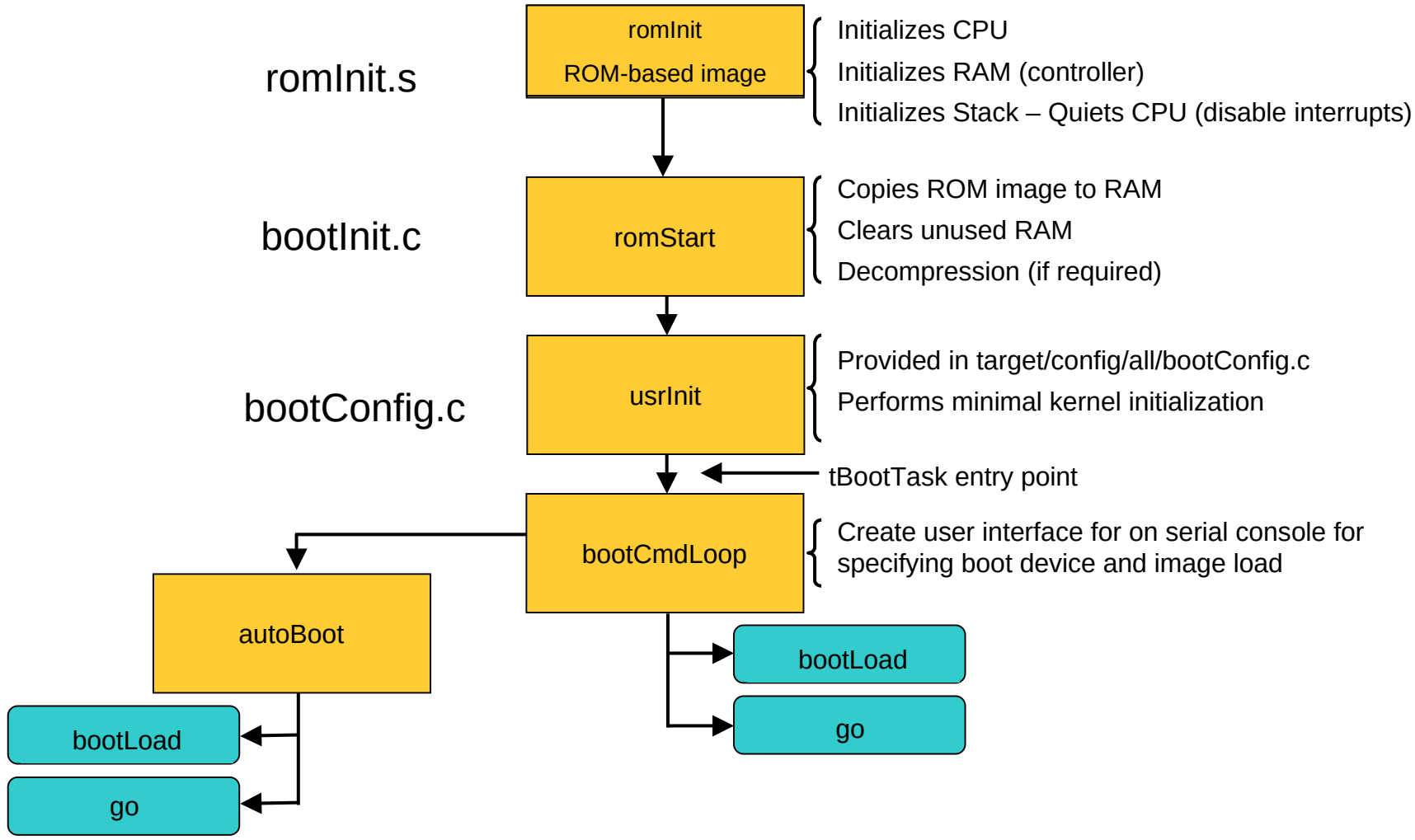
启动流程

Agenda

启动流程

- 启动流程概述
- 细节描述

Boot Sequence Using a Boot ROM



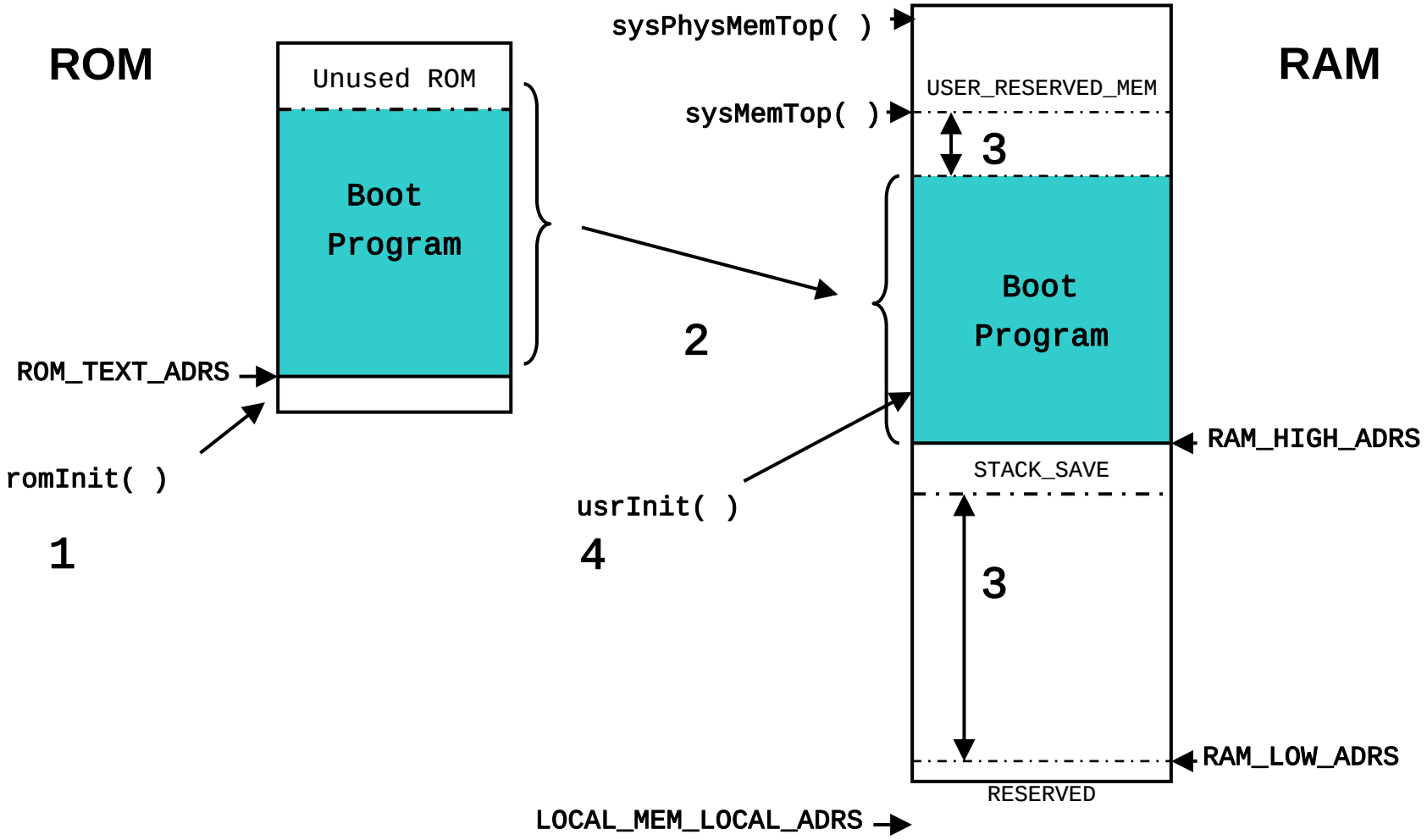
Cold vs. Warm Boots

- 启动类型
  - 冷启动: 上电或硬复位
  - 热启动: 调用 reboot( ), ctrl+X, 或 exception
  - 把启动类型传递给bootRom的函数是 sysLib.c 中的 sysToMonitor( ).
- 启动类型决定从 romInit( ) 的哪个位置开始执行
  - 冷启动: 在 romInit( ) 的入口开始执行. 启动类型强制置为 *BOOT_COLD*
  - 热启动: 在 romInit( ) 加一个小的偏移处执行 (通常偏移4个字节). 启动类型使用 sysToMonitor 传过来的值.
- 启动类型 (冷/热) 由sysToMonitor存储在寄存器中, 启动后传给 romStart( ), romStart( ) 由此决定是否清空内存

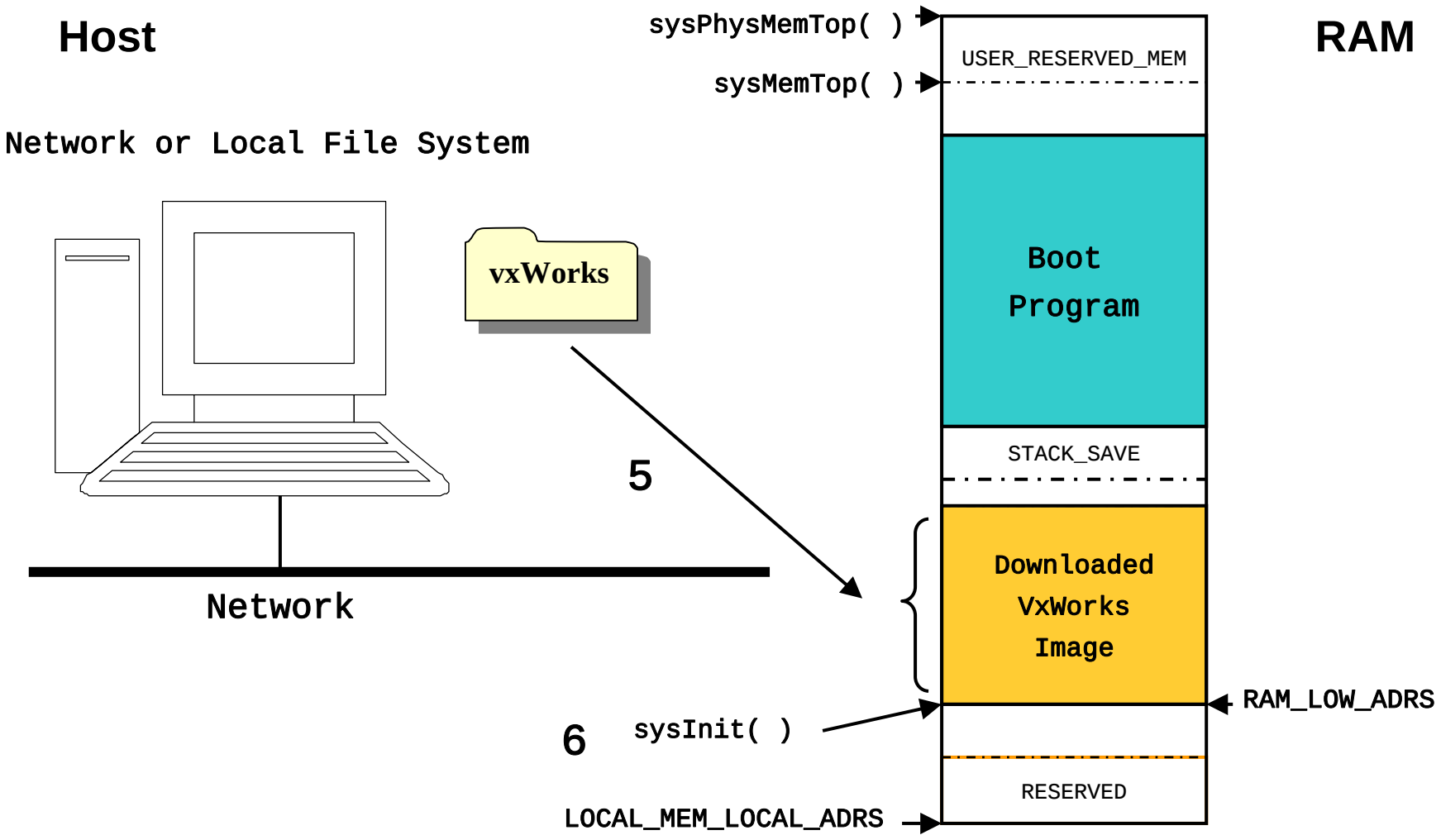
What Executes Where?

- 对于ROM驻留的image，所有代码的执行都在 ROM/Flash 中完成
  - 只有数据段被拷贝到 RAM
- 对于所有的 image 类型，汇编代码 romInit( ) 都在 ROM 或 Flash 中运行
- 在最小化的硬件初始化完成以后，vxWorks在RAM中初始化了一个很小的堆栈(一般16字节)，然后计算C代码入口函数romStart()的地址并跳转到romStart()中运行
- romStart()也是在ROM/Flash中运行的
- 对于基于ROM启动的image，romStart()会拷贝或解压vxWorks image到RAM，然后跳转到指定的入口函数执行
  - usrInit( ) 大多数image的入口函数
  - usrEntry( ) 工程编译的image入口函数，函数内部调用usrInit( )

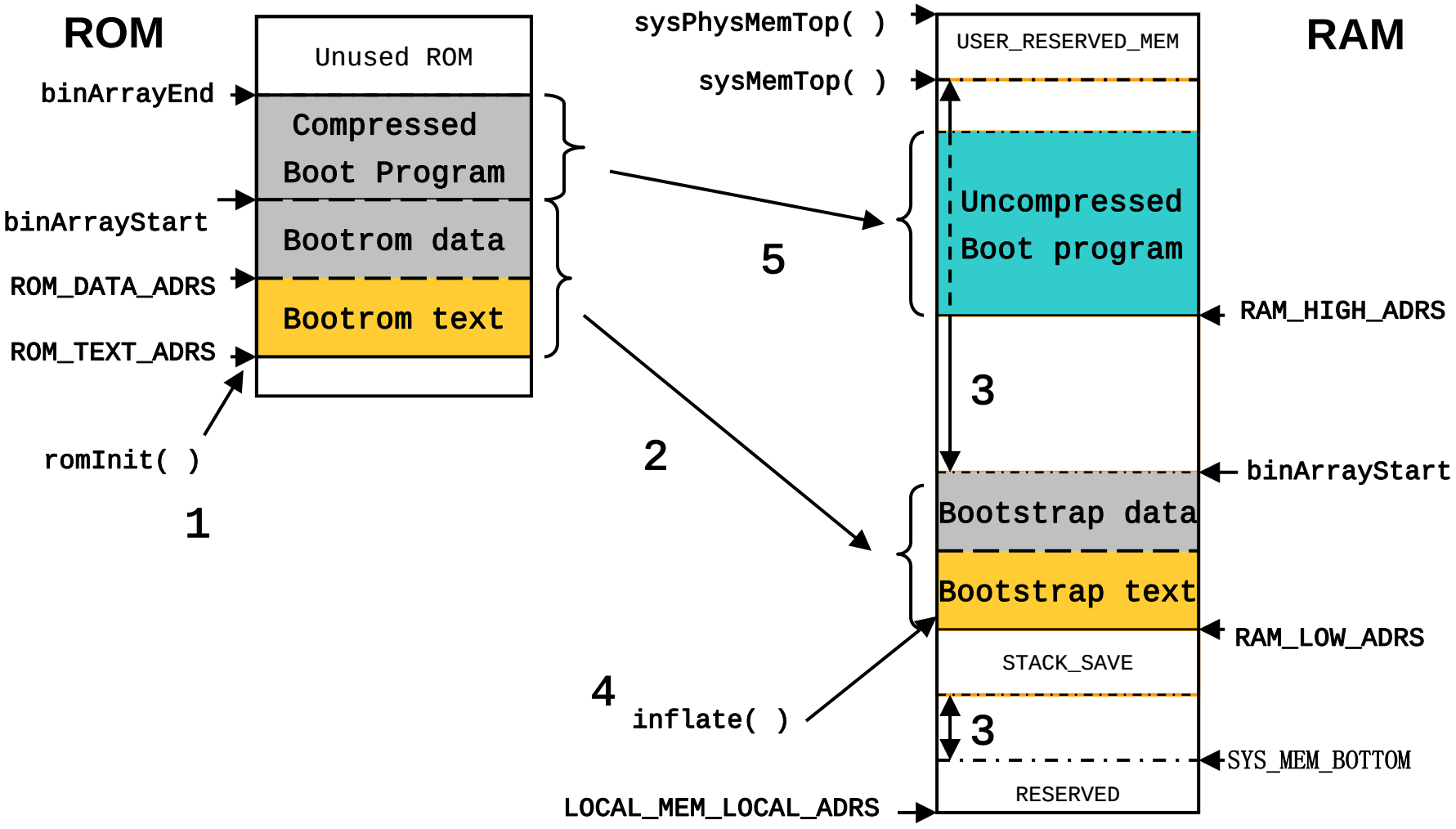
bootrom_uncmp



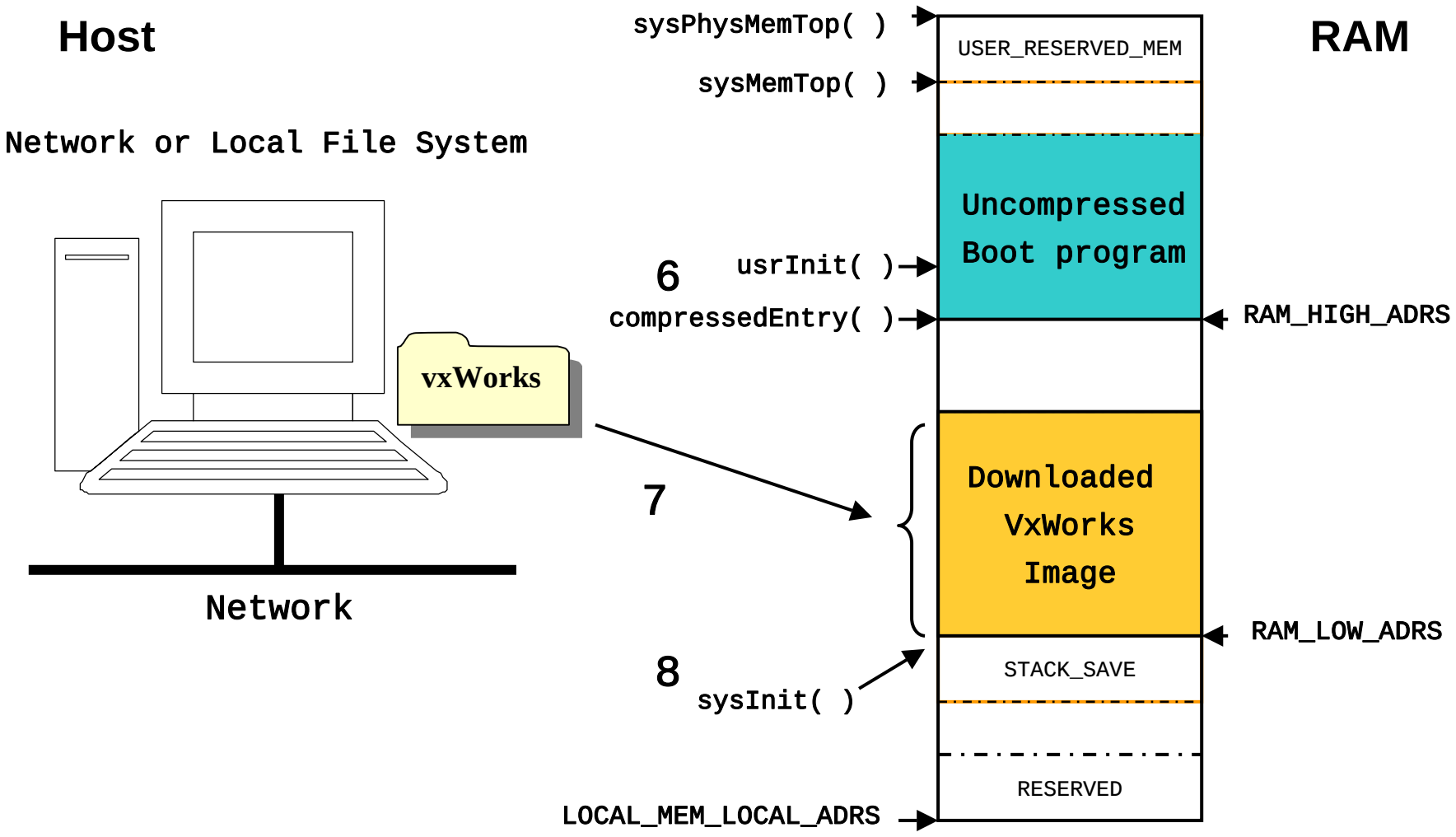
bootrom_uncmp (Continued)



Boot ROM (compressed)



Boot ROM (compressed) (Continued)



Agenda

启动流程

- 启动流程概述
- 细节描述

Step 1: Execute romInit()

- **boot ROM**的入口函数
- 函数实体在 romInit.s 中, 用汇编语言编写.
- 配置必需的少数寄存器
- 完整的硬件初始化在后续的 sysHwInit() 中完成

Processor Specific Initialization

- romInit( ) 一般跟**CPU**强相关, 如果**CPU**类型相同, 可以从参考板的**BSP**中拷贝
  - 关闭所有中断
  - 初始化一级**cache**
  - 把**sp**寄存器指向 STACK_ADRS 作为函数调用的栈, 以便于调用 romStart( ) 并传递启动类型参数

BSP-Specific Initialization

- 内存初始化
  - 等待内存硬件初始化完成
  - 配置刷新率
  - 配置片选 **(bridge/bus/memory controllers, etc.)**
  - 禁用二级**cache (**如果需要**)**

romInit( ) - PIC

- **PIC (Position Independent Code)**
- **romInit( ),** 运行在 **ROM/Flash,** 软件逻辑上必须与所处在地址无关, 以便于支持启动过程中存在变化的地址关系**(ROM->**内存**)**
- **PIC code is program counter (PC) relative**
- 如果存在绝对地址的操作, 例如函数调用, 必须根据运行时刻文本段所在的位置重新计算**.** 因为编译的地址是根据 **RAM_HIGH_ADRS** 获得的, 运行在 **ROM/Flash** 中时, 地址不匹配
  - 减去 **_romInit (),** 得到**bootRom**内的相对地址
    - **_romInit()** 即**bootRom**的第一个函数
  - 加上 *ROM_TEXT_ADRS* **,** 得到函数在**ROM**中的绝对地址
    - *ROM_TEXT_ADRS*是**bootRom**烧录在**ROM**中的地址

romInit( ) - Some Do's and Don'ts

- 执行启动过程中必须的初始化, 此时运行在**ROM/Flash**中, 速度较慢. 其他初始化待后续的 **sysHwInit( )** 中进行, 此时代码已经运行在**RAM**中, 速度较快.
- 不要调用外部的函数
  - 当编译压缩的**bootrom**时, 可能会引发链接问题
  - 调用函数时使用绝对的地址
- 确保 **romInit( )** 是 **romInit.s** 中的第一个函数
- 从 **romInit( )** 开始执行
- 确保下面两个宏在 **Makefile** 和 **config.h** 中都是正确的, 并且是一致的
  - ***ROM_TEXT_ADRS***
  - ***ROM_SIZE***

romInit.s Code Example

```
_romInit:
romInit:
    /* This is the cold boot entry (ROM_TEXT_ADRS) */
    bl  cold
 _romInitWarm:
romInitWarm:
    bl  warm
cold:
    li  r11, BOOT_COLD
    bl  start       /* skip over next instruction */
warm:
  orr11, r3, r3      /* startType to r11 */
start:
  /* Zero-out registers */
  addis   r0,r0,0
```

Step 2: Execute romStart()

- 从 **romInit( )** 跳转过来并把参数传递给 **romStart( )**
  - 在大多数**cpu**中, 都是通过寄存器来传递的
- 执行必要的代码搬移, 解压, 和内存初始化(清**0**)
  - 如果需要, 拷贝相应的**ROM image**到内存中
  - 清空内存中未使用的部分(冷启动)
  - 如果需要, 执行解压
  - 把启动类型传递给 usrInit( )
  - 代码用 **C** 语言编写, 位置在 ../target/config/all/bootInit.c
  - 执行位置在**ROM**中, 应该使用相对地址, 最后一个函数调用跳转到内存
  - 一般不需要修改本部分代码. 功能的调整可以通过配置宏来解决

Code Relocation

- 需要搬移的段 **romStart( )**
  - **ROM-based – Text and data**
  - **ROM-resident – Data only**
- 搬移到内存的目的地址
  - **Uncompressed vxWorks boot -** *RAM_HIGH_ADRS*
  - **Compressed vxWorks boot -** *RAM_HIGH_ADRS*
  - **Uncompressed vxWorks -** *RAM_LOW_ADRS*
  - **Compressed vxWorks -** *RAM_LOW_ADRS*
  - **ROM-resident vxWorks boot -** *RAM_HIGH_ADRS*
  - **ROM-resident vxWorks -** *RAM_LOW_ADRS*
- 搬移到内存需要花费比较长的时间**(**如何优化**?)**
- 对于非压缩的**bootrom,** 只需要一次搬移

Compressed Image Relocations

- 压缩的 **ROM images** 包含了压缩部分和未压缩部分
- romInit.s 和 bootInit.c 在未压缩部分. 其余的都在压缩部分
- 压缩的**bootrom**需要两次搬移
  - 第一次在 **romStart()** 中将未压缩部分从 **ROM** 搬到 **RAM**, 本次搬移中, **romStart**本身所在的 **TEXT** 也被搬移到内存, 提高了后续代码的执行速度
  - 第二次搬移是将压缩部分从 **ROM** 解压到内存, 到最终目的地. 这一次搬移也在 **romStart()** 中执行, 只不过运行代码所处的位置在内存而不是**ROM**

Clearing Memory for Cold Boots

- 冷启动的情况下, 需要对内存清 **0**
- 对于某些硬件, 内存清 **0** 可以减少内存访问错 **(**通常被未初始化的内存访问触发**)**
- 在 **romStart( )** 完成从 **ROM** 到 **RAM** 的搬移以后, 对代码段和数据段之外的内存清 **0**
- 内存清 **0** 需要花费比较长的时间
- 下列内存是不清 **0** 的**:**
  - 保留内存 *USER_RESERVED_MEM* **(**config.h**)**
  - 保留内存 *RESERVED* **(**configAll.h**)**
  - 保留内存 *STACK_SAVE* **(**configAll.h**)**

romStart( ) Stack

- **romStart( )** 的堆栈指针在**romInit( )** 中被初始化为 ***STACK_ADRS***
- **romStart( )** 不会返回 **–** 它的栈一直被用到内核启动完成, **kernelInit( )** 退出时为止
- **VxWorks** 内核被 **kernelInit( )** 激活, 它会启动一个 **tRootTask** 任务来完成剩余的系统配置和初始化工作, 这个任务有自己的任务栈, 后续再启动的其他应用层任务也都有自己的栈

ROM-resident Data Segment

- 对于**ROM**驻留的**bootrom,** 只有数据段需要搬移**,** 搬移完数据段以后应该校验数据段 **(data)** 的正确性

```
static int testVal = 13; /* data segment var */
                    ....
          if (testVal != 13)
             somethingWrongWithData();
```

- 如果**RAM**工作正常**,** 但数据段不对**,** 需要检查数据段搬移导**RAM**的过程是否有错
- 对于**ROM**驻留的**bootrom, romStart( )** 拷贝数据段到内存指定位置**,** 内存的位置是根据**ROM**中文本段的结束地址加一个偏移计算出来的
- 检查偏移

Modifying romStart( )

- 在**BSP**开发过程中可以在 bootInit.c 中增加调试信息
- 不要修改 ../target/config/all/bootInit.c
  - 把这个文件拷贝一份到具体的**BSP**目录下, 然后修改本地副本
- 在 **makefile** 的所有**rule include**之后, 增加下面的定义. 它将修改 **bootInit.c** 的来源, 把修改后的副本加到**BSP**中
  - **BOOTINIT = bootInit.c**
- 宏 *BOOTINIT*在 rules.bsp 中用到, 用于编译的时候查找 bootInit.c 文件
- 原始的**BOOTINIT**定义在 ../target/h/make/defs.$(WIND_HOST_TYPE)
  - BOOTINIT = $(CONFIG_ALL)\bootInit.c (即../target/config/all/bootInit.c)

romStart( ) Configuration Macros

- 配置宏控制 romStart( ) 的行为
- 这些宏定义在 config.h, Makefile, configAll.h, 和 bootInit.c 中
- 这些宏用于:
  - **BSP (或CPU架构)** 的依赖
  - **Image** 类型
  - 编译过程中 rules.bsp 的地址

romStart( ) Configuration Macros

- 定义在 config.h 中的配置宏
  - *LOCAL_MEM_LOCAL_ADRS* – 内存起始地址
  - *LOCAL_MEM_SIZE* – 内存大小
  - *USER_RESERVED_MEM* – 用户保留的字节数. 保留的内存处于内存顶端, 并且冷启动不会清0
  - *RAM_HIGH_ADRS* – 非 **ROM** 驻留的 **bootrom** 搬移到内存的地址
  - *RAM_LOW_ADRS* – 非 **ROM** 驻留的 **bootrom** 加载 **vxWorks** 应用的内存地址
  - *ROM_TEXT_ADRS* – **bootrom** 入口地址
  - *ROM_SIZE* – **ROM** 的大小
  - *ROM_BASE_ADRS* – **ROM** 的基地址

romStart( ) Configuration Macros

- 定义在 Makefile 中的宏
  - *RAM_HIGH_ADRS* – 值必须与 **config.h** 一致
  - *RAM_LOW_ADRS* –值必须与 **config.h** 一致
  - *ROM_TEXT_ADRS* – 值必须与 **config.h** 一致
  - *ROM_SIZE* - **Must agree with config.h**
- 定义在 configAll.h 中的宏
  - *RESERVED* – 保留内存. **RAM底部的保留内存(一般用于中断等, 例如PPC的0x4400), 冷启动不会清0**
  - *STACK_SAVE* - **romStart( ) 的栈空间大小. 跟CPU架构相关, 冷启动不会清0**
  - *STACK_ADRS* – 栈的基地址, 通常是拷贝到内存的第一个**image的地址 (例如romInit) , 向下生长***STACK_SAVE* 的大小

romStart( ) Configuration Macros

- 定义在 bootInit.c 中的宏
  - *USER_RESERVED_MEM* – 如果在 **config.h** 中没有定义, 将被定义为**0**
  - *SYS_MEM_BOTTOM* – 对于冷启动方式, 系统从这里开始对内存清**0**
    - *LOCAL_MEM_LOCAL_ADRS + RESERVED*
  - *SYS_MEM_TOP* – 对于冷启动方式, 系统清内存清到此处为止.
    - *LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE -USR_RESERVED_MEM*
  - *UNCMP_RTN* – 解压函数的名字 (地址)
    - *Inflate( )*
  - *ROM_OFFSET* – 用于重新计算函数绝对地址的宏, **romStart**中除跳转到内存的函数 **usrInit** 以外, 其他函数调用的地址都要用这个宏计算

romStart( ) Configuration Macros

- *RAM_DST_ADRS* – 压缩的 **image** 最终的搬移地址. 缺省值是 *RAM_HIGH_ADRS*, 必要时在 rules.bsp 中重新定义
- *RESIDENT_DATA* – **CPU**架构相关. **MIPS**和**PowerPC**中定义为 *RAM_DST_ADRS*. 其他架构中定义为数据段的起始地址
- *ROM_COPY_SIZE* – 需要搬移的代码的大小
- *ROM_BASE_ADRS* – 定义在 **config.h** 中. 如果 **romInit** 中定义了宏 *BOOTCODE_IN_RAM*, 则默认内存中已经有了**boot code**, 并且内存不需要初始化, 此时将重定义*ROM_BASE_ADRS*
- binArrayStart – 二进制代码的起始地址, 仅压缩的**bootRom**才需要转换二进制
- binArrayEnd –二进制代码的结束地址

romStart.c Example Code

```
void romStart
    (
    FAST int startType      /* start type */
    ){
    volatile FUNCPTR absEntry = (volatile
  FUNCPTR)RAM_DST_ADRS;
/* If cold booting, clear memory to avoid parity errors
  */
#ifdef ROMSTART_BOOT_CLEAR
    if (startType & BOOT_CLEAR)
        bootClear();
#endif
/* copy the main image into RAM */
    copyLongs ((UINT *)ROM_DATA(binArrayStart),
        (UINT *)UNCACHED(RAM_DST_ADRS),
        (&binArrayEnd - binArrayStart) / sizeof (long));
    /* and jump to it */
    absEntry (startType);
    }
```