南 京 工 程 学 院

毕业设计说明书(论文)

作 者：  **朱宇**  学 号： **202150340**

院 系： **计算机工程学院**

专 业： **计算机科学与技术**

题 目： **基于Cortex-A9多核处理器的**

**操作系统内核设计和实现**

指导者： **施 展** **讲 师**

评阅者：

2019年5月 南 京

**毕 业 论 文 中 文 摘 要**

|  |
| --- |
| **题目：** 基于Cortex-A9多核处理器的  操作系统内核设计和实现  摘要：  在科技日益发展的今天，新兴的技术层出不穷，我国的计算机行业也发展迅猛，然而，不可否认的是在基础软件领域任然落后于发达国家。特别像操作系统领域，还缺少一个认可度较高，通用性强，稳定的操作系统内核软件项目。随着新型处理器架构和物联网的兴起，势必需要一个新的操作系统内核，因此内核在未来具有深刻的研究价值。本课题基于Cortex-A9多核处理器，实现了一个支持多任务和文件系统的操作系统内核。内核的具体功能模块包括内存管理、进程管理（添加，删除，切换）、中断管理、设备驱动、文件系统和系统调用等。  本文以Linux内核作为主要参考对象，来编写一个操作系统雏形，并运行在s5p4418开发板上。内核的性能并没有专门优化，单纯展示内核是如何设计和实现。同时内核尽可能遵循POSIX标准，实现通用的编程接口。  关键词：Cortex-A9　ARM　操作系统　内核　POSIX |

**毕 业 论 文 外 文 摘 要**

|  |
| --- |
| **Title** Design and Implementation of Operating System Kernel Based on Cortex-A9 Multi-core Processor  **Abstract:**  Today, with the development of science and technology, emerging technologies are emerging one after another. China's computer industry is also developing rapidly. However, it is undeniable that it still lags behind developed countries in the field of basic software. Especially like the operating system field, there is still a lack of a highly recognized, versatile, and stable operating system kernel software project. With the rise of new processor architectures and the Internet of Things, a new operating system kernel is bound to be required, so the kernel has profound research value in the future. Based on the Cortex-A9 multi-core processor, this project implements an operating system kernel that supports multitasking and file systems. Specific functional modules of the kernel include memory management, process management (add, delete, switch), interrupt management, device drivers, file systems, and system calls.  This article uses the Linux kernel as the main reference object to write a prototype of the operating system and runs on the s5p4418 development board. The performance of the kernel is not specifically optimized, simply showing how the kernel is designed and implemented. At the same time, the kernel follows the POSIX standard as much as possible to implement a common programming interface.  **Keywords:** Cortex-A9 ARM operating system kernel POSIX |

目　　录

[前言 1](#_Toc9201653)

[第一章 绪论 2](#_Toc9201654)

[1.1 研究背景和意义 2](#_Toc9201655)

[1.2 设计目标 2](#_Toc9201656)

[第二章 硬件与软件 3](#_Toc9201657)

[2.1 硬件平台 3](#_Toc9201658)

[2.2 软件框架 4](#_Toc9201659)

[第三章 开发框架搭建 6](#_Toc9201660)

[3.1 启动镜像制作 6](#_Toc9201661)

[3.2 USB下载驱动 7](#_Toc9201662)

[3.3 启动代码 7](#_Toc9201663)

[3.4 Makefile构建框架 10](#_Toc9201664)

[3.5 ld srcipt链接脚本 12](#_Toc9201665)

[第四章 功能模块设计 14](#_Toc9201666)

[4.1 内存管理 14](#_Toc9201667)

[4.2 设备管理 18](#_Toc9201668)

[4.3 中断管理 21](#_Toc9201669)

[4.4 文件系统 21](#_Toc9201670)

[4.5 系统调用 21](#_Toc9201671)

[4.6 任务管理 21](#_Toc9201672)

[第五章 接口和性能测试 22](#_Toc9201673)

[5.1 接口测试 22](#_Toc9201674)

[5.2 性能测试 22](#_Toc9201675)

[参考文献 23](#_Toc9201676)

[附录：英文技术资料翻译 24](#_Toc9201677)

# 前言

本文主要面向实践，通过具体的代码介绍操作系统内核各个默模块的设计与实现。从启动代码开始，一步一步，逐渐完善，通过小半年时间编写代码，不断磨合，精益求精，最终形成一个比较完整的操作系统内核。

众所周知，操作系统内核作为最重要计算机基础软件之一，设计难度不是一般的大。如此短的时间，作者自然无法写出能满足生产实际的内核，但好在主要的模块一个不少，仅以此抛砖引玉，期待更好的作品。

操作系统往往非常复杂，而初学者一下子面对完整的操作系统，想要理清代码，更是难上加难。每个操作系统都是前辈们积累了几十年的智慧结晶，动辄就是几十上百万行代码，因此想要学习操作系统，无疑一个精简的操作系统会事半功倍。

基于上面一些考虑，我很早就开始考虑自己实现一个操作系统内核。最开始在x86架构的CPU上编写，相关的资料比较全面。但我很快发现x86由于多年的积累，其设计已经很复杂。作为CISC处理器的代表，不能满足我写操作系统的初衷，那为什么不选择RISC呢？于是我很快投入ARM处理器的内核研发中，本文也是我历时两年的研究成果的集中展示。

本文基于ARM Cortex-A9处理器，设计了一个能实际运行的操作系统内核。本内核具有以下功能：

1. 参考kbuild系统设计了一套基于Makefile的跨平台构建框架。
2. 支持VFS，支持文件读写，目录创建，重命名等操作。
3. 比较完善的设备驱动框架，支持json格式的设备树
4. 支持部分POSIX接口，容易移植

# 第一章 绪论

## 1.1 研究背景和意义

## 1.2 设计目标

# 第二章 硬件与软件

## 2.1 硬件平台

### 2.1.1 ibox4418

随着4412 芯片的即将停产，九鼎创展科技率先推出 ibox4418 卡片电脑。之所以选择S5P4418的开发板基于以下几点考虑：

1. 从性能上，采用更先进的工艺，一点不弱于4412，；
2. 从价格上，比4412 更便宜；
3. 从兼容性上，Pin to Pin 兼容 6818；
4. 从供货周期上，能持续批量供货，即使 4418停产，也可以直接升级 6818，即使 6818 停产，也可以直接换成 5430，有10年生命期。

### 2.1.2 配置和接口

* 内核：ARM Cortex-A9四核；
* 主频：1.4GHz\*4；
* 内存：1GB DDR3，可兼容2GB DDR3；
* Flash：支持4GB/8GB/16GBemmc可选，标配8GB emmc；
* 24位RGB接口；
* 8位LVDS接口；
* 两路USB HOST接口，支持更多的USB设备同时使用；
* USB OTG接口；
* 四路TTL电平UART接口；
* 2路TF卡接口；
* 两路LED指示；
* 复位按钮；
* 软件开关机按钮；
* 支持外置扬声器；
* 支持MIC输入；
* 支持耳机输出接口；
* 独家支持免启动配置开关设计；
* 支持背光无级调节；
* 支持HDMI接口；
* 支持5点电容触摸；
* 板载USB接口WIFI/BT二合一模块；
* 支持多种SPI，I2C，UART，等外围器件扩展；
* 支持MPEG4，H.263，H.264，MJPEG视频编码；
* 支持几乎全格式视频解码；
* 支持2D，3D高性能图形加速；
* 支持RTC时钟实时保存；
* 支持千兆有线以太网RTL8211E；
* 支持BT656/BT601/MIPI摄相头接口；
* 支持GPS接口；
* 支持GPRS接口；
* 支持外置USB 3G模块；
* 支持USB鼠标，键盘；
* 支持红外一体化接收头；

## 2.2 软件框架

### 2.2.1 xboot驱动框架

XBOOT是一个代码结构规整，代码抽象度很高的嵌入式裸机框架，支持各类主流的CPU架构，同时支持多款SoC。应用程序采用Lua虚拟机执行，功能强大，支持GUI开发。

本项目主要使用了XBOOT的驱动框架部分。原因有三点：

1. 适配s5p4418处理器，包含UART、LCD和SDCARD等驱动；
2. 代码抽象度高，接口简单，代码清晰，方便移植；
3. 支持设备树。通过修改json文件即可完成设备驱动的配置，方便快捷。

### 2.2.2 MINE内核

本文提及的内核早期源码是从MINE内核移植过来。随着版本的变迁，MINE内核的源码已发生很多的改动，并且我也添加和删除了一些功能。不过不可否认，MINE内核的设计给了我灵感和设计参考。

# 第三章 开发框架搭建

## 3.1 启动镜像制作

S5P4418处理器的启动过程有四步，如下图：

系统上电

IROM

2ndboot

uboot.bin

在IROM代码中，会检测当前设备支持的启动方式如emmc、sdcard等。之后会按优先级从相应的设备运行2ndboot。在2ndboot部分，用户可以做更详细的启动配置。最后2ndboot启动用户定义的启动文件，也可以理解为3rdboot。整个过程就像火箭发射，一级推动一级。

下面是一个镜像在EMMC中分布图：

2nboot

uboot.bin

**0 1 2 ... 62 63 64 …**

绿块包含了sd/emmc卡的信息，紫块是Nsih.bin，包含了启动配置信息，如DDR的频率，CPU的频率，启动方式等。

我们的并不想依赖uboot启动，而是从2ndboot直接启动，即第四步换成我们自己kernel.bin。因此我们需要需要修改2ndboot。

这里的2ndboot由三星官方提供代码。在下载代码后，配置为usb启动，并编译生成2ndboot。

然后使用九鼎创展提供的工具生成启动镜像，命令如下：

mk4418 usbboot.bin nsih.txt 2ndboot 2ndboot

工具名为mk4418，nsih.txt为系统配置文件。最后一个参数是3rdboot，因为我们选择usb启动，随便写一个文件就行。

最后我们用插上sdcard，/dev/sdb是我们的sdcard设备。使用dd命令将usbboot.bin从Block1开始烧写，命令如下：

dd if=usbboot.bin of=/dev/sdb bs=512 seek=1

这样我们就完成了启动镜像的制作。

## 3.2 USB下载驱动

之前我们配置2ndboot为usb启动，因此需要通过usb将内核二进制文件传给开发板，则免不了要编写usb下载驱动。

方便起见，我们使用libusb在用户态编写驱动。参考libusb的示例代码，定义厂家和设备编码如下：

#define X4418\_SECBULK\_IDVENDOR 0x04E8

#define X4418\_SECBULK\_IDPRODUCT 0x1234

这样就能识别开发板了，之后添加文件读取和批量传输的代码，即可完成驱动的编写。

## 3.3 启动代码

到这里，我们已经能正常下载和启动自己的程序了，按照道理应该把启动代码放在代码分析部分，但启动代码并不算具体哪个模块，因此独立出来，单独分析。主要包括异常向量表的设置，CPU初始化，各模式栈初始化，具体代码如下：

/\* Exception vector table \*/

.text

.arm

.global \_start

\_start:

b reset

ldr pc, \_vector\_und

ldr pc, \_vector\_swi

ldr pc, \_vector\_pabt

ldr pc, \_vector\_dabt

ldr pc, \_vector\_addrexcptn

ldr pc, \_vector\_irq

ldr pc, \_vector\_fiq

\_vector\_und:

.word vector\_und

\_vector\_swi:

.word vector\_swi

\_vector\_pabt:

.word vector\_pabt

\_vector\_dabt:

.word vector\_dabt

\_vector\_addrexcptn:

.word vector\_addrexcptn

\_vector\_irq:

.word vector\_irq

\_vector\_fiq:

.word vector\_fiq

/\* The actual reset code \*/

reset:

/\* Set cpu to svc32 mode and disable interrupt \*/

mrs r0, cpsr

bic r0, r0, #0x1f

orr r0, r0, #0xd3

msr cpsr, r0

/\* Enable neon/vfp unit \*/

mrc p15, #0, r1, c1, c0, #2

orr r1, r1, #(0xf << 20)

mcr p15, #0, r1, c1, c0, #2

mov r1, #0

mcr p15, #0, r1, c7, c5, #4

mov r0, #0x40000000

fmxr fpexc, r0

/\* Cache init \*/

mrc p15, 0, r0, c0, c0, 0

and r1, r0, #0x00f00000

and r2, r0, #0x0000000f

orr r2, r2, r1, lsr #20-4

cmp r2, #0x30

mrceq p15, 0, r0, c1, c0, 1

orreq r0, r0, #0x6

mcreq p15, 0, r0, c1, c0, 1

/\* Invalidate L1 I/D \*/

mov r0, #0

mcr p15, 0, r0, c8, c7, 0

mcr p15, 0, r0, c7, c5, 0

/\* Disable mmu stuff and caches \*/

mrc p15, 0, r0, c1, c0, 0

bic r0, r0, #0x00002000

bic r0, r0, #0x00000007

orr r0, r0, #0x00000002

orr r0, r0, #0x00000800

mcr p15, 0, r0, c1, c0, 0

/\* Set Vector Base Address Register \*/

ldr r0, =\_start

mcr p15, 0, r0, c12, c0, 0

mrc p15, 0, r0, c1, c0, 0

bic r0, #(1<<13)

mcr p15, 0, r0, c1, c0, 0

/\* Initialize stacks \*/

mrs r0, cpsr

bic r0, r0, #0x1f

orr r1, r0, #0x1b

msr cpsr\_cxsf, r1

ldr sp, \_stack\_und\_start

bic r0, r0, #0x1f

orr r1, r0, #0x17

msr cpsr\_cxsf, r1

ldr sp, \_stack\_abt\_start

bic r0, r0, #0x1f

orr r1, r0, #0x12

msr cpsr\_cxsf, r1

ldr sp, \_stack\_irq\_start

bic r0, r0, #0x1f

orr r1, r0, #0x11

msr cpsr\_cxsf, r1

ldr sp, \_stack\_fiq\_start

bic r0, r0, #0x1f

orr r1, r0, #0x13

msr cpsr\_cxsf, r1

ldr sp, \_stack\_srv\_end

/\* Copyself to link address \*/

adr r0, \_start

ldr r1, =\_start

cmp r0, r1

beq 1f

ldr r0, \_image\_start

adr r1, \_start

ldr r2, \_image\_end

sub r2, r2, r0

bl memcpy

1: nop

/\* Clear bss section \*/

ldr r0, \_bss\_start

ldr r2, \_bss\_end

sub r2, r2, r0

mov r1, #0

bl memset

/\* Call \_main \*/

ldr r1, =\_main

mov pc, r1

\_main:

mov r0, #0

mov r1, #0

mov fp, #0

bl main

b \_main

## 3.4 Makefile构建框架

### 3.4.1名词约定

1. 用户Makefile:指的是一般用户就可以修改的Makefile,主要用来指定参与编译的文件，比如init/Makefile。
2. 系统Makefile:指的是对体系架构和内核有深入了解的开发人员才可以修改的Makfile文件，比如arch/$(ARCH)/$(MACH)目录下的Makefile.mach、Makefile.head和Makefile.arch。
3. 核心Makefile:指的是对编译系统有深入了解的开发人员才可以修改的Makfile文件比如：
4. Makefile
5. include/Makefile
6. scripts/ Makefile.build
7. scripts/ Makefile.include。
8. 源码目录：也可以叫输入目录。代表当前项目的源码根目录。
9. 输出目录：你可以将生成的文件导出到输出目录，来避免污染源码。默认和源码目录一致。
10. 编译目录：当前正在编译文件的目录。
11. 工作目录：工作目录始终在输出目录的根目录。如果你没有指定输出目录，则和源码目录一致。

### 3.4.2总体框架

1. 顶层Makefile：它是所有Makefile文件的核心，从总体上控制着内核的编译、连接。
2. 配置文件：包括include/kconfigs.h和arch/$(ARCH)/$(MACH)/include/config.h前者是所有平台通用的配置，后者与具体机器和CPU架构相关。编译系统会根据这两个文件自动生成include/config/autoconf.h和include/config/auto.conf。前者用于C语言，后者用于Makefile。
3. 用户Makefile:决定哪些文件参与内核的生成。
4. scripts/Makefile.\* : Makefile共用的通用规则、脚本等
5. scripts/Makefile.build:用于递归的编译项目
6. scripts/Makefile.include:一些通用的变量和函数
7. scripts/fixdep.c:修复gcc生成的依赖文件
8. arch/$(ARCH)/$(MACH)/Makfile.mach:

用于设置机器相关的工具链参数

1. arch/$(ARCH)/$(MACH)/Makfile.arch:

用于设置架构相关的工具链参数

### 3.4.3使用方法

1. 编译

make

1. 清除

make clean

1. 导出生成目录

make O=output #编译

make clean O=output #清除

1. Makefile会自动创建output目录，所以你应该保证目录创建成功。使用这个功能请确保指定了一个空目录，否则可能在清除时，删除一些重要文件。
2. 添加编译目录或文件。假设添加init/yy目录，你应该找到init目录下的Makefile（如果没有请创建）添加如下内容：

obj-y += yy/

这里yy目录下的所有.S和.c文件都会参与编译。值得注意的是yy目录的子目录并不会参与编译，防止将一些错误的文件的编译进内核。

1. 添加编译文件。这时假设添加init/xx.c文件，你应该在到init目录下的Makefile，添加如下内容：

obj-y += xx.c

## 3.5 ld srcipt链接脚本

链接脚本主要有四个部分组成。通过一系列的定义描述，控制最终生成的可执行文件布局。

3.5.1 总体描述

1. 定义输出格式

OUTPUT\_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")

1. 定义体系架构

OUTPUT\_ARCH(arm)

1. 设置入口函数

ENTRY(\_start)

3.5.2常量定义

主要是各个模式栈大小定义：

STACK\_UND\_SIZE = 0x12;

STACK\_ABT\_SIZE = 0x12;

STACK\_IRQ\_SIZE = 0x12;

STACK\_FIQ\_SIZE = 0x12;

STACK\_SRV\_SIZE = 0x100000;

3.5.3 内存区域划分

主要分为RAM、DMA和HEAP区域，其位置和大小如下：

ram: org = 0x40100000, len = 128M - 1M

dma: org = 0x48000000, len = 128M

heap: org = 0x50000000, len = 256M

3.5.4 段划分

这里定义了最终可执行文件有哪些段，这些段有什么组成。由于代码冗长，择主要的说明。

.text包含了所有的可执行代码，描述如下：

.text :

{

PROVIDE(\_\_image\_start = .);

PROVIDE(\_\_text\_start = .);

\_text = .;

arch/arm32/mach-x4418/head.o;

\*(.text\*);

\*(.init.text);

\*(.exit.text);

PROVIDE(\_\_text\_end = .);

\_etext = .;

} > ram

.initcall包含了所有的初始化函数，系统关键的初始化程序都是通过这个段调用的，总共有10个等级，描述如下：

.initcall ALIGN(8) :

{

PROVIDE(\_\_initcall\_start = .);

KEEP(\*(.initcall\_0.text))

KEEP(\*(.initcall\_1.text))

KEEP(\*(.initcall\_2.text))

KEEP(\*(.initcall\_3.text))

KEEP(\*(.initcall\_4.text))

KEEP(\*(.initcall\_5.text))

KEEP(\*(.initcall\_6.text))

KEEP(\*(.initcall\_7.text))

KEEP(\*(.initcall\_8.text))

KEEP(\*(.initcall\_9.text))

PROVIDE(\_\_initcall\_end = .);

} > ram

# 第四章 功能模块设计

## 4.1 内存管理

内存管理单元是操作系统内核最基础的模块，一直以来是一个重要的研究对象。任何程序最重要的资源必定是内存，而系统关键的数据结构都依赖于内存分配。为此，曾提出很多不尽相同的算法，这些算法优势也各不相同。本章的内存管理主要负责内存的分配和MMU的控制。内存分配能让内核更好的管理内存，而MMU则可以让不同的程序共享同一段内存地址空间。在设计算法主要借鉴的Linux内核的SLAB分配器，它能有效减少内存碎片的问题。另外，本章也极力模仿Linux的API设计。

### 4.1.1 SLAB内存池

虽然SLAB是建立在更基础的分配算法上，但由于相对简单易懂，因此放在最前面。我们编写程序申请内存时，其实会申请很多一样大小的数据，如果我们用通用的分配算法申请，难免会出浪费的情况。举个例子，当你申请30B的数据时，可能获得的空间是32B。并且，采用通用算法，算法会更复杂，操作更耗时。Linux在处理这种问题就采用了SLAB内存池的技术。通俗来讲，就是提前申请一大块空间，然后划分出许多大小一致的小内存区域。当有程序申请内存，直接找一块空闲的块就行了。下面看一下Slab\_cache的结构体定义：

struct Slab\_cache {

unsigned long size; //分配单元大小

unsigned long total\_using;//已分配个数

unsigned long total\_free;//未分配个数

struct Slab \* cache\_pool;//slab链表

struct Slab \* cache\_dma\_pool;

void \*(\* constructor)(void \* Vaddress, unsigned long arg);//构造函数

void \*(\* destructor)(void \* Vaddress, unsigned long arg);//析构函数

};

这里反映了一个slab总体信息，用户可以自定义构造函数和析构函数。cache\_pool指向具体的内存池链表，如果内存池用光，就需要将申请新的内存池并链入链表。具体定义如下：

struct Slab {

struct list\_head list; //链表节点

struct Page \* page; //内存池所在Page

unsigned long using\_count; //使用数

unsigned long free\_count; //空闲数

void \* Vaddress; //Page对应的地址

unsigned long color\_length;// color\_map的bit数

unsigned long color\_count; // color\_map的字节数

unsigned long \* color\_map; // color\_map的地址

};

color\_map是一个动态分配的数组，包含了内存池的分配情况。其中的每一个bit代表一个单元。

下面介绍一下slab的几个API。

slab创建：

struct Slab\_cache \* slab\_create(

unsigned long size, //分配单元大小

void \* (\* constructor)(void \* Vaddress, unsigned long arg),

void \* (\* destructor)(void \* Vaddress, unsigned long arg),

unsigned long arg

);

slab销毁：

unsigned long slab\_destroy(struct Slab\_cache \* slab\_cache);

slab分配对象：

void \* slab\_malloc(

struct Slab\_cache \* slab\_cache,

unsigned long arg

);

slab释放对象：

unsigned long slab\_free(

struct Slab\_cache \* slab\_cache,

void \* address,

unsigned long arg

);

### 4.1.2 页管理单元

这一小节介绍内核最底层的分配算法，不管是SLAB内存池还是通用内存分配模块，其根本都是调用页管理单元。

本文内核的页都是2MB大小，Linux的页分配最小则是4KB。在粒度上相对比较大，但也因此碎片化的问题会少很多。Linux的采用了伙伴系统算法，能有效抑制碎片化，而我们为了简单起见，单纯采用最先适配算法。下面介绍内存大致的区域划分。

从左往右依次是.text段、.data段、page bit map、page stuct、zone struct、page和异常向量表。

内存首先会被划分为不同的区域即zone，内核定义了ZONE\_DMA、ZONE\_NORMAL和ZONE\_UNMAPED。每个zone包含若干page struct，每个page struct对应一个page。

主要的API如下：

1. 初始化内存

void init\_memory();

1. 页分配

struct Page \* alloc\_pages(

int zone\_select,

int number,

unsigned long page\_flags

);

zone\_select用于选定在哪个zone分配页，像LCD控制器申请framebuffer时就要指定ZONE\_DMA。Number代表要申请的page数，page\_flags暂时没有用到，写零即可

1. 页释放

void free\_pages(struct Page \* page, int number);

### 4.1.3 通用内存管理单元

为了处理程序任意的内存申请，肯定需要一个比较通用的解决方案。用到的算法其实和slab一致，就连结构体用的都是同一个。但是这里的slab是预先定义好的，并且和slab的具体实现略有不同。Slab事实上是依赖通用内存管理单元的，因为color\_map大小肯定是不固定的。总之，他们都是使用内存池技术，却是两套不同的实现，只不过通用内存管理单元的实现更偏底层一点罢了。既然是通过slab实现的，那我们先看看是怎么定义slab\_cache的。代码如下：

struct Slab\_cache kmalloc\_cache\_size[16] = {

{32 , 0 , 0 , NULL , NULL , NULL , NULL},

{64 , 0 , 0 , NULL , NULL , NULL , NULL},

{128 , 0 , 0 , NULL , NULL , NULL , NULL},

{256 , 0 , 0 , NULL , NULL , NULL , NULL},

{512 , 0 , 0 , NULL , NULL , NULL , NULL},

{1024 , 0 , 0 , NULL , NULL , NULL , NULL}, //1KB

{2048 , 0 , 0 , NULL , NULL , NULL , NULL},

{4096 , 0 , 0 , NULL , NULL , NULL , NULL}, //4KB

{8192 , 0 , 0 , NULL , NULL , NULL , NULL},

{16384 , 0 , 0 , NULL , NULL , NULL , NULL},

{32768 , 0 , 0 , NULL , NULL , NULL , NULL},

{65536 , 0 , 0 , NULL , NULL , NULL , NULL}, //64KB

{131072 , 0 , 0 , NULL , NULL , NULL , NULL}, //128KB

{262144 , 0 , 0 , NULL , NULL , NULL , NULL},

{524288 , 0 , 0 , NULL , NULL , NULL , NULL},

{1048576, 0 , 0 , NULL , NULL , NULL , NULL}, //1MB

};

可以发现我们一共定义了16的内存池，从32B到1MB大小不等。当程序申请一个500B的内存空间，它会匹配到最接近的512B的内存池，以此类推，也就是说我们支持0B到1MB之间任意的内存申请。下面简单介绍一下相关API。

1. 通用内存管理单元初始化：

unsigned long slab\_init()

1. 内存分配

void \*kmalloc(unsigned long size, unsigned long gfp\_flages)

1. 内存释放

unsigned long kfree(void \*address)

## 4.2 设备管理

内核采用xboot的驱动框架。这一小节集中介绍这个框架的实现原理。

### 4.2.1 initcall

首先介绍initcall机制，这是框架的基础部分。正常在编写裸机程序时，如果我们想要添加一个驱动，先要编写驱动，然后在主函数中初始化，调用等。这样编写驱动需要反复修改主函数，代码之间的耦合度就会很高，而initcall机制能够解决这种问题。下面通过具体的代码来详细介绍。

static struct driver\_t sdhci\_xl00500 = {

.name = "sdhci-xl00500",

.probe = sdhci\_xl00500\_probe,

.remove = sdhci\_xl00500\_remove,

.suspend = sdhci\_xl00500\_suspend,

.resume = sdhci\_xl00500\_resume,

};

static \_\_init void sdhci\_xl00500\_driver\_init(void)

{

register\_driver(&sdhci\_xl00500);

}

static \_\_exit void sdhci\_xl00500\_driver\_exit(void)

{

unregister\_driver(&sdhci\_xl00500);

}

driver\_initcall(sdhci\_xl00500\_driver\_init);

driver\_exitcall(sdhci\_xl00500\_driver\_exit);

这是一段sdhci驱动的代码，内核加载时，会自动调用sdhci\_xl00500\_driver\_init函数完成驱动的注册。当内核退出时也会自动调用sdhci\_xl00500\_driver函数。那这是怎么实现的呢？我们可以发现在上面这两个函数定义时加了\_\_init和\_\_exit修饰。这是两个宏，定义如下：

#define \_\_init \_\_attribute\_\_ ((\_\_section\_\_ (".init.text")))

#define \_\_exit \_\_attribute\_\_ ((\_\_section\_\_ (".exit.text")))

所以代码的意思是将这两个函数加入.init.text和.exit.text段。回顾之前的链接脚本的代码，你会发现这两个段最终都加入了.text段。接着往下看，还有driver\_initcall和driver\_exitcall这两个宏。driver\_initcall定义如下：

#define driver\_initcall(fn) \_\_define\_initcall("4", fn, 4)

\_\_define\_initcall定义如下：

#define \_\_define\_initcall(level, fn, id) \

static const initcall\_t \_\_initcall\_##fn##id \

\_\_attribute\_\_((\_\_used\_\_, \_\_section\_\_(".initcall\_" level ".text"))) = fn

不难发现，其实是定义了一个static修饰的函数指针，并将其加入.initcall4.text段。再回顾一下链接脚本的内容，有如下代码：

.initcall ALIGN(8) :

{

PROVIDE(\_\_initcall\_start = .);

…

KEEP(\*(.initcall\_4.text))

…

PROVIDE(\_\_initcall\_end = .);

} > ram

这意味着.initcall段存放着所有初始化的函数。完成所有初始化工作的函数如下：

void do\_initcalls(void)

{

initcall\_t \* call;

call = &(\*\_\_initcall\_start);

while(call < &(\*\_\_initcall\_end))

{

(\*call)();

call++;

}

}

### 4.2.2 register

register分为驱动和设备，分别对应register\_driver和register\_device函数。我们在编写一个新的驱动需要像上一小节代码一样通过initcall调用register\_driver。只有当在设备树文件中实例化设备，框架才会调用register\_device。当然由于设备繁多，也会提供register\_sdhci这样的包装函数来简化操作，基本每一类设备都会有类似的函数。其实框架采用面向对象的方式编写C程序，这样有助于扩展，也能降低耦合。

我们具体来看一下如何编写一个新的驱动。

首先我们要填充一下driver\_t结构体：

static struct driver\_t sdhci\_xl00500 = {

.name = "sdhci-xl00500",

.probe = sdhci\_xl00500\_probe,

.remove = sdhci\_xl00500\_remove,

.suspend = sdhci\_xl00500\_suspend,

.resume = sdhci\_xl00500\_resume,

};

name是驱动的名字，设备定义时会根据这个名字找到具体的驱动，所以这个名字肯定是不能重复的。之后是设置四个函数指针，当设备初始化会调用里面的probe函数。这样一个驱动就定义完成，后面仿照前一节的initcall相关内容填写即可。这里的驱动是没有类别的，只有在probe时才会决定什么类型驱动。

### 4.2.3 设备树（DTS）

有关设备树的概念最早出现在Linux内核PowerPC架构的源码中。随着ARM的碎片化越来越严重，不可能每个驱动参数都放在板级描述文件中。试想，Linux每支持一个板子，就要修改源代码，无疑会增加不必要的开发成本。就像之前提到的initcall机制，为解决这种耦合性过高的问题，ARM、MIPS、X86 等架构也相继引入了设备树机制。至此，新的板子只要在对应芯片下修改设备树文件即可。并且，这种方式比直接修改C语言源代码更直观，实际上相当于强迫开发者设计一个简单直接的接口来配置设备。

xboot同样有自己的设备树机制，不同的是，xboot的设备树文件是采用json描述。开发者不必像Linux一样学习新的语法，很快就能上手编写设备树。并且xboot设计更为激进一点，并没有像Linux一样保留C代码添加设备的方式。下面简单介绍一下如何编写设备树。

"sdhci-xl00500@0xC0062000": {

"clock-name": "pll2"

}

以上是一个最简单的语法实例。sdhci-xl00500是设备的名字，0xC0062000可以理解为设备的唯一编号，可以填写sdhci控制器的设备地址保证唯一性。括号内的是设备的参数，这里clock-name指的是sdhci控制器的时钟源。所有以上这些信息最终都会传入驱动的probe函数，从而完成设备初始化。当然你可能会好奇，xboot是如何完成这一过程的，下面通过代码简单讲解一下，太多细节的东西就不仔细仔细介绍了。

首先系统通过initcall机制调用subsys。代码如下：

extern unsigned char \_\_dtree\_start;

extern unsigned char \_\_dtree\_end;

static void subsys\_init\_dt(void)

{

char \* json;

int len = 0;

len = &\_\_dtree\_end - &\_\_dtree\_start + 1;

json = &\_\_dtree\_start;

probe\_device(json, len, "hello");

}

static \_\_init void subsys\_init(void)

{

subsys\_init\_dt();

}

subsys\_initcall(subsys\_init);

然后会调用probe\_device完成设备树解析移植设备初始化。核心代码如下：

for(i = 0; i < v->u.object.length; i++)

{

p = (char \*)(v->u.object.values[i].name);

n.name = strsep(&p, "@");

n.addr = p ? strtoull(p, NULL, 0) : 0;

n.value = (struct json\_value\_t \*)(v->u.object.values[i].value);

if(strcmp(dt\_read\_string(&n, "status", "okay"), "disabled") != 0)

{

drv = search\_driver(n.name);

if(drv && (dev = drv->probe(drv, &n)))

LOG("Probe device '%s' with %s", dev->name, drv->name);

else

LOG("Fail to probe device with %s", n.name);

}

}

通过调用search\_driver找到之前注册的设备驱动，然后调用probe函数，完成初始化。那么驱动是如何解析设备树的信息的呢？我们以sdhci-xl00500为例看一下。

physical\_addr\_t phy = dt\_read\_address(n);

if(phy == 0xc0062000)

port = 0;

else if(phy == 0xc0068000)

port = 1;

else if(phy == 0xc0069000)

port = 2;

else

return FALSE;

其中dt\_read\_address函数读取的便是@右面的数值。

clk\_name = dt\_read\_string(n, "clock-name", NULL);

if(!clk\_name)

return FALSE;

dt\_read\_string读取的则是参数clock-name的值，NULL代表默认值。

当驱动根据参数完成一系列初始化后，调用register\_sdhci完成设备的注册，后续的设置则由sdhci驱动框架完成，这里就不多赘余了。

## 4.3 中断管理

中断就如同其名字会中断CPU，强迫进入中断模式。中断的出现很大程度上解放了CPU，并且提高了设备请求响应的实时性。中断通常由外部设备（如GPIO、存储设备、USB等）产生。随着技术的发展，处理器能处理的中断数量也从几十到上百不等。像子中断，共享中断源，甚至设置设备控制器自定义中断也加深了中断处理的难度。为此，抽象出一个统一的中断模型至关重要。

### 4.3.1 中断控制器注册

S5P4418默认采用VIC中断控制器，共有两个控制器，每个控制器最多能管理32个中断源。设备树的定义如下：

"irq-pl192@0xc0002000": {

"interrupt-base": 0,

"interrupt-count": 32

},

"irq-pl192@0xc0003000": {

"interrupt-base": 32,

"interrupt-count": 32

},

像GPIO会统一挂在某个中断号下比如GPIOA：

"irq-s5p4418-gpio@0xc001a000": {

"interrupt-base": 64,

"interrupt-count": 32,

"interrupt-parent": 53

},

不管是irq-pl19还是irq-s5p4418-gpio都可以理解为中断控制器，只是级别不同而已。他们最终会通过填充struct irqchip\_t并调用register\_irqchip或register\_sub\_irqchip来完成注册过程，其结构体定义如下：

struct irqchip\_t

{

char \* name;

int base;

int nirq;

struct irq\_handler\_t \* handler;

void (\*enable)(struct irqchip\_t \* chip, int offset);

void (\*disable)(struct irqchip\_t \* chip, int offset);

void (\*settype)(struct irqchip\_t \* chip, int offset, enum irq\_type\_t type);

void (\*dispatch)(struct irqchip\_t \* chip);

void \* priv;

};

nirq为控制器能处理中断数，handler为设置的处理函数，并且可以发现我们可以enable和disable中断，并设置中断触发类型等等。

### 4.3.2中断分发

中断分发的功能体现在interrupt\_handle\_exception函数中，具体代码如下：

void interrupt\_handle\_exception(void \* regs)

{

struct device\_t \* pos, \* n;

struct irqchip\_t \* chip;

list\_for\_each\_entry\_safe(pos, n, &\_\_device\_head[DEVICE\_TYPE\_IRQCHIP], head)

{

chip = (struct irqchip\_t \*)(pos->priv);

if(chip->dispatch)

chip->dispatch(chip);

}

}

其实就是将注册的中断控制器dispatch逐个调用一遍。我们注册了两个，所以是调用两遍。irq\_pl192的dispatch函数代码如下：

static void irq\_pl192\_dispatch(struct irqchip\_t \* chip)

{

struct irq\_pl192\_pdata\_t \* pdat = (struct irq\_pl192\_pdata\_t \*)chip->priv;

u32\_t val = read32(pdat->virt + VIC\_IRQSTATUS);

if(val != 0)

{

u32\_t offset = \_\_ffs(val);

if((offset >= 0) && (offset < chip->nirq))

{

(chip->handler[offset].func)(chip->handler[offset].data);

write32(pdat->virt + VIC\_SOFTINTCLEAR, 0x1 << offset);

write32(pdat->virt + VIC\_ADDRESS, 0);

}

}

}

通过VIC\_IRQSTATUS寄存器读取去是谁触发了中断，然后调用handler中的func函数指针，完成处理过程。那我们的二级中断控制器irq\_s5p4418\_gpio是怎么处理的呢？其实早在register\_sub\_irqchip就已经将自己dispatch设置为父中断控制器相应handler的func函数指针。因此当irq\_pl192调用func函数指针时，其实是在调用子控制器的dispatch了。不得不说这种设计还是很巧妙的。当然你也会发现，这种方式不支持三级控制器。

### 4.3.3 中断请求

这一小节

## 4.4 文件系统

### 4.4.1 FAT32文件系统

### 4.4.2 虚拟文件系统

### 4.4.3 VFS实现

## 4.5 系统调用

### 4.5.1实现原理

### 4.5.2 open系统调用分析

## 4.6 任务管理

4.6.1进程调度

4.6.2 内核同步

4.6.3init进程

# 第五章 接口和性能测试

## 5.1 接口测试

## 5.2 性能测试

# 参考文献

[1] 田宇．一个64位操作系统的设计与实现[M]．北京：人民邮电出版社，2018 [02] 左飞，万晋森，刘航．Visual C++数字图像处理开发入门与编程实践．电子工业出版社，2008

[2] (荷)安德鲁 S. 塔嫩鲍姆(Andrew S. Tanenbaum)，(荷)赫伯特·博斯(Herbert Bos)．现代操作系统[M]．北京：机械工业出版社，2017

[3] (美)Abraham Silberschatz．操作系统概念[M]．北京：高等教育出版社，2004

[4] 彭东．深度探索嵌入式操作系统[M]．北京：机械工业出版社，2015

[5] (美) William Stallings．操作系统：精髓与设计原理[M]．北京：电子工业出版社，2006

[6] Maurice J.Bach．UNIX操作系统设计[M]．北京：机械工业出版社，2000

[7] (美)塔嫩鲍姆．操作系统设计与实现[M]．北京：电子工业出版社，2007

[8] 于渊．Orange'S一个操作系统的实现[M]．北京：电子工业出版社，2006

[9] 郑钢．操作系统真象还原[M]，北京：人民邮电出版社，2016

[10] 李无言．一步步写嵌入式操作系统[M]．北京：电子工业出版社，2011

# 附录：英文技术资料翻译

**英文原文：**

**Chapter 6 Memory Management Unit**

**6.1 About the MMU**

The MMU works with the L1 and L2 memory system to translate virtual addresses to physical addresses. It also controls accesses to and from external memory.

The Virtual Memory System Architecture version 7 (VMSAv7) features include the following:

• Page table entries that support 4KB, 64KB, 1MB, and 16MB.

• 16 domains.

• Global and address space identifiers to remove the requirement for context switch TLB flushes.

• Extended permissions check capability.

See the ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition for a full architectural description of the VMSAv7.

The processor implements the ARMv7-A MMU enhanced with Security Extensions and multiprocessor extensions to provide address translation and access permission checks. The MMU controls table walk hardware that accesses translation tables in main memory. The MMU enables fine-grained memory system control through a set of virtual-to-physical address mappings and memory attributes.

Note:

In VMSAv7 first level descriptor formats page table base address bit [9] is implementation-defined. In Cortex-A9 processor designs this bit is unused.

The MMU features include the following:

• Instruction side micro TLB.

— Hardware configurable 32 or 64 fully associative entries.

• Data side micro TLB.

— 32 fully associative entries.

• Unified main TLB.

— 2-way associative:

2x32 entry TLB for the 64-entry TLB.

2x64 entry TLB for the 128-entry TLB.

2x128 entry TLB for the 256-entry TLB.

2x256 entry TLB for the 512-entry TLB.

— 4 lockable entries using the lock-by-entry model.

— Supports hardware page table walks to perform lookups in the L1 data cache.

This section contains the following subsections:

• 6.1.1 Memory Management Unit on page 6-110.

**6.1.1 Memory Management Unit**

The MMU checks the Virtual Address and ASID, domain access permissions, and memory attributes.

The MMU also performs the following operations:

• Virtual-to-physical address translation.

• Support for four page (region) sizes.

• Mapping of accesses to cache, or external memory.

• TLB loading for hardware and software.

Domains

The Cortex-A9 processor supports 16 access domains.

TLB

The Cortex-A9 processor implements a 2-level TLB structure. Four entries in the main TLB are lockable.

ASIDs

Main TLB entries can be global, or can be associated with particular processes or applications using Address Space Identifiers (ASIDs). ASIDs enable TLB entries to remain resident during context switches, avoiding the requirement of reloading them subsequently.

Related concepts

Invalidate TLB Entries on ASID Match on page 4-100.

System control coprocessor

TLB maintenance and configuration operations are controlled through a dedicated coprocessor, CP15, integrated within the processor. This coprocessor provides a standard mechanism for configuring the level one memory system.

**6.2 TLB Organization**

The TLB is organized as a micro TLB and a main TLB.

This section contains the following subsections:

• 6.2.1 Micro TLB on page 6-112.

• 6.2.2 Main TLB on page 6-112.

**6.2.1 Micro TLB**

The first level of caching for the page table information is a micro TLB of 32 entries on the data side, and configurable 32 or 64 entries on the instruction side. These blocks provide a fully associative lookup of the virtual addresses in a single CLK signal cycle.

The micro TLB returns the physical address to the cache for the address comparison, and also checks the protection attributes to signal either a Prefetch Abort or a Data Abort.

All main TLB related operations affect both the instruction and data micro TLBs, causing them to be flushed. In the same way, any change of the Context ID Register causes the micro TLBs to be flushed.

**6.2.2 Main TLB**

The main TLB catches the misses from the micro TLBs. It also provides a centralized source for lockable translation entries.

Accesses to the main TLB take a variable number of cycles, according to competing requests from each of the micro TLBs and other implementation-dependent factors. Entries in the lockable region of the main TLB are lockable at the granularity of a single entry. As long as the lockable region does not contain any locked entries, it can be allocated with non-locked entries to increase overall main TLB storage size.

The main TLB is implemented as a combination of:

• A fully-associative, lockable array of four elements.

• A 2-way associative structure of 2x32, 2x64,2x128 or 2x256 entries.

TLB match process

Each TLB entry contains a virtual address, a page size, a physical address, and a set of memory properties. Each is marked as being associated with a particular application space, or as global for all application spaces. CONTEXIDR determines the selected application space.

A TLB entry matches if bits [31:N] of the modified virtual address match, where N is log2 of the page size for the TLB entry. It is either marked as global, or the ASID matched the current ASID.

A TLB entry matches when these conditions are true:

• Its virtual address matches that of the requested address.

• Its Non-secure TLB ID (NSTID) matches the Secure or Non-secure state of the MMU request.

• Its ASID matches the current ASID or is global.

The operating system must ensure that, at most, one TLB entry matches at any time.

Supersections, sections, and large pages are supported to permit mapping of a large region of memory while using only a single entry in a TLB. If no mapping for an address is found in the TLB, then the translation table is automatically read by hardware and a mapping is placed in the TLB.

TLB lockdown

The TLB supports the TLB lock-by-entry model as described in the ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition.

Related references

4.3.26 TLB lockdown operations on page 4-97.

**6.3 Memory access sequence**

When the processor generates a memory access, the MMU performs a lookup for the requested virtual address and current ASID and security state in the relevant instruction or data micro TLB.

If there is a miss in the micro TLB, the MMU performs a lookup for the requested virtual address and current ASID and security state in the main TLB. If there is a miss in the main TLB, the MMU performs a hardware translation table walk.

You can configure the MMU to perform hardware translation table walks in cacheable regions by setting the IRGN bits in the Translation Table Base Registers. If the encoding of the IRGN bits is write-back, then an L1 data cache lookup is performed and data is read from the data cache. If the encoding of the IRGN bits is write-through or non-cacheable then an access to external memory is performed.

The MMU might not find a global mapping, or a mapping for the selected ASID, with a matching Non-secure TLB ID (NSTID) for the virtual address in the TLB. In this case, the hardware does a translation table walk if the translation table walk is enabled by the PD0 or PD1 bit in the TTB Control Register. If translation table walks are disabled, the processor returns a Section Translation fault.

If the MMU finds a matching TLB entry, it uses the information in the entry as follows:

1. The access permission bits and the domain determine if the access is enabled. If the matching entry does not pass the permission checks, the MMU signals a memory abort. See the ARM® Architecture

Reference Manual, ARMv7-A and ARMv7-R edition for a description of access permission bits, abort types and priorities, and for a description of the IFSR and Data Fault Status Register (DFSR).

2. The memory region attributes specified in both the TLB entry and the CP15 c10 remap registers control the cache and write buffer, and determine if the access is

• Secure or Non-secure.

• Shared or not.

• Normal memory, Device, or Strongly-ordered.

3. The MMU translates the virtual address to a physical address for the memory access.

If the MMU does not find a matching entry, a hardware table walk occurs.

**6.4 MMU enabling or disabling**

You can enable or disable the MMU as described in the ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition.

**6.5 External aborts**

External memory errors are defined as those that occur in the memory system rather than those that are detected by the MMU. External memory errors are expected to be extremely rare. External aborts are caused by errors flagged by the AXI interfaces when the request goes external to the processor. External aborts can be configured to trap to Monitor mode by setting the EA bit in the Secure Configuration Register.

This section contains the following subsections:

• 6.5.1 External aborts on data read or write on page 6-116.

• 6.5.2 Synchronous and asynchronous aborts on page 6-116.

**6.5.1 External aborts on data read or write**

Externally generated errors during a data read or write can be asynchronous. This means that the r14\_abt on entry into the abort handler on such an abort might not hold the address of the instruction that caused the exception.

The DFAR is UNPREDICTABLE when an asynchronous abort occurs.

In the case of a load multiple or store multiple operation, the address captured in the DFAR is that of the address that generated the synchronous external abort.

**6.5.2 Synchronous and asynchronous aborts**

To determine a fault type, read the DFSR for a data abort or the IFSR for an instruction abort.

The processor supports an Auxiliary Fault Status Register for software compatibility reasons only. The processor does not modify this register because of any generated abort.

**中文译文：**

**第六章内存管理单元**

**6.1关于MMU**

MMU与L1和L2内存系统一起工作，将虚拟地址转换为物理地址。它还控制对外部内存的访问和来自外部内存的访问。

虚拟内存系统体系结构版本7 (VMSAv7)的特性包括:

• 支持4KB、64KB、1MB和16MB的页表条目。

• 16域。

• 全局和地址空间标识符消除了上下文切换TLB刷新的要求。

• 扩展权限检查功能。

查看ARM®体系结构参考手册,ARMv7-A和ARMv7-R版以获取VMSAv7h的完整架构描述。

处理器通过安全扩展和多处理器扩展来实现ARMv7-A MMU，以提供地址转换和访问权限检查。MMU控制访问主存中的转换表的表遍历硬件。MMU通过一组虚拟地址到物理地址的映射和内存属性，进行细粒度的内存系统控制。

注意:

在VMSAv7一级描述符格式中，页表基本地址位[9]有具体定义的。在Cortex-A9处理器设计中，这个位是却未使用的。

MMU的特点包括:

• 指令微TLB。

- 硬件可配置32或64个全相联条目。

• 数据微TLB。

- 32个全相联条目。

• 统一主TLB。

- 双路相联:

2x32 entry TLB 对于 64-entry TLB.

2x64 entry TLB 对于 128-entry TLB.

2x128 entry TLB 对于 256-entry TLB.

2x256 entry TLB 对于 512-entry TLB.

- 4个使用逐项锁定模型的可锁定条目。

-支持硬件页表遍历来进行L1数据缓存查找。

本节包括以下各节:

• 6.1.1内存管理单元，见第6-110页。

**6.1.1内存管理单元**

MMU检查虚拟地址和ASID、域访问权限和内存属性。

MMU还执行以下操作:

• 虚拟地址到物理地址的转换。

• 支持四页(区域)大小。

• 缓存或外部内存访问的映射。

• 硬件和软件的TLB加载。

域

Cortex-A9处理器支持16个访问域。

TLB

Cortex-A9处理器实现了一个2级的TLB结构。主TLB中的4个条目是可锁定的。

ASIDs

主TLB条目可以是全局的，也可以与使用地址空间标识符(ASIDs)的特定进程或应用程序相关联。ASIDs允许TLB条目在上下文切换期间保持驻留，从而避免随后重新加载它们的需求。

相关的概念

使ASID匹配的TLB条目失效，见第4-100页。

系统控制协处理器

TLB的维护和配置操作通过集成在处理器中的专用协处理器CP15进行控制。这个协处理器提供了配置一级内存系统的标准机制。

**6.2 TLB组织**

TLB组织为微TLB和主TLB。

本节包括以下各节:

• 6.2.1 微 TLB，见第6-112页。

• 6.2.2 主TLB，见第6-112页。

**6.2.1微TLB**

页表信息的第一级缓存是32个条目的数据微TLB，32或64个指令条目。这些块在单个CLK信号周期中提供虚拟地址的全相联查找。

微TLB将物理地址返回缓存以进行地址比较，并检查保护属性以发出预取中止或数据中止的信号。

所有与TLB相关的主要操作都会影响指令和数据微TLBs，导致它们被刷新。同样，上下文ID寄存器的任何更改都会导致微TLBs被刷新。

**6.2.2主TLB**

主TLB从微TLBs中捕捉缺失。它还为可锁定的转换条目提供了一个集中的源。

对主TLB的访问周期是不固定的，具体有来自每个微TLBs和其他依赖实现的因素的竞争性请求。，主TLB的可锁定区域的中的每个条目是可在单个条目的粒度上锁定的。只要可锁定区域不包含任何已锁定项，就可以使用非锁定项分配该区域，以增加主TLB总体存储大小。

主要的TLB实现为:

• 四个元素组成的全相联可锁定数组。

• 2x32、2x64、2x128或2x256双路相联结构。

TLB匹配过程

每个TLB条目包含一个虚拟地址、一个页面大小、一个物理地址和一组内存属性。每个都标记为与特定的应用程序空间相关联，或者作为所有应用程序全局空间。CONTEXIDR决定所选的应用程序空间。

如果修改后的虚拟地址匹配位[31:N]，则TLB条目匹配，其中N为TLB条目页面大小的log2。它要么标记为全局的，要么与当前ASID匹配。

当这些条件为真时，TLB条目匹配:

• 它的虚拟地址与请求地址匹配。

• 其非安全TLB ID (NSTID)匹配MMU请求的安全或非安全状态。

• 它的ASID匹配当前ASID或是全局的。

操作系统必须确保在任何时候最多匹配一个TLB条目。

可以使用单个条目在TLB中映射大内存区域比如超节、节和大页。如果在TLB中没有找到地址的映射，那么硬件将自动读取转换表，并在TLB中设置映射。

TLB封锁

TLB支持TLB lock-by-entry模型，具体描述在ARMv7-A 和ARMv7-R版的ARM®体系结构参考手册。

相关的参考文献

4.3.26 TLB锁定操作，见第4-97页。

**6.3内存访问顺序**

当处理器生成内存访问时，MMU在相关指令或数据微TLB中查找请求的虚拟地址、当前ASID和安全状态。

如果在微TLB中出现遗漏，MMU将对请求的虚拟地址、当前ASID和主TLB中的安全状态执行查找操作。如果在主TLB中出现缺失，MMU将执行硬件转换表遍历。

你可以通过设置转换表基地址寄存器的IRGN位，将MMU配置为在可缓存区域中执行硬件转换表遍历。如果IRGN位的编码是写回的，则执行L1数据缓存查找并从数据缓存中读取数据。如果IRGN位的编码是写通的或不可缓存的，则执行对外部内存的访问。

MMU可能无法在TLB中找到一个全局映射或特定ASID的映射的虚拟地址，这个映射匹配非安全TLB ID (NSTID)。在这种情况下，如果转换表遍历由TTB控制寄存器中的PD0或PD1位启用，硬件将执行转换表遍历。如果转换表的遍历被禁用，处理器将返回一个段转换错误。

如果MMU找到一个匹配的TLB条目，它将使用如下条目信息:

1.访问权限位和域决定是否启用访问。如果匹配的条目没有通过权限检查，MMU将发出内存中止的信号。查看ARMv7-A和ARMv7-R版本的ARM®体系结构参考手册中关于访问权限位、中止类型和优先级的描述，以及IFSR和数据故障状态寄存器(DFSR)的描述。

2.在TLB条目和CP15 c10重映射寄存器中指定的内存区域属性控制缓存和写缓冲区，并确定访问是否是

• 安全或不安全。

• 是否共享。

• 正常内存、设备或强顺序。

3.MMU将虚拟地址转换为物理地址以便进行内存访问。

如果MMU没有找到匹配的条目，就会出现硬件表遍历。

**6.4 MMU启用或禁用**

您可以启用或禁用MMU，具体描述在ARMv7-A ARMv7-R版的ARM®架构参考手册中描。

**6.5外部中止**

外部内存错误定义为发生在内存系统中的错误，而不是MMU检测到的错误。外部内存错误是非常罕见的。外部中止是由请求到处理器外部时AXI接口标记的错误引起的。通过在安全配置寄存器中设置EA位，可以将外部中止配置为陷阱至监视模式。

本节包括以下各节:

· 6.5.1数据读写的外部中止，见第6-116页。

· 6.5.2同步和异步中止，见第6-116页。

**6.5.1数据读写的外部中止**

数据读写过程中外部生成的错误可以是异步的。这意味着在进入中止处理程序时r14\_abt可能不包含导致异常的指令的地址。

当发生异步中止时，DFAR是不可预测的。

在加载多个或存储多个操作的情况下，DFAR中捕获的地址是产生同步外部中止的地址。

**6.5.2同步和异步中止**

若要确定故障类型，请读取数据中止的DFSR或指令中止的IFSR。

由于软件兼容性的原因，处理器支持辅助故障状态寄存器。处理器不会因为任何生成的中止而修改这个寄存器。