usrInit()函数执行过程研究

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	2Nov2010	usrInit()函数执行过程研究
V1.1/正式	聂勇	12Nov2010	细节修改

目　录

1. 函数说明

usrInit()函数的实现在 bootConfig.c 文件，这是代码被解压缩之后运行的第一个函数。

在链接成 tmp.o 文件的时候，可以看到 usrInit ()函数是作为入口点函数被链接的。并且 tmp.o 的文本段的开始位置是 RAM_HIGH-ADRS，也就是 0x81c0.0000。如下图粗体所示。

compileOut.txt

……

ldmips -o tmp.o -EB -X -N **-e usrInit -Ttext 81c00000** bootConfig.o version.o sysALib.o sysLib.o srecLoad.o ns16550Sio.o cacheLib.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 flashDrvLib.o flashFsLib.o ftpXfer2.o flashUtil.o nvramstubs.o bcmsrom.o

……

下面根据 usrInit ()函数完成的功能顺序，对其进行详细的介绍。

2. data section 检测

如果 data section 加载不是在正确的位置上，就会出现 trap。使用了两个 magic cookies 去检验 data section 是否对准。如果出现了错误，则代码不会继续向下运行，进入了死循环。

3. MIPS gp 设置

设置 MIPS 的全局指针（global pointer）。函数 sysGpInit()在 sysAlib.s 文件中实现。

sysAlib.s

```
……
    .ent sysGpInit
sysGpInit:
    la   gp, _gp            /* set global pointer from compiler */
    j    ra
    .endsysGpInit
……
```

为什么需要设置这个指针，作用是什么，在 romInit()函数中已经设置了一次，为什么这里还需要设置？有关 gp 的相关内容，请参考《MIPS 汇编_NY.docx》，其中有对于这些问题的解释。

4. Cache 的初始化

Cache 的初始化时这里的重点，我们将结合在 romInit()函数中所作的设置来讲解。

首先看在 usrStart()函数中调用的 cache 初始化函数。

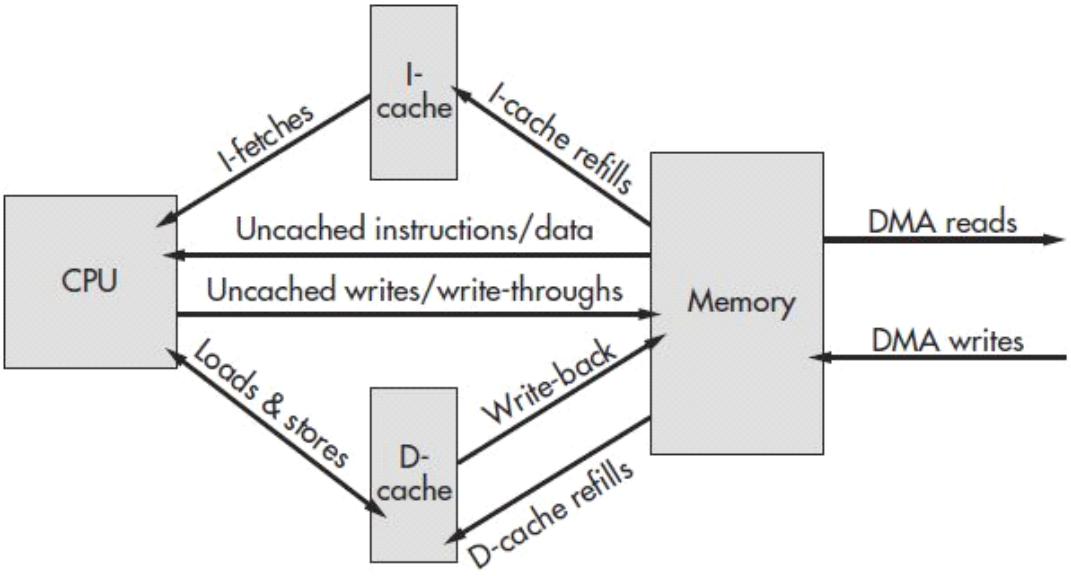
bootConfig.c
…… cacheLibInit (USER_I_CACHE_MODE, USER_D_CACHE_MODE) ……

在 config.h 文件中，USER_I_CACHE_MODE 和 USER_D_CACHE_MODE 都被定义为了 CACHE_COPYBACK 模式。有关 cache 的操作模式，可以参考 cache 的专业资料，在这里，cache 的操作模式就被定义为 copyback 模式，也就是回写模式。

到这里，有必要以俯视的角度看看 cache，看看 cache 是如何和 CPU，内存完成我们的工作的。

4.1 俯视整个 cache

首先，需要给出一张 CPU，Cache 和 Memory 的关系图。如下所示：



4.1.1.1 cache 和 DMA

cache 联系 CPU 和 memory，DMA 联系 memory 和外部设备（例如以太网卡，FLASH 等）。所以，在 DMA 对内存中的数据进行操作的时候，需要注意。

- 外部设备请求内存数据，DMA 要将内存中的数据拿出去，首先要保证内存中的数据已经和 D-Cache 中的数据同步了。所以此时要先将 D-Cache 中的数据 Write-back。

- 外部设备加载数据到内存中，最好把 D-Cache 中的条目置位无效。这样 CPU 使用 DMA 加载到内存的数据的时候，就不会拿到 D-Cache 中的旧数据了。

这里涉及到 DMA 的典型例子就是网络驱动中的 ET0 和 ET1。

那么，在代码中又是如何实现上面这两点的呢？请看下面 osl_dma_map()函数的实现。这个函数在 dma_txfast()函数中被调用，目的就是为了将要发送的包的数据使内存和 Cache 同步。

Vx_osl.c

```
……
void*
osl_dma_map(void *dev, void *va, uint size, uint direction)
{
    if (direction == DMA_TX)
        cacheFlush(DATA_CACHE, va, size);
    else
        cacheInvalidate(DATA_CACHE, va, size);
    return ((void*)CACHE_DMA_VIRT_TO_PHYS(va));
}
……
```

4.1.1.2 synci

出现这个的原因是，CPU 在从 rom 加载程序段的时候，会将这些程序段的代码当做数据，首先加载到 D-Cache 中。而这些程序段应该被写回到内存中，然后再加载到 I-Cache 中。为了完成这个操作过程，MIPS 提供了 synci 这条指令，意思是同步 I-Cache 中的数据。这条指令其实做了两节事情，一是将 D-Cache 中的数据（也就是程序段代码）write-back 到内存，二是把 I-Cache 中的条目置位无效。

5. 清零 BSS 段

清零 BSS 段需要用到的函数是 bzero()，函数的输入参数是编译时的特殊符号 edata 和 end，注意，这是 tmp.o 的，而不是在 romStart()函数中用到的 bootrom 的特殊符号。

有关可执行文件中的相关段的知识，请参考专业知识点。

6. 中断&异常向量初始化

函数调用如下：

bootConfig.c

```
……
    intVecBaseSet ((FUNCPTR *) VEC_BASE_ADRS);      /* set vector base table */


     excVecInit ();                  /* install exception vectors */
……
```

可以查到，VEC_BASE_ADRS 在 configAll.h 中定义，定义为 0x0000.0000。请问，这一个地址意味着什么？

中断和异常处理的过程时如何的。这与 CP0 Startus register 中的 BEV 的设置又有什么关系呢？

excVecInit()函数实现位于 E:\Tornado2.2.1-mips\target\src\arch\mips\excArchLib.c 文件中。

在此处的作用是安装 RAM 异常处理向量。在 romInit 函数中初始化的异常处理向量是位于 ROM 中的，现在要安装到 RAM 中。最后设置了 CP0 的 SR 的 BEV 位为 0，也就是将异常向量的位置由原来的 0xbcf0.0000 改成了 0x8000.0180，很显然，这是从 flash 的地址空间到了 sdram 的地址空间了。

7. 系统硬件初始化

硬件的初始化只有一个目的，那就是为 vxWorks Kernel 的配置和启动设置好硬件环境。调用函数 sysHwInit()函数，该函数在 sysLib.c 文件中实现，主要做的事情就是对 CP0 的 status register 进行了配置。

sysLib.c

```
……
    /* set default task status register for BCM47xx */
    sr = BCM47XX_SR;


     /* init status register but leave interrupts disabled */
     taskSRInit (sr);
     intSRSet (sr & ~SR_IE);
……
```

在完成了上一步之后，还调用了几个 Ds1743.c 文件中的函数初始化结构体。

sysLib.c

```
……
    typedef struct _systodfunctions
        {
        FUNCPTR init;
        FUNCPTR get;
        FUNCPTR set;
        FUNCPTR getSecond;
        FUNCPTR watchdogArm;
        } SYSTODFUNCTIONS;

SYSTODFUNCTIONS sysTodFuncs;
……
    /* Dallas TOD/NVRAM */
    sysTodFuncs.init = (FUNCPTR) ds1743_tod_init;
    sysTodFuncs.get = (FUNCPTR) ds1743_tod_get;
    sysTodFuncs.set = (FUNCPTR) ds1743_tod_set;
    sysTodFuncs.getSecond = (FUNCPTR) ds1743_tod_get_second;
    sysTodFuncs.watchdogArm = (FUNCPTR) NULL;
……
```

Q：ds1743 文件时干什么中途的？

8. vxWorks kernel 配置

调用函数 usrKernelInit()，这里主要是配置 vxWorks 的内核。这个函数的实现是在 vxWorks 中，在 BSP 中是找不到其实现的。我们编译时也是调用了其库文件。

对 tmp.o 文件进行反汇编，可以看到 usrKernelInit()函数主要做的事情有如下些。

```
81c00040 <usrKernelInit>:
81c00040:       27bdffe8        addiu   $sp,$sp,-24
81c00044:       afbf0014        sw      $ra,20($sp)
81c00048:       afbe0010        sw      $s8,16($sp)
81c0004c:       03a0f021        move    $s8,$sp
81c00050:       0c725ace        jal     81c96b38 <classLibInit>
81c00054:       00000000        nop
81c00058:       0c7310a1        jal     81cc4284 <taskLibInit>
81c0005c:       00000000        nop
81c00060:       0c72848b        jal     81ca122c <taskHookInit>
81c00064:       00000000        nop
81c00068:       0c730891        jal     81cc2244 <semBLibInit>
81c0006c:       00000000        nop
81c00070:       0c730d2e        jal     81cc34b8 <semMLibInit>
81c00074:       00000000        nop
81c00078:       0c7309a7        jal     81cc269c <semCLibInit>
81c0007c:       00000000        nop
81c00080:       0c72fe2b        jal     81cbf8ac <eventLibInit>
81c00084:       00000000        nop
81c00088:       0c731cc2        jal     81cc7308 <wdLibInit>
81c0008c:       00000000        nop
81c00090:       0c730216        jal     81cc0858 <msgQLibInit>
81c00094:       00000000        nop
81c00098:       3c0481cf        lui     $a0,0x81cf
81c0009c:       2484b6b0        addiu   $a0,$a0,-18768
81c000a0:       3c0581ce        lui     $a1,0x81ce
```

可以看到，在整个过程中，主要对 vxWorks 的类，任务，各种信号量，事件，看门狗以及消息队列进行了初始化。有关详细的初始化过程，可以参考 vxWorks 的相关资料。

9. 使能 I-Cache

调用函数 cacheEnable(INSTRUCTION_CACHE)完成对指令 Cache 的使能。该函数是 vxWorks 的一个标准 API，因为在前面的 cache 的初始化中，已经安装的 cache 的驱动，所以 vxWorks 就会调用这个驱动，也就是 cacheBcm47xxEnable()来开启 cache 功能。

10. 启动 kernel&建立根任务

这里启动了 vxWorks 内核并且建立了一个根任务，根任务函数是 usrRoot()。usrRoot()函数也是在 bootConfig.c 文件中定义。

bootConfig.c
…… /* start the kernel specifying usrRoot as the root task */

```
        kernelInit ((FUNCPTR) usrRoot, ROOT_STACK_SIZE,
          (char *) MEM_POOL_START_ADRS,
          sysMemTop (), ISR_STACK_SIZE, INT_LOCK_LEVEL);
……
```

从这里开始，usrStart()函数结束，下面就是要运行这个根任务 usrRoot()。

11. 根任务 usrRoot()

这是 vxWorks 执行的第一个任务，具有最高优先级 0，又叫做根任务。该任务中完成众多硬件的初始化。例如网络等等。最后会初始化一个命令行窗口 shell。

对 BSP 的简化的重点，就是这个函数。具体的相关内容，请参考《usrRoot()函数.docx》。