VxWorks 的内存配置和管理

殷战宁，刘　琳
（中国电子科技集团公司 51 所，上海 201802）

摘要：VxWorks 操作系统环境下内存配置和管理决定了 VxWorks 的系统性能，重点从软硬件配置、接口函数、应用优化 3 个方面对内存配置和管理方法进行了阐述和研究。

关键词：VxWorks 操作系统；内存控制器；内存管理
中图分类号：TP316　　**文献标识码**：A　　**文章编号**：CN32-1413(2012)03-0104-06

Memory Configuration and Management of VxWorks

YIN Zhan-ning，LIU Lin
(The 51st Research Institute of CETC，Shanghai 201802，China)

Abstract：The memory configuration and management in the environment of VxWorks operating system determine the performance of VxWorks system. This paper expatiates and studies the memory configuration and management method from three aspects：software-hardware configurations，interface function and application optimization.

Key words：VxWorks operating system；memory controller；memory management

0　引　言

对于一般编程人员尤其是 Windows 等通用平台的编程人员来说，内存配置基本上只要考虑全局变量、局部变量（堆栈变量）的合理安排，内存管理基本上只是使用 Malloc、Free 等编程接口。但对于 VxWorks 嵌入式操作系统而言，由于内存芯片选型、内存的地址空间分配[1]、中央处理器（CPU）级内存管理硬件初始化、映像文件的地址定位、内存的静态和动态占用、内存碎片对系统稳定性的影响等诸多内容都由设计人员来规划和控制，所以涉及内容广泛，需要有一个较好的通篇认识才能解决好与内存有关的问题。

1　软硬件配置

1.1　CPU 和内存芯片选型

内存需要相关的内存控制器来操作，以便能保证内存访问的时序、刷新等操作。内存控制器有些 CPU 自带，如 MPC5200 带双倍速率（DDR）控制器、同步动态随机存储（SDRAM）控制器，PPC860 通过用户可编程机制（UPM）来控制随机存储器（RAM）、S3c2410 带 SDRAM 控制器等；有些 CPU 不带内存控制器，如 X86 类型，需要桥片来对内存进行时序控制。

对内存芯片进行选型，首先要弄清楚内存控制器的特性，如支持的芯片类型、配置参数的标度方法等，然后选取类型无误、数据手册方便进行参数转换或计算的内存，合适的选型可以降低参数配置的难度，提高内存的访问效率。

由于内存参数配置是主板上电后立即就要做的事情，所以，顺利与否以及稳定与否，极大地影响随后的代码重定位等工作；通过硬件仿真器来调试主板，工作也往往是从配置和测试内存开始的。摘录 MPC5200 内存配置寄存器 2 的部分位定义，如图 1 所示。

可以看到该寄存器的设置牵涉到 t_{RCD}、t_{RP}、t_{RFC} 等内存片的参数。内存片 MT48LC32M8A2 的数据手册如表 1 所示。

收稿日期：2012-03-05

表 1　MT48LC32M8A2 内存数据表

Data-out high-impedance time　CL=3	$t_{HZ}(3)$		5.4	ns	10
Data-out low-impedance time	t_{LZ}	1		ns	
Data-out hold time(load)	t_{OH}	3		ns	
Data-out hold time(no load)	t_{OH_N}	1.8		ns	28
ACTIVE to PRECHARGE command	t_{RAS}	42	120 000	ns	
ACTIVE to ACTIVE command period	t_{RC}	60		ns	
ACTIVE to READ or WRITE delay	t_{RCD}	18		ns	
Refresh period(8 192 rows)	t_{REF}		64	ms	
AUTO REFRESH period	t_{RFC}	60		ns	
PRECHARGE command period	t_{RP}	18		ns	
ACTIVE bank a to ACTIVE bank b command	t_{RRD}	12		ns	7

Bit	Name	Description
13:15	act2rw	Active to Read/Write delay. Suggested value at 132 MHz = 0x02 Rule: t_{RCD}/MEM_CLK-1. Round up to nearest integer. EXAMPLE: If t_{RCD} = 20ns and MEM_CLK = 99 MHz 20ns / 10.1 ns = 1.98; round to 2; write 0x1. If t_{RCD} = 20 ns and MEM_CLK = 132 MHz 20ns / 7.5 ns = 2.66; round to 3; write 0x2.
16	—	Reserved
17:19	pre2act	Precharge to Active or Refresh delay. Suggested value at 132 MHz = 0x02 Rule: t_{RP}/MEM_CLK-1. Round up to nearest integer. EXAMPLE: If t_{RP} = 20ns and MEM_CLK = 99 MHz 20ns / 10.1 ns = 1.98; round to 2; write 0x1. If t_{RP} = 20 ns and MEM_CLK = 132 MHz 20ns / 7.5 ns = 2.66; round to 3; write 0x2.
20:23	ref2act	Refresh to Active delay. Suggested value at 132 MHz = 0x9 Rule: t_{RFC}/MEM_CLK - 1. Round up to nearest integer. EXAMPLE: If t_{RFC} = 75ns and MEM_CLK = 99 MHz 75ns / 10.1ns = 7.425; round to 8; write 0x7. If t_{RFC} = 75ns and MEM_CLK = 132 MHz 75ns / 7.5ns = 10; round to 9; write 0x9.

图 1　MPC5200 配置寄存器 2 的部分定义

可以看到 CPU 寄存器配置值的参数很方便就从内存手册里查到。如果选型不合适，则需要进行参数的换算，参数的具体关系可以参看专业介绍内存时序的资料。

这一步工作结果将具体反映到板级支持包(BSP)的 RomInit. s 文件中，如 MPC5200 的 BSP 里面，由 RomInitSdram 函数来实现内存配置，其重要配置就有：

```
#define CONFIG1    0xD2322800
#define CONFIG2    0x8AD70000
// Write Config 1.
lis    r4,HI(CONFIG1)
```

```
    ori   r4,r4,LO(CONFIG1)
    stw   r4,MBAR_SDRAM+SDRAM_CONF1_
OFF(r3)
    eieio
    // Write Config 2.
    lis   r4,HI(CONFIG2)
    ori   r4,r4,LO(CONFIG2)
    stw   r4,MBAR_SDRAM+SDRAM_CONF2_
OFF(r3)
    eieio
```

完成配置后，再通过内存控制器的 Command 寄存器等启动 Refresh，使能操作。

X86 的 VxWorks 由于有基本输入输出系统(BIOS)完成了内存的配置，所以在 RomInit. s 里面看不到类似的过程，其它类型 CPU 的主板基本上都需要该操作。

1.2 CPU 和内存地址分配

内存的地址对不同的 CPU 可以有不同的分配方法，对于 X86 体系，内存一般从地址 0 开始，跳过 A、B 段(显示内存)，C、D 段(卡式设备内存空间)，E、F 段(BIOS 空间)后，继续从 0x100000 连续分配。对于 CPU 命令架构()类 CPU，则一般有 CSx 类寄存器，可以配置与某个 CS 引脚对应的地址范围和操作宽度等，一般也配置成从 0 开始；对于增强的精简指令集机制(ARM)类 CPU，做法会有许多种，需要查询具体的芯片手册，如 S3c2410，则固定 CS6 和 CS7 用作 SDRAM 的片选，地址范围固定从 0x30000000 开始。

此类寄存器的设置也需要在 RomInit. s 里面设置好。

1.3 CPU 级内存管理硬件初始化

CPU 一般都有存储器管理单元(MMU)、块地址翻译(BAT)等硬件机制来对内存空间分段或分页管理，对不同的段页配置不同的虚实地址对应关系、读写属性、Cache 属性、保护属性等。这部分操作 VxWorks 开发人员无需直接访问寄存器，只需填写用来配置 MMU 或 BAT 的结构数组，如 MMU 配置：

```
PHYS_MEM_DESC sysPhysMemDesc [] =
{
    {
    /*  Vector Table and Interrupt Stack */
    (void  *)  LOCAL  __MEM  __LOCAL  __
ADRS,     /* virtual addr */
    (void  *)  LOCAL  __MEM  __LOCAL  __
ADRS,     /* real addr */
    (RAM_LOW_ADRS - LOCAL_MEM_LO-
CAL_ADRS),     /* size */
    VM_STATE_MASK_VALID | VM_STATE
_MASK_WRITABLE | VM_STATE_MASK_
CACHEABLE | VM_STATE_MASK_MEM_CO-
HERENCY,     /* attrib mask */
    VM_STATE_VALID        | VM_STATE_
WRITABLE      | VM_STATE_CACHEABLE
| VM_STATE_MEM_COHERENCY     /* at-
trib */
    },
    …,
    …
}
```

该配置器就描述了一段内存的虚地址、实地址、大小、属性使能、属性等；主内存地址空间一般都配置为不做虚实转换、可写、可 Cache 等。该表项会由 UsrMmuInit 函数里面读取并配置到 MMU，而且也可以通过 SysMmuMapAdd 等维护函数做增删。

BAT 表一般用于 CPU 命令架构(PPC)类寄存器，可以定义代码段、数据段、段大小、Cache 属性等，可以与 MMU 组合使用，也可以单独使用，相关的 BSP 也提供结构数组进行维护。

MMU 和 BAT 表项一般在 BSP 的 SysLib. c 文件里面。

对于 X86 类 CPU，还有全局描述符表(GDT)、中断描述符表(IDT)等内存描述符需要初始化，内存描述符规定了一段空间的大小和属性，组成表格由段寄存器来选取，段寄存器对地址选址的计算实际上是通过内存描述符翻译后进行的。VxWorks 的 BSP 简化了该操作，几个段寄存器都使用一个能覆盖所有地址空间的全功能的 GDT 进行工作。

对于 Cache 使能否，一般还会有针对所有的 Cache、代码 Cache、数据 Cache、一级 Cache、二级 Cache 等不同的硬件开关，可能会在 RomInit. s，SysAlib. s 通过汇编来操作，也可以在 UsrInit，UsrRoot 函数里面通过 CacheEnable 等函数来操作，需要检查确认。

1.4 映像文件的地址定位

一般而言，CPU 上电后先是在只读存储器(ROM)(或 Flash 等)内运行，为了提高运行速度，往往需要把代码搬移到内存去继续执行，数据相关段(Data、没有初值的全局变量符号(BSS)段)则需指向内存区域并得到正确的初始化，堆栈段也需要指向内存，段与段之间需要满足一定的定位关系，比如不互相重叠等。

以最常用的 bootrom_uncmp + VxWorks 启动组合为例：

(1) 上电后执行在只读存储器(ROM)里面的 bootrom_uncmp 映像，初始化 CPU、内存控制器等，堆栈底设置在宏 RAM_HIGH_ADRS 决定的位置(栈顶朝地址低端)。

(2) ROM 里面的程序会把 ROM 里面的 bootrom_uncmp 映像拷贝到起始位置是 RAM_HIGH_ADRS 的内存区域，然后跳转到内存来继续执行。此时代码段在 RAM_HIGH_ADRS 位置，数据段紧跟在代码段后面，BSS 段紧跟在数据段后面，BSS 段后面是中断使用的堆栈，然后是 bootrom_uncmp 将要使用的内存池，任务堆栈段会从内存池申请。所以，要想让 bootrom_uncmp 正常执行，需要确保 RAM_HIGH_ADRS 下面有足够的空间够 ROM 做搬移时的堆栈，RAM_HIGH_ADRS 上面有足够空间存放代码、数据和用来做内存申请。

(3) 内存里面 bootrom_uncmp 的执行会下载 VxWorks，把 VxWorks 映像拷贝到起始位置是 RAM_LOW_ADRS 的内存区域，然后跳到该内存继续执行。为了确保 bootrom_uncmp 拷贝 VxWorks 期间不会发生 VxWorks 覆盖 bootrom_uncmp 的现象，“RAM_LOW_ADRS＋VxWorks 代码段大小＋VxWorks 数据段大小”所占的空间要尽量避免与 bootrom_uncmp 需要使用的空间重叠。

(4) VxWorks 获得运行经过再次初始化后，代码段在 RAM_LOW_ADRS 位置，数据段紧跟在代码段后面，BSS 段紧跟在数据段后面，BSS 段后面是中断使用的堆栈，然后是 WDB 专用的内存池，然后是 VxWorks 将要使用的内存池，堆栈会先设置在 RAM_LOW_ADRS 开始朝下生长以满足 usrRoot 启动前的使用，然后会设置到靠近系统内存顶端以满足 usrRoot 的使用并利于回收到内存池，以后的任务堆栈就是从内存池里面申请。

可以看到 RAM_HIGH_ADRS 和 RAM_LOW_ADRS 2 个宏有着重要的定位意义，尤其是 RAM_LOW_ADRS 很大程度上决定了 VxWorks 的内存布局，需要斟酌一个好的位置以便既保证正常启动、又不至于造成内存浪费(太高了会使内存池缩小)。因为既影响代码，也影响链接，所以这两个宏的修改需要在 config.h 里面和 makefile 里面同时修改并确保一致。

其它类型组合，如 bootrom、bootrom_res、VxWorks_rom 等，结合编译链接时产生的 map 文件或符号表文件，同样可以做出类似的分析。

根据地址定位关系可以大概知道内存池的起始位置，通过 map 文件或符号表文件来看，一般是 BSS 段的结束 + ISR_STACK_SIZE + WDB_STACK_SIZE，而内存池的结束位置由 SysMemTop 函数决定，一般来说是 LOCAL_MEM_LOCAL_ADRS＋LOCAL_MEM_SIZE－ USER_RESERVED_MEM。

2 接口函数

接口函数主要是指创建内存池、动态申请和释放内存的函数，但需要额外说明的是，当书写 C/C＋＋源码时，如果定义了一个全局变量(指函数体外的变量)并赋予了初值，则该变量会静态占用数据段的空间，如果定义了一个全局变量但没有赋予初值，则该变量会静态占用 BSS 段的空间，如果定义了一个函数体内的变量，则该变量会动态占用堆栈空间，这些编程实际上是在隐蔽地申请(或释放)内存。下面列举常用的针对内存池的接口函数。

2.1 第 1 类函数

第 1 类函数是主内存的参数设置和使用：

```
void memOptionsSet[2]
    (
    unsigned options /* options for system
    partition */
    )
```

设置内存池的属性，包括：

MEM_ALLOC_ERROR_LOG_FLAG

当内存分配出错则打出 log 信息：

MEM_ALLOC_ERROR_SUSPEND_FLAG

当任务内存分配出错，则把任务挂起(除非任务设置了 VX_UNBREAKABLE 属性)：

MEM_BLOCK_ERROR_LOG_FLAG

当内存释放出错则打出 log 信息：

MEM_BLOCK_ERROR_SUSPEND_FLAG

当任务内存释放出错，则把任务挂起（除非任务设置了 VX_UNBREAKABLE 属性）：

```
void *malloc
  (
  size_t nBytes /* number of bytes to allo-
  cate */
  )
```

申请大小为 nBytes 的内存，返回该内存的 ptr（其实就是该段内存的起始地址）：

```
void free
  (
  void * ptr /* pointer to block of memory
  to free */
  )
```

释放已申请的一段内存，传入该段内存的 ptr（起始地址）作为参数：

```
void *calloc
  (
  size __ t elemNum, /* number of
  elements */
  size_t elemSize /* size of elements */
  )
```

申请 elemSize*elemNum 大小的内存，该段内存会被清 0：

```
void *memalign
  (
  unsigned alignment, /* boundary to align
  to (power of 2) */
  unsigned size/* number of bytes to allocate
  */
  )
```

申请 size 的内存，内存起始地址满足 alignment 的要求。

2.2 第 2 类函数

第 2 类函数是单独的内存池的使用，实际上，主内存池也是一个单独的内存池，其指针是全局变量 memSysPartId：

```
PART_ID memPartCreate
  char *    pPool,    /* pointer to memory
  area */
  unsigned poolSize /* size in bytes */
  )
```

创建一个新的内存池，起始地址是 pPool，大小是 poolSize，该段内存可以是原来系统内存池之外的某一段离散的内存，也可以是从系统内存池里面申请到的一段内存。建立了单独的内存池，可以在该段内存里面单独申请和释放内存，而不影响其它的内存。返回值是内存池指针：

```
STATUS memPartOptionsSet
  (
  PART_ID partId, /* partition to set op-
  tion for */
  unsigned options /* memory management
  options */
  )
```

内存池 partId 的参数配值，含义同 memOptionsSet：

```
void *memPartAlloc
  (
  PART_ID  partId, /* memory partition
  to allocate from */
          unsigned  nBytes/* number of
  bytes to allocate */
  )
```

在 partId 的内存池里面申请 nBytes 的内存：

```
STATUS memPartFree
  (
  PART_ID partId, /* memory partition to
  add block to */
  char *   pBlock   /* pointer to block of
  memory to free */
  )
```

释放已申请的 pBlock 内存回 partId 的内存池：

```
void *memPartAlignedAlloc
  (
  PART_ID  partId,    /* memory parti-
  tion to allocate from */
  unsigned nBytes,    /* number of bytes to
  allocate */
  unsigned alignment /* boundary to align to
  */
  )
```

在 partId 里面申请 nBytes 的内存，内存起始地址满足 alignment 的要求。

2.3 第3类函数

第3类函数是内存池的维护函数：

```
void memAddToPool
    (
    char *    pPool,    /* pointer to memory block */
    unsigned  poolSize  /* block size in bytes */
    )
```

增加一段新的内存给主内存池：

```
STATUS memPartAddToPool
    (
    PART_ID  partId,    /* partition to initialize */
    char *    pPool,    /* pointer to memory block */
    unsigned  poolSize  /* block size in bytes */
    )
```

增加一段新的内存给内存池 partId：

```
void memShow
    (
    int type /* 1 = list all blocks in the free list */
    )
```

察看主内存池的信息列表：

```
STATUS memPartShow
    (
    PART_ID partId, /* partition ID */
    int     type    /* 0 = statistics, 1 = statistics & list */
    )
```

察看内存池 partId 的信息列表。

3 应用优化

VxWorks 申请内存时使用空间首先满足的算法，找到合适的块，多出来的部分会单独形成一个空块，释放内存时会进行相邻空闲内存块的归并，却不会做碎片搬移和整理。因此动态内存虽然使用方便，但大量的小内存操作偶尔再穿插大内存的操作会造成内存池的碎片，最终没有足够的内存使用。而且动态申请和释放会带来时间上的损失。所以，在应用层，需要考虑静态和动态的平衡，考虑到动态情况下大量相同内存操作的优化。

频繁申请和释放的内存建议改成静态的方式，以避免时间上的损失，如果不想使用全局数组或结构这样的方式，也可以使用动态方式申请下来一块内存，然后进行强制类型转换。

如果跟任务动态运行有关，可以考虑放在任务的函数体内，成为堆栈变量（任务的堆栈大小在创建任务时确定，如果担心堆栈紧张，可以考虑对较大的变量只是把指针放在堆栈里面，而指针所指的内存则动态申请），不仅是动态的，而且了实现了任务与任务的隔离。

VxWorks 上的网卡驱动就是采用这样的方式来管理接收和发送缓冲：通过动态方式申请下来一块内存，建立不同大小的 cluster 的数组，并为 cluster 设置管理属性，然后建立申请和释放 cluster 的函数。

动态的大量相同内存操作建议自己建立一套机制来管理，如通过 message queue 的协助来管理。

4 结束语

VxWorks 是一个实时嵌入式操作系统，所以从嵌入式的角度来说，内存配置工作是必不可少的。从实时的角度来说，掌握了通用的内存管理函数后，还需要进一步了解这些函数对实时性和安全性的影响，从而规划一个比较稳健的内存管理系统。

参考文献

[1] 王金刚，高伟，苏琪. VxWork 程序员指南[M]. 北京：清华大学出版社，2003.

[2] 周户平，张杨. VxWork 程序员速查手册[M]. 北京：机械工业出版社，2005.

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道

Windows CE:

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发

PowerPC:

1. Freescale MPC8536 开发板原理图

ARM:

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码

Hardware:

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计