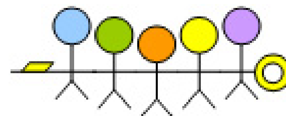


VxWorks 启动过程描述及主要宏开关含义

1 三种不同的 VxWorks 映象比较

VxWorks 是一种灵活的、可裁剪的嵌入式实时操作系统。用户可以根据需要创建自己的 VxWorks 映象，由它来引导目标系统，而后下载并运行应用程序。

根据应用场合的不同，VxWorks 映象可分为三类：可加载的 VxWorks 映象、基于 ROM 的 VxWorks 映象和驻留 ROM 的 VxWorks 映象。

1.1 可加载的 VxWorks 映象

这是一种运行于 RAM 的 VxWorks 映象。它不包含搬移程序，需要借助于一些外部的程序如 bootRom 才能加载到 RAM 的低端 RAM_LOW_ADRS 地址处。这是缺省的开发映象。

在开发的初期阶段，用户可以根据需要添加或删除一些 VxWorks 组件，生成自己的可加载的 VxWorks 映象，存放在开发主机的某个目录下。目标板上电后，由烧结在 BOOT 中的起始引导程序(BootStrap Programs)将 BOOT 中的 ROM 引导程序（ROM Boot Programs）拷贝到 RAM 的高端地址 RAM_HIGH_ADRS 处，并跳转至该地址执行 ROM 引导程序，配置好所选的加载方式（缺省为网络方式），将指定的主机目录下的可加载的 VxWorks 映象下载到目标板的 RAM 地址 RAM_LOW_ADRS 处，并跳转到此处执行。如图 1 所示。

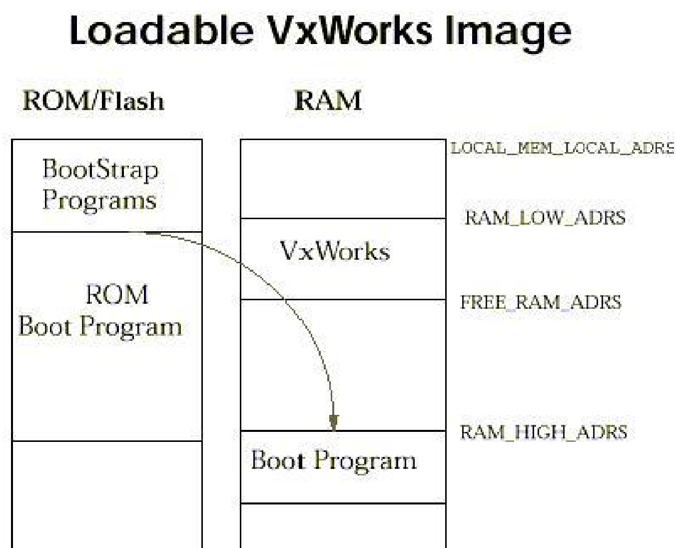


图 1、可加载的 VxWorks 映象

这种映象的优点是生成的 VxWorks 映象可以存放在开发主机 PC 机上，不用烧到 BOOT 中，节省了 BOOT 容量，也便于随时修改不同的 VxWorks 映象，适用于调试的初期阶段。不足之处是需要在主机上维护一个正确的 VxWorks 映象，对于调试硬件无关的上层应用程

序显得不是很方便。

在 Tornado 工作台的 Build 窗口中，选择 Rules 属性页中的 VxWorks 即可生成可加载的 VxWorks 映象。

1.2 基于 ROM 的 VxWorks 映象

这是一种运行于 RAM 中，但起初存放于 ROM 中的 VxWorks 映象。即该映象需要和搬移程序一起固化在 BOOT 中。目标板上电后，首先运行 BOOT 中的引导搬移程序，将整个 VxWorks 映象拷贝到 RAM 地址 RAM_LOW_ADRS 处，并跳转到此处执行。如图 2 所示。

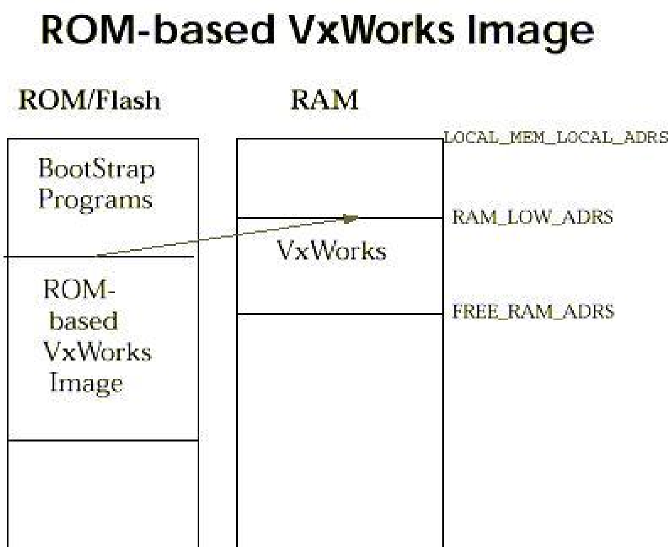


图 2 基于 ROM 的 VxWorks 映象

该映象根据是否被压缩又可分为：

- 基于 ROM 的未压缩的 VxWorks 映象，可直接从 ROM 拷贝到 RAM 中
- 基于 ROM 的压缩的 VxWorks 映象，这种映象主要是为了节约 BOOT 空间，在从 ROM 拷贝到 RAM 的过程中需要解压缩，因此与上述未压缩的映象相比，它的引导过程相对较慢，但两者在 RAM 中的运行速度是一样的。

1.3 驻留 ROM 的 VxWorks 映象

这种映象起初也和搬移程序一起固化在 BOOT 中。目标板上电后，首先运行 BOOT 中的引导搬移程序，但仅将 VxWorks 映象的数据段和 BSS 段拷贝到 RAM 地址 RAM_LOW_ADRS 处，映象的代码段仍旧留在 ROM 中，从 ROM 中开始执行。如图 3 所示。

这种映象的优点是具有最快的引导速度，占用最少的 RAM 空间，适用于 RAM 空间有限的目标板。但是由于该映象在 ROM 中运行，运行速度在三种映象中是最慢的。

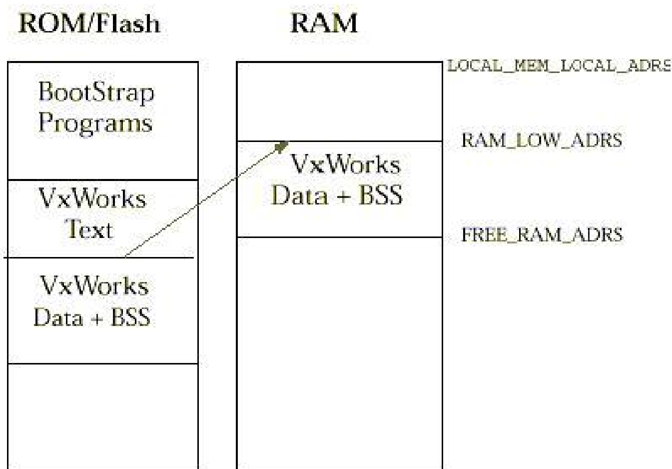


图 3 驻留 ROM 的 VxWorks 映象

2 几种不同的 BOOTROM 的比较

针对上述三种不同的 VxWorks 映象，可以生成以下几种不同的 BOOTROM，主要体现在执行搬移程序 romStart( )(位于 bootInit.c 文件中)时不同：

2.1 用于可加载 VxWorks 映象的 BOOTROM

由图 1 所示可知，用于可加载 VxWorks 映象的 BOOTROM 包含两部分：起始引导程序（BootStrap Programs)和 ROM 引导程序（ROM Boot Programs）。

起始引导程序驻留在 ROM 中，主要包含：

- 汇编级的硬件初始化程序 romInit.s，用于系统的基本初始化，设置一些重要寄存器的初始值，进行存储器的映射
- 搬移程序 **bootInit.c**，将 **ROM** 引导程序拷贝至 **RAM** 的高端地址 **RAM_HIGH_ADRS**，然后跳转到此处执行 **ROM** 引导程序。

ROM 引导程序起初存放在 ROM 中，初始化时被拷贝到 RAM 中，主要用于系统的进一步初始化，并配置加载方式，将 VxWorks 映象加载至 RAM。可分为三种不同的类型：

- 压缩的 ROM 引导程序，在拷贝的过程中需要解压缩，在 RAM 中执行
- 未压缩的 ROM 引导程序，可直接拷贝，在 RAM 中执行
- 驻留 ROM 的 ROM 引导程序，仅拷贝 ROM 引导程序的数据段，代码段仍旧在 ROM 中执行

在 Tornado 开发环境中，通过在主窗口点击 Build|Build Boot ROM…可以选择生成以上三种 BOOTROM，分别为：bootrom_uncmp.hex(未压缩的 BOOTROM)，bootrom.hex(压缩的 BOOTROM)，bootrom_res.hex（驻留的 BOOTROM）。

静态连接到可加载的 VxWorks 映象的系统初始化代码执行并完成整个初始化过程。

引导过程成功以后，RAM 中 ROM 引导程序占用的空间（从 RAM_HIGH_ADRS 开始）可以重新被系统利用。

图 1 中所示的各地址含义为：

- LOCAL_MEM_LOCAL_ADRS 是 RAM 的起始地址
- RAM_LOW_ADRS 是 VxWorks 的加载点，也是 VxWorks 代码段的起始位置
- FREE_RAM_ADRS 是 VxWorks 映象的结束点。通常也是系统内存池和目标服务器内存池的起始地址
- RAM_HIGH_ADRS 是 ROM 引导程序的加载点。它也是 ROM 引导程序（除驻留 ROM 引导程序之外）的代码段的起始位置，或驻留 ROM 引导程序数据段的起始位置。

2.2 用于基于 ROM 的 VxWorks 映象的 BOOTROM

由图2所示可知，用于该映象的BOOTROM包含两部分：起始引导程序（BootStrap Programs)和基于ROM的VxWorks映象。搬移程序bootInit.c负责将**VxWorks**映象的文本段和数据段搬移到用户定义的低端内存地址RAM_LOW_ADRS，如果需要进行必要的解压缩，然后直接启动VxWorks映像。

因此 BOOTROM 的容量相对于 2.1 中描述的 BOOTROM 要大一些，但无需在主机目录下维护一个可用的 VxWorks 映象。

基于 ROM 的 VxWorks BOOTROM 有压缩和未压缩之分。在 Tornado 工作台的 Build 窗口中，选择 VxWorks 映象 Rules 属性页中的 VxWorks_rom 即可生成基于 ROM 的未压缩的 VxWorks BOOTROM，选中 VxWorks_romCompress 即可生成基于 ROM 的压缩的 VxWorks BOOTROM。

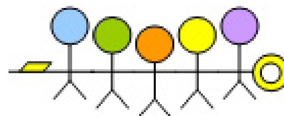
2.3 用于驻留 ROM 的 VxWorks 映象的 BOOTROM

由图 3 所示可知，用于该映象的 BOOTROM 包含两部分：起始引导程序（BootStrap Programs)和驻留 ROM 的 VxWorks 映象，VxWorks 系统文本段驻留在 ROM，搬移程序 bootInit.c 负责将数据段和 bss 段搬移到用户定义的低端内存地址 RAM_LOW_ADRS，直接启动 VxWorks 映像（含符号表）。此时，RAM_LOW_ADRS 是 VxWorks 映象的加载点，它也是 VxWorks 数据段的起始点。

在 Tornado 工作台的 Build 窗口中，选择 VxWorks 映象 Rules 属性页中的 VxWorks_romResident 即可生成驻留 ROM 的 VxWorks BOOTROM。

3 VxWorks 的启动过程

根据上述所采用的 BOOTROM 的不同，VxWorks 的启动过程会有所不同，下面主要介绍一下使用可加载 VxWorks 映像的启动过程。此时，从目标板上电复位到启动用户定义的任务的整个流程如下：



BootStrap 程序
在 ROM 中执行

romInit.s : **romInit**
设置机器状态字及其它硬件相关寄存器，关闭中断，禁止程序和数据 CACHE，初始化内存，并设置堆栈指针

bootInit.c : **romStart( )**
将 ROM 中的程序搬移至 RAM 中

ROM Boot 程序
被搬移到 RAM 中执行

bootConfig.c : **usrInit( )**
设置 cache 的工作模式，板级硬件初始化，调用 sysHwInit( ),usrKernelInit( ),KernelInit( ),初始化 Win 内核，产生根任务 usrRoot( )

bootConfig.c : **usrRoot( )**
初始化内存，系统时钟，I/O 系统，标准输入输出，异常处理，外围设备初始化，产生任务 bootCmdLoop

bootConfig.c : **bootCmdLoop( )**
调用自动引导程序 autoboot( )，此函数若成功则不返回

bootConfig.c: **autoboot( )**
延时 7s，以默认参数启动

用户按键中断

bootConfig.c :**bootCmdLoop( )**
启动命令行用于配置 VxWorks 启动参数

等待超时

bootConfig.c: **bootLoad( )**
加载 VxWorks 映象，并转向它进行重启

用户输入‘@ ’

开始在 RAM 中运行
VxWorks

sysALib.s : **sysInit( )**
锁住中断，关闭 cache（如果使用了话），初始化处理器的寄存器（包括 C 堆栈指针）至缺省值

usrConfig.c : **usrInit( )**
设置 cache 的工作模式，板级硬件初始化，初始化 Win 内核，启动 usrRoot( )

usrConfig.c : **usrRoot( )**
初始化内存，系统时钟，I/O 系统，标准输入输出错，异常处理，添加用户应用程序

3.1 可加载 VxWorks 映象的 BOOT 过程

Boot 中几个关键宏定义：

#define LOCAL_MEM_LOCAL_ADRS 0x00000000

#define ROM_TEXT_ADRS 0x100 ROM Boot 程序执行起始地址(romStart( ))

#define ROM_OFFSET(adr) (((UINT)adr - (UINT)romInit) + ROM_TEXT_ADRS)

define BOOT_LINE_OFFSET 0x1200

#define BOOT_LINE_ADRS ((char *) (LOCAL_MEM_LOCAL_ADRS+BOOT_LINE_OFFSET))

ldFileFromMch 时，从该起始地址读取加载要用到的 tBootParams

define RAM_LOW_ADRS 0x10000 boot Rom 将控制权交给 VxWorks 的起始进入点 (usrInit( )). boot 启动之后，将系统映像从 Flash 上 copy 或解压到 RAM_LOW_ADRS 地址处，并跳转到该地址执行

define VERSION_START_ADRS 0x10000 版本加载完后的执行入口地址.

define FREE_RAM_ADRS (end) start right after bss of VxWorks

define FREE_MEM_START_ADRS (FREE_RAM_ADRS + WDB_POOL_SIZE)

bootRom 中的 pMemPoolStart

在本系统中, BPC 以及 FS 单板的启动流程可概括如下:

3.1.1 BOOTROM 的启动过程

1、 romInit.s

目标板加电之后，程序指针指向 RESET 中断程序入口处，开始执行初始化程序 romInit.s，设置机器状态字及其它硬件相关寄存器，关闭中断，禁止程序和数据 CACHE，初始化内存，并设置堆栈指针，保存启动类型，调用 romStart.c 中的 romStart( )。

The default base address for the internal memory map register(IMMR) is 0xFF40_0000. Because IMMRBAR is at offset 0x0 from the beginning of the local access registers, IMMRBAR always points to itself., 在romInit中首先会将这个默认值改为我们自定义的一个内存映像基地址,在本系统中这个歌基地址定义为0xE0000000。

System Configuration Registers			
0x0_0000	IMMRBAR—Internal memory map base address register	R/W	0xFF40_0000

```
/* initialize the IMMR register */
lis  r4, HI (CCSBAR)
ori  r4, r4, LO (CCSBAR)
lis  r8,HI(CCSBAR_INIT)  /* IMMR was at 0xff400000 */
ori  r8, r8, LO(CCSBAR_INIT) /* IMMR now at CCSBAR */
stw  r4,0(r8)
isync
```

```
/*set windows size and attribute, config boot at 0xfff00000~0xffffffff */
 WRITEADR(r6,r7,M83XX_LBLAWBARn(CCSBAR,0), 0x80000000)
 WRITEADR(r6,r7,M83XX_LBLAWARn(CCSBAR,0),(LAWAR_ENABLE|LAWAR_SIZE_2GB ))
/* DUART 波特率设定 0x6c8 for 266M */
 WRITEADRB(r17,r18,MPC83XX_UDMB1(CCSBAR),0x06)
 WRITEADRB(r17,r18,MPC83XX_UDLB1(CCSBAR),0xc8)
```

SPMF 4:1 ie 4*66.67Mhz = 266Mhz CSB. the desired baud rate = platform clock frequency / (16 x [UDMB||UDLB]= 266M/（16×0x06c8）=9600

2、 romStart( )

sync 程序跳到第一个 C 程序 romStart.c 的函数 romStart( )入口地址，根据堆栈中的参数决定是否清零内存 RAM（如是冷启动 cold start,则清零），根据不同的 bootRom 文件，把 ROM 中数据段和文本段拷贝到 RAM（如果 ROM 代码是压缩的，还要解压）；

3、 usrInit( )

程序跳到 RAM 入口地址(usrConfig.c 中的函数 usrInit( ))，usrInit( )中清零 bss 段（这也是未赋初始值的全局变量在编译后初始值为 0 的原因），调用 excVecInit() 安装异常向量（excVecInit 会将 excIntHandle 注册到相应的异常上），初始化异常处理程序，调用 cacheLibInit（）,设置 cache 的指令与数据工作模式，调用 sysHwInit( )对板级硬件初始化，调用 usrKernelInit( )配置 wind Kernel, 调用 KernelInit( ) 进行内核初始化。

4、 KernelInit( )

初始化内核及内存池，主要是中断堆栈及根任务堆栈初始化，初始化任务 Tcb 并生成根任务 usrRoot( )。

```
*kernelInit - initialize the kernel. The routine kernelRoot() is called before the user's root routine so
*that memory management can be initialized.
*The memory setup is as follows For _STACK_GROWS_DOWN:
*         - HIGH MEMORY -
*         ---------------------------------- <--- pMemPoolEnd
*         |                                |   We have to leave room for this block headers
*         |       1 BLOCK_HDR              |      so we can add the root task memory to the pool.
*         ----------------------------------
*         |        WIND_TCB                |
*         ---------------------------------- <--- pRootStackBase;
*         |       ROOT STACK               |
*         ----------------------------------
*         |    1 FREE_BLOCK                |   We have to leave room for these block headers
*         |    1 BLOCK_HDR                 |      so we can add the root task memory to the pool.
*         ---------------------------------- <--- pRootMemStart;
*         ----------------------------------
*         ~     FREE MEMORY POOL          ~      pool initialized in kernelRoot()
*         ---------------------------------- <--- pMemPoolStart + intStackSize; vxIntStackBase
```

```
*        |  INTERRUPT STACK    |
*        ----------------------------------- <--- pMemPoolStart; vxIntStackEnd
*              - LOW   MEMORY -
```

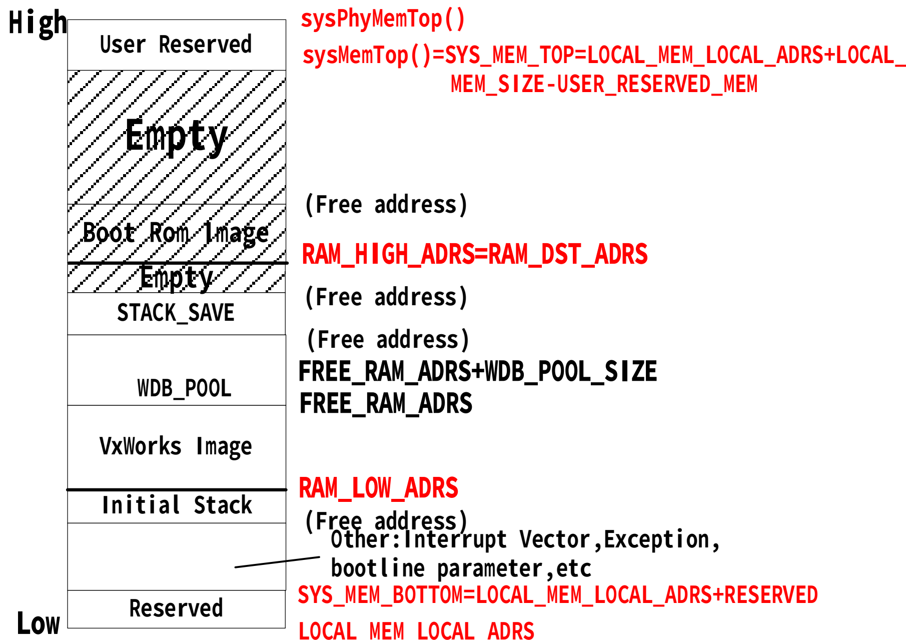
kernelInit( ) 函数参数包括：

```
FUNCPTR    rootRtn,    /* user start-up routine */
Unsigned    rootMemSize, /* memory for TCB and root stack */
char *pMemPoolStart,  /* beginning of memory pool */
char *pMemPoolEnd,   /* end of memory pool */
unsigned    intStackSize,     /* interrupt stack size */
int      lockOutLevel     /* interrupt lock-out level (1-7) */
```

- rootRtn: 用以产生作为根任务的应用程序，典型的为 usrRoot( )
- rootMemSize: 使用的堆栈大小
- pMemPoolStart: 可用的起始内存地址，位于 VxWorks 映象(bootRom 启动过程中，位于从 ROM 中搬迁过来的 boot 镜像)的代码段，数据段和 bss 段之后，如果包含可选的主机内存池，则还要加上 WDB_POOL_SIZE。
- pMemPoolEnd: 由 sysMemTop( )定义的内存顶部
- intStackSize: 中断堆栈的大小
- lockOutLevel: 中断封锁级别

kernelInit( ) 调用 intLockLevelSet( )，关闭循环模式，创建一个中断堆栈（如果结构支持的话）。然后从内存池的顶部创建一个根堆栈和 TCB，创建一个根任务 usrRoot，并终止 usrInit( )线程的执行。此时使能中断，所有的中断源已被关闭，未决中断已被清除。

VxWorks RAM 空间分配图，斜线部分可以被 malloc 申请：



VxWorks for PowerPC 的内存分配图：

- **Interrupt Vector Table.** Table of exception/interrupt vectors.
- **SM Anchor.** Anchor for the shared memory network and VxMP shared memory objects (if there is shared memory on the board).
- **Boot Line.** ASCII string of boot parameters.
- **Exception Message.** ASCII string of the fatal exception message.
- **Initial Stack.** Initial stack for **usrInit( )**, until **usrRoot( )** is allocated a stack.
- **System Image.** The VxWorks system image itself (three sections: text, data, and bss). The entry point for VxWorks is at the start of this region, which is BSP dependent (see BSP-specific documentation).
- **Host Memory Pool.** Memory allocated by host tools. The size depends on the the macro **WDB_POOL_SIZE**. Modify **WDB_POOL_SIZE** under **INCLUDE_WDB**.
- **Interrupt Stack.** Size is defined by **ISR_STACK_SIZE** under **INCLUDE_KERNEL**. Location depends on system image size.
- **System Memory Pool.** Size depends on the size of the system image. The **sysMemTop( )** routine returns the address of the end of the free memory pool.

PowerPC 体系结构的内存结构包括 5 大部分，分别为系统映像（System Image）之前的系统启动相关的低端内存，系统映像，Host Memory Pool，中断堆栈以及系统内存池（System Memory Pool）。下面就各部分进行介绍。

1. 系统映像之前的低端内存

包括中断向量表（Interrupt Vector Table），共享内存标志（SM Anchor），启动参数（Boot Line），异常信息（Exception Message）和初始化堆栈（Initial Stack）。

中断向量表（异常向量表）占据 0x0 到 0x3000 地址的 12KB 的空间，保存有重要的中断向量信息；

共享内存标志占据 0x4100 到 0x4200 地址的 100 字节，它的作用是标志是否有网络共享内存和 VxMP 共享内存对象；

启动参数是保留 VxWorks 启动的时候所用的参数，如：

qefcc(0,0)host:vxWorks h=192.1.1.1 e=192.254.0.4 u=cca pw=cca tn=cca

异常信息，起始地址是 0x4300，如果启动过程中出现致命异常，则系统将异常信息保留在这段内存中。如果系统启动过程中失败，我们首先要看的是这段地址中记录的异常信息，可以使用 d 0x4300 命令查看其中记录的内容。

初始化堆栈，是给 usrInit()使用的初始化堆栈，直到 usrRoot()分配堆栈。起始地址是 0x4C00。

2. 系统映像

系统映像是 ELF 格式的文件，boot 启动之后，将系统映像(boot 映像或版本映像)从 Flash 上 copy 或解压（如果是压缩版本）到 RAM_LOW_ADRS 地址处，并跳转到该地址执行。

系统映像包括三部分：TEXT 段、DATA 段、BSS 段。其中 TEXT 段是代码段，使用的内存基本是必须的；DATA 段是数据段，包括已经初始化的全局变量和数组；而 BSS 段是未初始化的数据段，包括未初始化的全局变量和数组，实

际上基本不占用 Flash 存储空间，在 VxWorks 系统启动的时候在内存将其进行扩展为全零。

代码段的起始地址：RAM_LOW_ADRS，终止地址：VxWorks 定义的 char etext[]；

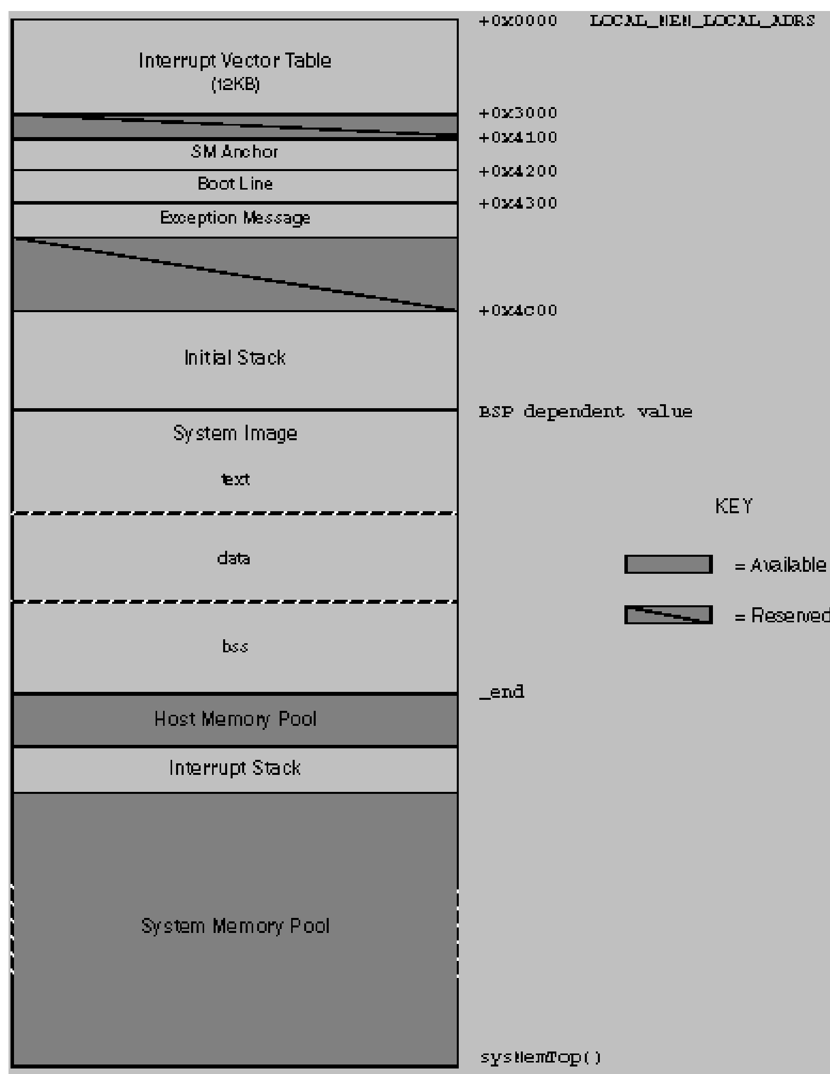
数据段的起始地址：VxWorks 定义的 char etext[]，终止地址：VxWorks 定义的 char edata[]；

BSS 段的起始地址：VxWorks 定义的 char edata[]，终止地址：VxWorks 定义的 char end[]。

FREE_RAM_ADRS 指向 VxWorks 定义的 char end[]，即 BSS 段的最后，也是映像的最后。

end 是由 loader 在动态加载时确定的，从源码里找不到。首先取得 end 变量的地址，再减去低 RAM_LOW_ADRS 的空间，即得到系统映像的大小：

dwImageSize = (WORD32)end - RAM_LOW_ADRS;



3. Host Memory Pool

Host Memory Pool 是在 VxWorks 上驻留的调试工具使用的内存空间，可以根

据 WDB_POOL_SIZE 宏值得到。该部分大小一般有十几 M 左右。

起始地址：VxWorks 定义的 char end[]，终止地址：end+WDB_POOL_SIZE。

4. 中断堆栈：中断堆栈的大小可以由宏 ISR_STACK_SIZE 定义可以得出。

5. 系统内存池

这部分是给 VxWorks 用户程序使用的存储空间，用户通过 malloc 动态申请获得，这部分可以说是最大的内存空间，当物理内存不够需要优化时需要重点考虑。

5、 usrRoot( )

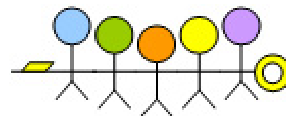
本系统的根任务函数 usrRoot( )在 prjConfig.c 中。在该任务中初始化内存，系统时钟，**I/O** 系统，标准输入输出错，异常处理，外围设备等。BPC 初始任务 usrRoot 具体所处理的内容如下：

```
void usrRoot (char *pMemPoolStart, unsigned memPoolSize)
{
    excIntNestLogInit();
    vxMsrSet(vxMsrGet() | taskMsrDefault);        /* Enable interrupts at appropriate point in root task */
    usrKernelCoreInit ();                         /* core kernel facilities */
    memInit (pMemPoolStart, memPoolSize);         /* full featured memory allocator */
    memPartLibInit (pMemPoolStart, memPoolSize);  /* core memory partition manager */
    usrMmuInit ();                                /* basic MMU component */
    sysClkInit ();                                /* System clock component */
    selectInit (NUM_FILES);                       /* select */
    usrIosCoreInit ();                            /* core I/O system */
    usrKernelExtraInit ();                        /* extended kernel facilities */
    usrIosExtraInit ();                           /* extended I/O system */
    usrNetworkInit ();                            /* Initialize the network subsystem */
    selTaskDeleteHookAdd ();                      /* install select task delete hook */
    usrToolsInit ();                              /* software development tools */
    cplusCtorsLink ();   /* run compiler generated initialization functions at system startup */
    usrAppInit ();       /* call usrAppInit() (in your usrAppInit.c project file) after startup. */
}
```

最后会调用 usrAppInit.c 中的 usrAppInit ( )进行用户级应用模块的初始化。

6、 usrAppInit( )

usrAppInit()会调用 BspFlashInit( )以及 usrMiiWrite( )对 Flash 以及 PHY 进行初始化；调用 MacIpReset( )获取在 FLASH 中设置的本板 IP 地址值，并把本板 IP 地址设置为该值；调用 extUartPPPInit( )初始化 tty 设备；调用 BootBoardInit( )（对主控板初始化文件系统、受控板初始化 flash、通讯串口、获取槽位号、IP 地址信息等）；usrAppInit( )最后会调用 BootApp.c 中的 BootAppStart( )进入 autoboot 或 bootLoad，根据单板设计选择不同方式加载 **VxWorks** 映像文件，如通



过串口、网口、硬盘等方式加载。

7、 BootAppStart( )

BootAppStart 中调用 Autoboot(); Autoboot 中调用 bootLoad(), bootLoad 会根据 *BootStartModeFlag = (ULONG *)(BOOT_MODE_ADDR)当中所存放的 boot 启动模式进入相应的加载模式，AMC 在正常启动模式中调用 normalBoot(); normalBoot 中使用看门狗，创建喂狗任务，然后调用 ldFileFromMch()从主控板上下载版本；ldFileFromMch 首先获取本地 IP 和主控板 IP，初始化 UDP 设备，申请发送缓存和接收缓存，初始化请求版本消息体，发送版本请求并接收请求版本应答消息，挂起喂狗任务并使用硬件喂狗，开始调用 netLoadBin()用 FTP 从 Mch 上

romStart.c:romStart()
usrConfig.c: usrInit()
excArchShow.c: excVecInit() ;
cacheLib.c: cacheLibInit() ;
sysLib.c: sysHwInit() ;
usrKernel.c: usrKernelInit() ;
KernelLib.c: KernelInit() ;
prjConfig.c: usrRoot() ;
usrAppInit.c: usrAppInit ();
BootCommon.c:BootBoardInit()
BootApp.c: BootAppStart();
bstop 启动命令行用于配置VxWorks启动参数
延时10s，以默认参数启动
BootApp.c: Autoboot()
BootApp.c: bmenu()
BootApp.c: bootLoad()
ulBootMode = GetBootMode();
RELEASE_VER_MODE
VERSION_DOWNLOAD_MODE
FTPSERVER_DOWNLOAD_MODE
DEBUG_VER_MODE
NormalBoot()
ldBootLoadFile()
从主机加载
FtpServerLoadFile()
FTPServer烧录模式
netLoad()
BspGetIsCtrlBoard()?
Y
N
ldFileFromFlash()
ldFileFromMch()
netLoadBin()

下载二进制映像版本，校验软件类型是否是 CPU 软件，计算版本的 CRC 校验值，将接收到的版本去版本头后，压缩数据解压缩到 RAM_LOW_ADRS 处，最终在*pEntry = (FUNCPTR)VERSION_START_ADRS 处开始启动运行加载后的 VxWorks 版本.

3.1.2 VxWorks 映象的启动过程

1. VxWorks 进入点 sysInit()

启动 VxWorks 系统的第一步就是将系统映象加载到主内存，在开发初期，这通常是在 VxWorks boot Rom 的控制下，从开发主机上下载，我们正式运行的系统中, AMC 的 wxworks 版本储存在主控板的 Flash 电子盘上，版本会从电子盘下载。VxWorks 映象被加载到 ram 后，boot Rom 会复位系统并将控制权交给 VxWorks 的起始进入点 sysInit()。 在 makefile 和 config.h 文件里，已将这个进入点设置成位于地址 RAM_LOW_ADRS。

函数 sysInit()位于系统特定的汇编语言模块 sysALib.s 中。它可以锁住中断，关闭 cache（如果使用了话），初始化处理器的寄存器（包括 C 堆栈指针）至缺省值。它还会关闭跟踪，清除所有未决的中断，并调用一个位于 usrConfig.c 模块的 C 语言子程序：usrInit()。对于某些目标板，sysInit()还执行一些必要的与系统有关的硬件初始化，以便在 usrInit()中执行完剩余的初始化内容。仅供 usrInit()使用的初始堆栈指针，被设置成位于系统映象(RAM_LOW_ADRS)以下，向量表以上的位置。

2. 初始化代码 usrInit()

函数 usrInit()（位于 usrConfig.c 中），储存有关引导类型的信息，处理在内核启动之前必须执行的初始化，而后启动内核执行。它是运行于 VxWorks 内的第一个 C 函数。此时，所有的中断都已被锁住。

许多 VxWorks 工具在 usrInit( )中都不能使用。这是因为此时还没有任务的上下文（没有 TCB 和任务堆栈），那些需要任务上下文的工具无法被调用。函数 usrInit( )仅做一些创建初始化任务 usrRoot( )所必须的工作。然后由 usrRoot( )完成启动过程。

usrInit( ) 中的初始化过程如下所述：

Cache 初始化

usrInit( ) 的起始代码初始化 cache，设置 cache 模式，并将 cache 放置在一个安全的位置。在 usrInit( )结束时，缺省情况下，指令 cache 和数据 cache 被使能。

对系统的 BSS 段清零

C 和 C++语言规定所有未初始化的变量缺省的初始值为零。这些未初始化的变量被放置在一个称为 bss 的段内。由于 usrInit( )是系统执行的第一个 C 代码，在它的一开始对包含 bss 段的内存清零。VxWorks 的 boot ROM 也会清内存，但 VxWorks 映象假设没有采用 boot ROM，仍然执行清内存的操作。

初始化中断向量

异常向量必须在使能中断和启动内核之前建立。首先，调用 intVecBaseSet( ) 建立向量表基地址。而后，调用 excVecInit( ) 初始化所有的异常向量至缺省句柄，以便安全地捕获和报告由程序错误或意外的硬件中断导致的异常。

初始化硬件至静止状态

通过调用系统相关函数 sysHwInit( )初始化系统硬件。该函数复位并关闭那些在中断使能（内核启动时）以后可能产生中断的硬件设备。这一点很重要，因为 VxWorks ISRs（用于 I/O 设备，系统时钟等）直到在任务 usrRoot( )中完成系统初始化以后，才被连接到它们的中断向量上。不要在 sysHwInit( ) 调用中试图为一个中断连接一个中断句柄（也就是不能使用 intConnect( )），因为此时内存池还没有初始化。

初始化内核

函数 usrInit( )结束时调用了两个内核初始化函数：

usrKernelInit( )(在 usrKernel.c 中定义)为每个指定的可选内核组件调用合适的初始化代码。详见表 1。

kernelInit( ) (kernelLib.c 的一部分)初始化多任务环境，不用返回。

初始化内存池

内存池的初始化是由 kernelInit( )来完成的。 kernelInit( ) 的参数指定了初始内存池的起始和终止地址。在缺省的 usrInit( )中，将内存池设置在紧接于引导的系统映象之后，并包含所有剩余的可用内存。

可用内存的大小由 sysMemTop( )决定。如果你的系统有其它的不连续的内存片，你可以在 usrRoot( )任务中通过调用 memAddToPool( )将它们包含进通用的内存池。

VxWorks 包含了一个位于 memPartLib 模块中的内存分配工具，它管理一个可用内存池。用户可以调用 malloc( )函数从内存池中获得可变大小的内存块。VxWorks 也利用 malloc( )函数来动态分配内存。许多 VxWorks 工具在初始化过程中需要分配数据结构。因此，内存池必须在任何其他的 VxWorks 工具初始化之前初始化。

Tornado 目标服务器也管理一部分目标内存以支持目标模块的下载和其他开发功能。VxWorks 使用 malloc( )函数为已下载的模块分配空间，为已产生的任务分配堆栈，在初始化时分配数据结构。用户也可以使用 malloc( )函数为自己的应用程序分配所需的内存空间。因此，推荐将所有的未用内存分配给 VxWorks 内存池，除非必须为一个特殊的应用保留一片固定的绝对内存。

3. 初始任务 usrRoot( )

当多任务内核启动执行以后，所有的 VxWorks 多任务工具就可以用了。控制权被传送至 usrRoot( )任务，并完成初始化系统。

usrRoot( ) 执行以下操作：

- 初始化系统时钟
- 初始化 I/O 系统和驱动
- 创建控制台设备
- 设置标准输入和标准输出
- 安装异常处理和登陆
- 初始化管道驱动器
- 初始化标准 I/O
- 创建文件系统设备并安装磁盘驱动器
- 初始化浮点支持
- 初始化性能监视工具
- 初始化网络
- 初始化可选的工具
- 初始化 WindView
- 初始化目标代理
- 执行一个用户提供的启动脚本
- 初始化 VxWorks Shell

下面对各个步骤进行详尽的描述：

初始化系统时钟

usrRoot( ) 任务执行的第一个操作就是初始化 VxWorks 时钟。通过调用 sysClkConnect( )
将系统时钟的中断向量连接到 usrClock( )函数上。调用 sysClkRateSet( )将系统时钟率设置为 60Hz。

sysClkConnect( ) 函数调用 sysHwInit2( )。风河的 BSP 采用 sysHwInit2( )执行在 sysHwInit( )中未完成的进一步的板级初始化。例如，可以利用 intConnect( )连接 ISR，因为此时已经分配了内存，系统处于多任务环境。

初始化 I/O 系统

如果在 configAll.h 中定义了 INCLUDE_IO_SYSTEM，就可以调用 iosInit( )函数初始化 VxWorks 的 I/O 系统。该函数的参数指定了可被顺序安装的最大驱动器的数目，可以在系统中同时打开的最大文件数目，和 VxWorks 的 I/O 系统包含的“空”设备的名字。

包含或去除 INCLUDE_IO_SYSTEM 还会影响是否创建控制台设备，是否设置标准的输入、输出和标准的出错信息。

创建控制台设备

如果包含了板上串口驱动器（定义了 INCLUDE_TTY_DEV），就可以通过调用驱动器的初始化函数（典型的是 ttyDrv( )）将它安装进 I/O 系统。实际的设备是通过调用驱动器的设备创建函数（典型的是 ttyDevCreate()）来创建和命名的。这个函数的参数包括设备名称，一个串行 I/O 通道描述字（从 BSP 获得），和输入输出缓存大小。

宏 NUM_TTY 定义了 tty 口的数量（缺省是 2）。宏 CONSOLE_TTY 指定了哪个口作为控制台口（缺省是 0），宏 CONSOLE_BAUD_RATE 指定了其比特率（缺省是 9600 bps）。这些宏都在 configAll.h 中定义，但对于那些具有非标口数

的单板可以在 config.h 中对它们进行重新定义。

设置标准输入、标准输出和标准出错信息

系统级的标准输入、标准输出和标准错误信息的配置是通过打开控制台设备并调用 ioGlobalStdSet( )来建立的。这些配置作为 VxWorks 的缺省设备用于与应用开发人员通讯。为了使控制台设备成为一个交互式的终端，调用 ioctl( )将设备选项设为 OPT_TERMINAL。

安装异常处理和登录

初始化 VxWorks 的异常处理工具（由 excLib 模块提供）和登录工具(由 logLib 库提供)。这些工具检查在根任务内部或者初始化各种工具时产生的程序错误。

当定义了宏 INCLUDE_EXC_HANDLING 和 INCLUDE_EXC_TASK 后，调用 excInit( )初始化异常处理工具。excInit( ) 函数产生一个异常支持任务 excTask( )。初始化以后，可以安全地捕获和报告导致硬件异常的程序错误，报告并解除没有初始化向量的中断。当定义了 INCLUDE_SIGNALS 后，调用 sigInit( )初始化 VxWorks 的信号工具，该工具用于任务的异常处理。

当定义了 INCLUDE_LOGGING 宏以后，调用 logInit( )初始化登录工具。其参数定义了显示登录信息的设备的文件描述字，和分配的登录信息缓存数。登录初始化还创建了一个登录任务 logTask( )。

初始化管道驱动

如果需要所谓的管道，在 configAll.h 中定义 INCLUDE_PIPE，就会自动地调用 pipeDrv( )初始化管道。而后任务就可以利用管道通过标准的 I/O 接口互相通讯了。管道必须由 pipeDevCreate( ) 函数创建。

初始化标准 I/O

当定义了宏 INCLUDE_STDIO 以后，VxWorks 就会包含一个可选的标准 I/O 包。

创建文件系统设备并初始化设备驱动

许多 VxWorks 配置至少包含一个磁盘驱动器，或带有 dosFs/rt11Fs/rawFs 文件系统的 RAM 磁盘。首先，通过调用驱动器的初始化代码安装一个磁盘驱动器。而后，驱动器的设备创建代码会定义一个设备。这个调用会返回一个指向描述设备的 BLK_DEV 结构的指针。

然后就可以调用文件系统的设备初始化代码－dosFsDevInit( ), rt11FsDevInit( ), or rawFsDevInit( )（如果定义了宏 INCLUDE_DOSFS, INCLUDE_RT11FS 和 INCLUDE_RAWFS）初始化和命名设备。在初始化一个设备之前，必须用 dosFsInit( ), rt11FsInit( )或 rawFsInit( )初始化文件系统模块。文件系统的设备初始化函数的参数取决于特定的文件系统，但典型的包括设备名称，由驱动器的设备创建代码产生的一个指向 BLK_DEV 结构的指针，可能还有一些文件系统特定的配置参数。

初始化浮点支持

如果在 configAll.h 中包含了 INCLUDE_FLOATING_POINT 宏定义，则调用 floatInit( )函数初始化浮点 I/O 支持。当定义了 INCLUDE_HW_FP，调用 mathHardInit( )初始化对浮点协处理器的支持。当定义了 INCLUDE_SW_FP，调用 mathSoftInit( )初始化对软件浮点仿真的支持。

包含性能仿真

VxWorks 具有两个内嵌的性能监视工具。一个由 spyLib 提供的任务活动综述，一个由 timexLib 提供的子程序执行定时器。如果在 configAll.h 中定义了宏 INCLUDE_SPY 和 INCLUDE_TIMEX，就会包含这些工具。

初始化网络

如果配置头文件中定义了 INCLUDE_NET_INIT，usrRoot( )就会调用 usrNetInit( ) 函数初始化网络（usrNetInit( ) 的源代码位于 installDir/target/src/config/usrNetwork.c）。usrNetInit( )函数使用了一个配置字符串作为它的参数。这个配置字符串通常是一条“引导行”，用于 VxWorks 的 boot ROM 引导系统。根据这个字符串，usrNetInit( )函数执行以下操作：

- 调用 netLibInit( )初始化网络子系统
- 连接并配置合适的网络驱动器
- 添加网关路由
- 初始化远程文件存取驱动器 netDrv，并添加一个远程文件存取设备
- 初始化远程登录工具
- 可选地初始化远端程序调用（RPC）
- 可选地初始化网络文件系统（NFS）工具

如前所述，是否包含这些网络工具由 configAll.h 中的宏定义决定。

初始化可选产品和其它组件

可选产品 VxMP 可提供共享内存目标。如果定义了宏 INCLUDE_SM_OBJ，usrRoot( )
就会调用 usrSmObjInit( )函数（源代码位于 installDir/target/src/config/usrSmObj.c)，初始化共享内存目标。

共享内存目标库需要 VxWorks 引导行中的域值。这些函数包含在 usrNetwork.c 文件中。如果不包含网络服务，usrNetwork.c 就不会被包含，共享内存初始化就会失败。工程工具计算所有的依存关系，但如果使用手工配置，可以将 INCLUDE_NETWORK 添加进 configAll.h，或是从 usrNetwork.c 文件中将引导行代码提取出来放置到其他地方。

如果定义了 INCLUDE_MMU_BASIC，就可以提供基本的 MMU 支持。如果定义了 INCLUDE_MMU_FULL，可选产品 VxVMI 就可以提供代码保护，向量表保护和一个虚拟内存接口。MMU 由函数 usrMmuInit( )初始化，该函数位于 installDir/target/src/config/usrMmuInit.c 文件中。如果还定义了宏 INCLUDE_PROTECT_TEXT 和 INCLUDE_PROTECT_VEC_TABLE，就会初始化代码保护和向量表保护。

初始化 WindView

可选产品 WindView 可提供内核测试工具。如果在 configAll.h 中定义了宏 INCLUDE_WINDVIEW，就可以在 usrRoot( )中调用 windviewConfig ( )初始化 WindView。其它的 WindView 常量控制特定的初始化步骤。

初始化目标代理

如果定义了 INCLUDE_WDB，调用函数 wdbConfig( )（位于 installDir/target/src/config/usrWdb.c）。这个函数初始化通讯接口，然后启动代理。

执行一个启动脚本

如果 VxWorks 配置了目标驻留的 shell，定义了 INCLUDE_STARTUP_SCRIPT，并且在 boot 引导过程中在启动脚本参数中输入了脚本文件的名称，usrRoot( )函数就可以执行一个用户提供的启动脚本。如果在引导过程中忽略了启动脚本参数，就不会执行启动脚本。

表 1、可加载 VxWorks 映象的初始化过程

函　　数	函　　数 功 能	所 在 文 件
sysInit()	(a)锁住中断；(b)禁用缓冲； (c)用缺省值初始化系统中断表（仅i960）； (d)用缺省值初始化系统错误表（仅i960）； (e)初始化处理器寄存器到一缺省值； (f)使回溯失效；(g)清除所有悬置中断； (h)激活 usrInit()，指明启动类型。	sysALib.s
usrInit()	(a)对bss清零； (b)保存bootType于sysStartType； (c)调用excVecInit(),初始化所有系统和缺省中断向量； (d)依次调用sysHwInit(), usrKernelInit(),kernelInit().	usrConfig.c
usrKernelInit()	依次调用 classLibInit(),taskLibInit(),taskHookInit(),semBLibInit(),semMLibInit(), semCLibInit(),semOLibInit(),wdLibInit() msgQLibInit(),qInit(),workQInit()	usrKernel.c
kernelInit()	初始化并启动内核。 (a)激活intLockLevelSet()； (b)从内存池顶部创建根堆栈和TCB；(c)调用 taskInit(), taskActivate(),用于usrRoot()； (d)调用usrRoot().	kernelLib.h
usrRoot()	初始化I/O系统，驱动器，设备（在configAll.h和config.h中指定） (a)调用sysClkConnect(),sysClkRateSet(), iosInit(),[ttyDrv()]； (b)初始化excInit(),logInit(),sigInit(). (c)初始化管道,pipeDrv()； (d)stdioInit(),mathSoftInit()或 mathHardInit() (e)wdbConfig():配置并初始化目标代理机	usrConfig.c

3.2 使用基于 ROM 的 VxWorks 映象的启动过程

此时 BOOTROM 中包含了引导程序和 VxWorks 映象。VxWorks 的入口点由两个函数 romInit()和 romStart()来完成，而非 sysInit()。具体过程如表 2 所示。

表 2、基于 ROM 的 VxWorks 映象的启动过程

函　　数	函 数 功 能	所 在 文 件
1.romInit()	(a)禁止中断； (b)保存启动类型； (c)硬件初始化； (d)调用romStart()；	romInit.s

2.romStart()	(a)将数据段从ROM拷贝到RAM，清内存； (b)将代码段从ROM拷贝到RAM,有必要的话解压缩; (c)依据引导类型调用usrInit();	bootInit.c
3.usrInit()	初始化程序	usrConfig.c
4.usrKernelInit()	如果相应的配置文件被定义，对应函数被调用	usrKernel.c
5.kernelInit()	初始化并启动内核	kernelLib.h
6.usrRoot()	初始化I/O系统，驱动器，创建设备	usrConfig.c
7.Application routine	应用程序代码	应用程序源文件

使用驻留 ROM 的 VxWorks 映象的启动过程与此类似，只是在执行搬移程序 romStart( )有所不同。

附 主要文件及宏开关介绍

以 est8260 评估板为例，说明编制 BSP 软件所涉及的主要文件及宏开关。

Makefile 文件

位于BSP文件目录下的编译文件Makefile，定义了启动文件bootrom首地址及大小，以及编译时调用的函数库，VxWorks映像文件的加载地址，用户需增加的目标模块。

该文件中的主要宏定义：

CPU = PPCEC603　目标板的CPU类型，MPC750为PPC604，MPC8260为PPCEC603

TOOL = gnu　主机工具为基于GNU的工具，用户不必修改

TGT_DIR = $(WIND_BASE)/target

target所在目录(WIND_BASE）已在torVars.bat中指定,为Tornado的安装目录

include $(TGT_DIR)/h/make/defs.bsp　定义编译选项

include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)　即文件make.ppc860gnu

include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)

即defs.x86-win32，定义和主机操作平台相关的工具

注意：　只能在这个位置之后重新定义make的定义。

TARGET_DIR = est8260　目标板BSP目录名，用户需指定为目标板BSP文件所在目录

VENDOR = EST　目标板制造商，和BSP程序无关，作注释用

BOARD　= est8260 Evaluation SBC　目标板名，和BSP程序无关，作注释用

BOOTINIT　= bootInit.c

USRCONFIG　= usrConfig.c

系统编译bootrom时会自动编译用户BSP文件目录下的usrConfig.c和bootInit.c，缺省文件位于All目录下。

以下地址值都为十六进制值，无需加入0x前缀。在config.h和Makefile中都定义了ROM_TEXT_ADRS, ROM_SIZE, RAM_HIGH_ADRS等常量，在2个文件中的对常量的定义要求一致。

ROM_BASE_ADRS = fff00000 ROM的物理起始点

ROM_TEXT_ADRS = fff00100 根启动设备Boot ROM首地址

ROM_SIZE = 00080000 BOOTROM大小（512K）

LOCAL_MEM_LOCAL_ADRS = 00000000 RAM的物理起始点

LOCAL_MEM_SIZE = 01000000 RAM大小（16M）

RAM_LOW_ADRS = 00100000 加载VxWorks的目标地址

RAM_HIGH_ADRS = 00a00000 拷贝boot ROM文本段和数据段的目标地址

HEX_FLAGS = -a $(ROM_TEXT_ADRS) Hex文件转化为bin文件的执行程序参数

MACH_EXTRA = m8260CpmEnd.obj 用户加入的目标模块名，后缀为obj

LDFLAGS = -X -N -Map est8260.map 在连接标识中增加-M est8260.map可以在BSP目录下生成包含编译信息的est8260.map文件。

注意：只能在这个位置之前重新定义make的定义。

include $(TGT_DIR)/h/make/rules.bsp

include $(TGT_DIR)/h/make/rules.$(WIND_HOST_TYPE)

定义编译各种BSP文件的生成规则

all/ConfigAll.h 文件

此文件包含了适用于所有目标板的缺省定义。主要定义了以下选项和参数：

内核配置参数

I/O 系统参数

NFS 参数

选择可选的软件模块

选择可选的设备控制器

cache 模式

最大数量的不同共享内存目标

设备控制器 I/O 地址，中断向量和中断级别

config.h 文件

此文件位于BSP目录下，包含仅适用于特定目标板的定义，还可以对configAll.h中的缺省定义进行重定义。其主要内容为：

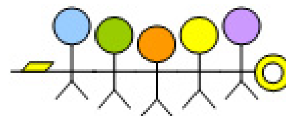
1、BSP版本号

#define BSP_VER_1_1 1

#define BSP_VERSION "1.2"

#define BSP_REV "/4"

2、configAll.h 包含文件，应放在 BSP_VERSION 和 BSP_REV 定义之后




```
#include "configAll.h"
```

3、定义缺省的BOOT引导设备

```
#define DEFAULT_BOOT_DEVICE   CPM_END
```

4、定义网络控制器

```
#define INCLUDE_CPM   ；使用SCC以太网控制器
```

5、定义缺省的BOOT引导参数

```
#define DEFAULT_BOOT_LINE \
 "$dev(0,procnum)host:dir\\file h=# e=# b=# g=# u=usr pw=passwd f=#   tn=targetname s=script o=other"
```

$dev -- boot device,启动的设备类型，必须是已包含的设备。

procnum -- 处理器序号，一般从零开始。

host -- 主机名

dir:\\file -- 被加载的VxWorks文件所在的完整路径

h -- 主机IP

e -- 目标板IP

b -- 背板IP，用户可不定义

g -- 网关，用户可不定义

u -- 用户名

pw -- 登录口令

f -- 定义网络加载方式。无此项时缺省值为零，为FTP

tn -- 目标板名

s -- 启动描述字符串，用户可不定义

o -- 从SCSI启动时指明网络接口

依据不同的启动设备类型，其中某些项可无。例如：

```
#define DEFAULT_BOOT_LINE \
"motec(0,0)diags:c:\\vxWorks h=172.96.36.88 e=172.96.78.23   g=172.16.0.1 u=anonymous pw=user f=0x00 tn=est8260"
```

6、 缓冲和 MMU 工作方式

```
#define INCLUDE_CACHE_SUPPORT
#define USER_I_CACHE_ENABLE
#define USER_D_CACHE_ENABLE
#define INCLUDE_MMU_BASIC
#define USER_I_MMU_ENABLE
#define USER_D_MMU_ENABLE
```

7、包含必要的网络驱动

```
#define   INCLUDE_NETWORK
#define   INCLUDE_END
```

8、 定义内存地址和大小

应与 Makefile 中的定义一致

#define USER_RESERVED_MEM 0x00000000 目标板用户保留内存大小

#define ROM_WARM_ADRS (ROM_TEXT_ADRS+8)

热启动地址，为ROM_TEXT_ADRS的一个偏移量，由启动文件romInit.s中跳转语句bl start相对第一条执行语句的偏移量 。

target.h 文件

本文件定义和目标板硬件相关的宏。用户应根据目标板硬件对时钟频率、内存空间分配、串口数目等进行设置。

romInit.s 文件

此文件是系统上电运行的第一个程序，也是制作 BOOTROM 时需要重点修改的程序，与硬件配置紧密相关。

可能需要修改：

机器状态寄存器 MSR

内部存储器映像寄存器 IMMR

总线配置寄存器 BCR

60x 总线总裁配置寄存器 PPC_ACR

系统保护控制寄存器 SYPCR

SIU 模式配置寄存器 SIUMCR

选项寄存器 ORx，基址寄存器 BRx

60x 总线分配 SDRAM 刷新时间 PSRT

60x 总线 SDRAM 模式寄存器 PSDMR

all/bootConfig.c 和 SysALib.s

一般用户不需修改，也可根据需要修改。

sysSerial.c

提供串口初始化的系统接口函数，定义串行通道参数和中断号等。串口设备驱动程序位于目录\tornado\target\src\drv\sio。根据用户所采用的串口及硬件连接作相应修改。

sysNet.c

一般只需修改网络设备初始化部分，如 MPC8260 中为函数 sysFccEnetEnable() 或 sysCpmEnetEnable( )。

网口和串口的驱动设计可参见相关的专题。