SAM926x 裸奔指南

文档编号	MAN3008A_CH				
文档版本	Rev. A				
文档摘要	系统阐述了基于 SAM926x 构建裸奔(非 Linux，非 WinCE 等 OS)系统的各个方面				
关键词	SAM926x 裸奔 软硬件设计				
创建日期	2009-12-22	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	2009-12-22	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 13957118045
FAX：+86-571-89908193
www.mcuzone.com www.atarm.com

1.概述

SAM926x 基于 ARM v5TEJ 的 ARM926ej-s 内核，在一般的开发中，这类内核比较适合使用 Linux 或者 WinCE 这类的 OS，可以发挥硬件的性能。但是实际使用中，还是有各种各样的原因，使得开发者不能使用或者不愿意使用这类的 OS，比如从 ARM7 一路裸奔过来的开发者，或者由于以前产品的芯片停产，为了性能而转到 SAM926x 的用户，为了产品的一致性而移植裸奔代码。
本站的 SAM926x 的 Linux 与 WinCE 资料很多，可以供相关用户参考。而本文是为了弥补 SAM926x 裸奔这个话题的空缺。
本文中的裸奔，是指不使用 Linux，WinCE 之类的大型 OS，而不排斥 uC/OS-II, FreeRTOS 之类的实时内核。

2. 认识裸奔

2.1 裸奔的定义

软件的“裸奔”可以认为直接从硬件层面开始编程，而没有像 Linux 或者 WinCE 那样的复杂开发环境。编译输出的二进制代码直接从板子上运行，而不是依赖于一个 OS 所提供的运行时环境，比如 XP 下的.exe 需要 XP 本身的运行。从这个意义上讲，uC/OS-II 与 FreeRTOS 之类的实时内核也在本文的讨论范围之内，因为这些实时内核与应用本身不能割裂，编译为一个执行文件。

2.2 裸奔的优势

在裸奔的情况下，程序员可以控制系统的所有资源，这意味着系统有很好的透明度。同时，由于可以直接访问硬件资源，往往意味着更好的系统性能。再者，不使用大型 OS，对程序员的要求有所降低(不意味着水平要求的降低)。

2.3 裸奔的代价

在裸奔的情况下，程序员可以控制系统的所有资源，系统的稳定性基本取决于程序员。同时，部分增加了软件维护的难度和移植的难度，因为这样的代码很可能与硬件耦合得过于紧密。再者，Linux 与 WinCE 上大量现成的程序很难借鉴，比如，Linux 内核可以包含稳定可靠的网络协议，而裸奔的话就只能自定义网络协议栈，工作量增加，测试和老化也需要不少时间。

3. 硬件系统

3.1 电源，时钟

这部分应当严格参考 ATMEL 官方的 EK，才能保证系统的可靠运行。由于芯片的电流消耗可观，应尽量选用 DC-DC 芯片为核心供电。
时钟应该使用官方推荐的时钟，这样的好处是软件系统中时钟修改比较小。

3.2 启动媒介

SAM926x 支持多种启动(boot)方式，支持从多种媒介(media)上启动，比如 nor flash，data flash，NAND，SD 卡。但是除了 nor flash，其它的方式都依赖于片上的 BOOTROM，该 BOOTROM 的功能和性能和芯片的版本密切相关。所以在设计之前，应该检索芯片数据手册，查看当前芯片版本和启动的限制，避免定错芯片。
从外部 nor flash 启动是不需要依赖于 BOOTROM 的，但是对 nor flash 有一些限制，最好选用 AT49BV 系列。而且从外部 nor flash 启动时，系统是完全没有初始化的，因此用户代码相当于完全从 0 开始，好处是和 ARM7 启动流程一样，缺点是所有的初始化都得写一遍，而且还得考虑 remap 的问题。
片外的 nor flash 上的代码，支持 XIP，也就是代码存储的空间可以说完全在 nor flash 上，能节省 RAM(泛指，可以是 SRAM，也可以是 SDRAM，下同)，但是低的总线位宽和较慢的访问速度将会是设计中的问题。相对的，使用 BOOTROM 启动，有 ATMEL 现成的方案支持(SAM-BA 软件和 Bootstrap 软件包)，可以说都是现成的，缺点是 data flash 与 NAND 都没有 XIP 的能力，代码必须 copy 到 RAM 中才能运行，这样会增加 RAM 的消耗。
从 layout 工程师角度看，nor flash 体积大，引脚多，而且都是并行的数据和地址信号，有一定的布线要求，而 data flash 体积小，引脚少，串行方式，对于 SAM926x 使用六层板来布线的系统来说，PCB 面积的增加带来了系统总成本的增加。
综上所述，不推荐使用片外 nor flash，对软件和硬件开发人员都会带来压力，而是使用 data flash 来替代，甚至是直接使用 NAND 启动。相同容量的 nor flash 和 data flash 有一定的价格差，但是扣除 PCB 面积和相关开发成本，还是值得使用。在可靠性方面，这两者类似。
本文中的 SAM926x 系统基于 data flash 启动，兼顾开发周期，成本与可靠性，基于 NAND 启动的系统与之类似，而基于 nor flash 启动的系统要做的工作就稍多。

3.3 Memory map

这里的 Memory map 主要指存储系统的 map，而网卡，总线上的外设不在此列。为了更好的软件兼容性，推荐外设应当按照官方 EK 上的来设计，比如网卡。下面着重讨论存储系统。
SAM926x 的存储系统，可以通俗理解为存放代码的地方，运行代码的地方，存储变量的地方，和存储数据的地方。存放代码的地方可以是 nor flash，data flash，或者 NAND，除了 nor flash，代码运行的地方只能是 SDRAM，因为 SAM926x 的 SRAM 比较小，不足以支撑实际应用。而大数据量的存储，比如采集的样本，文本资料，可以放到 NAND 或者 SD 卡。
首先讨论 data flash，由于其接口相同，不同容量的芯片可以换用，而硬件上不需要更改，所以給设计带来了很强的适应性和弹性。开发初期可以使用大容量的，而成品可以使用容量正好的。

NAND flash 与 data flash 类似，有统一的硬件接口，设计上的弹性也很大。
以上两者都基本不需要修改硬件设计就可以更换部件，只需要部分的软件处理即可。而 SDRAM 就不同了，由于官方 EK 使用了 32MB x 2 的两片 SDRAM 构成了 64MB，32bit 宽的主存储系统，虽然在 Linux 和 WinCE 下还嫌不够，但对于绝大多数裸奔系统来说都是个很大的空间，用户也很可能裁剪这部分。但是这里也是和代码结合比较紧密的地方，因为 bootstrap 是为 64MB 系统写的，修改了这里也就需要修改其代码，才能确保 SDRAM 能被正确初始化。定制 SDRAM 需要从硬件和软件两方面入手，但都是需要参考数据手册 SDRAM 控制器的章节。
本文不讨论定制 SDRAM 的内容，仍以官方 EK 的 64MB 为标配。

4. 软件系统

4.1 系统启动

先温习一下 SAM926x 数据手册上关于 BOOT 的描述：当使用内部 BOOTROM 启动时，系统根据一定的顺序检索系统中可能的存储媒介，比如 data flash， NAND，SD 卡。当检索到 data flash 连接时，系统进一步读取 data flash 从绝对地址 0 开始的若干字节，如果是合法的 ARM 程序，这些字节都应该是系统的向量，包含了 ARM 的 B 或者 LDR 指令，系统校验这些字节，然后从 0x14 处驱动启动信息(这就是烧写 bootstrap 时要选择 send boot file 的原因)并加载这部分代码到内部的 SRAM 并跳转到其中运行，这样系统的控制权从 BOOTROM 转移到 data flash 中的程序。

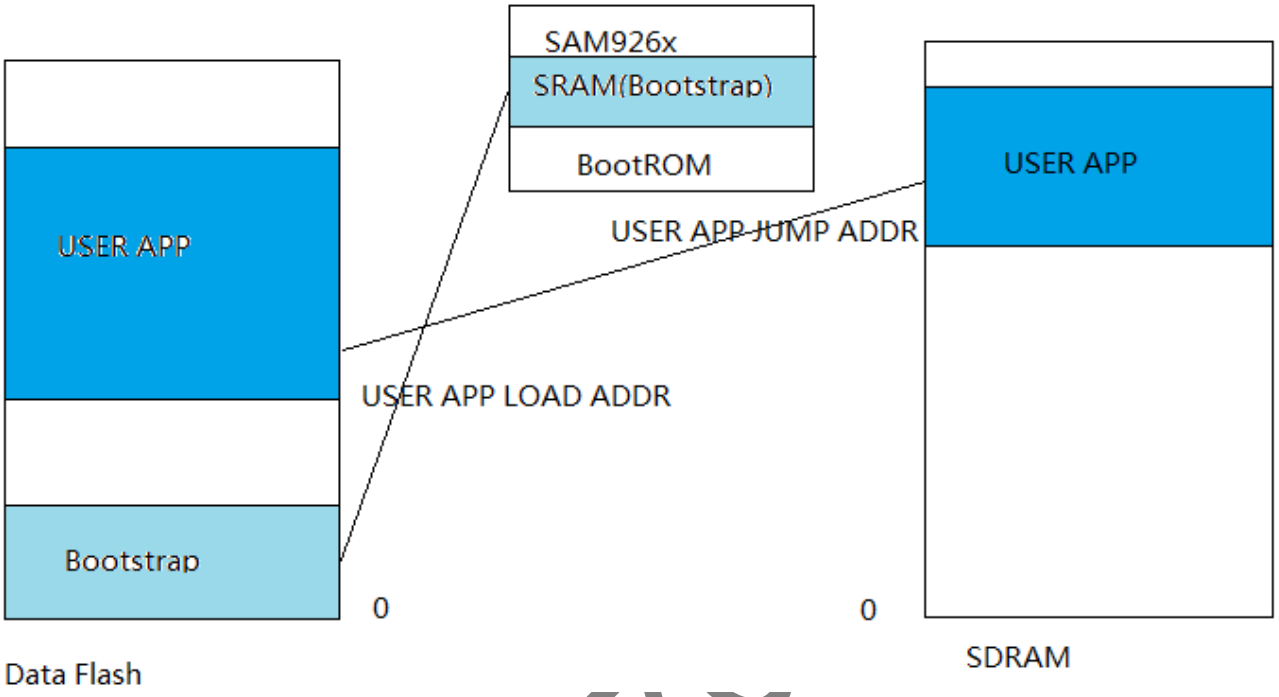
4.2 Bootstrap

由于 SAM926x 的内部 SRAM 极其有限，这也就限制了 BOOTROM 所能加载的代码的大小。那么对于一个比较大的裸奔程序，如何加载？这里就需要用到 Bootstrap。Bootstrap 可以理解为一个程序代码，也可以理解为一个过程。
Bootstrap 的作用就是作为一个加载用户应用的桥梁，它被烧写到 data flash 开始的位置(必须使用 send boot file，因为该选项 SAM-BA 会自动修改 0x14 位置处的特殊向量，满足 BOOTROM 的启动要求)，因为其体积很小(大约 4KB)。系统启动后，BOOTROM 发现了 Bootstrap 并加载其到 SRAM，Bootstrap 开始运行后，将初始化环境，包括时钟系统，data flash，SDRAM，然后就是最重要的工作，就是从 data flash 的指定位置，加载一段指定的代码到 SDRAM 的指定位置，并跳转到那个 SDRAM 的指定位置运行。可以看到，采用这种方式，那段被加载的代码可以很大，因为 SDRAM 的空间很大。
ATMEL 官方提供了 Bootstrap 的源代码，但是需要使用 GCC 进行编译。

4.3 用户代码

在使用 BOOTROM 并使用 Bootstrap 记载的情况下，用户代码在空间上可以被存储在 data flash 或者 NAND flash 上，而在 SDRAM 中运行。由于复制的过程由 Bootstrap 完成，用户代码认为它就是从 SDRAM 的加载点开始

运行，也就是说用户代码的 Linker 地址设置就是 SDRAM 的加载点地址。
由于在 SDRAM 中运行，用户代码的大小可以很大。为了更好的控制用户代码的 memory map，建议 link 使用脚本控制，比如使用 ARM 的 scatter load file。
综上，使用 Bootstrap 下的用户代码启动过程入下图：



4.4 开发环境

对于 SAM926x，很多编译器都可以为之编译代码，比如 ARM RealView，MDK，IAR，GCC 等。但是基于性能的考虑推荐使用 ARM 自己的编译器，RealView 或者 MDK。由于不少开发工程师从 ADS1.2 转过来，下面的描述中使用 RealView 2.2 作为软件开发环境。

4.5 仿真器

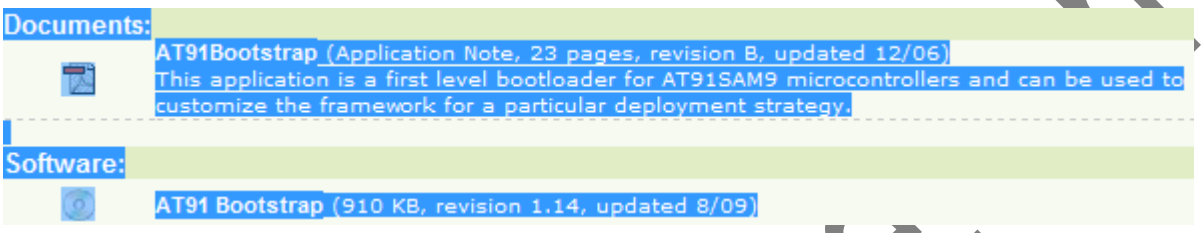
可以用于调试 SAM926x 的仿真器很多，而 RealView2.2 也支持多种方式的仿真器调试，选择面还是很宽的。

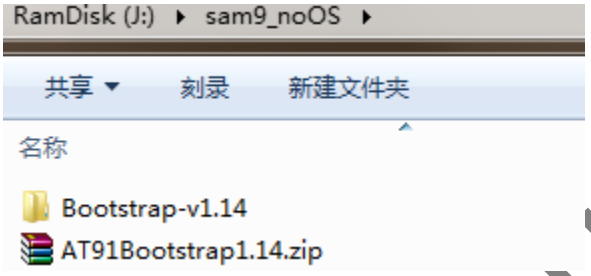
5. 开发实战

5.1 编译 bootstrap

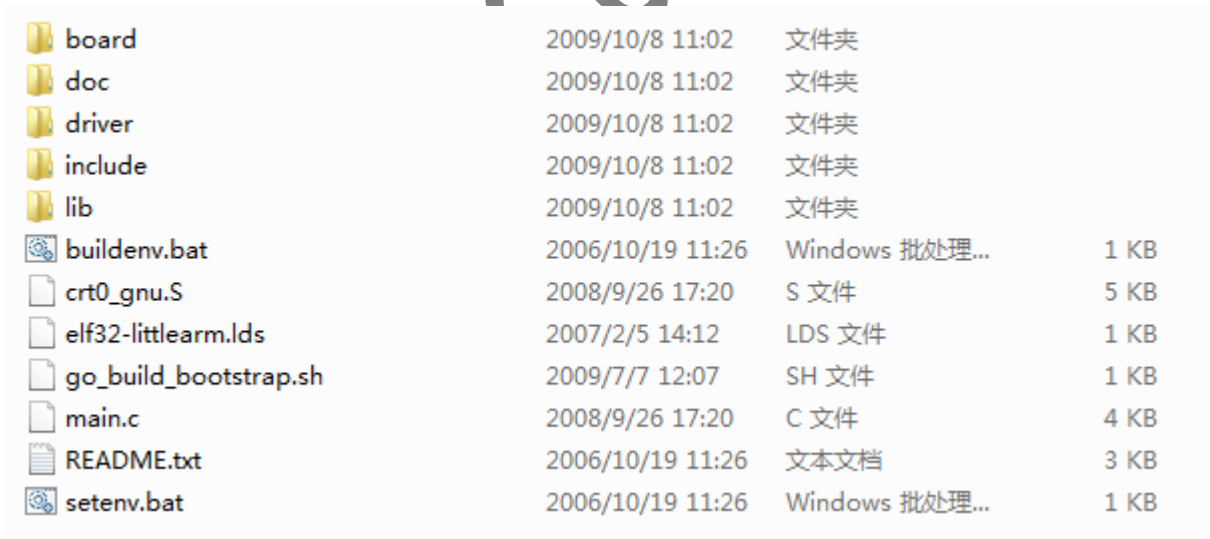
5.1.1 配置编译环境

首先从 ATMEL 下载 Bootstrap 的源码并展开在工作目录下：





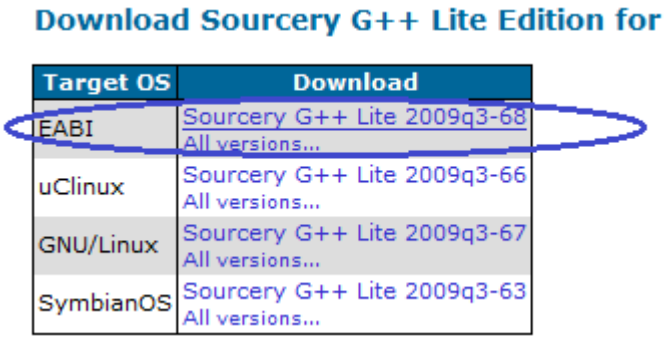
文件夹中的内容：



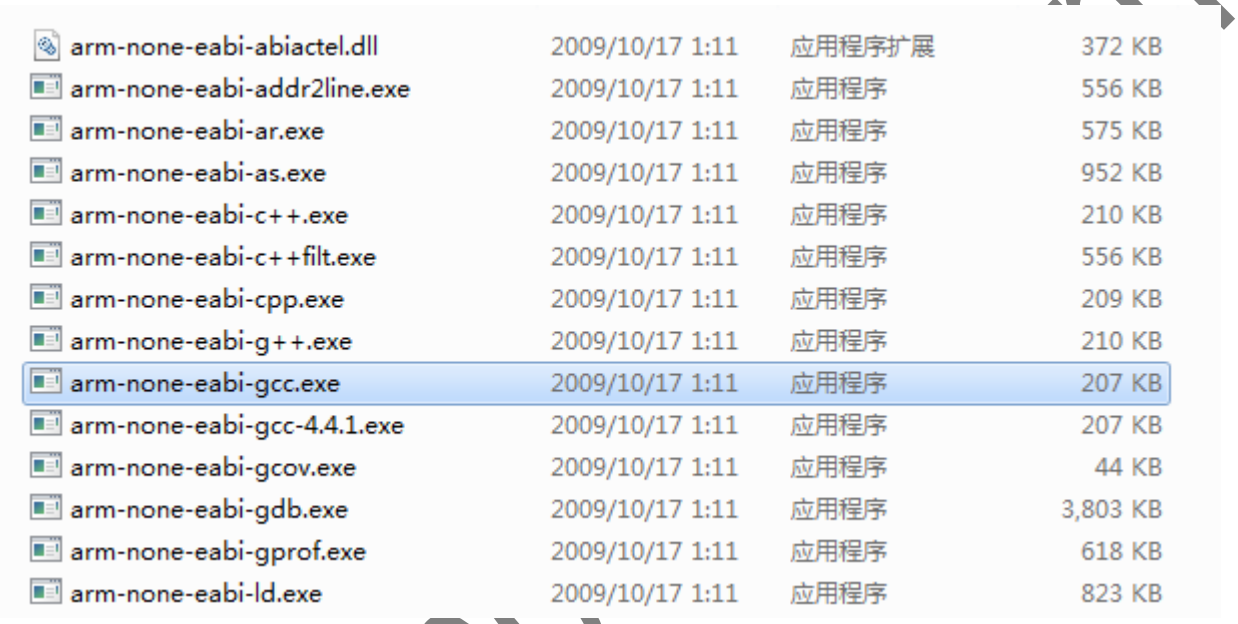
其中的 board 文件夹包含了各种 SAM9 板子的支持，并且每块板子还有不同的启动配置，比如 dataflash，nandflash。

推荐首先阅读 doc 文件夹下的文档，可以帮助理解。driver 文件夹下包含了一些公用的设备驱动代码，include 文件夹下是公用的头文件。

编译 Bootstrap 需要 GCC 支持，并且可以在 windows 上编译。这里推荐使用 codesourcery 的 arm-none-eabi 工具链，



将下载的文件展开，可以看到需要的编译器：



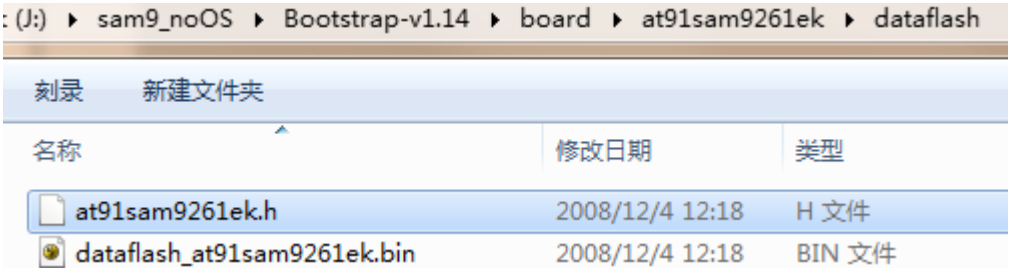
安装后并将其 bin 路径添加到 Windows 系统路径：

```
$ arm-none-eabi-gcc -v
Using built-in specs.
Target: arm-none-eabi
Configured with: /scratch/julian/2009q3-respin-eabi-lite/src/gcc-4.4/configure --build=i686-pc-linux-gnu --host=i686-min
gw32 --target=arm-none-eabi --enable-threads --disable-libmudflap --disable-libssp --disable-libstdcxx-pch --enable-extr
a-sgxxlite-multilibs --with-gnu-as --with-gnu-ld --with-specs='%{O2:%{!fno-remove-local-statics: -fremove-local-statics}
} %{O*:%{O|O0|O1|O2|Os:;:%{!fno-remove-local-statics: -fremove-local-statics}}}' --enable-languages=c,c++ --disable-shar
ed --disable-lto --with-newlib --with-pkgversion='Sourcery G++ Lite 2009q3-68' --with-bugurl=https://support.codesourcer
y.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery --with-headers=yes --with-sysroot=/opt/codesourcery/arm-non
e-eabi --with-build-sysroot=/scratch/julian/2009q3-respin-eabi-lite/install/host-i686-mingw32/arm-none-eabi --with-libic
onv-prefix=/scratch/julian/2009q3-respin-eabi-lite/obj/host-libs-2009q3-68-arm-none-eabi-i686-mingw32/usr --with-gmp=/sc
ratch/julian/2009q3-respin-eabi-lite/obj/host-libs-2009q3-68-arm-none-eabi-i686-mingw32/usr --with-mpfr=/scratch/julian/
2009q3-respin-eabi-lite/obj/host-libs-2009q3-68-arm-none-eabi-i686-mingw32/usr --with-ppl=/scratch/julian/2009q3-respin-
eabi-lite/obj/host-libs-2009q3-68-arm-none-eabi-i686-mingw32/usr --with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lst
dc++,-Bdynamic -lm' --with-cloog=/scratch/julian/2009q3-respin-eabi-lite/obj/host-libs-2009q3-68-arm-none-eabi-i686-ming
w32/usr --disable-libgomp --enable-poison-system-directories --with-build-time-tools=/scratch/julian/2009q3-respin-eabi-
lite/obj/tools-i686-pc-linux-gnu-2009q3-68-arm-none-eabi-i686-mingw32/arm-none-eabi/bin --with-build-time-tools=/scratch
/julian/2009q3-respin-eabi-lite/obj/tools-i686-pc-linux-gnu-2009q3-68-arm-none-eabi-i686-mingw32/arm-none-eabi/bin
Thread model: single
gcc version 4.4.1 (Sourcery G++ Lite 2009q3-68)
```

5.1.2 配置 Bootstrap

对于 Bootstrap 的配置，如果只是为了加载用户应用，只要修改对应 board 下对应配置的.h 文件即可。
本文的描述基于本站的 VC9261-EK 开发板，使用 data flash 启动，配置文件就是 board\at91sam9261ek\dataflash

下的 at91sam9261ek.h：



使用文本编辑工具打开该文件。其中定义了 CPU 的时钟参数：

```
#define MASTER_CLOCK        (198656000/2)
#define PLL_LOCK_TIMEOUT    1000000

#define PLLA_SETTINGS       0x2060BF09
#define PLLB_SETTINGS       0x10483F0E
```

用户程序所在的 data flash 所位于的 SPI 片选：

```
/*.....*/
#define AT91C_SPI_PCS_DATAFLASH     AT91C_SPI_PCS0_DATAFLASH    /* Boot on SPI NCS0 */
```

以及和加载程序关系最大的参数：

```
#define IMG_ADDRESS     0x8400      /* Image Address in DataFlash */
#define IMG_SIZE        0x33900     /* Image Size in DataFlash    */

#define MACH_TYPE       0x350       /* AT91SAM9261-EK */
#define JUMP_ADDR       0x23F00000  /* Final Jump Address         */
```

其中：

IMG_ADDRESS: 用户代码在 data flash 上的起始地址，决定了用户代码烧写时的位置。

IMG_SIZE: Bootstrap 所要加载的代码长度(用户代码的长度)，不能小于用户代码长度，否则会加载不全。

JUMP_ADDR: 用户代码加载到 SDRAM 中的目标地址，也就是用户代码开始运行的地址。

这些参数中，IMG_ADDRESS 建议不要修改。IMG_SIZE 在用户代码开发过程中选择一个较大的值，这样可以不必频繁编译 Bootstrap。而发布之前应该按照用户代码的实际大小修改并重新编译，以避免复制不必要的区域，缩短启动时间。JUMP_ADDR 原来的值是給 u-boot 使用，位于 64MB SDRAM 的最后 1MB。可能对大部分用户代码都不是很合适，针对裸奔系统，建议将其修改为 0x2000 0000，也就是 SDRAM 的起始地址。

5.1.3 编译 Bootstrap

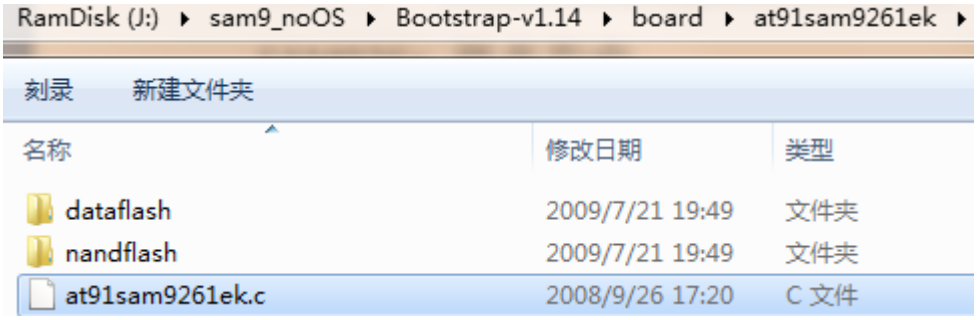
针对上一节描述，修改配置如下：

```
 99  #define IMG_ADDRESS     0x8400      /* Image Address in DataFlash */
100  #define IMG_SIZE        0x40000     /* Image Size in DataFlash    */
101
102  #define MACH_TYPE       0x350       /* AT91SAM9261-EK */
103  #define JUMP_ADDR       0x20000000  /* Final Jump Address         */
```

同时打开 DEBUG 选项，以输出一些信息：

```
108  #define CFG_DEBUG
109  #define CFG_DATAFLASH
110  #define CFG_HW_INIT
111  #define CFG_SDRAM
```

打开上层目录的 at91sam9261ek.c 文件：



修改一下 DEBUG 信息，方便串口识别：

```
104 #ifdef CFG_DEBUG
105     /* Enable Debug messages on the DBGU */
106     dbg_init(BAUDRATE(MASTER_CLOCK, 115200));
107     dbg_print("Start AT91Bootstrap...\n\rMCUzone updated for MAN3008\n\r");
108 #endif /* CFG_DEBUG */
```

启动命令行，一路 cd 到 data flash 配置路径下(cygwin 环境，windows 命令行 cmd 与之类似)：

```
/cygdrive/j/sam9_noOS/Bootstrap-v1.14/board/at91sam9261ek/dataflash
$ ls
Makefile  at91sam9261ek.h
```

首先 make clean：

```
$ make clean
rm -f *.o *.bin *.elf *.map
```

然后 make，注意指明工具链：

```
$ make CROSS_COMPILE=arm-none-eabi-
rm -f *.o *.bin *.elf *.map
arm-none-eabi-gcc -g -mcpu=arm9 -c -Os -Wall -DAT91SAM9261 -I./../../../board/at91sam9261ek/dataflash -DTOP_OF_MEM=0x328
000 ../../../crt0_gnu.S -o crt0_gnu.o
```

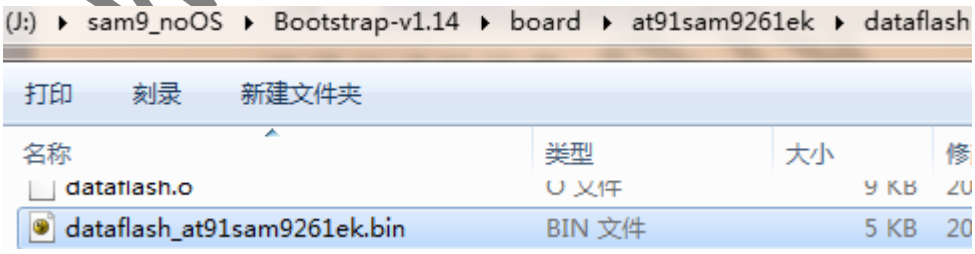
注意其中的 TOP_OF_MEM，这个指明了 SRAM 的界限，如果是 SAM9261S 需要做相应修改。该宏定义在 Makefile 中：

```
20 #Link Address and Top_of_Memory
21 LINK_ADDR=0x300000
22 TOP_OF_MEMORY=0x328000
23 #Name of current directory
24 PROJECT=dataflash
```

编译完成的信息：

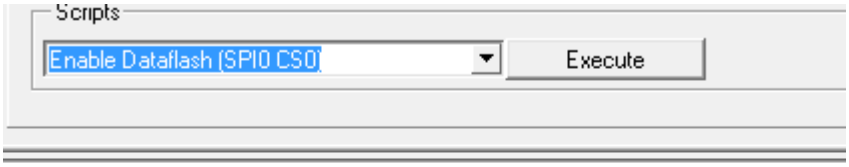
```
arm-none-eabi-gcc -nostartfiles -nostdlib -Wl,-Map=dataflash_at91sam9261ek.map,--cref -T ../../../elf32-littlearm.lds -T
text 0x300000 -n -o dataflash_at91sam9261ek.elf crt0_gnu.o at91sam9261ek.o main.o gpio.o pmc.o debug.o sdramc.o dataflas
h.o _udivsi3.o _umodsi3.o div0.o udiv.o string.o
arm-none-eabi-objcopy --strip-debug --strip-unneeded dataflash_at91sam9261ek.elf -O binary dataflash_at91sam9261ek.bin
```

编译输出的 bin 文件：



5.1.4 测试 Bootstrap

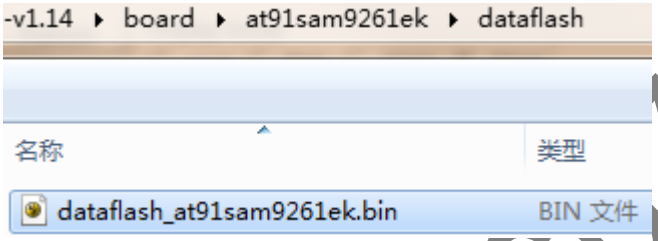
使得 VC9261-EK 进入 SAM-BA 状态，并连接到 PC，启动 SAM-BA 软件并与之相连。
首先运行 enable data flash，这既是一个初始化过程，也是一个检查的过程：



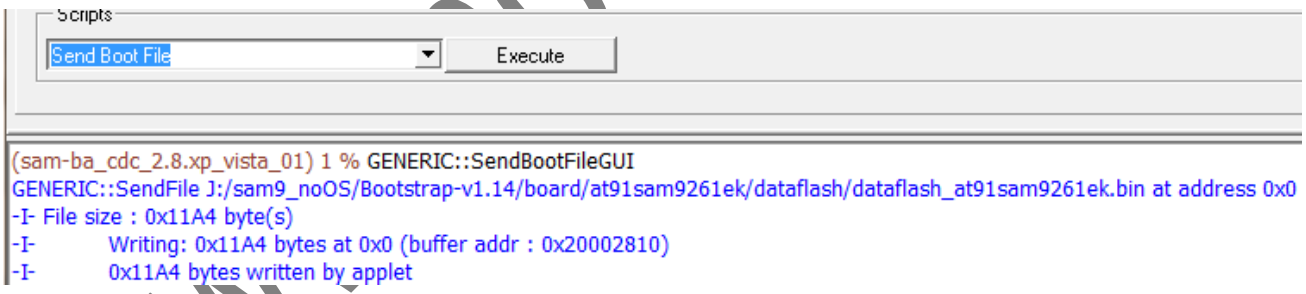

```
-I- Loading applet isp-dataflash-at91sam9261.bin at address 0x20000000
-I- Memory Size : 0x420000 bytes
-I- Buffer address : 0x20002810
-I- Buffer size: 0x40740 bytes
-I- Applet initialization done
```

然后一定要使用 Send Boot File 功能来烧写 Bootstrap：

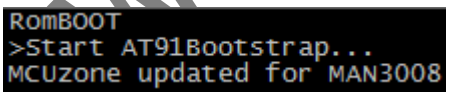
选择对应的文件：



由于文件很小，烧写工作很快就会完成：



使用一条串口线连接开发板的 DBGU 到 PC 串口，并打开串口调试软件，设置为 115200,n,8,1。
按下开发板的 reset 按钮，PC 上收到如下信息：

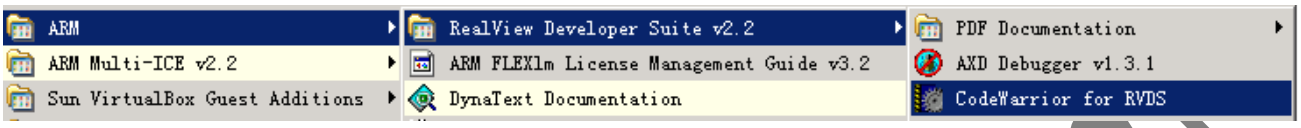


OK，Bootstrap 正常启动。

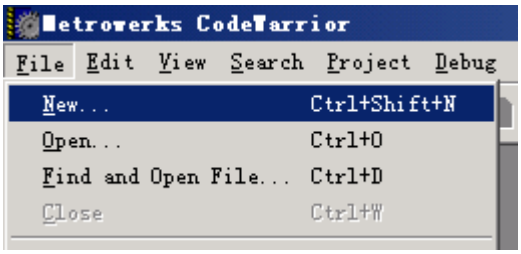
5.2 编译用户代码

5.2.1 新建工程

完整安装 RealView2.2 开发环境，然后运行 IDE：



选择新建：



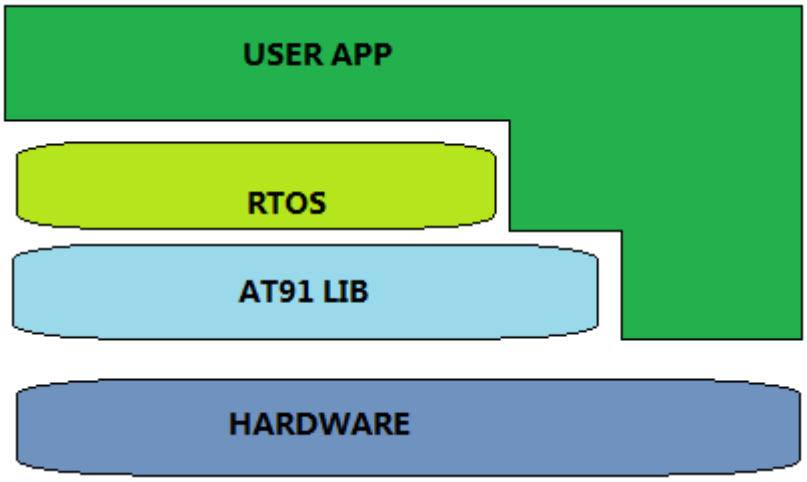
选择创建 ARM 可执行 Image ，并设置工程名为 man2008，因为在 ARM9 上裸奔，代码长度不是个大问题，性能才是大问题：

生成的工程框架：

其中的.mcp 文件就是工程文件。

5.2.2 添加代码

首先看下本例程的软件框架：

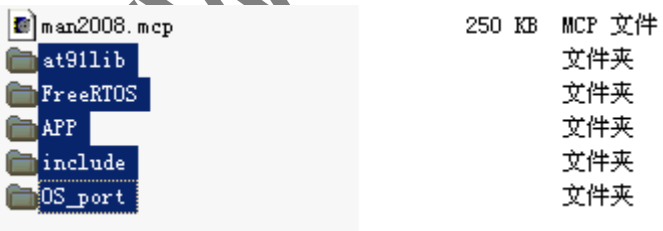


在 Hardware 之上的是由 ATMEL 提供的 AT91 LIB，也就是底层硬件的操作代码，包含在 soft package 中。
软件系统使用了 RTOS，采用 FreeRTOS，开源，且 Royalty-free：

License feature comparison

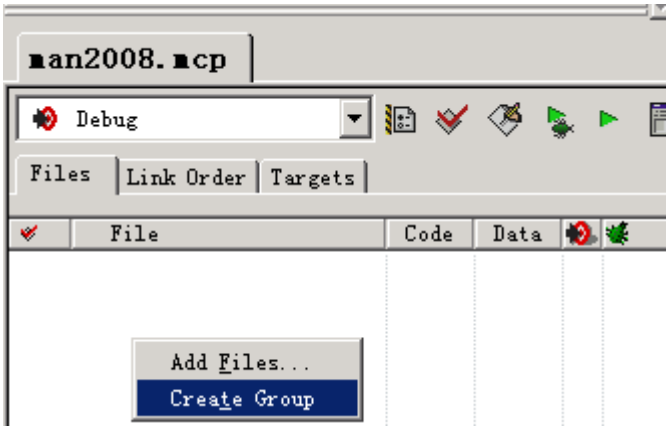
	FreeRTOS Modified GPL license
Is it free?	Yes
Can I use it in a commercial application?	Yes
Is it royalty free?	Yes
Do I have to open source my application code that makes use of the FreeRTOS services?	No, as long as the code provides functionality that is distinct from that provided by FreeRTOS
Do I have to open source my changes to the kernel?	Yes
Do I have to document that my product uses FreeRTOS.org?	Yes a WEB link to the FreeRTOS.org site is sufficient
Do I have to offer to provide the FreeRTOS.org code to users of my application?	Yes
Can I receive professional technical support on a commercial basis?	No, FreeRTOS is supported by an online community
Is a warranty provided?	No
Is legal protection provided?	No

注意，FreeRTOS 的移植不在本文的论述范围。
在工程文件夹下为项目中的各种文件创建文件夹：

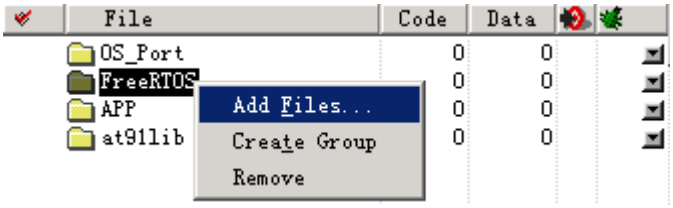


其中 include 文件夹，也按照头文件范围，创建不同的文件夹：

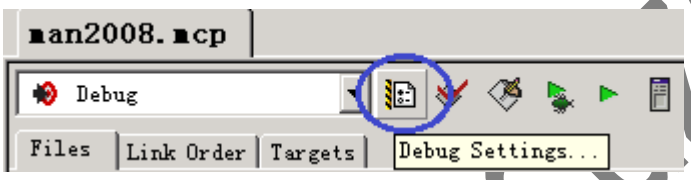
将 at91 lib 及 FreeRTOS 的代码复制到对应文件夹。
在 IDE 中也为工程创建相应的程序组：



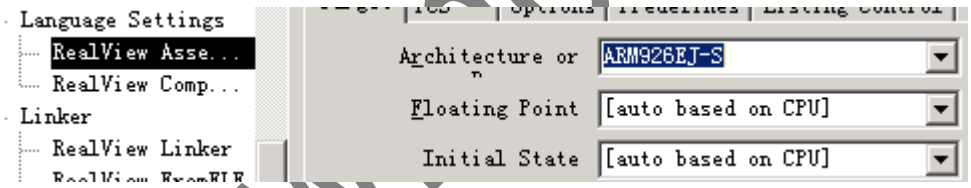
并将文件添加到不同的组：

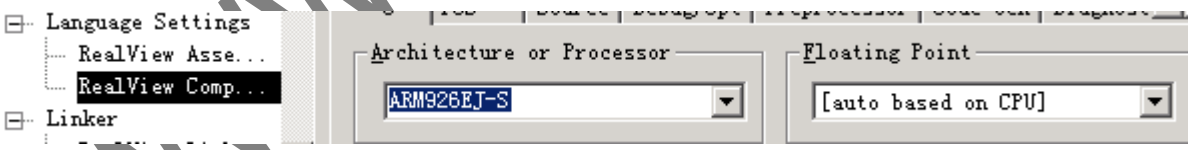


对工程进行配置：

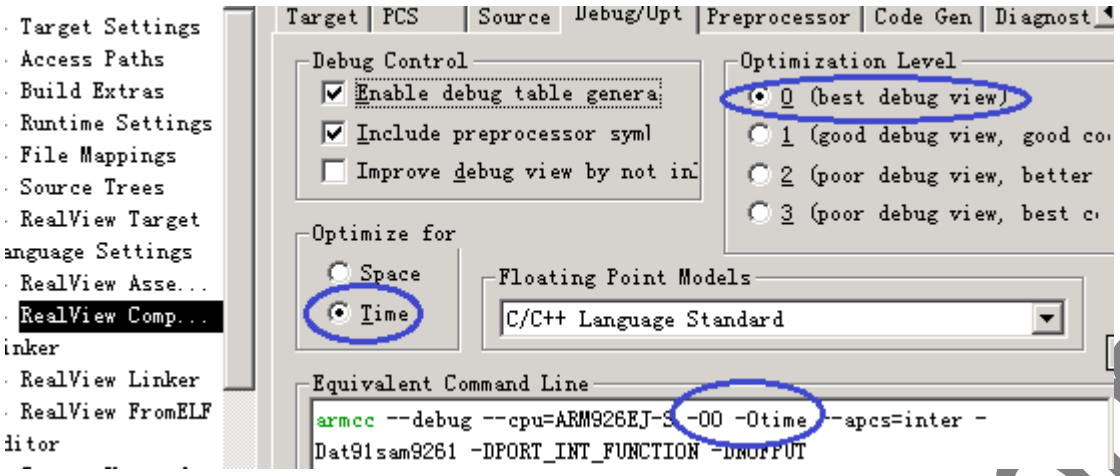


两处设置 ARCH 为 ARM926EJ-S:

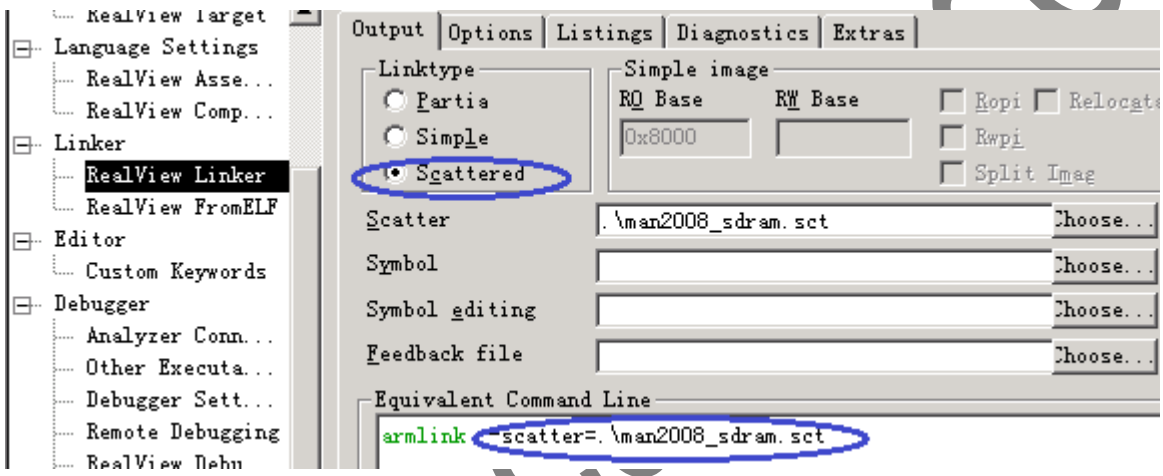




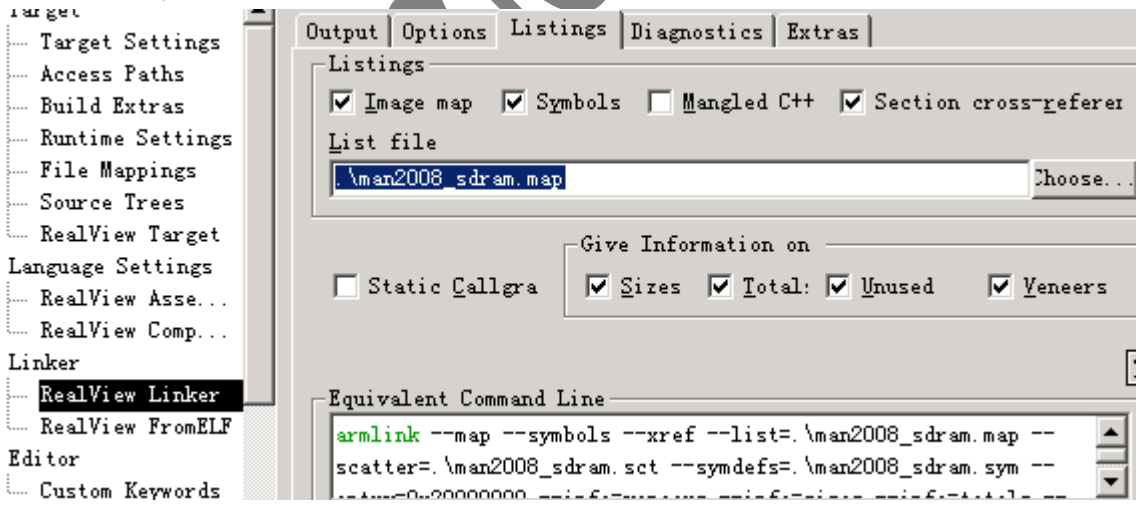
优化等级先设置为 0，对 debug 有利，并且为速度优化：



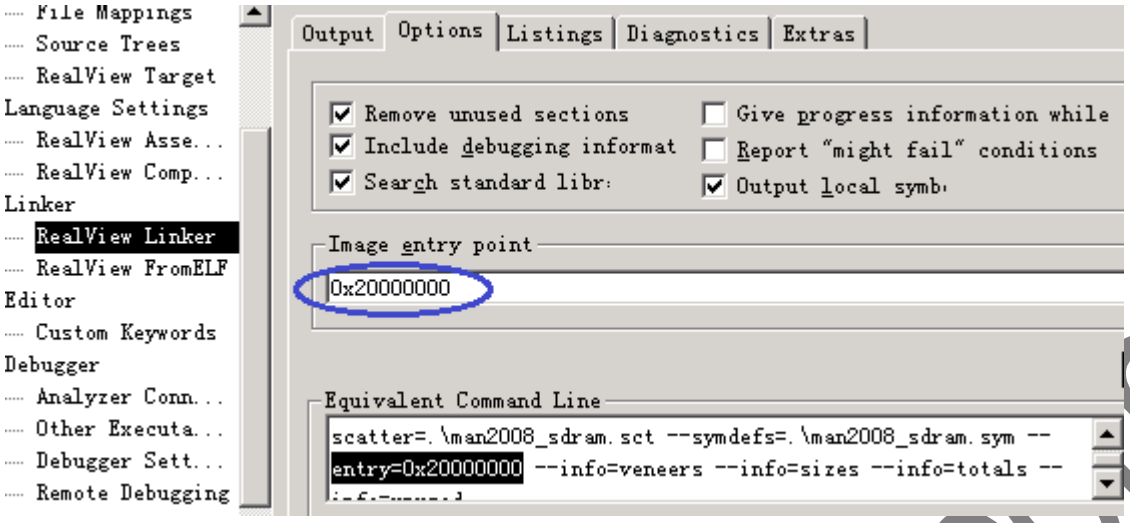
Linker 设置中选择使用 scatter load(参考 ARM linker 的文档)，并指定 scatter load 文件：



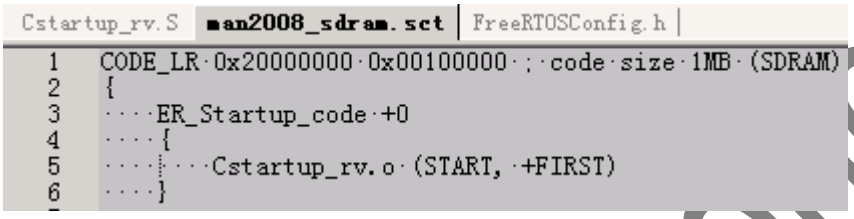
选择输出 map 信息，对分析有利：



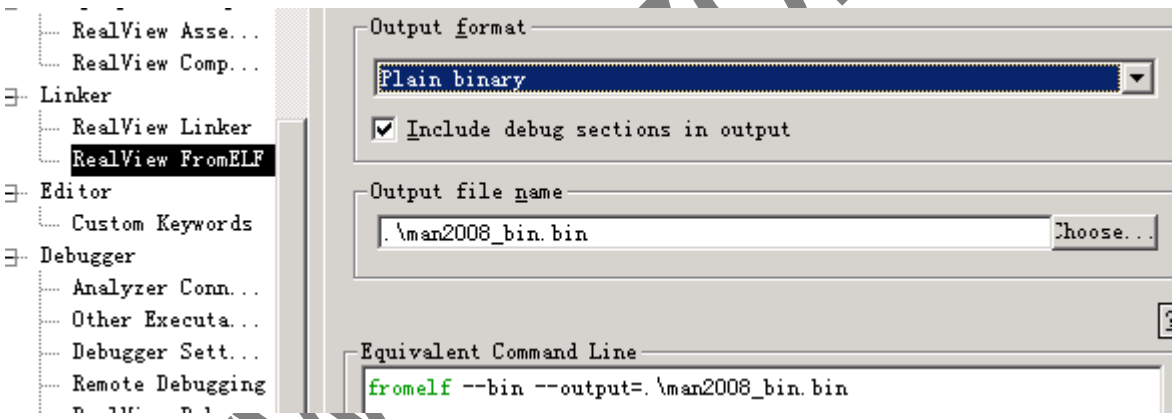
指定 image 的 entry point：



这个也就是 image link 的起始地址，与 sct 文件中的指定一致：

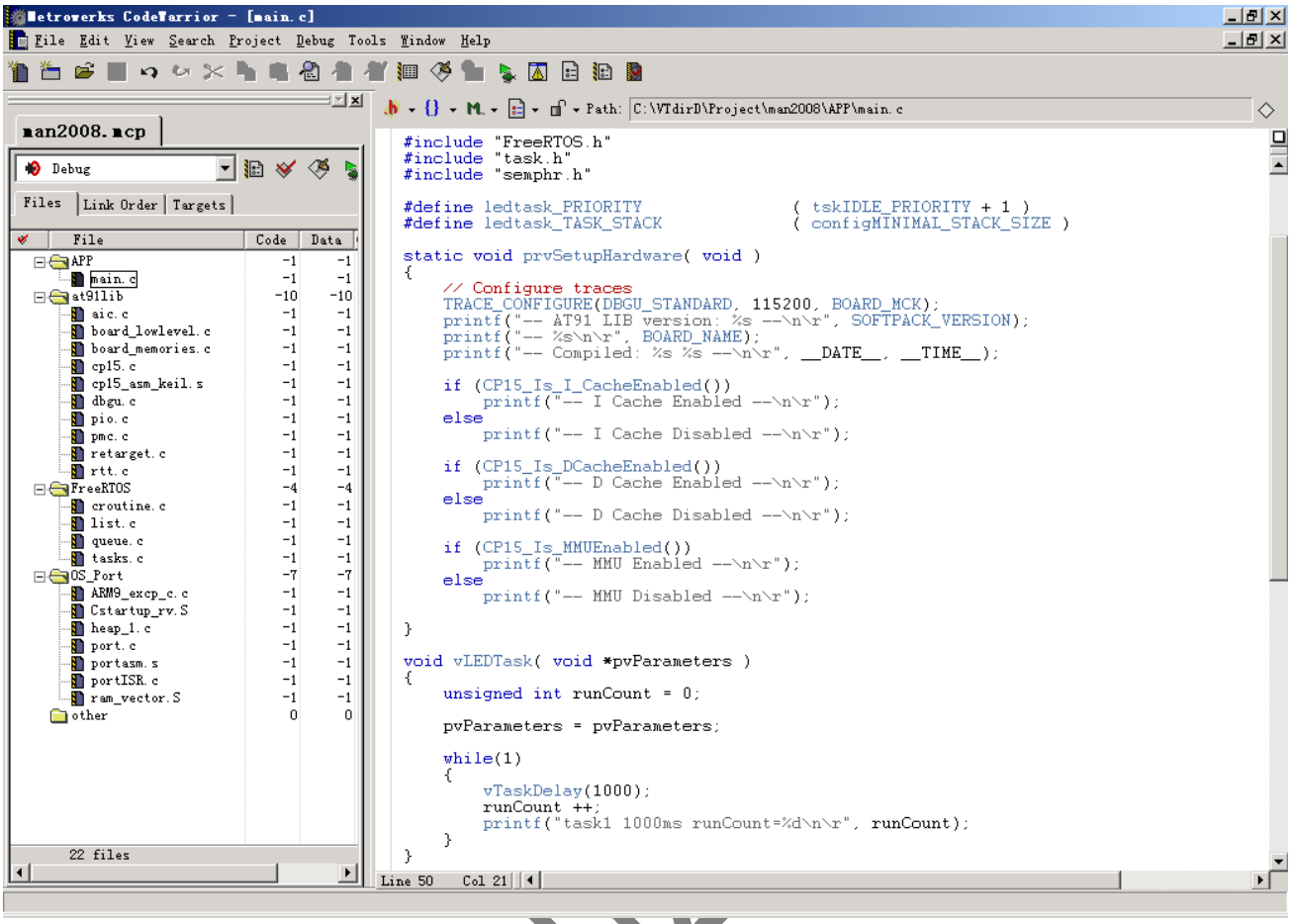


在 FromELF 中选择输出 bin 文件，并指定文件名：



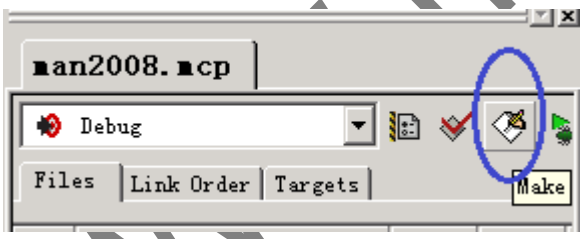
设置完毕，选择 apply 保存所有修改：

添加完成源代码的工程如图：

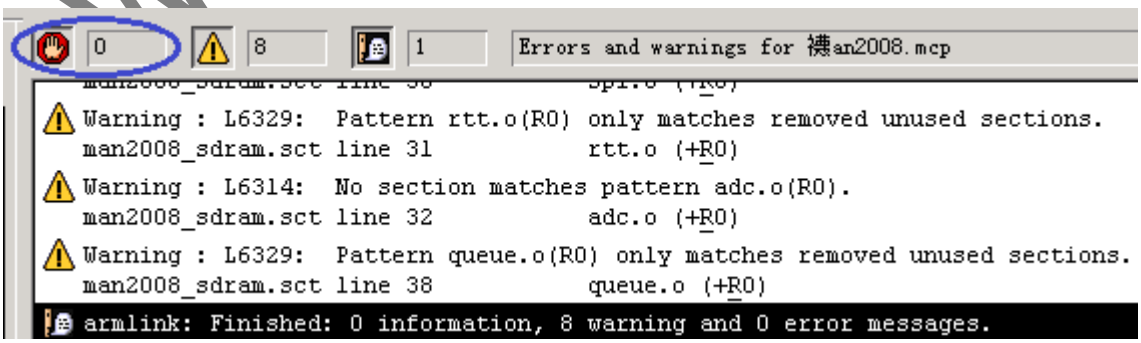


5.2.3 编译代码

点击 Make 按钮开始 build：



没有 error 即是编译通过：

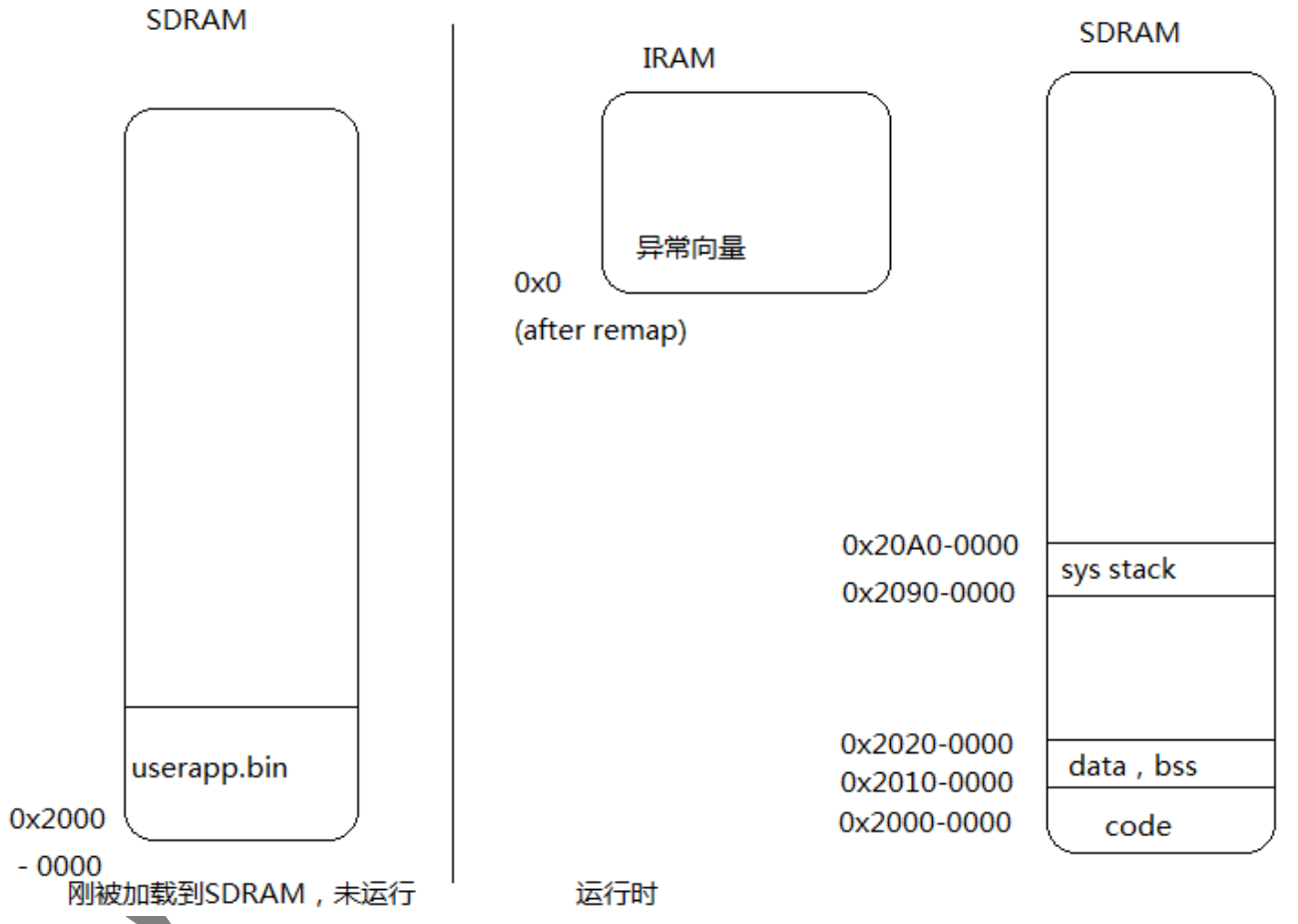


同时生成对应的 bin 文件。

编译完成后通过查看 map 文件确认 image 的 mem map，同时该文件也提供了段，变量的地址信息。
例如，下图中的 Region ER_IRAM_VECTOR 运行于 0 地址，也就是 IRAM remap 之后的地址。该 region 包含了来自 ram_vector.o 与 portasm.o 的代码(code):

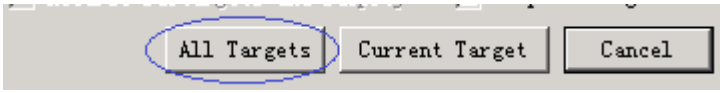
```
Cstartup_rv.S | man2008_sdram.sct | FreeRTOSConfig.h | man2008_sdram.map
625    0x200034d4   0x00000022   Data   RO          359    .constdata          _printf_hex.o(c_a__un.l)
626    0x200034f6   0x00000011   Data   RO          390    .constdata          __printf.o(c_a__un.l)
627    0x20003507   0x00000142   Data   RO          403    .constdata          defsig.o(c_a__un.l)
628
629
630    Execution Region ER_IRAM_VECTOR (Base: 0x00000000, Size: 0x000001e8, Max: 0x00001000, ABSOLUTE)
631
632    Base Addr    Size         Type   Attr      Idx    E Section Name       Object
633
634    0x00000000   0x000000c4   Code   RO          297  * RAM_VECTOR        ram_vector.o
635    0x000000c4   0x00000124   Code   RO          124    FreeRTOS_PORT_ASM   portasm.o
636
637
638    Execution Region ER_data_ram (Base: 0x20100000, Size: 0x0002016c, Max: 0x00100000, ABSOLUTE)
```

该例程的 mem map 如下：



如果要全新编译，需要选择 project->remove 前面编译的所有 objects：

这样可以使得所有的代码都被重新编译一遍，保证编译的确定性。在弹出的菜单中选择清除所有 target：



5.2.4 remap 与中断

由于 ARM 的异常向量相对固定，用户代码从 0x2000-0000 开始运行就要考虑中断向量的问题。
可以使用两种办法解决这个问题，一个是使用 MMU 将 SDRAM 映射到 0 地址，这样用户代码就和一般的应用没有什么区别。另一个办法是利用 scatter loader 文件，将异常向量放置在 IRAM 中，运行时，将这个段(section/region)复制到 IRAM 并将 IRAM 映射到 0，也能实现中断处理。本文中的例子使用第二个方法。

5.3 调试代码

5.3.1 调试的难点

由于用户代码被 link 在 SDRAM 的地址范围，而 SDRAM 在没有初始化之前是不能使用的，所以直接 debug 用户代码是不能凑效的，因为 debugger 无法加载 image 到工作地址。
为了保证在加载代码前 SDRAM 处于可操作状态，可以采取的方法有：1) 编写脚本文件，在调试之前初始化硬件环境；2) 先调试一个在 IRAM 中运行并且能初始化 SDRAM 的代码，待这个代码运行过初始化 SDRAM 的代码后再加载用户代码。

5.3.2 调试实战

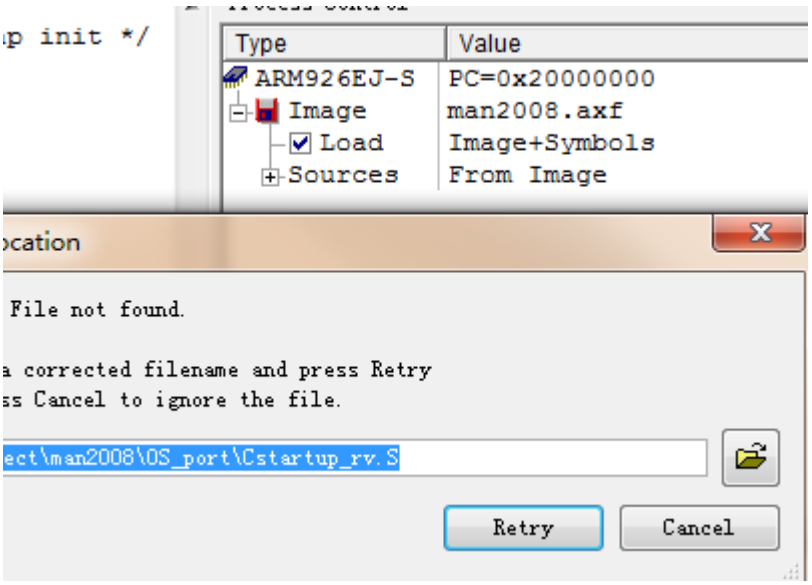
这里叙述使用第二种办法的调试过程。而第一个被加载的文件就是 Bootstrap 的代码，因为其满足那两个要求。
将仿真器连接上开发板，并启动 RVD(不是 AXD)，首先加载 bootstrap 的 elf 文件:

```
178
179  /* Branch to the application at the end of the bootstrap init */
180  _go:
181          ldr     r1, =MACH_TYPE
182          mov     lr, pc
183          bx      r0
184
185          .align
186  _lp_data:
```

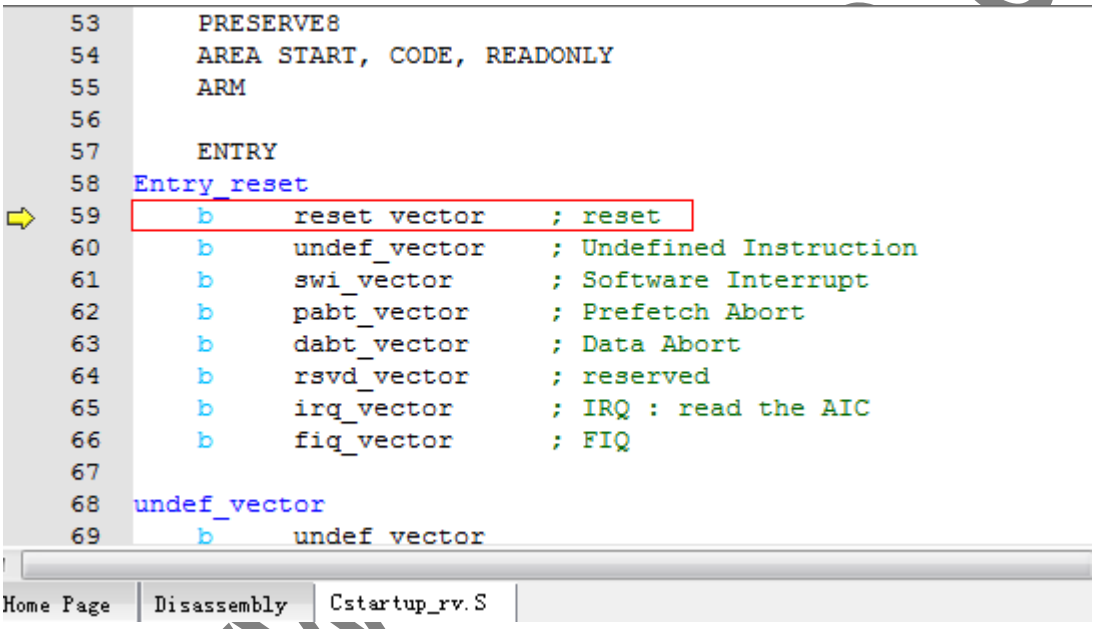
Process Control

Type	Value
ARM926EJ-S	PC=0x00300000
Image	dataflash_at91sam9261ek.elf
Load	Image+Symbols
Sources	From Image

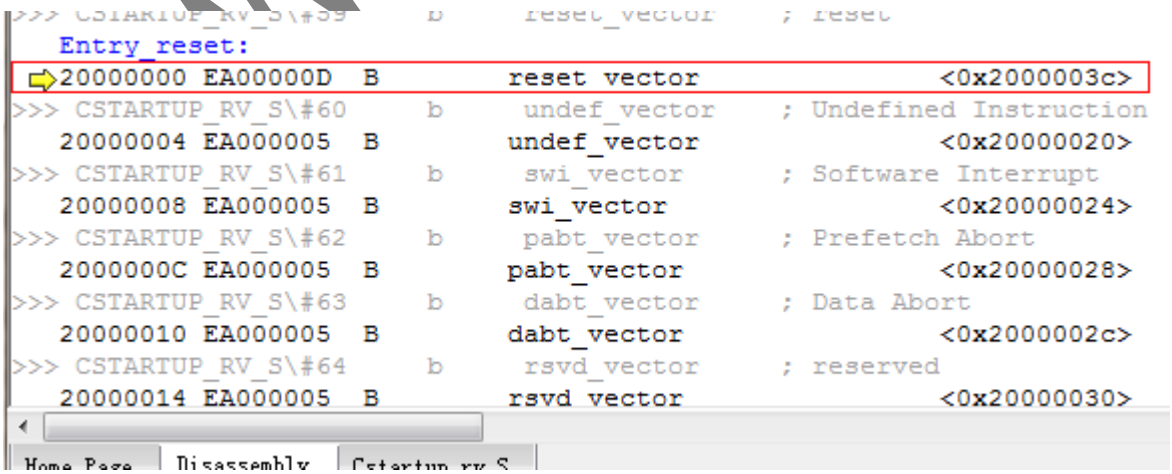
并在上图所示位置打一个断点，这里就是 Bootstrap 完成加载任务并跳转到用户代码的地方。而此时 SDRAM 已经初始化完成。
此时，加载用户代码的 axf 文件：



提示查找文件时，给出对应文件的路径即可。如果有对应的文件，将显示源文件视图：



可以查看反汇编：

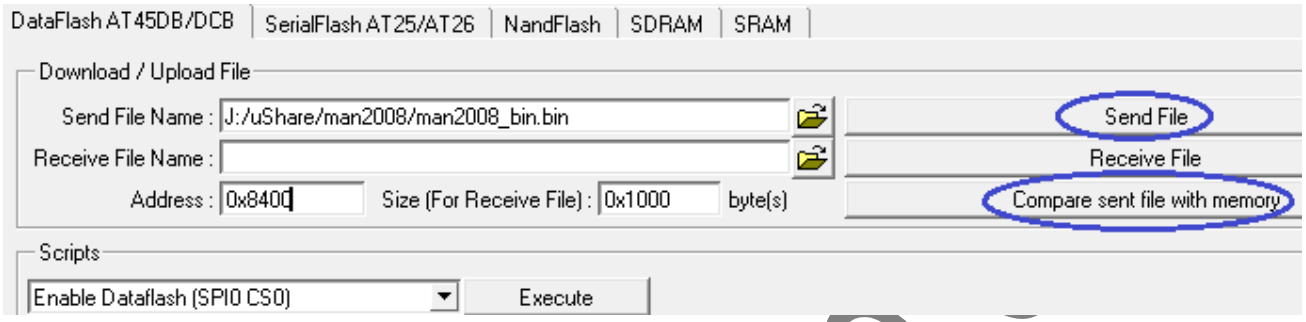


到了这里，就可以对用户代码进行调试了。

5.4 烧写与运行

5.4.1 烧写代码

将 Bootstrap 的 bin 文件用 send boot file 的方式烧写到 data flash。然后 用户代码的 bin 文件按照.h 中的配置烧写到 0x8400：



使用 Send File 烧写文件，烧写完成不要忘记 compare 一下，相当于校验。

5.4.2 运行代码

将板子的 DBGU 连接到 PC，设置为 115200,n,8,1，按下复位键，可以看到代码被加载运行：

```
RomBOOT
>Start AT91Bootstrap...
MCUzone updated for MAN3008
-- AT91 LIB version: 1.5 --
-- AT91SAM9261-EK
-- Compiled:                --
-- I Cache Enabled --
-- D Cache Disabled --
-- MMU Disabled --
```

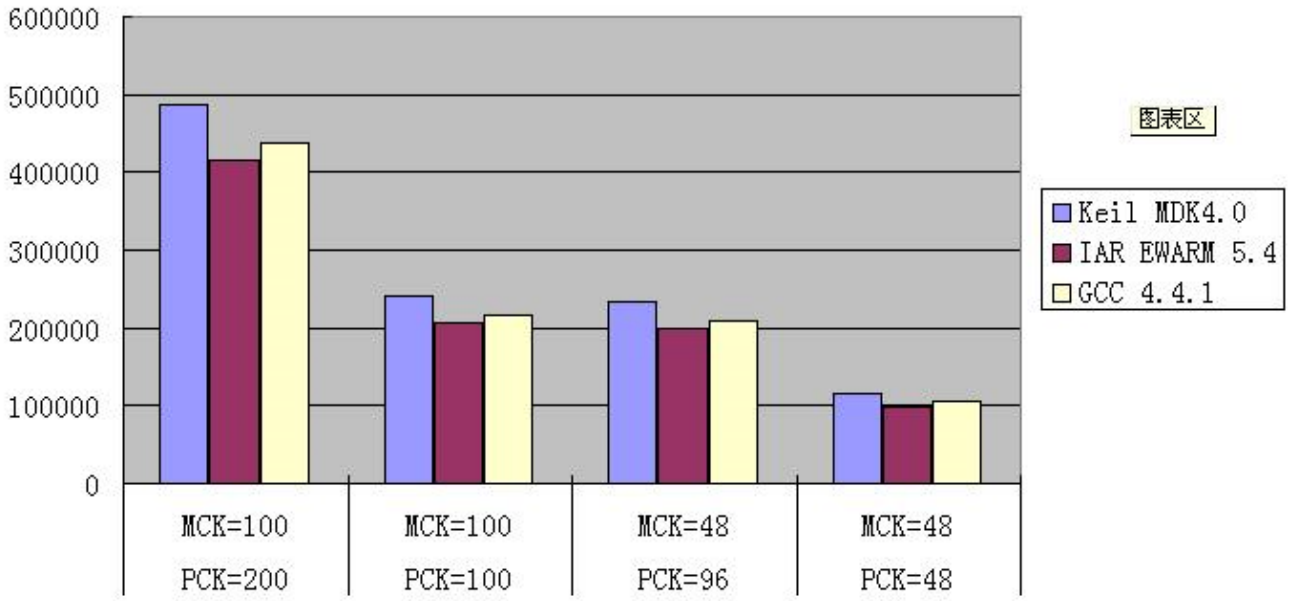
片刻之后，系统中的两个任务就开始运行并打印相应的输出：

```
task1 1000ms runCount=1
task2 2000ms runCount=1
task1 1000ms runCount=2
task1 1000ms runCount=3
task2 2000ms runCount=2
task1 1000ms runCount=4
task1 1000ms runCount=5
task2 2000ms runCount=3
task1 1000ms runCount=6
task1 1000ms runCount=7
task2 2000ms runCount=4
task1 1000ms runCount=8
task1 1000ms runCount=9
task2 2000ms runCount=5
task1 1000ms runCount=10
```

6. 提高性能

6.1 选择编译器

不同编译器的性能不同，应该优选性能好的，本站有过对比：



评测的具体内容请参考 MAN3004 三大编译器效能大比拼。同时编译器的优化策略也会影响输出的代码性能。

6.2 cache

SAM926x 基于 ARM926ej-s 内核构建，具有指令与数据 cache。使用 cache 可以大幅度提升系统的性能，因为 CPU 访问 SDRAM 是比较慢的，而访问 Cache 就快很多。具体的提升数值请参考 MAN3004 三大编译器效能大比拼。

指令 cache，I-Cache，可以加速取指令的速度。又由于大部分的代码在时间和空间上都比较集中，比如循环的代码，就是一个很好的例子。被 cache 后，只要循环体足够小，CPU 都不需要到 SDRAM 中去取指令，可以直接 cache 命中，节省很多的周期。

I-Cache 是一个单向的 cache，只缓存从存储器来的指令数据，因此比较简单，可以很容易的打开。

数据 cache，D-Cache，是为了解决快速的数据读写与较慢的外存储器的矛盾而产生的。数据是可以读写的，因此，D-Cache 是双向的。首先，读取数据的时候可以从 Cache 取，如果 Cache 命中，就不需要从 memory 中读取，这种性能提升对数据表特别明显。其次，写数据的时候，可以先将数据放置在写缓冲中，而不是直接写到 SDRAM，等数据积累一段时间，或者确有必要，才写 memory，这就是 write back 模式，这种模式可以节省写 SDRAM 的时间。当然也可以设置成数据一改就写回到 memory 的 write through 模式，但是就没有优化效果了。

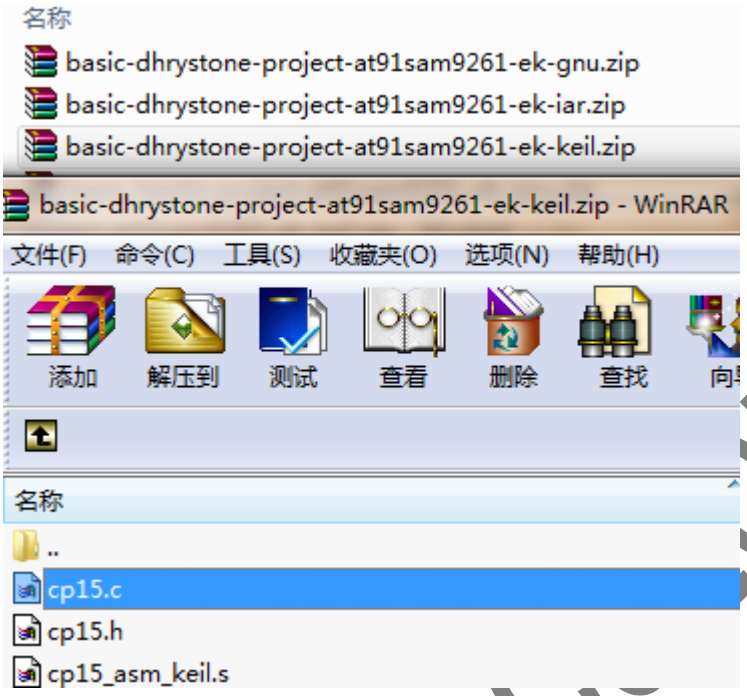
D-Cache 虽然好，但是也带来了一致性问题，如果 memory 被 CPU 之外的控制器更新了，比如 DMA，而 CPU 不知道，继续用 cache 中的数据，就出问题了。

使用 Cache 还会带来不确定问题，因为某个时刻，cache 可以命中，也可以不命中，访问时间就不同了。

另外，ARM926ej-s 的 D-Cache 是依赖于 MMU 的，如果 MMU 没有使能，单独开启 D-Cache 是无效的。

6.3 MMU

MMU 提供两大功能：存储映射与存储保护。
正如前面所说的，使用 MMU 可以将 SDRAM 映射到 0 地址，从而简化中断处理。同时在代码中也可以利用 MMU 的按照权限访问的功能将不同的数据区域保护起来，提升系统的安全性。
关于 MMU 和 Cache 的内容请参考 ARM 的 ARM926 TRM。
ATMEL 在 at91 soft package 中提供了操作的源代码，可以参考其中的例子学习：



7. 其它

SAM926x 的裸奔这个命题很大，绝不是这么寥寥数页纸可以搞定。
应该说，由于 SAM926x 的资源很多，系统也比较复杂，尽可能从现成的应用着手，比如参考 ATMEL 的 lib，或者将工程基于 AT91 LIB(使用 LIB 的时候也要注意 LIB 本身可能也有 bug)，这样一来，可以省掉开发共有代码的时间。
为了加速开发，应该尽量避免纯裸奔，推荐选择 RTOS 构建应用系统。
受到条件和水平的限制，本文难免错漏之处，请到微控电子论坛讨论。