BSP 编译过程研究

SWD 聂勇

版 本 历 史

版本/状态	责任人	起止日期	备注
V1.0/正式	聂勇	11Oct2010	BCM 平台下 BSP 编译过程研究

目　录

1. 说明

本文档为 Boot ROM 和 vxWorks 的编译过程研究结果。平台为 bcm5836，使用的开发方案为 broadcom 公司提供的 bcm_harrier 方案。

主要参考的资料

- 《 BCM5836 VxWorks BSP User's Guide.pdf 》 Section 4 Build Boot ROM and VxWorks Image
- 《程序员的自我修养-链接、装载与库》
- 《计算机组织与系统结构》北京大学程旭老师讲义——MIPS 指令系统结构

2. Makefile 文件

这里使用的是 Tornodo 开发工具（Tornodo development tools），这既可以看成是一个集成开发环境（IDE Integrate Development Environment），又可以看成是一个工具链（tool chains）。

在 Makefile 文件中，经常使用到的一个变量就是 Tornodo 开发工具的安装目录 $(WIND_BASE)。

2.1 常用路径

1. $(WIND_BASE)

   例如：E:\Tornado2.2.1-mips　　Tornodo 开发工具的安装目录。

2. $(WIND_BASE)\host\x86-win32\bin

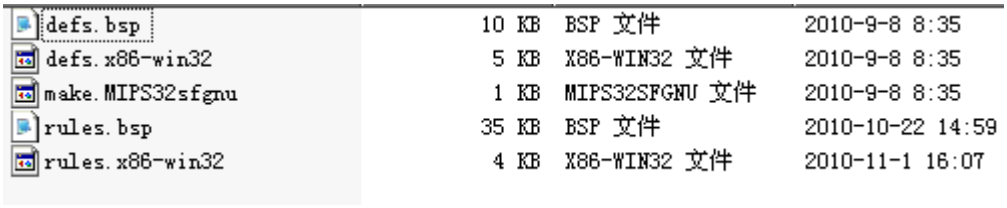
   例如：E:\Tornado2.2.1-mips\host\x86-win32\bin

   Tornodo 开发工具所在的目录，包含我们用到的所有开发工具的可执行文件（***.exe）。可以是 IDE 的窗口界面，例如 Tornado.exe；可以是 GNU 工具，例如 ccmips.exe，ldmips.exe，objcopymips.exe；也可以是其它的工具，例如 tar.exe 等等。

2.2 Makefile 类型

本文档中提到的 Makefile 文件主要有两种，standard Makefile 和 default Makefile。

standard Makefile 是 Tonardo 开发工具提供，对于所有的 BSP 都是适用的。存放的路径一般为 $(WIND_BASE)\target\h\make。在 default Makefile 肯定包含某几个特定的 standard Makefile。standard Makefile 一般不要去修改。在我们的 default Makefile 中需要包含的几个 standard Makefile 如下图所示。



defs.bsp	10 KB	BSP 文件	2010-9-8 8:35
defs.x86-win32	5 KB	X86-WIN32 文件	2010-9-8 8:35
make.MIPS32sfgnu	1 KB	MIPS32SFGNU 文件	2010-9-8 8:35
rules.bsp	35 KB	BSP 文件	2010-10-22 14:59
rules.x86-win32	4 KB	X86-WIN32 文件	2010-11-1 16:07

在以后的工作中，有需要涉及到 standard Makefile 文件的阅读、修改，我们再针对性的做一些讲解。

default Makefile 存放在需要编译的 BSP 目录中，用于编译该 BSP。需要调用 Tonardo 开发工具提供的 standard Makefile 完成大部分工作。大部分时候，通过修改 default Makefile 来配置 BSP 的编译。

2.3 Makefile 分析

下面就 Makefile 文件（主要是 default Makefile）中经常遇到的问题记录总结。

2.3.1 变量 CONFIG_ALL

变量 CONFIG_ALL 的定义位于 standard Makefile 中的 defs.x86-win32 中，控制着 BSP 中 all 文件夹的位置。默认路径为$(TGT_DIR)\config\all。如果不加以修改，那么在编译过程中，就不会使用当前路径下的 all 文件夹下的文件，其中主要有 bootInit.c，bootConfig.c，usrConfig.c，configAll.h 等等，如下图所示：

defs.x86-win32

```
……
CONFIG_ALL      = $(TGT_DIR)\config\all
……
USRCONFIG       = $(CONFIG_ALL)\usrConfig.c
BOOTCONFIG      = $(CONFIG_ALL)\bootConfig.c
BOOTINIT     = $(CONFIG_ALL)\bootInit.c
DATASEGPAD      = $(CONFIG_ALL)\dataSegPad.c
CONFIG_ALL_H  = $(CONFIG_ALL)\configAll.h
……
```

如果想使用在当前 BSP 目录下的 all 文件夹，则需要在 default Makefile 中添加对 CONFIG_ALL 变量的修改为当前路径下。（注意，修改必须要在 include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)）。

Makefile

```
……
include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)
include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)
```

```
CONFIG_ALL=./all
USRCONFIG      = $(CONFIG_ALL)\usrConfig.c
BOOTCONFIG     = $(CONFIG_ALL)\bootConfig.c
BOOTINIT       = $(CONFIG_ALL)\bootInit.c
DATASEGPAD     = $(CONFIG_ALL)\dataSegPad.c
CONFIG_ALL_H   = $(CONFIG_ALL)\configAll.h
……
```

做完这一步调整之后，编译的时候可能会提示有些 C 文件找不到，这是因为在 bootConfig.c 文件中，有很多相对路径的 C 文件包含。这个相对路径是相对于默认的 CONFIG_ALL 路径下的 bootConfig.c 文件。

一种解决的办法就是将所有的相对路径都修改为绝对路径 E:/Tornado2.2.1-mips/target/config/comps/src。

3. 编译 Boot ROM

3.1 cmd 编译步骤

第一步：启动 cmd 终端，设置编译的环境变量。可以拖曳 $(WIND_BASE)\host\x86-win32\bin\torVars.bat 到 cmd 窗口执行。需要使用到的环境变量的设置如下：

torVars.bat

```
……
set WIND_HOST_TYPE=x86-win32
set WIND_BASE=E:\Tornado2.2.1-mips
set PATH=%WIND_BASE%\host\%WIND_HOST_TYPE%\bin;%PATH%
……
```

查看变量 PATH，输入以下命令：PATH 或者 echo %PATH%

查看其他变量，使用 echo 命令，例如：echo %WIND_BASE%

注意，cmd 的命令不区分大小写

第二步：切换路径到 BSP 的文件夹下，输入命令 make bootrom.bin。这样，make 程序会自动以当前路径下的 Makefile 文件为输入，编译整个工程。也可以用选项-f 选择默认的 Makefile 文件，例如：make –f Makefile_ny bootrom.bin。不能够输入命令 make，而没

有参数 bootrom.bin，这样 standard Makefile 文件 rules.bsp 和 rules.$(WIND_HOST_TYPE) 会编译整个 vxWorks 的 SDK，而不是编译 boot rom。

第三步：输入 make clean 清除编译获得的目标文件和可执行文件。

3.2 编译链接顺序

输入命令 make bootrom.bin 之后，Boot ROM 的编译链接顺序是怎么样的呢？是如何由源文件生成目标文件，目标文件又是如何链接成最后的可执行文件的呢？本节的 Boot ROM 编译链接顺序就是来解决这些问题的。

整个编译链接的顺序主要依靠于 Tornado 开发工具提供的 standard Makefile，也就是 rules.bsp 文件和 rules.$(WIND_HOST_TYPE)文件。由于一般需要修改的是 BSP 工程中的 default Makefile，若无特别说明，下文中提到的 Makefile 是指工程中的 default Makefile。

概要的说，这个编译链接过程由下面几步组成

1）将 bootconfig、version、sysAlib.s,sysLib.c 等文件（除 romInit.s 和 bootInit.c 外）编译链接为 tmp.o（在此之前还有头文件包含关系文件 depend 的生成）

2）将 tmp.o 编译为 tmp.out

3）将 tmp.out 压缩为 tmp.z

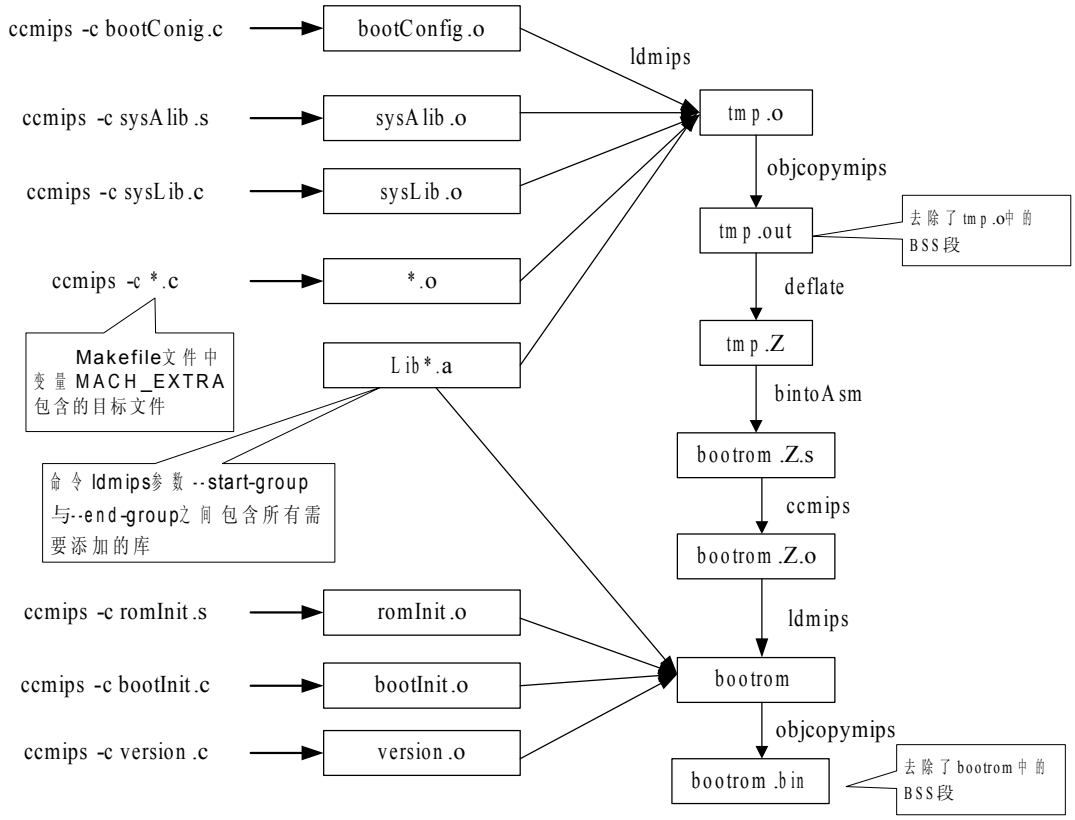
4）将 tmp.z 利用工具 binToAsm 转化为汇编文件 bootrom.Z.s，其文件内容起始标签为 binArrayStart，结束标签为 binArrayEnd

5）将 bootrom.Z.s 编译为 bootroom.Z.o

6）将 romInit.s 和 bootInit.c 和 bootrom.Z.o 编译链接为 ELF 文件 bootrom

7）最后将 bootrom 转化为 bin 文件

整个流程可以由下图表示：



3.2.1 第一步 depend 文件

第一步就是生成 depend.$(TARGET_DIR)文件。扩展名由变量 TARGET_DIR 决定，可以在 Makefile 中设置，默认为 bcm95836cpci。

depend 文件是需要编译的目标文件所依赖的头文件的集合。就是找到编译每个目标文件所需要的头文件的位置。头文件在当前工程中，则直接列出文件，无需路径；如若使用的是于 vxWorks 相关的头文件，则列出绝对路径。从处理过程来看，首先处理的是 C 文件，然后处理两个汇编文件（romInit.s 和 sysAlib.s）。下面举例说明。

rootInit.s 源文件包含的头文件如下表所示。

romInit.s
……
#include "vxWorks.h"
#include "arch/mips/ivMips.h"
#include "arch/mips/asmMips.h"
#include "arch/mips/esfMips.h"
#include "sysLib.h"
#include "config.h"

```
#include "bcm4704.h"
#include "sbmemc.h"
#include "sbconfig.h"
……
```

下表为生成的 depend 文件对应的头文件列表。可以看出，调用的系统的头文件都放在路径$(WIND_BASE)/target/h/下面。最后 mbz.h bcm4704.h sbmemc.h sbconfig.h 四个文件为 BSP 目录下原有的文件，所以没有写出绝对路径。

depend.bcm4704

```
……
romInit.o: romInit.s $(WIND_BASE)/target/h/vxWorks.h \
  $(WIND_BASE)/target/h/types/vxCpu.h \
  $(WIND_BASE)/target/h/types/vxArch.h \
  $(WIND_BASE)/target/h/arch/mips/archMips.h \
  $(WIND_BASE)/target/h/tool/gnu/toolMacros.h \
  $(WIND_BASE)/target/h/arch/mips/ivMips.h \
  $(WIND_BASE)/target/h/arch/mips/asmMips.h \
  $(WIND_BASE)/target/h/arch/mips/esfMips.h \
  $(WIND_BASE)/target/h/arch/mips/regsMips.h \
  $(WIND_BASE)/target/h/sysLib.h config.h all/configAll.h \
  $(WIND_BASE)/target/h/smLib.h \
  $(WIND_BASE)/target/h/vwModNum.h \
  $(WIND_BASE)/target/h/vme.h $(WIND_BASE)/target/h/iv.h \
  mbz.h bcm4704.h sbmemc.h sbconfig.h
……
```

3.2.2 第二步 编译目标文件

这里需要编译的目标文件有： bootInit.c, romInit.s, bootConfig.c, sysALib.s, sysLib.c, version.c，以及在 Makefile 文件中使用 MACH_EXTRA 变量定义的文件。

注意：有且仅有 bootInit.c 文件在编译的时候使用了选项-fpic。

3.2.3 第三步 生成 tmp.o

将上一步生成的目标文件 bootConfig.o, sysALib.o, sysLib.o, version.o ，MACH_EXTRA 变量定义生成的目标文件，以及一些库文件（library object file ）链接成文件 tmp.o。

下面为编译时在终端输出的结果。几个重要的参数（加粗突出）需要注意：

-e usrInit 该参数指明了 tmp.o 文件的入口点

-Ttext 81800000 tmp.o 中文本段的起始地址，也就是在 Makefile 文件中定义的变量 RAM_HIGH_ADRS。有关链接 ld 的相关知识。参考第五节附件。

Build Output
…… ldmips -o tmp.o -EB -X -N **-e usrInit** \ **-Ttext 81800000** bootConfig.o version.o sysALib.o sysLib.o srecLoad.o ns16550Sio.o ca cheLib.o cacheALib.o pciConfigLib.o pciIntLib.o sysSerial.o et_vx.o etc.o etc47xx.o vx_osl .o hnddma.o sbutils.o bcmutils.o m48t59y.o ds1743.o flash29l640DrvLib.o flash29l320DrvLib. o flash29l160DrvLib.o flash28f320DrvLib.o flash28f640DrvLib.o flash29gl128DrvLib.o flashDr vLib.o flashFsLib.o ftpXfer2.o flashUtil.o nvramstubs.o bcmsrom.o \ **--start-group** -LE:\Tornado2.2.1-mips\target\lib\mips\MIPS32\sfgnu -LE:\Tornado2.2.1-mips\target\lib\mips\MIPS32\sfcommon -LE:\Tornado2.2.1-mips\target\lib\mips\MIPS32\sfgnu -LE:\Tornado2.2.1-mips\target\lib\mips\MIPS32\sfcommon E:\Tornado2.2.1-mips\target\lib\mips\MIPS32\sfcommon_rc32xxx\libarch.a -lcplus -lgnucplus -lvxcom -larch -lcci -lccidef -lccigmp -lcommoncc -ldcc -ldrv -lgcc -lnet -los -lrpc -ltffs -lusb -lwdb -lwi nd -lwindview -lcplus -lgnucplus -lvxcom -larch -lcci -lccidef -lccigmp -lcommonc c -ldcc -ldrv -lgcc -lnet -los -lrpc -ltffs -lusb -lwdb -lwind -lwindview E:\Tornado2.2.1-mips\target\lib\libMIPS32sfgnuvx.a **--end-group** \ -T E:\Tornado2.2.1-mips\target\h\tool\gnu\ldscripts\link.RAM ……

上表中标记的--start-group 于--end-group 之间，就是指明调用哪些库文件（ library object file ）。

4. 编译 vxWorks

尚未进行……

5. 附件

5.1 链接器 ld 及其选项

使用到的链接器 ld 中最重要的两个是选项是-e 和-Ttext。具体的知识请参考《程序员的自我修养-链接、装载与库》中的 4.6 节，链接过程控制。

5.2 链接重定位

在调试的过程中，需要对目标文件或者可执行文件进行反汇编。对目标文件和可执行文件反汇编的结果可能不一样。出现这样的差异，主要是链接中的重定位造成。下面 以romStart()函数的反汇编代码为例说明产生这种差别的原委。

对目标文件 bootInit.o 的反汇编结果：

```
bootInit.o:     file format elf32-bigmips

Disassembly of section .text:

0000000000000000 <romStart>:
   0:   27bdffe0        addiu   $sp,$sp,-32
   4:   afbf001c        sw      $ra,28($sp)
   8:   afbe0018        sw      $s8,24($sp)
   c:   03a0f021        move    $s8,$sp
  10:   afc40020        sw      $a0,32($s8)
  14:   3c030000        lui     $v1,0x0
  18:   246300e8        addiu   $v1,$v1,232
  1c:   3c020000        lui     $v0,0x0
  20:   24420000        addiu   $v0,$v0,0
```

对可执行文件 bootrom 的反汇编结果：

```
        ...

80100f20 <romStart>:
80100f20:       27bdffe0        addiu   $sp,$sp,-32
80100f24:       afbf001c        sw      $ra,28($sp)
80100f28:       afbe0018        sw      $s8,24($sp)
80100f2c:       03a0f021        move    $s8,$sp
80100f30:       afc40020        sw      $a0,32($s8)
80100f34:       3c038010        lui     $v1,0x8010
80100f38:       24631008        addiu   $v1,$v1,4104
80100f3c:       3c028010        lui     $v0,0x8010
80100f40:       24420000        addiu   $v0,$v0,0
80100f44:       00621823        subu    $v1,$v1,$v0
80100f48:       3c02bfc0        lui     $v0,0xbfc0
80100f4c:       00623821        addu    $a3,$v1,$v0
```

注意红色方框中的汇编指令差别。通过分析代码，可以肯定，这几句代码是在寻找被调用函数 copyLongs 的地址。那么，如何知道由目标文件 bootInit.o 链接成可执行文件的时候，这几句代码需要修改呢，其修改又是根据什么规则呢？

对目标文件 bootInit.o 进行分析，查看其重定位表，结果如下：

```
E:\source\bcm4704_BSP_v1>objdumpmips -r bootInit.o

bootInit.o:     file format elf32-bigmips

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000014 R_MIPS_HI16       .text+0x00000000000000e8
0000000000000018 R_MIPS_LO16       .text+0x00000000000000e8
000000000000001c R_MIPS_HI16       romInit
0000000000000020 R_MIPS_LO16       romInit
0000000000000030 R_MIPS_HI16       romInit
0000000000000034 R_MIPS_LO16       romInit
0000000000000040 R_MIPS_HI16       wrs_kernel_data_end
0000000000000044 R_MIPS_LO16       wrs_kernel_data_end
0000000000000048 R_MIPS_HI16       romInit
```

可以看到，在目标文件 bootInit.o 的重定位表中，已经明确规定了，在文本段.text 的 OFFSET 为 0x14 和 0x18 的地方，其中的值在链接的时候需要重新定位。有关重定位的具体细节，可以参考《程序员的自我修养-链接、装载与库》4.2 节-符号接卸与重定位。

将目标文件反汇编，使用指令 objdumpmips －d，例如：

objdumpmips －d bootInit.o>bootInit.asm

参看目标文件的重定位表，使用指令 objdumpmips －r，例如：

objdumpmips －r bootInit.o

5.3 特殊符号

在刚启动的过程中，将用到很多和编译器，连接器有关的特殊符号，下面以 romStart()函数来说明其中的一些原委。相关知识请参考《程序员的自我修养-链接、装载与库》3.5.2 节-特殊符号，4.6 节-链接过程控制。

下面以 romStart()函数中一段为例说明：

bootInit.c Line 386

```
……
#if   (CPU_FAMILY == MIPS)
      volatile FUNCPTR absUncompress = (FUNCPTR) UNCMP_RTN;


      if ((absUncompress) ((UCHAR *)ROM_OFFSET(binArrayStart),
                  (UCHAR *)K0_TO_K1(RAM_DST_ADRS),
                  (int)((UINT)&binArrayEnd - (UINT)binArrayStart)) != OK)
……
```

其中，binArrayStart 和 binArrayEnd 就是编译器中的特殊符号。将 tmp.z 利用工具 binToAsm 转化为汇编文件 bootrom.Z.s，其文件内容起始标签为 c，结束标签为 binArrayEnd 就由编译器确定了。

获得 bootrom 文件的符号表，输出到 bootrom_symbol.txt 中：

```
E:\source\bcm4704_BSP_v1>readelfmips -s bootrom>bootrom_symbol.txt
```

可以看到 binArrayEnd 等符号的值如下：

```
  142: 8015d6b0     0 OBJECT  GLOBAL DEFAULT  ABS wrs_kernel_data_end
  143: 80100478     0 OBJECT  GLOBAL DEFAULT    1 romReboot
  144: 801043b0     4 OBJECT  GLOBAL DEFAULT    2 runtimeName
  145: 8015d6b0     0 OBJECT  GLOBAL DEFAULT    3 wrs_kernel_bss_start
  146: 8015d15c     0 OBJECT  GLOBAL DEFAULT    2 _binArrayEnd
  147: 80100000     0 OBJECT  GLOBAL DEFAULT    1 _wrs_kernel_text_start
  148: 80100000     0 OBJECT  GLOBAL DEFAULT    1 romInit
  149: 80177330     0 OBJECT  GLOBAL DEFAULT  ABS wrs_kernel_bss_end
  150: 801043c0     0 OBJECT  GLOBAL DEFAULT    2 _binArrayStart
  151: 8015d47c     4 OBJECT  GLOBAL DEFAULT    2 inflateCksum
  152: 8015d15c     0 OBJECT  GLOBAL DEFAULT    2 binArrayEnd
  153: 8015d6b0     0 OBJECT  GLOBAL DEFAULT  ABS _edata
  154: 80177330     0 OBJECT  GLOBAL DEFAULT  ABS _end
  155: 801043c0     0 OBJECT  GLOBAL DEFAULT    2 binArrayStart
  156: 8015d160   163 OBJECT  GLOBAL DEFAULT    2 copyright_wind_river
  157: 801043b8     4 OBJECT  GLOBAL DEFAULT    2 vxWorksVersion
  158: 801008cc     0 FUNC    GLOBAL DEFAULT    1 displaymsg
```

5.4 获得调试用反汇编代码

在调试的过程当中，可能要快速的确定函数的地址，也可能要对汇编代码进行分析，这个时候就需要对目标代码或者可执行代码进行反汇编。在本次工作中，主要是 tmp.o 和 c。

tmp.o 获得的是从 usrInit()函数开始，从 RAM_HIGH_ADDR 开始执行的所有程序的反汇编代码。在默认情况下，编译得到 bootrom.bin 文件的时候，tmp.o 文件是被删除掉的。可以修改 rules.bsp 文件取消对 tmp.o 文件的删除。

将$(WIND_BASE)\target\h\make\rules.bsp 文件中，bootrom.Z.s 目标描述一段中的 -@$(RM)tmp.o 的行注释掉。

```
768 bootrom.Z.s : depend.$(BSP_NAME) bootConfig.o $(MACH_DEP) $(LDDEPS) \
769                         $(patsubst -l%,lib%.a,$(LIBS)) $(CC_LIB)
770         @ $(RM) $@
771 #- @ $(RM) tmp.o
772         - @ $(RM) tmp.out
773         - @ $(RM) tmp.Z
774         - @ $(RM) version.o
775         $(CC) $(OPTION_OBJECT_ONLY) $(CFLAGS) -o version.o \
776             $(CONFIG_ALL)/version.c
777         $(LD) -o tmp.o $(LDFLAGS) $(ROM_LDFLAGS) $(LD_ENTRY_OPT) $(USR
778             $(LD_HIGH_FLAGS) bootConfig.o version.o $(MACH_DEP) $(CC_L
779             $(LD_START_GROUP) $(LD_LINK_PATH) $(LIBS) $(LD_END_GROUP)
780             $(CC_LIB) $(LD_SCRIPT_RAM)
781         $(LDOUT_HOST) tmp.o
782         $(EXTRACT_BIN) tmp.o tmp.out
783         $(COMPRESS) < tmp.out > tmp.Z
784         $(BINTOASM) tmp.Z >bootrom.Z.s
785 #- @ $(RM) tmp.o
786         - @ $(RM) tmp.out
787         - @ $(RM) tmp.Z
```

重新编译一遍 bootrom，就会得到 tmp.o 文件。使用 objdummips –d 命令获得反汇编代码。

```
E:\source\bcm4704_BSP_v1>objdumpmips -d tmp.o>tmp.asm
```

tmp.o 的反汇编代码如下所示：

```
tmp.o:     file format elf32-bigmips

Disassembly of section .text:

81c00000 <compressedEntry>:
81c00000:       27bdffe8        addiu   $sp,$sp,-24
81c00004:       afbf0014        sw      $ra,20($sp)
81c00008:       afbe0010        sw      $s8,16($sp)
81c0000c:       03a0f021        move    $s8,$sp
81c00010:       afc40018        sw      $a0,24($s8)
81c00014:       3c1c81cf        lui     $gp,0x81cf
81c00018:       279ca3e0        addiu   $gp,$gp,-23584
81c0001c:       8fc40018        lw      $a0,24($s8)
81c00020:       0c70074f        jal     81c01d3c <usrInit>
81c00024:       00000000        nop
81c00028:       03c0e821        move    $sp,$s8
81c0002c:       8fbf0014        lw      $ra,20($sp)
```

反汇编可执行文件 bootrom 可以获得从 romInit()函数开始的所有在 flash 执行的汇编代码，这其中主要就是 romInit()函数和 romStart()函数。

5.5 PIC(Position Independent Code)

“romInit()函数必须设计成与地址无关的代码（PIC，Position Independent Code）”——《VxWorks 下设备驱动程序及 BSP 开发指南》.周启平等编著 P209。

- 什么是地址无关的代码（PIC）呢？

  在我们的开发中，本质是代码运行的地址（PC 寄存器）和链接时指定的地址（-Ttext）并不相同，结果导致我们代码中不能够出现绝对地址的引用等问题。

- 为什么 romInit()函数需要设计成 PIC 呢？

  这里，需要设计为 PIC 的原因应该是，romInit()，romStart()等在 flash 中执行的函数，在编译链接的时候，使用的-Ttext 为 RAM_LOW_ADRS（0x8010.0000），而却把代码放到了从 ROM_TEXT_ADRS（0xbfc0.0000）处，并且要从这里开始执行。

  这里的地址无关并不是说这份代码可以在 flash 中随意放置位置都可以运行（准确的说，不考虑 CPU 复位时的地址唯一这一点，应该是可以运行的，但是会发生问题）。关于 Status Register 的 BEV 位对应的地址 0xbfc00380 就论证了这一点。如果随意放置的话，那么发生异常时，如何找得到这个地址呢？

- 如何才能够设计成 PIC 呢？

  在 BSP 中，只有 bootInit.c 文件在编译的时候使用了选项-fpic。在“为什么 romInit()函数需要设计成 PIC”中提到，只有函数 romInit()和 romStart()运行地址和编译时确定的地址并不相同，所以，只有这两个函数对应的文件需要编译成 PIC，也就是 romInit.s 文件和 bootInit.c 文件。

  那么，为什么 romInit.s 文件无需这个编译参数呢？这是因为这是一个汇编文件，地址的确定都需要在代码中实现。

complieOutput.txt

```
……

ccmips -c -G 0 -mno-branch-likely -mips2 -EB -ansi -fno-builtin -O0 -Wall -I/h -I. -IE:\Tornado2.2.1-mips\target\config\all -IE:\Tornado2.2.1-mips\target/h -IE:\Tornado2.2.1-mips\target/h/wrn/coreip -IE:\Tornado2.2.1-mips\target/src/config -IE:\Tornado2.2.1-mips\target/src/drv -DCPU=MIPS32 -DTOOL_FAMILY=gnu -DTOOL=sfgnu -D_WRS_KERNEL -DMIPSEB -DSOFT_FLOAT -DMIPSEB -DSOFT_FLOAT -DBCM56218 -fpic -msoft-float E:\Tornado2.2.1-mips\target\config\all\bootInit.c

……
```

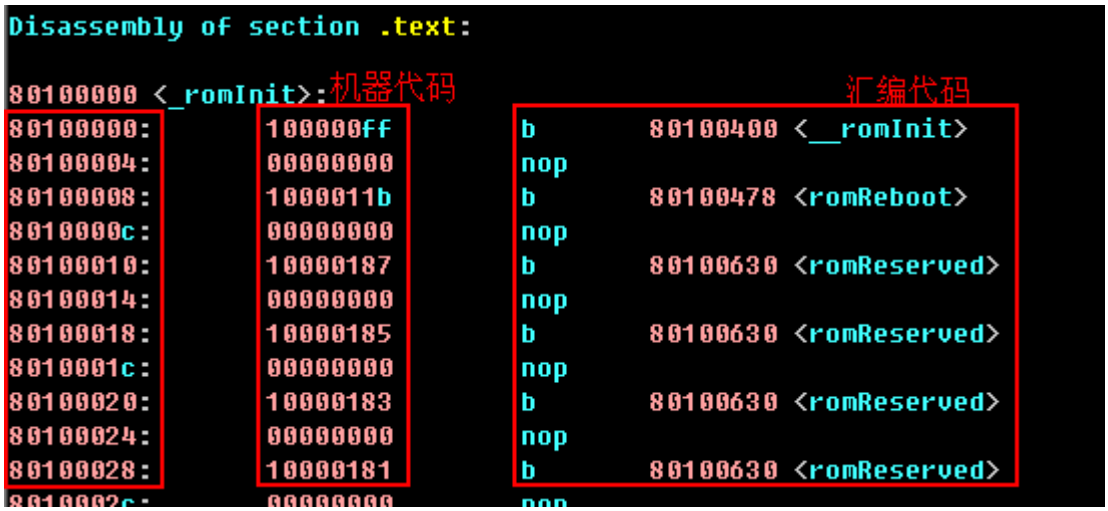
带着这些问题，下面重点分析 romInit()函数最后跳转到 C 语言代码 romStart()函数的相关代码。请参考《romInit()函数.docx》文档，最后对这一段代码有详细的注解。

有关 PIC 的相关知识，请参考《程序员的自我修养-链接、装载与库》的 7.3 节，地址无关代码。

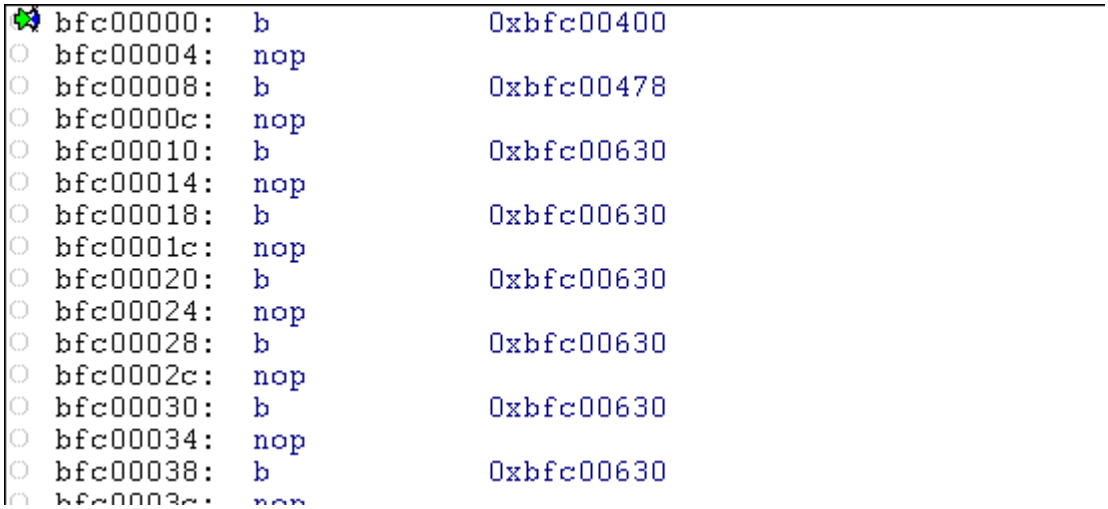
5.6 汇编代码与机器代码

以调试中的一个问题，引出汇编代码与机器代码的区别（Assembly Language and Machine Language）。

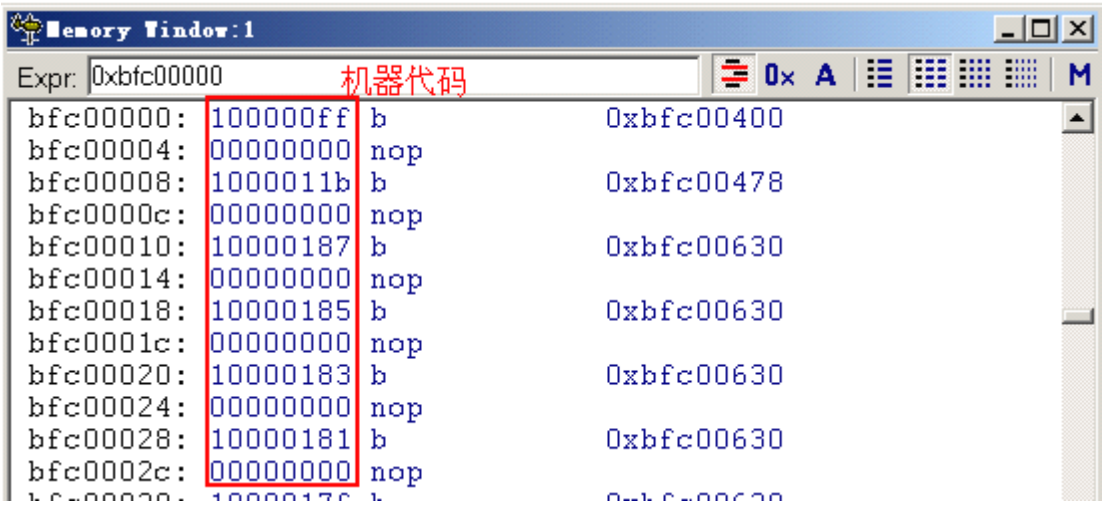
查看 bootrom 的反汇编代码，开头一段如下所示：



但是，使用 EPI 进行调试的时候，可以看到调试环境中显示的反汇编代码如下：



对比上下两个图，我们可以看到，地址不一样了。第一个图是链接时参数-Ttext 传入的地址，第二个图是代码运行时的运行地址。那么，是不是代码 b 0x8010.0400 在烧写到 flash 中之后，就变成了 b 0xbfc0.0000 呢？其实，这些都是汇编代码，是反汇编程序，或者调试器为了人阅读的方便，将相对地址跳转根据文本段地址或者当前的运行地址转化了绝对地址。



在 EPI 中，通过工具参看 0xbfc0.0000 处的二进制码，就可以看到，其实，在 flash 中的代码，就是我们编译时生成的机器码。

在考虑地址无关代码的时候，不要被表面上的汇编代码迷惑了。其实在 PIC 中，最重要的一点就是相对地址的设置。