASE_RTTC));

81    LED_Configure(2);
```

注意此时 PC 的值，

F	-	R5	05050505	R13	20801068	+0C	00000000
T	-	R6	06060606	R14	14141414	+10	4F000000
J	-	R7	07070707	PC	20000404	+14	6D542D53

位于 SDRAM 中。

一般的调试过程就是断点，运行，单步的过程，根据寄存器，内存的数据判断程序的状态，排除可能存在的问题。