# ARMv8-a架构简介

## 1. 前言

ARMv8（当前只有A系列，即ARMv8-A）架构，是ARM公司为满足新需求而重新设计的一个架构，是近20年来，ARM架构变动最大的一次。它引入的Execution State、Exception Level、Security State等新特性，已经和我们对旧的ARM架构的认知，有很大差距了。

因此，本文从ARMv8-A产生的背景开始，对它进行一个简单的介绍，使大家从整体上，对ARMv8有一个简单的了解。

## 2. 背景

本节参考自“<http://www.arm.com/zh/files/downloads/ARMv8_white_paper_v5.pdf>”，感兴趣的同学可以自行阅读。

有一点是可以确定的，ARM诞生时，对Intel主导的PC市场，没有（也不敢有）一点点的非分之想。最初的ARMv4（ARM7系列），到最近的ARMv7（Cortex-A,-M,-R系列），都是针对功耗比较敏感的移动设备的，就性能而言，基于ARM处理器的设备，始终无法和PC相提并论。

但从ARMv7开始，情况开始有些转变，ARM的市场开始扩展到移动设备之外的其它领域，这也是ARMv7划分为A（Application）、R（Real-time）和M（Microcontroller）三个系列的原因，其实质就是三个细分市场，其中的A系列，就是针对性能要求较高的应用。

特别是在Cortex-A9之后，ARM的处理性能有很大的提高，渐渐的吸引了一些PC用户。因此基于ARM的类PC产品，如平板电脑，开始大量涌现。此时，ARM的处理能力，已经有机会应用于其它领域了，如企业设备、服务器等，当然，其优势依然是低功耗。

与此同时，新的趋势正在酝酿，主要包括大内存（Large Memory）、虚拟化（Virtualization）和安全（Security）。Virtualization在ARMv7上已经有简单的硬件实现，Security也有可能基于当前架构扩展，唯有Large memory的需求，有点棘手。

由于处理器性能越来越强，运行于其上的软件也来越复杂，复杂到单一应用对内存的需求可能超出32-bit架构所能支持的最大内存（4G），这就是Large memory需求的起因。不过，后来的Cortex-A15（ARMv7架构）通过Large Physical Address Extensions (LPAE) 技术，可以支持高达40bits的物理地址空间。但受限于32-bit的指令集，虚拟地址空间依旧只有32bits（4G），如果有应用需要更大的虚拟内存，怎么办？只能定义一个新的架构，使用64-bit的指令集（也即我们常说的ARM64）。

毫无疑问，在现阶段，需要超过4G虚拟内存的应用场景，是非常少的。但ARM还是定义了一个新的架构--ARMv8，为什么呢？下面是ARM的解释（只有伟大的公司才有伟大的理念！）：

Trends. That’s really what ARM has to look at when defining a new architecture. That is the nature of our business, we need to look a long way forward, and plan.

当然，ARMv8并不仅仅是为了解决虚拟地址的问题，它也要解决现有架构的一些问题。不过，新的问题又来了：一个新的架构？用户为什么要使用新的架构？因此，ARMv8的定义，必须先满足如下前提条件：

1）对上兼容。

2）能解决现存架构的已知问题。

3）相比现存架构，必须具备优势明显的新特性，哪怕软件从来不使用这些新特性。

以上就是ARMv8-a产生的背景，也是ARMv8-a架构之所以是“这个”样子的直接原因。那么到底是什么样子呢？我们继续介绍。

## 3. ARMv8-a架构简介

基于上面的前提条件，ARMv8-a架构的主要特性包括：

1. 新增一套64-bit的指令集，称作A64。
2. 由于需要向前兼容ARMv7，所以同时支持现存的32-bit指令集，称作A32和T32（也即我们熟悉的ARM和Thumb指令集）。
3. 定义AArch64和AArch32两套运行环境（称作Execution state），分别执行64-bit和32-bit指令集。软件可以在需要的时候，切换Execution state。
4. AArch64最大的改动，使用新的概念（exception level），重新解释了processor mode、privilege level等概念，具体可参考第4章的介绍。
5. 在ARMv7 security extension的基础上，新增security model，支持安全相关的应用需求。
6. 在ARMv7 virtualization extension的基础上，提供完整的virtualization框架，从硬件上支持虚拟化。

## 4. AArch64 Exception level

Exception level，是ARMv8-a引入的一个新概念，用于整合之前架构中processor mode和privilege level相关的功能。

**4.1 ARMv7之前的实现**

我们知道，以前的ARM架构，处理器可以工作在多种模式（称作processor mode）下，包括User、FIQ、IRQ、Abort、Undefined、System等，之所以存在不同的模式，主要有2个方面的考虑：

1. 不同的处理器模式，有不同的硬件访问权限，称作privilege level。
2. 主要有2个level，privilege和non-privilege。其中只有User模式属于non-privilege level，其它均是privilege level。
3. 安全起见，大多数时候，软件都运行在User mode。一旦需要其它操作，则需要切换到相应的privilege模式下。这是最原始、最朴素的安全思想，当然，只防君子，不防小人。
4. 这些处理器模式，除User模式外，其它模式基本上和各类异常一一对应。而不同的模式，都有一些自己独有的寄存器，例如R13(SP)、R14(LR)等等，可以使模式切换过程（也是异常处理过程）更为高效、便利。

**4.2 ARMv7-a的实现**

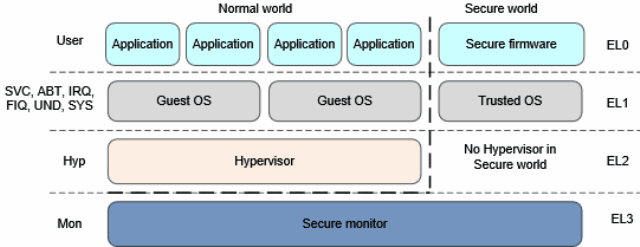
ARMv7-a基本保留了之前的设计，不同之处，将privilege level命名了，称作PL0和PL1（也许您猜到了，后来出现了PL2，用于虚拟化扩展（Virtualization Extension）。

另外，增加了两个模式：Monitor和Supervisor，分别用于security扩展和virtualization扩展。

**4.3 ARMv8-a的实现**

可能ARMv8-a的设计者觉得之前的设计有些啰嗦，就把processor mode的概念去掉（或者说淡化）了，取而代之的是4个固定的Exception level，简称EL0-EL3。同时，也淡化了privilege level的概念。Exception level本身就已经包好了privilege的信息，即ELn的privilege随着n的增大而增大。类似地，可以将EL0归属于non-privilege level，EL1/2/3属于privilege level。

这些Exception level的现实意义是（如下图，先忽略Secure model有关的内容）：

[](http://www.wowotech.net/content/uploadfile/201507/7ffa71881f6356c740ef254b01cecf1520150707143059.gif)

**ARMv8-a Exception level有关的说明如下：**

1）首先需要注意的是，AArch64中，已经没有User、SVC、ABT等处理器模式的概念，但ARMv8需要向前兼容，在AArch32中，就把这些处理器模式map到了4个Exception level。

2）Application位于特权等级最低的EL0，Guest OS（Linux kernel、window等）位于EL1，提供虚拟化支持的Hypervisor位于EL2（可以不实现），提供Security支持的Seurity Monitor位于EL3（可以不实现）。

3）只有在异常发生时（或者异常处理返回时），才能切换Exception level（这也是Exception level的命名原因，为了处理异常）。当异常发生时，有两种选择，停留在当前的EL，或者跳转到更高的EL，EL不能降级。同样，异常处理返回时，也有两种选择，停留在当前EL，或者调到更低的EL。

注1：有关ARMv8-a异常处理的具体细节，会在其它文章中描述。

**5. security model**

ARMv8-a的security模型基本沿用了ARMv7 security extension的思路，主要目的保护一些安全应用的数据，例如支付等。它不同于privilege level等软件逻辑上的保护，而是一种物理上的区隔，即不同security状态下，可以访问的物理内存是不同的。

ARMv8-a架构有两个security state（参考上面图片），Security和non-Security。主要的功效是物理地址的区隔，以及一些system control寄存器的访问控制：

在Security状态下，处理器可以访问所有的Secure physical address space以及Non-secure physical address space；

在Non-security状态下，只能访问Non-secure physical address space，且不能访问Secure system control resources。

6. virtualization

硬件虚拟化包括指令集虚拟化、异常处理虚拟化、MMU虚拟化、IO虚拟化等多个议题，比较复杂，这里先不描述了。

7. 总结

本文简单介绍了ARMv8-a中的一些概念，后续文章将会重点关注异常处理模型、security模型、virtualization模型。

# ARM64的启动过程 内核第一个脚印

ARM64的启动过程之（一）：内核第一个脚印

一、前言

kernel的整个启动过程涉及的内容很多，不可能每一个细节都描述清楚，因此我打算针对部分和ARM64相关的启动步骤进行学习、整理，并方便后续查阅。本文实际上描述在系统启动最开始的时候，bootloader和kernel的交互以及kernel如何保存bootloader传递的参数并进行校验，此外，还有一些最基础的硬件初始化的内容。

本文中的source来自4.1.10内核，这是一个long term的版本，后续一段时间的文章都会基于这个long term版本进行。

二、进入kernel之前

系统启动过程中，linux kernel不是一个人在战斗，在kernel之前bootloader会执行若干的动作，然后把控制权转移给linux kernel。需要特别说明的是：这里bootloader是一个宽泛的概念，其实就是为kernel准备好执行环境的那些软件，可能是传统意义的bootloader（例如Uboot），也可能是Hypervisor或者是secure monitor。具体bootloader需要执行的动作包括：

1. 初始化系统中的RAM并将RAM的信息告知kernel
2. 准备好device tree blob的信息并将dtb的首地址告知kernel
3. 解压内核（可选）
4. 将控制权转交给内核。当然，bootloader和kernel的交互的时候需求如下：

MMU = off, D-cache = off, I-cache = on or off   
   x0 = physical address to the FDT blob

这里需要对data cache和instruction cache多说几句。我们知道，具体实现中的ARMv8处理器的cache是形成若干个level，一般而言，可能L1是分成了data cache和instruction cache，而其他level的cache都是unified cache。上面定义的D-cache off并不是说仅仅disable L1的data cache，实际上是disable了各个level的data cache和unified cache。同理，对于instruction cache亦然。

此外，在on/off控制上，MMU和data cache是有一定关联的。在ARM64中，SCTLR, System Control Register用来控制MMU icache和dcache，虽然这几个控制bit是分开的，但是并不意味着MMU、data cache、instruction cache的on/off控制是彼此独立的。一般而言，这里MMU和data cache是绑定的，即如果MMU 是off的，那么data cache也必须要off。因为如果打开data cache，那么要设定memory type、sharebility attribute、cachebility attribute等，而这些信息是保存在页表（Translation table）的描述符中，因此，如果不打开MMU，如果没有页表翻译过程，那么根本不知道怎么来应用data cache。当然，是不是说HW根本不允许这样设定呢？也不是了，在MMU OFF而data cache是ON的时候，这时候，所有的memory type和attribute是固定的，即memory type都是normal Non-shareable的，对于inner cache和outer cache，其策略都是Write-Back，Read-Write Allocate的。

更详细的ARM64 boot protocol请参考Documentation/arm64/booting.txt文档。

三、参数的保存和校验

最开始的ARM64启动代码位于arch/arm64/kernel/head.S文件中，代码如下：

ENTRY(stext)   
    bl    preserve\_boot\_args   
    bl    el2\_setup            // Drop to EL1, w20=cpu\_boot\_mode   
    adrp    x24, \_\_PHYS\_OFFSET   
    bl    set\_cpu\_boot\_mode\_flag

bl    \_\_vet\_fdt    
    ……   
ENDPROC(stext)

1、preserve\_boot\_args

preserve\_boot\_args:   
    mov    x21, x0－－－－－－将dtb的地址暂存在x21寄存器中，释放出x0以便后续做临时变量使用

    adr\_l    x0, boot\_args－－－x0保存了boot\_args变量的地址   
    stp    x21, x1, [x0]－－－－保存x0和x1的值到boot\_args[0]和boot\_args[1]   
    stp    x2, x3, [x0, #16] －－－保存x2和x3的值到boot\_args[2]和boot\_args[3]

    dmb    sy－－－－－－－－－full system data memory barrier

    add    x1, x0, #0x20－－－－x0和x1是传递给\_\_inval\_cache\_range的参数   
    b    \_\_inval\_cache\_range   
ENDPROC(preserve\_boot\_args)

**由于MMU = off, D-cache = off**，因此写入boot\_args变量的操作都是略过data cache的，直接写入了RAM中（前面说过了，这里的D-cache并不是特指L1的data cache，而是各个level的data cache和unified cache），为了安全起见（也许bootloader中打开了D-cache并操作了boot\_args这段memory，从而在各个level的data cache和unified cache有了一些旧的，没有意义的数据），需要将boot\_args变量对应的cache line进行清除并设置无效。在调用\_\_inval\_cache\_range之前，x0是boot\_args这段memory的首地址，x1是末尾的地址（boot\_args变量长度是4x8byte＝32byte，也就是0x20了）。

为何要保存x0～x3这四个寄存器呢？因为ARM64 boot protocol对启动时候的x0～x3这四个寄存器有严格的限制：x0是dtb的物理地址，x1～x3必须是0（非零值是保留将来使用）。在后续setup\_arch函数执行的时候会访问boot\_args并进行校验。

对于invalidate cache的操作而言，我们可以追问几个问题：如果boot\_args所在区域的首地址和尾部地址没有对齐到cache line怎么办？具体invalidate cache需要操作到那些level的的cache？这些问题可以通过阅读\_\_inval\_cache\_range的代码获得答案，这里就不描述了。

还有一个小细节是如何访问boot\_args这个符号的，这个符号是一个虚拟地址，但是，现在没有建立好页表，也没有打开MMU，如何访问它呢？这是通过adr\_l这个宏来完成的。这个宏实际上是通过adrp这个汇编指令完成，通过该指令可以将符号地址变成运行时地址（通过PC relative offset形式），因此，当运行的MMU OFF mode下，通过adrp指令可以获取符号的物理地址。不过adrp是page对齐的（adrp中的p就是page的意思），boot\_args这个符号当然不会是page size对齐的，因此不能直接使用adrp，而是使用adr\_l这个宏进行处理，如果读者有兴趣可以自己看source code。

最后，我们来解释一下dmb    sy这一条指令。在ARM ARM文档中，有关于数据访问指令和 data cache指令之间操作顺序的约定，原文如下：

All data cache instructions, other than DC ZVA, that specify an address can execute in any order relative to loads or stores that access any address with the Device memory attribute,or with Normal memory with Inner Non-cacheable attribute unless a DMB or DSB is executed between the instructions.

因此，在Non-cacheable的情况下，必须要使用DMB来保证stp指令在dc ivac指令之前执行完成。

2、el2\_setup

程序执行至此，CPU处于哪一个exception level呢？根据ARM64 boot protocol，CPU要么处于EL2（推荐）或者non-secure EL1。如果在EL1，情形有些类似过去arm处理器的感觉，处于EL2稍微复杂一些，需要对virtualisation extensions进行基本的设定，然后将cpu退回到EL1。代码太长了，我们分成两段来阅读，第一段如下：

ENTRY(el2\_setup)   
    mrs    x0, CurrentEL－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    cmp    x0, #CurrentEL\_EL2－－－－－－判断是否处于EL2   
    b.ne    1f－－－－－－－－－－－－－－不是的话，跳到1f   
    mrs    x0, sctlr\_el2－－－－－－－－－－－－－－－－－－－－－－－－－（2）   
CPU\_BE(    orr    x0, x0, #(1 << 25)    )    // Set the EE bit for EL2   
CPU\_LE(    bic    x0, x0, #(1 << 25)    )    // Clear the EE bit for EL2   
    msr    sctlr\_el2, x0－－－－写回sctlr\_el2寄存器   
    b    2f   
1:    mrs    x0, sctlr\_el1－－－－－－－－－－－－－－－－－－－－－－－－－（3）   
CPU\_BE(    orr    x0, x0, #(3 << 24)    )    // Set the EE and E0E bits for EL1   
CPU\_LE(    bic    x0, x0, #(3 << 24)    )    // Clear the EE and E0E bits for EL1   
    msr    sctlr\_el1, x0   
    mov    w20, #BOOT\_CPU\_MODE\_EL1－－－－w20寄存器保存了cpu启动时候的Eexception level   
    isb－－－－－－－－－instruction memory barrier   
    ret

2:    mov    x0, #(1 << 31) －－－－－－－－－－－－－－－－－－－－－－－－（4）   
    msr    hcr\_el2, x0

    mrs    x0, cnthctl\_el2 －－－－－－－－－－－－－－－－－－－－－－－－－（5）   
    orr    x0, x0, #3                 // Enable EL1 physical timers   
    msr    cnthctl\_el2, x0   
    msr    cntvoff\_el2, xzr        // Clear virtual offset

    mrs    x0, id\_aa64pfr0\_el1 －－－－－－－－－－－－－－－－－－－－－－－（6）   
    ubfx    x0, x0, #24, #4 －－－－取出24 bit开始的4个bit的值并将该值赋给x0   
    cmp    x0, #1   
    b.ne    3f －－－－－不支持system register接口

    mrs\_s    x0, ICC\_SRE\_EL2   
    orr    x0, x0, #ICC\_SRE\_EL2\_SRE    // Set ICC\_SRE\_EL2.SRE==1   
    orr    x0, x0, #ICC\_SRE\_EL2\_ENABLE    // Set ICC\_SRE\_EL2.Enable==1   
    msr\_s    ICC\_SRE\_EL2, x0   
    isb                    // Make sure SRE is now set   
    msr\_s    ICH\_HCR\_EL2, xzr        // Reset ICC\_HCR\_EL2 to defaults

3:               ……

（1）当前的exception level保存在PSTATE中，程序可以通过MRS或者MSR来访问PSTATE，当然需要传递一个Special-purpose register做为参数，CurrentEL就是获取PSTATE中current exception level域的特殊寄存器。

（2）sctlr\_el2也是一个可以通过MRS/MSR指令访问的寄存器，当CPU处于EL2状态的时候，该寄存器可以控制整个系统的行为。当然，这里仅仅是设定EL2下的数据访问和地址翻译过程中的endianess配置，也就是EE bit[25]。根据配置，CPU\_BE和CPU\_LE包围的指令只会保留一行。对于little endian而言，实际上就是将sctlr\_el2寄存器的EE（bit 25）设定为0。顺便说一下，这个bit不仅仅控制EL2数据访问的endianess以及EL2 stage 1的地址翻译过程中的endianess（当然，EL2只有stage 1），还可以控制EL1和EL0 stage 2地址翻译的过程的endianess（这时候有两个stage的地址翻译过程）。

（3）执行到这里说明CPU处于EL1，这种状态下没有权限访问sctlr\_el2，只能是访问sctlr\_el1。sctlr\_el1可以通过EE和E0E来控制EL1和EL0状态下是little endian还是big endian。EE bit控制了EL1下的数据访问以及EL1和EL0 stage 1地址翻译的过程的endianess。E0E bit用来控制EL0状态下的数据访问的endianess。此外，需要注意的是：由于修改了system control register（设定endianess状态），因此需要一个isb来同步（具体包括两部分的内容，一是确认硬件已经执行完毕了isb之前的所有指令，包括修改system control寄存器的那一条指令，另外一点是确保isb之后的指令从新来过，例如取指，校验权限等）。

（4）执行到这里说明CPU处于EL2，首先设定的是hcr\_el2寄存器，Hypervisor Configuration Register。该寄存器的大部分bit 值在reset状态的时候就是0值，只不过bit 31（Register Width Control）是implementation defined，因此这里set 31为1，确保Low level的EL1也是Aarch64的

（5）这一段代码是对Generic timers进行配置。要想理解这段代码，我们需要简单的了解一些ARMv8上Generic timer的运作逻辑。一个全局范围的system counter、各个PE上自己专属的local timer以及连接这些组件之间的bus或者信息传递机制组成了Generic Timer。对于PE而言，通过寄存器访问，它能看到的是physical counter（实际的system counter计数）、virtual counter（physical counter基础上的offset）、physical timer、virtual timer等。NTHCTL\_EL2，Counter-timer Hypervisor Control register，用来控制系统中的physical counter和virutal counter如何产生event stream以及在EL1和EL0状态访问physical counter和timer的硬件行为的。在EL1（EL0）状态的时候访问physical counter和timer有两种配置，一种是允许其访问，另外一种就是trap to EL2。这里的设定是：不陷入EL2（对应的bit设置为1）。更详细的信息可以参考ARMv8 ARM文档。cntvoff\_el2是virtual counter offset，所谓virtual counter，其值就是physical counter的值减去一个offset的值（也就是cntvoff\_el2的值了），这里把offset值清零，因此virtual counter的计数和physical counter的计数是一样的。

（6）这一段代码是对GIC V3进行配置。ID\_AA64PFR0\_EL1，AArch64 Processor Feature Register 0，该寄存器描述了PE实现的feature。GIC bits [27:24]描述了该PE是否实现了system register来访问GIC，如果没有（GIC bits 等于0）那么就略过GIC V3的设定。ICC\_SRE\_EL2，Interrupt Controller System Register Enable register (EL2)，该寄存器用来（在EL2状态时候）控制如何访问GIC CPU interface模块的，可以通过memory mapped方式，也可以通过system register的方式。将SRE bit设定为1确保通过system register方式进行GIC interface cpu寄存器的访问。将enable bit设定为1确保在EL1状态的时候可以通过ICC\_SRE\_EL1寄存器对GIC进行配置而不是陷入EL2。

下面我们进入第二段代码：

    mrs    x0, midr\_el1 －－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    mrs    x1, mpidr\_el1   
    msr    vpidr\_el2, x0   
    msr    vmpidr\_el2, x1

    mov    x0, #0x0800            // Set/clear RES{1,0} bits －－－－－－－－－－－－－－－（2）   
CPU\_BE(    movk    x0, #0x33d0, lsl #16    )    // Set EE and E0E on BE systems   
CPU\_LE(    movk    x0, #0x30d0, lsl #16    )    // Clear EE and E0E on LE systems   
    msr    sctlr\_el1, x0

    mov    x0, #0x33ff－－－－－－－Disable Coprocessor traps to EL2   
    msr    cptr\_el2, x0            // Disable copro. traps to EL2

#ifdef CONFIG\_COMPAT－－－－－是否支持64 bit kernel上运行32bit 的application   
    msr    hstr\_el2, xzr            // Disable CP15 traps to EL2   
#endif

    mrs    x0, pmcr\_el0－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（3）   
    ubfx    x0, x0, #11, #5            // to EL2 and allow access to   
    msr    mdcr\_el2, x0            // all PMU counters from EL1    
    msr    vttbr\_el2, xzr －－－－清除Stage-2 translation table base address register

    adrp    x0, \_\_hyp\_stub\_vectors   
    add    x0, x0, #:lo12:\_\_hyp\_stub\_vectors   
    msr    vbar\_el2, x0 －－－－－－－－－－－－－－－设定EL2的异常向量表的基地址

    mov    x0, #(PSR\_F\_BIT | PSR\_I\_BIT | PSR\_A\_BIT | PSR\_D\_BIT |\   
              PSR\_MODE\_EL1h)   
    msr    spsr\_el2, x0 －－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（4）   
    msr    elr\_el2, lr   
    mov    w20, #BOOT\_CPU\_MODE\_EL2        // This CPU booted in EL2   
    eret－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（5）  
ENDPROC(el2\_setup)

（1）midr\_el1和mpidr\_el1都属于标识该PE信息的read only寄存器。MIDR\_EL1，Main ID Register主要给出了该PE的architecture信息，Implementer是谁等等信息。MPIDR\_EL1，Multiprocessor Affinity Register，该寄存器保存了processor ID。vpidr\_el2和vmpidr\_el2是上面的两个寄存器是对应的，只不过是for virtual processor的。

（2）这段代码实际上是将0x33d00800（BE）或者0x30d00800（LE）写入sctlr\_el1寄存器。BE和LE的设定和上面第一段代码中的描述是类似的，其他bit的设定请参考ARMv8 ARM文档

（3）PMCR\_EL0，Performance Monitors Control Register，该寄存器的[15:11]标识了支持的Performance Monitors counter的数目，并将其设定到MDCR\_EL2（Monitor Debug Configuration Register (EL2)）中。MDCR\_EL2中其他的bit都设定为0，其结果就是允许EL0和EL1进行debug的操作（而不是trap to EL2），允许EL1访问Performance Monitors counter（而不是trap to EL2）。

（4）当系统发生了异常并进入EL2，SPSR\_EL2，Saved Program Status Register (EL2)会保存处理器状态，ELR\_EL2，Exception Link Register (EL2)会保存返回发生exception的现场的返回地址。这里是设定SPSR\_EL2和ELR\_EL2的初始值。w20寄存器保存了cpu启动时候的Eexception level ，因此w20被设定为BOOT\_CPU\_MODE\_EL2。

（5）eret指令是用来返回发生exception的现场。实际上，这个指令仅仅是模拟了一次异常返回而已，SPSR\_EL2和ELR\_EL2都已经设定OK，执行该指令会使得CPU返回EL1状态，并且将SPSR\_EL2的值赋给PSTATE，ELR\_ELR就是返回地址（实际上也恰好是函数的返回地址）。

完成了el2\_setup这个函数分析之后，我们再回头思考这样的问题：为何是el2\_setup？为了没有el3\_setup？当一个SOC的实现在包括了EL3的支持，那么CPU CORE缺省应该进入EL3状态，为何这里只是判断EL2还是EL1，从而执行不同的流程，如果是EL3状态，代码不就有问题了吗？实际上，即便是由于SOC支持TrustZone而导致cpu core上电后进入EL3，这时候，接管cpu控制的一定不是linux kernel（至少目前来看linux kernel不会做Secure monitor），而是Secure Platform Firmware（也就是传说中的secure monitor），它会进行硬件平台的初始化，loading trusted OS等等，等到完成了secure world的构建之后，把控制权转交给non-secure world，这时候，CPU core多半处于EL2（如果支持虚拟化）或者EL1（不支持虚拟化）。因此，对于linux kernel而言，它感知不到secure world（linux kernel一般也不会做Trusted OS），仅仅是在non-secure world中呼风唤雨，可以是Hypervisor或者rich OS。

**3、set\_cpu\_boot\_mode\_flag**

在进入这个函数的时候，有一个前提条件：w20寄存器保存了cpu启动时候的Eexception level ，具体代码如下：

ENTRY(set\_cpu\_boot\_mode\_flag)   
    adr\_l    x1, \_\_boot\_cpu\_mode   
    cmp    w20, #BOOT\_CPU\_MODE\_EL2   
    b.ne    1f   
    add    x1, x1, #4   
1:    str    w20, [x1]            // This CPU has booted in EL1   
    dmb    sy   
    dc    ivac, x1            // Invalidate potentially stale cache line   
    ret   
ENDPROC(set\_cpu\_boot\_mode\_flag)

由于系统启动之后仍然需要了解cpu启动时候的Eexception level（例如判断是否启用hyp mode），因此，有一个全局变量\_\_boot\_cpu\_mode用来保存启动时候的CPU mode。代码很简单，大家自行体会就OK了，我这里补充几点描述：

（1）本质上我们希望系统中所有的cpu在初始化的时候处于同样的mode，要么都是EL2，要么都是EL1，有些EL2，有些EL1是不被允许的（也许只有那些精神分裂的bootloader才会这么搞）。

（2）所有的cpu core在启动的时候都处于EL2 mode表示系统支持虚拟化，只有在这种情况下，kvm模块可以顺利启动。

（3）set\_cpu\_boot\_mode\_flag和el2\_setup这两个函数会在各个cpu上执行。

（4）变量\_\_boot\_cpu\_mode定义如下：

ENTRY(\_\_boot\_cpu\_mode)  
    .long    BOOT\_CPU\_MODE\_EL2－－－－－－－－A  
    .long    BOOT\_CPU\_MODE\_EL1－－－－－－－－B

如果cpu启动的时候是EL1 mode，会修改变量\_\_boot\_cpu\_mode A域，将其修改为BOOT\_CPU\_MODE\_EL1。如果cpu启动的时候是EL2 mode，会修改变量\_\_boot\_cpu\_mode B域，将其修改为BOOT\_CPU\_MODE\_EL2。

4、\_\_vet\_fdt

在进入具体函数之前，x21和x24都被设定成了指定的值。x21被设定为fdt在RAM中的物理地址（参考preserve\_boot\_args函数），x24被设定为\_\_PHYS\_OFFSET，定义为：

#define \_\_PHYS\_OFFSET    (KERNEL\_START - TEXT\_OFFSET)

#define KERNEL\_START    \_text

KERNEL\_START是kernel开始运行的虚拟地址，更确切的说是内核正文段开始的虚拟地址。 在链接脚本文件中（参考arch/arm64/kernel下的vmlinux.lds.S），KERNEL\_START被设定为：

. = PAGE\_OFFSET + TEXT\_OFFSET;

.head.text : {   
    \_text = .;   
    HEAD\_TEXT   
}

因此，KERNEL\_START的值和PAGE\_OFFSET以及TEXT\_OFFSET这两个offset的设定有关。TEXT\_OFFSET标识了内核正文段的offset，其实如果该宏被定义为KERNEL\_TEXT\_OFFSET会更好理解。我们知道，操作系统运行在内核空间，应用程序运行在用户空间，假设内核空间的首地址是x（一般也是RAM的首地址），那么是否让kernel运行在x地址呢？对于arm，在内核空间的开始有32kB（0x00008000）的空间用于保存内核的页表（也就是进程0的PGD）以及bootload和kernel之间参数的传递，对于ARM64，在其Makefile中定义了这个offset是512KB（0x00080000）。

ifeq ($(CONFIG\_ARM64\_RANDOMIZE\_TEXT\_OFFSET), y)   
TEXT\_OFFSET := $(shell awk 'BEGIN {srand(); printf "0x%03x000\n", int(512 \* rand())}')   
else   
TEXT\_OFFSET := 0x00080000   
endif

kernel image的开始部分包括了一个ARM64 image header的内容，这个header定义了bootloader如何来copy kernel image。ARM64 image header中有一个域（text\_offset）就是告知bootloader，它应该按照多大的偏移来copy kernel image。当然了，也许有些bootloader不鸟这些，对于ARM64平台，反正大家一直都是固定为0x80000，因此，bootloader也没有什么太大的动力来修改支持这个特性。怎么破？虽然目前ARM64的kernel的TEXT\_OFFSET就是固定为0x80000，但是也许将来内核会修改这个offset啊。在这种情况下，内核的开发者提供了一个CONFIG\_ARM64\_RANDOMIZE\_TEXT\_OFFSET选项，在编译内核的时候可以randomize内核的TEXT\_OFFSET值，以此来测试bootloader是否能够正确的copy kernel image到正确的内存偏移位置上去。通过这样一个配置项，可以尽快的暴露问题，确保了整个系统（bootloader ＋ kernel）稳定的运行。

搞定了TEXT\_OFFSET，我们再来看看PAGE\_OFFSET，在arch/arm64/include/asm/memory.h中，PAGE\_OFFSET被定义为：

#define VA\_BITS            (CONFIG\_ARM64\_VA\_BITS)   
#define PAGE\_OFFSET        (UL(0xffffffffffffffff) << (VA\_BITS - 1))

VA\_BITS定义了虚拟地址空间的bit数（该值也就是定义了用户态程序或者内核能够访问的虚拟地址空间的size），假设VA\_BITS被设定为39个bit，那么PAGE\_OFFSET就是0xffffffc0-00000000。PAGE\_OFFSET的名字也不好（个人观点，可能有误），OFFSET表明的是一个偏移，内核空间被划分成一个个的page，PAGE\_OFFSET看起来应该是定义以page为单位的偏移。但是，以什么为基准的偏移呢？PAGE\_OFFSET的名字中没有给出，当然实际上，这个符号是定义以整个address space的起始地址（也就是0）为基准。另外，虽然这个地址是要求page对齐，但是实际上，这个符号仍然定义的是虚拟地址的offset（而不是page的offset）。根据上面的理由，我觉得定义成KERNEL\_IMG\_OFFSET会更好理解一些。一句话总结：PAGE\_OFFSET定义了将kernel image安放在虚拟地址空间的哪个位置上。

OK，经过漫长的说明之后，\_\_PHYS\_OFFSET实际上就是kernel image的首地址（并不是\_\_PHYS\_OFFSET的位置开始就是真实的kernel image，实际上从\_\_PHYS\_OFFSET开始，首先是TEXT\_OFFSET的保留区域，然后才是真正的kernel image）。实际上，\_\_PHYS\_OFFSET定义的是一个虚拟地址而不是物理地址，这里的PHYS严重影响了该符号的含义，实际上adrp这条指令可以将一个虚拟地址转换成物理地址（在没有打开MMU的时候）。而函数\_\_vet\_fdt主要是对这个bootloader传递给kernel的fdt参数进行验证，看是否OK，主要验证的内容包括：

（1）是否是8字节对齐的

（2）是否在kernel space的前512M内

\_\_vet\_fdt:   
    tst    x21, #0x7－－－－是否是8字节对齐的   
    b.ne    1f   
    cmp    x21, x24－－－－－是否在小于kernel space的首地址   
    b.lt    1f   
    mov    x0, #(1 << 29)   
    add    x0, x0, x24   
    cmp    x21, x0   
    b.ge    1f－－－－－－－是否大于kernel space的首地址＋512M   
    ret   
1:   
    mov    x21, #0－－－－－－－－－－传递的fdt地址有误，清零   
    ret   
ENDPROC(\_\_vet\_fdt)

四、参考文献

1、Documentation/arm64/booting.txt

2、ARM Architecture Reference Manual

# 创建启动阶段的页表

ARM64的启动过程之（二）：创建启动阶段的页表

## 一、前言

本文主要描述了ARM64启动过程中，如何建立初始化阶段页表的过程。我们知道，从bootloader到kernel的时候，MMU是off的（顺带的负作用是无法打开data cache），为了提高性能，加快初始化速度，我们必须某个阶段（越早越好）打开MMU和cache，而在此之前，我们必须要设定好页表。

在初始化阶段，我们mapping三段地址，一段是identity mapping，其实就是把物理地址mapping到物理地址上去，在打开MMU的时候需要这样的mapping（ARM ARCH强烈推荐这么做的）。第二段是kernel image mapping，内核代码欢快的执行当然需要将kernel running需要的地址（kernel txt、dernel rodata、data、bss等等）进行映射了，第三段是blob memory对应的mapping。

在本文中，我们会混用下面的概念：page table和translation table、PGD和Level 0 translation table、PUD和Level 1 translation table、PMD和Level 2 translation table、Page Table和Level 3 translation table。最后，还是说明一下，本文来自4.1.10内核（部分来自4.4.6），有兴趣的读者可以下载来对照阅读本文。

## 二、基础知识

为了更好的理解\_\_create\_page\_tables的代码，我们需要准备一些基础知识。由于ARM64太复杂了，各种exception level、各种stage translation、各种地址宽度配置等等让虚拟地址到物理地址的映射变得非常复杂，因此，本文focus在一种配置：Non-secure EL1和EL0、stage 1 translation、VA和PA的地址宽度都是48个bit。

**1、虚拟地址空间的size是多少？**

在32-bit的ARM时代，这个问题问的有点白痴，大家都是耳熟能详的一句话就是每一个进程都有4G的独立的虚拟地址空间（0x0～0xffffffff）。对于ARM64而言，进程需要完全使用2^64那么多的虚拟地址空间吗？如果需要，那么CPU的MMU单元需要能接受来自处理器发出的64根地址线的输入信号，并对其进行翻译，这也就意味着MMU需要更多的晶体管来支持64根地址线的输入，而CPU也需要驱动更多的地址线，但是实际上，在短期内，没有看出有2^64那么多的虚拟地址空间的需求，因此，ARMv8实际上提供了TCR\_ELx （Translation Control Register (ELx)可以对MMU的输入地址（也就是虚拟地址）进行配置。为了不把问题复杂化，我们先不考虑TCR\_EL2和TCR\_EL3这两个寄存器。通过TCR\_EL1寄存器中的TxSZ域可以控制虚拟地址空间的size。对于ARM64（是指处于AArch64状态的处理器）而言，最大的虚拟地址的宽度是48 bit，因此虚拟地址空间的范围是0x0000\_0000\_0000\_0000 ～ 0x0000\_FFFF\_FFFF\_FFFF，总共256TB。 当然，具体实现的时候可以选择如下的地址线数目：

config ARM64\_VA\_BITS   
    int   
    default 36 if ARM64\_VA\_BITS\_36   
    default 39 if ARM64\_VA\_BITS\_39   
    default 42 if ARM64\_VA\_BITS\_42   
    default 47 if ARM64\_VA\_BITS\_47   
    default 48 if ARM64\_VA\_BITS\_48

在代码中，有一个宏定义如下：

#define VA\_BITS            (CONFIG\_ARM64\_VA\_BITS)

这个宏定义了虚拟地址空间的size。

**2、物理地址空间的size是多少？**

问过虚拟地址空间的size是多少这个问题之后，很自然的会考虑物理地址空间。基本概念和上一节类似，符合ARMv8的PE最大支持的物理地址宽度也是48个bit，当然，具体的实现可以自己定义（不能超过48个bit），具体的配置可以通过ID\_AA64MMFR0\_EL1 （AArch64 Memory Model Feature Register 0）这个RO寄存器获取。

**3、和地址映射相关的宏定义**

|  |  |
| --- | --- |
| 宏定义符号 | 描述 |
| VA\_START | 内核地址空间的起始地址 |
| TEXT\_OFFSET | bootloader会把kernel image从外设copy到RAM中，那么具体copy到什么位置呢？从RAM的起始地址开始吗？实际上是从TEXT\_OFFSET开始的，偏移这么一小段内存估计是为了bootloader和kernel之间传递一些信息。所以，这里TEXT是指kernel text segment，而OFFSET是相对于RAM的首地址而言的。  TEXT\_OFFSET必须要4K对齐并且TEXT\_OFFSET的size不能大于2M。 |
| PAGE\_OFFSET | kernel image的起始虚拟地址，一般而言也就是系统中RAM的首地址，在该地址TEXT\_OFFSET之后保存了kernel image。  PAGE\_OFFSET必须要2M对齐 |
| TASK\_SIZE | 一般而言，用户地址空间从0开始，大小就是TASK\_SIZE，因此，这个宏定义的全称应该是task userspace size。对于ARM64的用户空间进程而言，有两种，一种是运行在AArch64状态下，另外一种是运行在AArch32状态，因此，实际上代码中又定义了TASK\_SIZE\_32和TASK\_SIZE\_64两个宏定义。 |
| PHYS\_OFFSET | 系统内存的起始物理地址。在系统初始化的过程中，会把PHYS\_OFFSET开始的物理内存映射到PAGE\_OFFSET的虚拟内存上去。 |

**4、虚拟地址空间到物理地址空间的映射**

MMU主要负责从VA（virutal address）到PA（Physical address）的翻译、memory的访问控制以及memory attribute的控制，这里我们暂时只关注地址翻译功能。不同的exception level和security state有自己独立的地址翻译过程，当然我们这里暂时只关注Non-secure EL1和EL0，在这种状态下，地址翻译可以分成两个stage，不过两个stage是为虚拟化考虑的，因此，为了简化问题，我们先只考虑一个stage。OK，做了这么多的简化之后，我们可以来看看地址翻译过程了（也就是Non-secure EL1和EL0 stage 1情况下的地址翻译过程）。

一个很有意思的改变（针对ARM32而言）是虚拟地址空间被分成了两个VA subrange：

Lower VA subrange ： 从0x0000\_0000\_0000\_0000 到 (2^(64-T0SZ) - 1)   
Upper VA subrange ： 从(2^64 - 2^(64-T1SZ)) 到 0xFFFF\_FFFF\_FFFF\_FFFF

为什么呢？熟悉ARM平台的工程师都形成了固定的印象，当进程切换地址空间的时候，实际上切换了内核地址空间+用户地址空间（total 4G地址空间），而实际上，每次进程切换的时候，内核地址空间都是不变的，实际变化的只有userspace而已。如果硬件支持了VA subrange，那么我们可以这样使用：

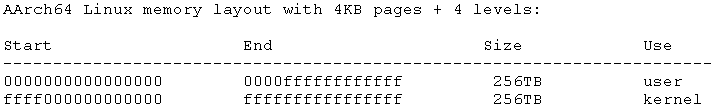
Lower VA subrange ： process-specific地址空间   
Upper VA subrange ： kernel地址空间

这样，当进程切换的时候，我们不必切换kernel space，只要切换userspace就OK了。

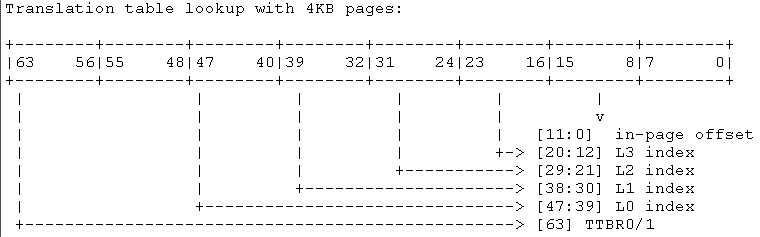
地址映射的粒度怎么配置呢？地址映射的粒度用通俗的语言讲就是page size（也可能是block size），传统的page size都是4K，ARM64的MMU支持4K、16K和64K的page size。除了地址映射的粒度还有一个地址映射的level的概念，在ARM32的时代，2 level或者3 level是比较常见的配置，对于ARM64，这和page size、物理地址和虚拟地址的宽度都是有关系的，具体请参考ARM ARM文档。

**5、AArch64 Linux中虚拟地址空间的布局**

把事情搞的太复杂了往往迷失了重点，我们这里再做一个简化就是固定page size是4K，并且VA宽度是48个bit，在这种情况下，虚拟地址空间的布局如下：

[](http://www.wowotech.net/content/uploadfile/201510/87d98e9346ad7283fe5e3aabc2a6ea1520151013101751.gif)

具体的映射过程如下：

[](http://www.wowotech.net/content/uploadfile/201510/9894aaea293dc8492f2f0744360a819f20151013101751.gif)

整个地址翻译的过程是这样的：首先通过虚拟地址的高位可以知道是属于userspace还是kernel spce，从而分别选择TTBR0\_EL1（Translation Table Base Register 0 (EL1)）或者TTBR1\_EL1（Translation Table Base Register 1 (EL1)）。这个寄存器中保存了PGD的基地址，该地址指向了一个lookup table，每一个entry都是描述符，可能是Table descriptor、block descriptor或者是page descriptor。如果命中了一个block descriptor，那么地址翻译过程就结束了，当然对于4-level的地址翻译过程，PGD中当然保存的是Table descriptor，从而指向了下一节的Translation table，在kernel中称之PUD。随后的地址翻译概念类似，是一个PMD过程，最后一个level是PTE，也就是传说中的page table entry了，到了最后的地址翻译阶段。这时候PTE中都是一个个的page descriptor，完成最后的地址翻译过程。

## 三、代码分析

本文涉及的代码就是\_\_create\_page\_tables这个函数。

1、initial translation tables的位置。

initial translation tables定义在链接脚本文件中（参考arch/arm64/kernel下的vmlinux.lds.S），如下：

. = ALIGN(PAGE\_SIZE);   
idmap\_pg\_dir = .;   
. += IDMAP\_DIR\_SIZE;   
swapper\_pg\_dir = .;   
. += SWAPPER\_DIR\_SIZE;

ARM32的的时候，kernel image在RAM开始的位置让出了32KB的memory保存了bootloader到kernel传递的tag参数以及内核空间的页表。在刚开始的时候，ARM64沿用了ARM32的做法，将这些初始页表放到了PHYS\_OFFSET和PHYS\_OFFSET+TEXT\_OFFSET之间（size是0x80000）。但是，其实这段内存是有可能被bootloader使用的，而且，这个时候，memory block模块（确定内核需要管理的memory block）没有ready，想要标记reservation memory也是不可能的。在这种情况下，假设bootloader在这段memory放了些数据，试图传递给kernel，但是kernel如果在这段memory上建立页表，那么就把有用数据给覆盖了。最后，initial translation tables被放到了kernel image的后面，位于bss段之后，从而解决了这个问题。

解决了位置问题之后，我们来看一看size，代码如下：

#if ARM64\_SWAPPER\_USES\_SECTION\_MAPS  
#define SWAPPER\_PGTABLE\_LEVELS    (CONFIG\_PGTABLE\_LEVELS - 1)  
#define IDMAP\_PGTABLE\_LEVELS    (ARM64\_HW\_PGTABLE\_LEVELS(PHYS\_MASK\_SHIFT) - 1)  
#else  
#define SWAPPER\_PGTABLE\_LEVELS    (CONFIG\_PGTABLE\_LEVELS)  
#define IDMAP\_PGTABLE\_LEVELS    (ARM64\_HW\_PGTABLE\_LEVELS(PHYS\_MASK\_SHIFT))  
#endif  
  
#define SWAPPER\_DIR\_SIZE    (SWAPPER\_PGTABLE\_LEVELS \* PAGE\_SIZE)  
#define IDMAP\_DIR\_SIZE        (IDMAP\_PGTABLE\_LEVELS \* PAGE\_SIZE)

ARM64\_SWAPPER\_USES\_SECTION\_MAPS这个宏定义是说明了swapper/idmap的映射是否使用section map。什么是section map呢？我们用一个实际的例子来描述。假设VA是48 bit，page size是4K，那么，在地址映射过程中，地址被分成9（level 0） ＋ 9（level 1） ＋ 9（level 2） ＋ 9（level 3） ＋ 12（page offset），对于kernel image这样的big block memory region，使用4K的page来mapping有点得不偿失，在这种情况下，可以考虑让level 2的Translation table entry指向一个2M 的memory region，而不是下一级的Translation table。所谓的section map就是指使用2M的为单位进行映射。当然，不是什么情况都是可以使用section map，对于kernel image，其起始地址是2M对齐的，因此block size是2M的情况下才OK，对于PAGE SIZE是16K，其Block descriptor指向了一个32M的内存块，PAGE SIZE是64K的时候，Block descriptor指向了一个512M的内存块，因此，只有4K page size的情况下，才可以启用section map。

OK，我们回到具体的初始阶段页表大小这个问题上。原来ARM32的时候，一个page就OK了，对于ARM64，由于虚拟地址空间变大了，因此我们需要更多的page来完成启动阶段的initial translation tables的构建。我们仍然用VA是48 bit，page size是4K为例子进行说明。根据前面的描述，我们知道，内核空间的地址大小是256T，48 bit的**地址被分成9 ＋ 9 ＋ 9 ＋ 9 ＋ 12，**因此**PGD（Level 0）、PUD（Level 1）、PMD（Level 2）、PT（Level 3）的translation table中的entry都是512项**，每个描述符是8个byte，因此这些translation table都是4KB，恰好是一个page size。根据链接脚本中的定义，idmap和swapper page tables （或者叫做translation table）分别保留了3个page的页面。3个page分别是3个level的translation table。等等，读者可能会问：上面不是说48 bit VA加上4K page size需要4阶translation table吗？这里怎么只有3个level？实际上，3级映射是PGD/PUM/PMD（每个table占据一个page），只不过PMD的内容不是下一级的table descriptor，而是基于2M block的mapping（或者说PMD中的描述符是block descriptor）。

2、创建页表前的准备

  代码如下：

\_\_create\_page\_tables:   
    adrp    x25, idmap\_pg\_dir －－－－－－获取idmap的页表基地址（物理地址）   
    adrp    x26, swapper\_pg\_dir －－－－－获取kernel space的页表基地址（物理地址）   
    mov    x27, lr －－－－－－保存lr

    mov    x0, x25 －－－－－－－－－－准备要invalid cache的地址段的首地址   
    add    x1, x26, #SWAPPER\_DIR\_SIZE －－－－－－－准备要invalid cache的地址段的尾地址   
    bl    \_\_inval\_cache\_range －－－－将idmap和swapper页表地址段对应的cacheline设定为无效

    mov    x0, x25 －－－－－－－这一段代码是将idmap和swapper页表内容设定为0   
    add    x6, x26, #SWAPPER\_DIR\_SIZE －－－－x0是开始地址，x6是结束地址   
1:    stp    xzr, xzr, [x0], #16   
    stp    xzr, xzr, [x0], #16   
    stp    xzr, xzr, [x0], #16   
    stp    xzr, xzr, [x0], #16   
    cmp    x0, x6   
    b.lo    1b

这段代码没有什么特别要说明的，除了adrp这条指令。adrp是计算指定的符号地址到run time PC值的相对偏移（不过，这个offset没有那么精确，是以4K为单位，或者说，低12个bit是0）。在指令编码的时候，立即数（也就是offset）占据21个bit，此外，由于偏移计算是按照4K进行的，因此最后计算出来的符号地址必须要在该指令的－4G和4G之间。由于执行该指令的时候，还没有打开MMU，因此通过adrp获取的都是物理地址，当然该物理地址的低12个bit是全零的。此外，由于在链接脚本中idmap\_pg\_dir和swapper\_pg\_dir是page size aligned，因此使用adrp指令也是OK的。

为什么要调用\_\_inval\_cache\_range来invalidate idmap\_pg\_dir和swapper\_pg\_dir对应页表空间的cache呢？根据boot protocol，代码执行到此，对于cache的要求是kernel image对应的那段空间的cache line是clean到PoC的，不过idmap\_pg\_dir和swapper\_pg\_dir对应页表空间不属于kernel image的一部分，因此其对应的cacheline很可能有一些旧的，无效的数据，必须要清理掉。

顺便再提一句，将idmap和swapper页表内容设定为0是有意义的。实际上这些translation table中的大部分entry都是没有使用的，PGD和PUD都是只有一个entry是有用的，而PMD中有效的entry数目是和mapping的地址size有关。将页表内容清零也就是意味着将页表中所有的描述符设定为invalid（描述符的bit 0指示是否有效，等于0表示无效描述符）。

**3、创建identity mapping**

identity mapping实际上就是建立了整个内核（从KERNEL\_START到KERNEL\_END）的一致性mapping，就是将物理地址所在的虚拟地址段mapping到物理地址上去。为什么这么做呢？ARM ARM文档中有一段话：

If the PA of the software that enables or disables a particular stage of address translation differs from its VA, speculative instruction fetching can cause complications. ARM strongly recommends that the PA and VA of any software that enables or disables a stage of address translation are identical if that stage of translation controls translations that apply to the software currently being executed.

由于打开MMU操作的时候，内核代码欢快的执行，这时候有一个地址映射ON/OFF的切换过程，这种一致性映射可以保证在在打开MMU那一点附近的程序代码可以平滑切换。具体的操作分成两个阶段，第一个阶段是通过create\_pgd\_entry建立中间level（也就是PGD和PUD）的描述符，第二个阶段是创建PMD的描述符，由于PMD的描述符是block descriptor，因此，完成PMD的设定后就完成了整个identity mapping页表的设定。具体代码如下：

    ldr    x7, =MM\_MMUFLAGS    
    mov    x0, x25－－－－－－－－－x0保存了idmap\_pg\_dir变量的物理地址   
    adrp    x3, KERNEL\_START－－－x3保存了内核image的物理地址

    create\_pgd\_entry x0, x3, x5, x6   
    mov    x5, x3                // \_\_pa(KERNEL\_START)   
    adr\_l    x6, KERNEL\_END            // \_\_pa(KERNEL\_END)   
    create\_block\_map x0, x7, x3, x5, x6

create\_pgd\_entry用来在PGD（level 0 translation table）中创建一个描述符，如果需要下一级的translation table，也需要同时建立，最终的要求是能够完成所有中间level的translation table的建立（其实每个table中都是只建立了一个描述符），对于identity mapping，这里需要PGD和PUD就OK了。该函数需要四个参数：x0是pgd的地址，具体要创建哪一个地址的描述符由x3指定，x5和x6是临时变量，create\_pgd\_entry具体代码如下：

    .macro    create\_pgd\_entry, tbl, virt, tmp1, tmp2   
    create\_table\_entry \tbl, \virt, PGDIR\_SHIFT, PTRS\_PER\_PGD, \tmp1, \tmp2   
    create\_table\_entry \tbl, \virt, TABLE\_SHIFT, PTRS\_PER\_PTE, \tmp1, \tmp2   
    .endm

create\_table\_entry这个宏定义主要是用来创建一个translation table的描述符，具体创建哪一个level的Translation table descriptor是由tbl参数指定的。怎么来创建描述符呢？如果是table descriptor，那么该描述符需要指向下一级页表基地址，当然，create\_table\_entry参数并没有给出，是在程序中hardcode实现：L（n）的translation table中的描述符指向的L（n＋1） Translation table位于L（n）translation table所在page的下一个page（太拗口了，但是我也懒得画图了）。shift和ptrs这两个参数用来计算页表内的index，具体算法可以参考下面的代码：

.macro    create\_table\_entry, tbl, virt, shift, ptrs, tmp1, tmp2   
lsr    \tmp1, \virt, #\shift－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
and    \tmp1, \tmp1, #\ptrs - 1 －－－－－－－－－－－－－－－－－－－－－－－－－（2）   
add    \tmp2, \tbl, #PAGE\_SIZE－－－－－－－－－－－－－－－－－－－－－－－－（3）   
orr    \tmp2, \tmp2, #PMD\_TYPE\_TABLE－－－－－－－－－－－－－－－－－－－－（4）   
str    \tmp2, [\tbl, \tmp1, lsl #3]－－－－－－－－－－－－－－－－－－－－－－－－（5）   
add    \tbl, \tbl, #PAGE\_SIZE －－－－－－－－－－－－－－－－－－－－－－－－－（6）   
.endm

（1）如果是PGD，那么shift等于PGDIR\_SHIFT，也就是39了。根据第二章的描述，我们知道L0 index（PGD index）使用虚拟地址的bit[47:39]。如果是PUD，那么shift等于PUD\_SHIFT，也就是30了（注意：L1 index（PUD index）使用虚拟地址的bit[38:30]）。要想找到virt这个地址（实际传入的是物理地址，当然，我们本来就是要建立和物理地址一样的虚拟地址的mapping）在translation table中的index，当然需要右移shift个bit了。

（2）除了右移操作，我们还需要mask操作（ptrs - 1实际上就是掩码）。对于PGD，其index占据9个bit，因此mask是0x1ff。同样的，对于PUD，其index占据9个bit，因此mask是0x1ff。至此，tmp1就是virt地址在translation table中对应的index了。

（3）如果是table描述符，需要指向另外一个level的translation table，在哪里呢？答案就是next page，读者可以自行回忆链接脚本中的3个连续的idmap\_pg\_dir的page定义。

（4）光有下一级translation table的地址不行，还要告知该描述符是否有效（set bit 0），该描述符的类型是哪一种类型（set bit 1表示是table descriptor），至此，描述符内容准备完毕，保存在tmp2中

（5）最关键的一步，将描述符写入页表中。之所以有“lsl #3”操作，是因为一个描述符占据8个Byte。

（6）将translation table的地址移到next level，以便进行下一步设定。

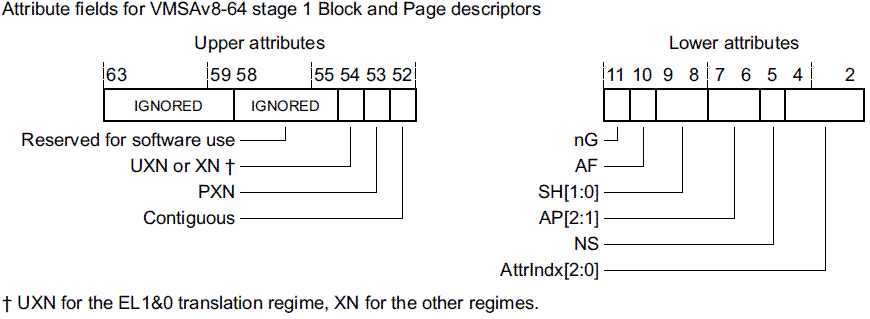
如果你足够细心，一定不会忽略这样的一个细节。获取KERNEL\_START和KERNEL\_END的代码是不一样的，对于KERNEL\_START直接使用了adrp    x3, KERNEL\_START，而对于KERNEL\_END使用了adr\_l    x6, KERNEL\_END。具体使用哪一个是和该地址是否4K对齐相关的。KERNEL\_START一定是4K对齐的，而KERNEL\_END就不一定了，虽然在4.1.10中KERNEL\_END也是4K对齐的，不过没有任何协议保证这一点，为了保险起见，代码使用了adr\_l，确保获取正确的KERNEL\_END的物理地址。

回到create\_pgd\_entry函数中，这个函数填充了内核image首地址对应的1G memory range所需要的Translation table描述符，听起来很吓人，不过就是两个描述符，一个是在PGD中，另外一个是在PUD中。虽然只有两个描述符，可以可以支持1G虚拟地址的mapping了。当然具体mapping多少（PMD中有多少entry），还是要看kernel image的size了。

OK，来到PMD部分的设定了，我们看看代码：

    .macro    create\_block\_map, tbl, flags, phys, start, end   
    lsr    \phys, \phys, #BLOCK\_SHIFT   
    lsr    \start, \start, #BLOCK\_SHIFT   
    and    \start, \start, #PTRS\_PER\_PTE - 1    // table index   
    orr    \phys, \flags, \phys, lsl #BLOCK\_SHIFT    // table entry   
    lsr    \end, \end, #BLOCK\_SHIFT   
    and    \end, \end, #PTRS\_PER\_PTE - 1        // table end index   
9999:    str    \phys, [\tbl, \start, lsl #3]        // store the entry   
    add    \start, \start, #1            // next entry   
    add    \phys, \phys, #BLOCK\_SIZE        // next block   
    cmp    \start, \end   
    b.ls    9999b   
    .endm

create\_block\_map的名字起得不错，该函数就是在tbl指定的Translation table中建立block descriptor以便完成address mapping。具体mapping的内容是将start 到 end这一段VA mapping到phys开始的PA上去。其实这里的代码逻辑和上面类似，我们这里就不详述，需要提及的是PTE已经进入了最后一个level的mapping，因此描述符中除了地址信息之外（占据bit[47:21]，还需要memory attribute和memory accesse的信息。对于这个场景，PMD中是block descriptor，因此描述符中还包括了block attribute域，分成upper block attribute[63:52]和lower block attribute[11:2]。对这些域的定义如下：

[](http://www.wowotech.net/content/uploadfile/201510/08cc60334ad8425f4eb5ee92ed9a2c1320151013101752.gif)

在代码中，block attribute是通过flags参数传递的，MM\_MMUFLAGS定义如下：

#define MM\_MMUFLAGS    PMD\_ATTRINDX(MT\_NORMAL) | PMD\_FLAGS

#define PMD\_FLAGS    PMD\_TYPE\_SECT | PMD\_SECT\_AF | PMD\_SECT\_S

MT\_NORMAL表示该段内存的memory type是普通memory（对应AttrIndx[2:0]），而不是device什么的。PMD\_TYPE\_SECT 说明该描述符是一个有效的（bit 0等于1）的block descriptor（bit 1等于0）。PMD\_SECT\_AF中的AF是access flag的意思，表示该memory block（或者page）是否被最近被访问过。当然，这需要软件的协助。如果该bit被设置为0，当程序第一次访问的时候会产生异常，软件需要将给bit设置为1，之后再访问该page的时候，就不会产生异常了。不过当软件认为该page已经old enough的时候，也可以clear这个bit，表示最近都没有访问该page。这个flag是硬件对page reclaim算法的支持，找到最近不常访问的那些page。当然在这个场景下，我们没有必要enable这个特性，因此将其设定为1。PMD\_SECT\_S对应SH[1:0]，描述memory的sharebility。这些内容和memory attribute相关，我们会在后续的文档中描述，这里就不偏离主题了。

广大人民群众最关心的当然也是最熟悉的是memory access control，这是通过AP[2:1]域来控制的。这里该域被设定为00b，表示EL1状态下是RW，EL0状态不可访问。UXN和PXN是用来控制可执行权限的，这里UXN和PXN都是0，表示EL1和EL0状态下都是excutable的。

4、创建kernel space mapping

要创建kernel space的页表了，遇到的第一个问题就是：mapping多少呢？kernel space辣么大，256T，不可能全部都mapping。OK，答案就是创建两部分的页表，一个从kernel image的开始地址（包括开始的那一段TEXT\_OFFSET的保留区域）到kernel image的结束地址（内核的正常运行需要这段mapping），这一段覆盖了内核的正文段、各种data段、bss段、各种奇奇怪怪段等。还有一个就是bootloader传递过来的blob memory对应的页表。我们先看第一段kernel image的mapping：

mov    x0, x26－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
mov    x5, #PAGE\_OFFSET－－－－－－－－－－－－－－－－－－－（2）   
create\_pgd\_entry x0, x5, x3, x6－－－－－－－－－－－－－－－－－（3）   
ldr    x6, =KERNEL\_END－－－－－－end address   
mov    x3, x24                // phys offset   
create\_block\_map x0, x7, x3, x5, x6－－－－－－－－－－－－－－－（4）

（1）swapper\_pg\_dir其实就是swapper进程（pid等于0的那个，其实就是idle进程）的地址空间，这时候，x0指向了内核地址空间的PGD的基地址。

（2）PAGE\_OFFSET是kernel image的首地址，对于48bit的VA而言，该地址是0xffff8000-00000000。

（3）创建PAGE\_OFFSET（即kernel image首地址）对应的PGD和PUD中的描述符。

（4）创建PMD中的描述符。x24保存了\_\_PHYS\_OFFSET，实际上也就是kernel image的首地址（物理地址）。

完成了kernel image的mapping，我们来看看blob mapping的建立。由于ARM64 boot protocol要求blob必须在内核空间开始的512MB内（同时要求8字节对齐，dtb image不能越过2M section size的边界），因此实际上PGD和PUD都不需要建立了，只要建立PMD的描述符就OK了。对应的PMD描述符的建立代码如下：

mov    x3, x21－－－－－－－－－－－－－－FDT phys address   
and    x3, x3, #~((1 << 21) - 1) －－－－－－2MB aligned   
mov    x6, #PAGE\_OFFSET－－－－－－－kernel space start virtual address   
sub    x5, x3, x24－－－－－－－－－－－－subtract kernel space start physical address

tst    x5, #~((1 << 29) - 1) －－－－－－－－within 512MB?   
csel    x21, xzr, x21, ne －－－－－－－－－bad blob parameter and zero the FDT pointer   
b.ne    1f   
add    x5, x5, x6 －－－－－－－－－－－－x5 equal blob virtual address   
add    x6, x5, #1 << 21 －－－－－－－－－mapping 2M size   
sub    x6, x6, #1    
create\_block\_map x0, x7, x3, x5, x6－－－create blob block descriptor in PMD

5、收尾

    mov    x0, x25－－－－－－－再次invalid上文中建立page table memory对应的cache   
    add    x1, x26, #SWAPPER\_DIR\_SIZE   
    dmb    sy   
    bl    \_\_inval\_cache\_range

    mov    lr, x27－－－－－－恢复lr   
    ret－－－－－－－－－－－返回   
ENDPROC(\_\_create\_page\_tables)

由于页表中写了新的内容，而且是在没有打开cache的情况下写的，这时候，cache line的数据有可能被speculatively load，因此再次invalid是一个比较保险的做法。

四、参考文献

1、Documentation/arm64/memory.txt

2、ARM Architecture Reference Manual

change log：

1、2015-12-1，修正对PAGE\_OFFSET的描述。

2、2016-7-12，（1）增加了和地址映射相关的几个宏定义的描述。（2）增加建立identity mapping的原因

3、2016-7-15，对initial translation tables的位置和size进行补充描述。

4、2016-9-9，修改DTB的限制。

# 为打开MMU而进行的CPU初始化

ARM64的启动过程之（三）：为打开MMU而进行的CPU初始化

## 一、前言

上一节主要描述了为了打开MMU而进行的Translation table的建立，本文延续之前的话题，主要是进行CPU的初始化（注：该初始化仅仅为是为了turn on MMU）。

本文主要分析ARM64初始化过程中的\_\_cpu\_setup函数，代码位于arch/arm64/mm/proc.S中。主要的内容包括：

1、cache和TLB的处理

2、Memory attributes lookup table的构建

3、SCTLR\_EL1、TCR\_EL1的设定

## 二、cache和TLB的处理

**1、oerview**

根据ARM64 boot protocol，我们知道，会bootloader将内核解压并copy到RAM中，同时将CPU core（BSP）的状态设定为：关闭MMU，disable D-cache，I-cache状态可以是enable，也可以是disable的。其实在bootloader将控制权交给Kernel之前，bootloader已经走过千山万水，为了性能，很可能是打开了MMU以及各种cache，只是在进入kernel的时候，受限于ARM64 boot protocol而将CPU以及cache、MMU等硬件状态设定为指定的状态。因此，实际上这时候，instruction cache以及TLB中很可能有残留的数据，因此需要将其清除。

**2、如何清除instruction cache的数据？**

听起来这个问题似乎有点愚蠢，实际上不是。随着人类不断向更快的计算机系统进发，memory hierarchy也变得异常复杂起来，cache也形成了cache hierarchy（ARMv8最大支持7个level，L1～L7），不同级别的cache中都包含了部分下一级cache（或者main memory）的内容。这时候，维护数据一致性变得复杂了，例如：当要操作（例如clean或者invalidate）某个地址对应的cacheline的时候，是仅仅操作L1还是覆盖L1和L2，异或将L1～L3中对应的cacheline都设置为无效呢？PoU（Point of Unification）和PoC（Point of Coherency）这两个术语就是用来定义cache操作范围的，它们其实都是用来描述计算机系统中memory hierarchy的一个具体的“点”，操作范围是从PE到该点的所有的memory level。

我们先看PoU，PoU是以一个特定的PE（该PE执行了cache相关的指令）为视角。PE需要透过各级cache（涉及instruction cache、data cache和translation table walk）来访问main memory，这些操作在memory hierarchy的某个点上（或者说某个level上）会访问同一个copy，那么这个点就是该PE的Point of Unification。假设一个4核cpu，每个core都有自己的L1 instruction cache和L1 Data cache，所有的core共享L2 cache。在这样的一个系统中，PoU就是L2 cache，只有在该点上，特定PE的instruction cache、data cache和translation table walk硬件单元访问memory的时候看到的是同一个copy。

PoC可以认为是Point of System，它和PoU的概念类似，只不过PoC是以系统中所有的agent（bus master，又叫做observer，包括CPU、DMA engine等）为视角，这些agents在进行memory access的时候看到的是同一个copy的那个“点”。例如上一段文章中的4核cpu例子，如果系统中还有一个DMA controller和main memory（DRAM）通过bus连接起来，在这样的一个系统中，PoC就是main memory这个level，因为DMA controller不通过cache访问memory，因此看到同一个copy的位置只能是main memory了。

之所以区分PoC和PoU，根本原因是为了更好的利用cache中的数据，提高性能。OK，我们回到本节开始的问题：如何清除instruction cache的数据？我们还是用一个具体的例子来描述好了：对于一个PoU是L2 cache的系统，清除操作应该到哪一个level？根据ARM64 boot protocol规定，kernel image对应的VA会被cleaned to PoC，这时候，各级的data cache的数据都是一致性的，按理说，BSP只需要清除本cpu core上的instruction cache就OK了。不过代码使用了PoU，也就是说操作到了L2，而实际上，L2是unified cache，其数据是有效的，清除了会影响性能，这里我也想的不是很清楚，先存疑吧。

**3、代码解析**

ENTRY(\_\_cpu\_setup)   
    ic    iallu －－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    tlbi    vmalle1is－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（2）   
    dsb    ish －－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（3）

    mov    x0, #3 << 20 －－－－－－－－－－－－－－－－－－－－－－－－－－－－（4）   
    msr    cpacr\_el1, x0            // Enable FP/ASIMD   
    msr    mdscr\_el1, xzr            // Reset mdscr\_el1

（1）ic iallu指令设置instruction cache中的所有的cache line是无效的，直到PoU。同时设置为无效状态的还包括BTB（Branch Target Buffer） cache。在处理器设计中，分支指令对性能的影响非常巨大（打破了pipeline，影响了并行处理），因此在处理器中会设定一个Branch target predictor单元用来对分支指令进行预测。Branch target predictor凭什么进行预测呢？所谓预测当然是根据过去推测现在，因此，硬件会记录分支指令指令的跳转信息，以便Branch target predictor对分支指令进行预测，这个硬件单元叫做Branch Target Buffer。程序中的分支指令辣么多，Branch Target Buffer不可能保存所有，只能cache近期使用到的分支跳转信息。

（2）tlbi这条指令通过猜测也知道是对TLB进行invalidation的操作，但是vmalle1is是什么鬼？它其实是vm-all-e1-is，vmall表示要invalidate all TLB entry，e1表示该操作适用于EL1，is表示inner sharebility。根据ARM ARM描述，这条指令的作用范围是inner shareable的所有PEs。这里有一个疑问：其实启动过程有些是只在BSP上进行，例如前面文章中的save boot parameter、校验blob、建立页表都是全局性的，只做一次就OK了。而这里的\_\_cpu\_setup函数是会在每一个cpu core上执行，因此应该尽量少的影响系统。如果这里是invalidation所有的inner shareable的PE的TLB，那么在secondary cpu core启动的时候会再执行一次，对系统影响很大，合理的操作应该是操作自己的TLB就OK了。

（3）step 1和step 2的操作和打开MMU操作有严格的时序要求，dsb这个memory barrier操作可以保证在执行打开MMU的时候，step 1和step 2都已经执行完毕。同样的，ish表示inner shareable。

（4）CPACR\_EL1（Architectural Feature Access Control Register）是用来控制Trace，浮点运算单元以及SIMD单元的。FPEN，bits [21:20]是用来控制EL0和EL1状态的时候访问浮点单元和SIMD单元是否会产生exception从进入上一个exception level。这里的设定运行用户空间（EL0）和内核空间（EL1）访问浮点单元和SIMD单元。MDSCR\_EL1（Monitor Debug System Control Register）主要用来控制debug系统的。

## 三、Memory attributes lookup table的构建

1、overview

MMU的作用有三个：地址映射，控制memory的访问权限，控制memory attribute。[ARM64的启动过程之（二）：创建启动阶段的页表](http://www.wowotech.net/linux_kenrel/create_page_tables.html)对前面两个功能有了简单的描述，关于memory attribute将在本节描述。在Translation table中描述符中除了地址信息还有一些attribute的信息，例如attribute index域，既然叫做index则说明该域并没有保存实际的memory attribute，实际的attribute保存在MAIR\_ELx中。在这个64 bit的寄存器中，每8个bit一组，形成一种类型的memory attribute。

**2、memory type**

我们知道，ARMv8采用了weakly-order内存模型，也就是说，通俗的讲就是处理器实际对内存访问（load and store）的执行序列和program order不一定保持严格的一致，处理器可以对内存访问进行reorder。例如：对于写操作，processor可能会合并两个写的请求。处理器这么任性当然是从性能考虑，不过这大大加大了软件的复杂度（软件工程师需要理解各种memory barrier操作，例如ISB/DSB/DMB，以便控制自己程序的内存访问的order）。

地址空间那么大，是否都任由processor胡作非为呢？当然不是，例如对于外设的IO地址，处理必须要保持其order。因此memory被分成两个基本的类型：normal memory和devicememory。除了基本的memory type，还有memory attribute（例如：cacheability，shareability）来进一步进行描述，我们在下一节描述。

标识为normal memory type的memory就是我们常说的内存而已，对其访问没有副作用（side effect），也就是说第n次和第n＋1次访问没有什么差别。device memory就不会这样，对一些状态寄存器有可能会read clear，因此n和n+1的内存访问结果是不一样的。正因为如此，processor可以对这些内存操作进行reorder、repeat或者merge。我们可以把程序代码和数据所在的memory设定为normal memory type，这样可以获取更高的性能。例如，在代码执行过程中，processor可能进行分支预测，从而提前加载某些代码进入pipeline（而实际上，program不一定会fetch那些指令），如果设定了不正确的memory type，那么会阻止processor进行reorder的动作，从而阻止了分支预测，进而影响性能。

对于那些外设使用的IO memory，对其的访问是有side effect的，很简单的例子就是设备的FIFO，其地址是固定不变的，但是每次访问，内部的移位寄存器就会将下一个数据移出来，因此每次访问同一个地址实际上返回的数据是不一样的。device不存在cache的设定，总是no cache的，处理器访问device memory的时候，限制会比普通memory多，例如不能进行Speculative data accesses（所谓不能进行Speculative data accesses就是说cpu对memory的访问必须由顺序执行的执行产生，不能由于自己想加快性能而投机的，提前进行某些数据访问）。

**3、 memory attribute**

上一节将memory分成两个大类：normal memory和device，但是这么分似乎有些粗糙，我们可以进一步通过memory attribute将memory分成更多的区域。一个memory range对应的memory attribute是定义在页表的描述符中（由upper attribues和lower attributes组成），最重要的attributes定义在lower attributes中的AttrIndx[2:0]，该域只是一个index而已，指向MAIR\_ELx中具体的memory attribute。8-bit的memory attribute的具体解释可以参考ARM ARM。

对于device type，其总是non cacheable的，而且是outer shareable，因此它的attribute不多，主要有下面几种附加的特性：

（1）Gathering 或者non Gathering (G or nG)。这个特性表示对多个memory的访问是否可以合并，如果是nG，表示处理器必须严格按照代码中内存访问来进行，不能把两次访问合并成一次。例如：代码中有2次对同样的一个地址的读访问，那么处理器必须严格进行两次read transaction。

（2）Re-ordering (R or nR)。这个特性用来表示是否允许处理器对内存访问指令进行重排。nR表示必须严格执行program order。

（3）Early Write Acknowledgement (E or nE)。PE访问memory是有问有答的（更专业的术语叫做transaction），对于write而言，PE需要write ack操作以便确定完成一个write transaction。为了加快写的速度，系统的中间环节可能会设定一些write buffer。nE表示写操作的ack必须来自最终的目的地而不是中间的write buffer。

对于normal memory，可以是non-cacheable的，也可以是cacheable的，这样就需要进一步了解Cacheable和shareable atrribute，具体如下：

（1）是否cacheable

（2）write through or write back

（3）Read allocate or write allocate

（4）transient or non-transient cache

最后一点要说明的是由于cache hierararchy的存在，memory的属性可以针对inner和outer cache分别设定，具体如何区分inner和outer cache是和具体实现相关，但通俗的讲，build in在processor内的cache是inner的，而outer cache是processor通过bus访问的。

**4、代码分析**

    ldr    x5, =MAIR(0x00, MT\_DEVICE\_nGnRnE) | \   
             MAIR(0x04, MT\_DEVICE\_nGnRE) | \   
             MAIR(0x0c, MT\_DEVICE\_GRE) | \   
             MAIR(0x44, MT\_NORMAL\_NC) | \   
             MAIR(0xff, MT\_NORMAL)   
    msr    mair\_el1, x5

页表中的memory attribute的信息并非直接体现在descriptor中的bit中，而是通过了一个间接的手段。描述符中的AttrIndx[2:0]是一个index，可以定位到8个条目，而这些条目就是保存在MAIR\_EL1（Memory Attribute Indirection Register (EL1)）中。对于ARM64处理器，linux kernel定义了下面的index：

#define MT\_DEVICE\_nGnRnE    0   
#define MT\_DEVICE\_nGnRE        1   
#define MT\_DEVICE\_GRE        2   
#define MT\_NORMAL\_NC        3   
#define MT\_NORMAL        4

NC是no cache，也就是说MT\_NORMAL\_NC的memory是normal memory，但是对于这种类型的memory的访问不需要通过cache系统。这些index用于页表中的描述符中关于memory attribute的设定，对于初始化阶段的页表都是被设定成MT\_NORMAL。

## 四、SCTLR\_EL1、TCR\_EL1的设定

1、寄存器介绍

SCTLR\_EL1是一个对整个系统（包括memory system）进行控制的寄存器，我们这里描述几个重要的域。这些域有两种类型，一种是控制EL0状态时候能访问的资源。例如：UCI bit[26]控制是否允许EL0执行cache maintemance的指令（DC或者IC指令），如果不允许，那么会陷入EL1。nTWE bit[18]控制是否允许EL0执行WFE指令，如果不允许，那么会陷入EL1。bit 16类似bit 18，但是是for WFI指令的。UCT bit[15]控制是否允许EL0访问CTR\_EL0（该寄存器保存了cache信息），如果不允许，那么会陷入EL1。UMA，bit [9]控制是否可以访问cpu状态寄存器的PSTATE.{D,A, I, F}比特。还有一种是实际控制memory system的域，例如：C bit[2]是用来enable或者disable EL0 & EL1 的data cache。具体包括通过stage 1 translation table访问的memory以及对stage 1 translation table自身memory的访问。I bit[12]是用来enable或者disable EL0 & EL1 的instruction cache。M bit[0]是用来enable或者disable EL0 & EL1 的MMU。

我们知道，kernel space和user space使用不同的页表，因此有两个Translation Table Base Registers，形成两套地址翻译系统，TCR\_EL1寄存器主要用来控制这两套地址翻译系统。TBI1，bit[38]和TBI0，bit[37]用来控制是否忽略地址的高8位（TBI就是Top Byte ignored的意思），如果允许忽略地址的高8位，那么MMU的硬件在进行地址比对，匹配的时候忽略高八位，这样软件可以自由的使用这个byte，例如对于一个指向动态分配内存的对象指针，可以通过高8位来表示reference counter，从而可以跟踪其使用情况，reference count等于0的时候，可以释放内存。AS bit[36]用来定义ASID（address space ID）的size，A1, bit [22]用来控制是kernel space还是user space使用ASID。ASID是和TLB操作相关，一般而言，地址翻译的时候并不是直接查找页表，而是先看TLB是否命中，具体判断的标准是虚拟地址＋ASID，ASID是每一个进程分配一个，标识自己的进程地址空间。这样在切换进程的时候不需要flush TLB，从而有助于performance。TG1，bits [31:30]和TG0，bits [15:14]是用来控制page size的，可以是4K，16K或者64K。当MMU进行地址翻译的时候需要访问页表，SH1, bits [29:28]和SH0, bits [13:12]是用来控制页表所在memory的Shareability attribute。ORGN1, bits [27:26]和ORGN0, bits [11:10]用来控制页表所在memory的outercachebility attribute的。IRGN1, bits [25:24]和IRGN0, bits [9:8]用来控制页表所在memory的inner cachebility attribute的。T1SZ, bits [21:16]和T0SZ, bits [5:0]定义了虚拟地址的宽度。

2、代码分析

代码位于arch/arm64/mm/proc.S中，该函数主要为打开MMU做准备，具体代码如下：

    adr    x5, crval －－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    ldp    w5, w6, [x5]   
    mrs    x0, sctlr\_el1   
    bic    x0, x0, x5            // clear bits   
    orr    x0, x0, x6            // set bits    
  
    ldr    x10, =TCR\_TxSZ(VA\_BITS) | TCR\_CACHE\_FLAGS | TCR\_SMP\_FLAGS | \   
            TCR\_TG\_FLAGS | TCR\_ASID16 | TCR\_TBI0   
    tcr\_set\_idmap\_t0sz    x10, x9 －－－－－－－－－－－－－－－－－－－－－－－（2）

    mrs    x9, ID\_AA64MMFR0\_EL1   
    bfi    x10, x9, #32, #3   
    msr    tcr\_el1, x10 －－－－－－－－－－－－－－－－－－－－－－－－－－－－（3）   
    ret                    // return to head.S －－－－－－－－－－－－－－－－－－－－－（4）   
ENDPROC(\_\_cpu\_setup)

（1）在调用\_\_enable\_mmu之前要准备好SCTLR\_EL1的值，该值在这段代码中设定并保存在x0寄存器中，随后做为参数传递给\_\_enable\_mmu函数。具体怎么设定SCTLR\_EL1的值呢？这是通过crval变量设定的，如下：

    .type    crval, #object   
crval:   
    .word    0xfcffffff－－－－－－－－－－－－－－－－SCTLR\_EL1寄存器中需要清0的bit   
    .word    0x34d5d91d －－－－－－－－－－－－－SCTLR\_EL1寄存器中需要设置成1的bit

由代码可知，EE和E0E这两个bit没有清零，因此实际上这些bit保持不变（在el2\_setup中已经设定）。这里面具体各个bit的含义清参考ARM ARM文档，我们不一一说明了。

（2）这里的代码是准备TCR寄存器的值。TCR\_TxSZ(VA\_BITS)是根据CONFIG\_ARM64\_VA\_BITS配置来设定内核和用户空间的size，其他的是进行page size的设定或者是page table对应的memory的attribute的设定，具体可以对照ARM ARM文档进行分析。

（3）到这里x10已经准备好了TCR寄存器的值，还缺省IPS（Intermediate Physical Address Size）的设定（IPS和2 stage地址映射相关，它和虚拟化有关，这里就不展开描述了，内容太多）。ID\_AA64MMFR0\_EL1, AArch64 Memory Model Feature Register 0，该寄存器保存了memory model和memory management的支持情况，该寄存器的PARange保存了物理地址的宽度信息，bfi    x10, x9, #32, #3 指令就是将x9寄存器的内容左移32bit，copy 3个bit到x10寄存器中（IPS占据bits [34:32]）。

（4）stext的代码如下：

ENTRY(stext)

……   
    ldr    x27, =\_\_mmap\_switched   
    adr\_l    lr, \_\_enable\_mmu   
    b    \_\_cpu\_setup   
ENDPROC(stext)

在调用\_\_cpu\_setup之前设定了lr的内容是\_\_enable\_mmu，而调用\_\_cpu\_setup使用的是b而不是bl指令，因此lr寄存器没有修改，因此，这里的ret返回到\_\_enable\_mmu函数。

五、参考文献

1、ARM Architecture Reference Manual

# ARM64的启动过程之（四）：打开MMU

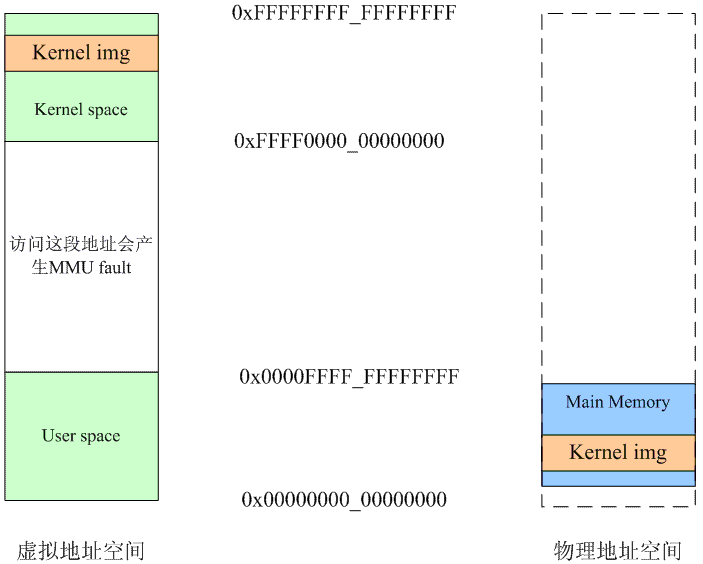
ARM64的启动过程之（四）：打开MMU

## 一、前言

经过漫长的前戏，我们终于迎来了打开MMU的时刻，本文主要描述打开MMU以及跳转到start\_kernel之前的代码逻辑。这一节完成之后，我们就会离开痛苦的汇编，进入人民群众喜闻乐见的c代码了。

## 二、打开MMU前后的概述

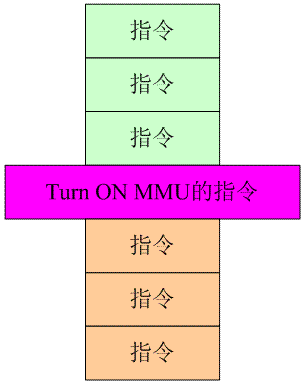
对CPU以及其执行的程序而言，打开MMU是一件很有意思的事情，好象从现实世界一下子走进了奇妙的虚幻世界，本节，我们一起来看看内核是如何“穿越”的。下面这张图描述了两个不同的世界：

[](http://www.wowotech.net/content/uploadfile/201512/60448863bf7d21ce6ac493a4b2d92de920151202015359.gif)

当没有打开MMU的时候，cpu在进行取指以及数据访问的时候是直接访问物理内存或者IO memory。虽然64bit的CPU理论上拥有非常大的address space，但是实际上用于存储kernel image的物理main memory并没有那么大，一般而言，系统的main memory在低端的一小段物理地址空间中，如上图右侧的图片所示。当打开MMU的时候，cpu对memory系统的访问不能直接触及物理空间，而是需要通过一系列的Translation table进行翻译。虚拟地址空间分成三段，低端是0x00000000\_00000000～0x0000FFFF\_FFFFFFFF，用于user space。高端是0xFFFF0000\_00000000～0xFFFFFFFF\_FFFFFFFF，用于kernel space。中间的一段地址是无效地址，对其访问会产生MMU fault。虚拟地址空间如上图右侧的图片所示。

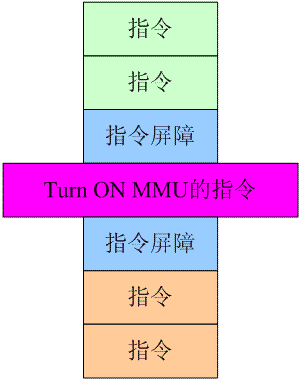
Linker感知的是虚拟地址，在将内核的各个object文件链接成一个kernel image的时候，kernel image binary code中访问的都是虚拟地址，也就是说kernel image应该运行在Linker指定的虚拟地址空间上。问题来了，kernel image运行在那个地址上呢？实际上，将kernel image放到kernel space的首地址运行是一个最直观的想法，不过由于各种原因，具体的arch在编译内核的时候，可以指定一个offset（TEXT\_OFFSET），对于ARM64而言是512KB（0x00080000），因此，编译后的内核运行在0xFFFF8000\_00080000的地址上。系统启动后，bootloader会将kernel image copy到main memory，当然，和虚拟地址空间类似，kernel image并没有copy到main memory的首地址，也保持了一个同样size的offset。现在，问题又来了：在kernel的开始运行阶段，MMU是OFF的，也就是说kernel image是直接运行在物理地址上的，但是实际上kernel是被linker链接到了虚拟地址上去的，在这种情况下，在没有turn on MMU之前，kernel能正常运行吗？可以的，如果kernel在turn on MMU之前的代码都是PIC的，那么代码实际上是可以在任意地址上运行的。你可以仔细观察turn on MMU之前的代码，都是位置无关的代码。

OK，解决了MMU turn on之前的问题，现在我们可以准备“穿越”了。真正打开MMU就是一条指令而已，就是将某个system register的某个bit设定为1之类的操作。这样我们可以把相关指令分成两组，turn on mmu之前的绿色指令和之后的橘色指令，如下图所示：

[](http://www.wowotech.net/content/uploadfile/201510/a4baf59a65665a921228cf031606bbe320151024043505.gif)

由于现代CPU的设计引入了pipe， super scalar，out-of-order execution，分支预测等等特性，实际上在turn on MMU的指令执行的那个时刻，该指令附近的指令的具体状态有些混乱，可能绿色指令执行的数据加载在实际在总线上发起bus transaction的时候已经启动了MMU，本来它是应该访问physical address space的。而也有可能橘色的指令提前执行，导致其发起的memory操作在MMU turn on之前就完成。为了解决这些混乱，可以采取一种投机取巧的办法，就是建立一致性映射：假设kernel image对应的物理地址段是A～B这一段，那么在建立页表的时候就把A～B这段虚拟地址段映射到A～B这一段的物理地址。这样，在turn on MMU附近的指令是毫无压力的，无论你是通过虚拟地址还是物理地址，访问的都是同样的物理memory。

还有一种方法，就是清楚的隔离turn on MMU前后的指令，那就是使用指令同步工具，如下：

[](http://www.wowotech.net/content/uploadfile/201510/4e9eff9c8d6e4d68316b168c76299b3620151024043506.gif)

指令屏障可以清清楚楚的把指令的执行划分成三段，第一段是绿色指令，在执行turn on mmu指令执行之前全部完成，随后启动turn on MMU的指令，随后的指令屏障可以确保turn on MMU的指令完全执行完毕（整个计算机系统的视图切换到了虚拟世界），这时候才启动橘色指令的取指、译码、执行等操作。

## 三、打开MMU的代码

具体打开MMU的代码在\_\_enable\_mmu函数中如下：

\_\_enable\_mmu:   
    ldr    x5, =vectors   
    msr    vbar\_el1, x5 －－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    msr    ttbr0\_el1, x25            // load TTBR0 －－－－－－－－－－－－－－－－－（2）   
    msr    ttbr1\_el1, x26            // load TTBR1   
    isb   
    msr    sctlr\_el1, x0 －－－－－－－－－－－－－－－－－－－－－－－－－－－（3）   
    isb   
    br    x27 －－－－－－－－－－－－－跳转到\_\_mmap\_switched执行，不设定lr寄存器   
ENDPROC(\_\_enable\_mmu)

传入该函数的参数有四个，一个是x0寄存器，该寄存器中保存了打开MMU时候要设定的SCTLR\_EL1的值（在\_\_cpu\_setup函数中设定），第二个是个是x25寄存器，保存了idmap\_pg\_dir的值。第三个参数是x26寄存器，保存了swapper\_pg\_dir的值。最后一个参数是x27，是执行完毕该函数之后，跳转到哪里去执行（\_\_mmap\_switched）。

（1）VBAR\_EL1, Vector Base Address Register (EL1)，该寄存器保存了EL1状态的异常向量表。在ARMv8中，发生了一个exception，首先需要确定的是该异常将送达哪一个exception level。如果一个exception最终送达EL1，那么cpu会跳转到这里向量表来执行。具体异常的处理过程由其他文档描述，这里就不说了。

（2）idmap\_pg\_dir是为turn on MMU准备的一致性映射，物理地址的高16bit都是0，因此identity mapping必定是选择TTBR0\_EL1指向的各级地址翻译表。后续当系统运行之后，在进程切换的时候，会修改TTBR0的值，切换到真实的进程地址空间上去。TTBR1用于kernel space，所有的内核线程都是共享一个空间就是swapper\_pg\_dir。

（3）打开MMU。实际上在这条指令的上下都有isb指令，理论上已经可以turn on MMU之前之后的代码执行顺序严格的定义下来，其实我感觉不必要再启用idmap\_pg\_dir的那些页表了，当然，这只是猜测。

## 四、通向start\_kernel

我痛恨汇编，如果能不使用汇编那绝对不要使用汇编，还好我们马上就要投奔start\_kernel：

\_\_mmap\_switched:   
    adr\_l    x6, \_\_bss\_start   
    adr\_l    x7, \_\_bss\_stop

1:    cmp    x6, x7   
    b.hs    2f   
    str    xzr, [x6], #8 －－－－－－－－－－－－－－－clear BSS   
    b    1b   
2:   
    adr\_l    sp, initial\_sp, x4 －－－－－－－－－－－建立和swapper进程的链接   
    str\_l    x21, \_\_fdt\_pointer, x5        // Save FDT pointer   
    str\_l    x24, memstart\_addr, x6        // Save PHYS\_OFFSET   
    mov    x29, #0   
    b    start\_kernel   
ENDPROC(\_\_mmap\_switched)

这段代码分成两个部分，一部分是清BSS，另外一部分是为进入c代码做准备（主要是stack）。clear BSS段就是把未初始化的全局变量设定为0的初值，没有什么可说的。要进入start\_kernel这样的c代码，没有stack可不行，那么如何设定stack呢？熟悉kernel的人都知道，用户空间的进程当陷入内核态的时候，stack切换到内核栈，实际上就是该进程的thread info内存段（4K或者8K）的顶部。对于swapper进程，原理是类似的：

.set    initial\_sp, init\_thread\_union + THREAD\_START\_SP

如果说之前的代码执行都处于一个孤魂野鬼的状态，“adr\_l    sp, initial\_sp, x4”指令执行之后，初始化代码终于找到了归宿，初始化代码有了自己的thread info，有了自己的task struct，有了自己的pid，有了进程（内核线程）应该拥有的一切，从此之后的代码归属idle进程，pid等于0的那个进程。

为了方便后面的代码的访问，这里还初始化了两个变量，分别是\_\_fdt\_pointer（设备树信息，物理地址）和memstart\_addr（kernel image所在的物理地址，一般而言是main memory的首地址）。 memstart\_addr主要用于main memory中物理地址和虚拟地址的转换，具体可以参考\_\_virt\_to\_phys和\_\_phys\_to\_virt的实现。

## 五、参考文献

1、ARM Architecture Reference Manual

change log：

1、2015-11-30，强调了初始化代码和idle进程的连接

2、2015-12-2，修改了物理空间和虚拟空间的视图

3、2016-9-21，修改对一致性映射的描述

# ARM64的启动过程之（五）：UEFI

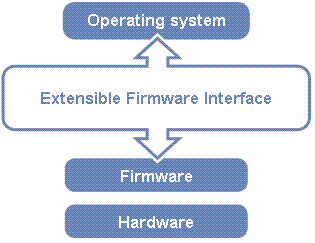
## 一、前言

在准备大刀阔斧进入start\_kernel之际，我又重新review了一下head.S文件，看看是否有一些遗漏的知识点，很不幸，看到了CONFIG\_EFI这个配置项。当然，在一年前阅读kernel代码的时候就了解过相关的内容，但是，做为一个嵌入式工程师总是或多或少对其有些排斥，因此习惯性的忽略掉CONFIG\_EFI相关的代码，逃避总不是办法，在本文中，我们一起来探讨ARM64平台上UEFI相关的内容。

## 二、背景介绍

1、UEFI是什么鬼？

在个人电脑刚兴趣的时代，能够进入BIOS（Basic Input/Output System）解决一些计算机的问题绝对是高手中的高手（当年我就是这么骗到老婆的）。所谓BIOS实际上就是IBM PC兼容机（多么古老的一个词汇啊）主板上的固件（firmware），这些固件可以在系统启动过程中初始化硬件，self test，加载bootloader或者OS kernel，并且能为OS提供一些基础的服务。由于各种存在的问题，后来，Intel提出来EFI（Extensible Firmware Interface）来取代BIOS interface。2005年，Intel终止了EFI规范的开发，替代它的是Unified EFI Forum负责的UEFI（Unified Extensible Firmware Interface）specification。UEFI在系统中的位置如下（图片来自wiki）：

[](http://www.wowotech.net/content/uploadfile/201510/e1acc41f3cee9de326092ff1aa24a84c20151030112749.gif)

随着PC和服务器的飞速发展，软件和硬件厂商都不断的研发各种新的产品来应对客户的需求，在整合成系统的时候，有大量的协调的工作需要做，并且是越来越复杂。为了加快整合，降低设计复杂度，需要一个统一的接口标准，也就是传说中的UEFI了。有了UEFI，OS（软件厂商阵营）和固件（硬件厂商阵营）就有了接口规格，这样，大家可以各自进行开发，只要符合UEFI规格就OK了。如果硬件厂商有了创新性的硬件特性，如果不需要修改UEFI接口，那么系统还是可以无缝的衔接，如果需要修改接口，那么提前修改接口规格，让参与整个系统构建的厂商可以同步前进。同样的，从软件角度看，如果创新性的软件算法需要HW的支持，那么可以通过UEFI这样的接口和硬件厂商阵营进行交互，大大加快了将整个系统交付给客户的时间。

**2、UEFI关ARM什么事？**

如果ARM仅仅是将目光放在移动（嵌入式）市场，那么UEFI当然不关ARM什么事情。在嵌入式ARM平台上，ROM code ＋ bootloader（例如Uboot）＋ linux kernel这样的组合可以很好的工作。但是，在推出ARMv8以及64 bit架构的的处理器之后，ARM的野心已经不满足在移动市场上称王了。不过嵌入式平台和server或者PC类的平台是有区别的：嵌入式平台往往是高度定制化的平台，各个硬件模块都是不可分割的。如果你购买了一个手机，如果你觉得LCD不满意，是不可能单独去市场购买一个LCD屏更换的。而server（PC）类产品则不然，各个模块是可以更换的。例如：可以自由的去购买一个硬盘或者显卡进行更换。

在移动平台上，firmware（ROM code）怎么做是自己的事情，只要在应用层面提供一致性的接口就OK了，反正硬件以及OS不会更换。来到服务器平台，ARM必须和她的合作伙伴（SOC，外围硬件，OS厂商等等）一起面对这样的问题：

（1）硬件平台（firmware）和OS之间的接口如何定义？

（2）如何向OS传递硬件信息？

为了让各个厂商能够协同工作，尽快将ARM服务器推向市场，选择一个标准让大家follow是一个不错的主意。我们以OS提供商为例描述选择标准的好处。如果定义了硬件平台和OS之间的标准，OS提供商可以为ARMv8 server发布一个image而不会因为任何一点硬件平台的修改就得发布一个新的OS。因此，ARMv8 server选择UEFI是很自然的事情了。

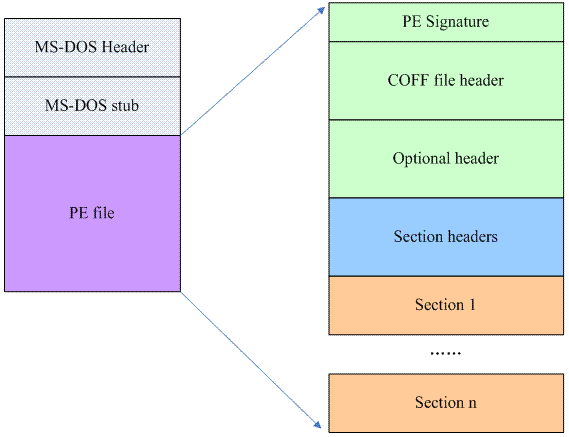
**3、UEFI如何定义系统的启动过程？**

相信大家对传统的嵌入式ARM平台的启动过程都是有所了解的，系统reset后，各个ARM SOC的从ROM代码开始执行（一般ARM reset之后，PC＝0，而ROM缺省地址就是0）。根据SOC厂商约定的规则，ROM code会从外部设备（串口、网络、NAND flash、USB磁盘设备或者其他磁盘设备）加载linux bootloader，bootloader会收集硬件信息，之后加载linux kernel。在UEFI规范中定义了BOOT manager，它会根据保存在NVRAM参数来决定如何load EFI Application（可能是bootloader或者其他的image file）。EFI Application的格式必须符合PE（Portable Executable ）格式。PE是一种二进制可执行文件的格式（在linux世界中，我们多半熟悉的是ELF格式），由微软开发，广泛应用在Windows平台上。

在ARMv8平台上，firmware中的boot manager可以加载支持UEFI的传统的bootloader（例如uboot），然后由uboot加载kernel，这样，kernel其实不必关心什么UEFI。当然这样有些不直观，本来OS kernel关心的那些firmeare提供的各种信息都是由bootloader进行转接，严重影响了系统整合的效率（bootloader和kernel是由不同的团队开发），因此，linux kernel image自身也可以包装成一个EFI image，由boot manager直接加载，完成启动过程。

**4、PE格式介绍**

下面的图片是一个PE文件格式的示意图：

[](http://www.wowotech.net/content/uploadfile/201510/0b7b5a396f9e2728331cca89c688f0c220151030112750.gif)

PE文件主要由两部分组成，一部分是为了兼容MS-DOS操作系统而包装的外壳（灰色block），主要由64B的MZ header和MS-DOS stub代码区组成。在遥远的MSDOS时代，其可执行文件就需要这样的一个header，MSDOS的program loader就会根据这个header加载程序运行。在Windows时代，微软提出了PE这种格式文件，它主要是运行在windows系列的操作系统中，但是，还需要考虑MSDSO的兼容性（也就是说当MSDOS执行PE格式的文件也能够提供足够的信息让用户知道如何处理）。MS-DOS stub block是一段stub code，这段区域的主要作用是：当PE格式的image在MS-DOS下加载运行的时候，程序会执行这个区域的代码（PE的代码都是for windows的，不可能在DOS下实际执行，因此，只能执行这些stub程序），当然运行的结果仅仅是打印“This program cannot be run in DOS mode”。

另外一个区域就是实际的PE格式的文件了。主要包括PE header（绿色block）、各种Section header（蓝色block，用于描述各个section）和各个section的实际的Data。各个域的具体含义我们会结合具体的代码在下一章描述。

## 三、代码分析：

1、MZ header。相关代码如下所示：

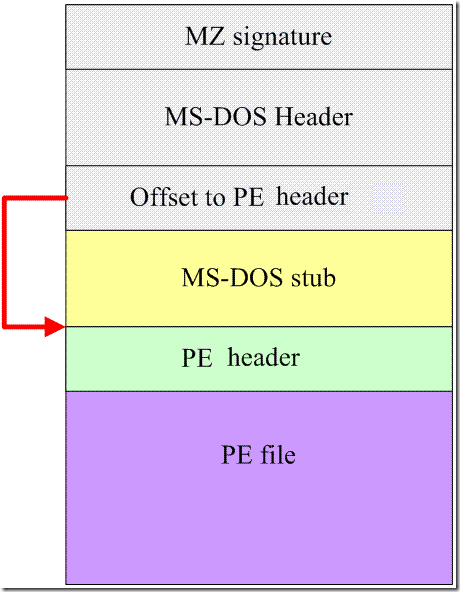
#ifdef CONFIG\_EFI   
efi\_head:   
    add    x13, x18, #0x16 －－－－－－－－－－－－－－－－－－－－－－－－－－（2）   
    b    stext   
#else   
    b    stext－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
    .long    0    
#endif   
    .quad    \_kernel\_offset\_le－－－－－－－－－－－－－－－－－－－－－－－－－（3）   
    .quad    \_kernel\_size\_le    
    .quad    \_kernel\_flags\_le   
    .quad    0                // reserved   
    .quad    0                // reserved   
    .quad    0                // reserved   
    .byte    0x41－－－－－－－－－－－－Magic number, "ARM\x64"   
    .byte    0x52   
    .byte    0x4d   
    .byte    0x64

#ifdef CONFIG\_EFI   
    .long    pe\_header - efi\_head－－－－－－－－－－－－－－－－－－－－－－－（4）   
#else   
    .word    0                // reserved   
#endif

这里定义了64字节的kernel image header，应对两种场景：一种是从普通的linux bootloader加载内核，另外一种是从UEFI firmware直接加载kernel（定义了CONFIG\_EFI ），在这种场景下，这64B的内容被解释为MZ header。

（1）大部分的kernel image header都是相同的，除了第一个8-Byte和最后的4-Byte。没有定义CONFIG\_EFI 是大家都比较熟悉的场景，当bootloader完成kernel image的从外设到RAM的搬移之后会执行kernel image的第一条指令。因此，这里是一条跳转到stext的指令。

（2）如果想把自己伪装成一个UEFI image，kernel需要符合PE格式，下面是一个简化版本的PE格式的示意图（仅仅包括部分格式，主要用来说明兼容MS-DOS 相关部分的内容）：

[](http://www.wowotech.net/content/uploadfile/201510/46c93364c68c86d081689dbb3371624020151030112755.gif)

上图中的灰色区域就是64-Byte的MZ header（对应kernel image header的内容），当然，对于linux kernel而言，它只是伪装成PE格式而已，只要能够提供足够的信息给UEFI firmware的boot manager就OK了。PE格式的文件除了包括一个MZ header，还包括一段MS-DOS stub（上图中的黄色区域），当然，对于linux kernel image，我们没有提供这部分的内容。这里“add    x13, x18, #0x16”这条指令没有任何实际的意义，这条指令的opcode实际上就是MZ signature，用来标识这是一个DOS MZ executable的image。

（3）对于UEFI firmware而言，MS-DOS header大部分的区域都是没有什么用处的，因此正好可以用来提供信息，以便让linux的bootloader可以知道如何加载kernel（非UEFI加载的情况）。\_kernel\_offset\_le标识加载kernel的位置，如果等于0，表示加载到RAM的0地址的位置上。\_kernel\_size\_le表示需要加载的kernel image的长度，\_kernel\_flags\_le是表示kernel的一些属性，目前仅仅使用了bit 0，表示kernel的endianess。

（4）在UEFI firmware加载kernel的情况下，需要找到PE header以及各个section的定义了，以便boot manager完成加载kernel image的任务。在MS-DOS header中（offset是0x3c）有四个字节指向了PE header，通过它可以找到如何加载内核的各种信息。这个过程是这样的：UEFI firmware的boot manager如果发现了MZ header，那么就认为这是一个符合标准的EFI image，并在0x3c处获取PE header的位置，并继续解析其内容以便加载kernel image。

2、PE header相关代码

PE header包括三部分的内容：PE signature、COFF（Common Object File Format）file header和optional header。PE signature和COFF file header的代码如下：

pe\_header:   
    .ascii    "PE" －－－－－－－－－－－－－－－－PE header signanature   
    .short     0   
coff\_header:   
    .short    0xaa64－－－－－－－－－－表示machine type是AArch64   
    .short    2－－－－－－－－－－－－该PE文件有多少个section   
    .long    0－－－－－－－－－－－－该文件的创建时间   
    .long    0－－－－－－－－－－－－符号表信息   
    .long    1－－－－－－－－－－－－符号表中的符号的数目   
    .short    section\_table - optional\_header －－－－－－－－optional header的长度   
    .short    0x206－－－－－－－－－－－－－－－Characteristics，具体的含义请查看PE规格书

上节我们说过，通过MZ header可以找到PE header，所谓PE header的开始位置实际上就是一个“PE\0\0”的signature，随后紧接着就是COFF file header，COFF file header具体的定义如下（该表格来自PE specification）：

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Size** | **Field** | **Description** |
| 0 | 2 | Machine | The number that identifies the type of target machine |
| 2 | 2 | NumberOfSections | The number of sections. This indicates the size of the section table, which immediately follows the headers. |
| 4 | 4 | TimeDateStamp | The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time\_t value), that indicates when the file was created. |
| 8 | 4 | PointerToSymbolTable | The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated. |
| 12 | 4 | NumberOfSymbols | The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated. |
| 16 | 2 | SizeOfOptionalHeader | The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see section 3.4, “Optional Header (Image Only).” |
| 18 | 2 | Characteristics | The flags that indicate the attributes of the file |

NumberOfSections定义了PE文件中的section的数目，对于linux kernel image的PE文件，包括了两个section，一个是.reloc section（这是EFI application loader需要的，我们这里只是提供了一个dummy版本的.reloc section），另外一个是.text section（整个kernel image）。

通过COFF file header中的SizeOfOptionalHeader域，UEFI firmware可以知道optional header的size。之所以是“optional”主要是因为这些header内容不一定会存在。例如：对于object文件，这些header不存在。当然，我们是UEFI image file（可执行文件），因此这些optional header是必须提供的。optional\_header的最开始的域是optional header magic number，用来确定该PE文件是PE32还是PE32+格式的。根据UEFI规范，UEFI application file应该是PE32+格式的。PE32+格式的optional header格式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Size** | **Header part** | **Description** |
| 0 | 28/24 | Standard fields | Fields that are defined for all implementations of COFF, including UNIX. |
| 28/24 | 68/88 | Windows-specific fields | Additional fields to support specific features of Windows (for example, subsystems). |
| 96/112 | Variable | Data directories | Address/size pairs for special tables that are found in the image file and are used by the operating system (for example, the import table and the export table). |

Standard fields包括了如何加载以及如何运行的信息。相关的代码如下：

optional\_header:   
    .short    0x20b                // PE32+ format   
    .byte    0x02                // MajorLinkerVersion   
    .byte    0x14                // MinorLinkerVersion   
    .long    \_end - stext            // SizeOfCode   
    .long    0                // SizeOfInitializedData   
    .long    0                // SizeOfUninitializedData   
    .long    efi\_stub\_entry - efi\_head    // AddressOfEntryPoint   
    .long    stext\_offset            // BaseOfCode

比较重要的信息包括：代码段在image file中的偏移（BaseOfCode），正文段的大小（SizeOfCode），data段的大小（SizeOfInitializedData），bss段的大小（SizeOfUninitializedData），加载到memory后入口函数（AddressOfEntryPoint，对于linux kernel而言，入口函数是efi\_stub\_entry）。

Windows-specific fields和Data directories主要被Windows操作系统的linker和loader使用的，这里就不详述了。

**3、Section table和section Data**

大家有兴趣可以自己查阅PE规格，我这里就偷懒啦，^\_^。

四、参考文献：

1、<https://lwn.net/Articles/584123/>

2、<http://www.linaro.org/blog/when-will-uefi-and-acpi-be-ready-on-arm/>

3、<https://lwn.net/Articles/574439/>

4、PE规格书

5、UEFI规格书

# ARM64的启动过程之（六）：异常向量表的设定

ARM64的启动过程之（六）：异常向量表的设定

## 一、前言

本文主要描述了4.1.10内核初始化过程中如何初始化异常向量表。当然，首先需要准备一些异常的基础知识，这主要在第二章，如果你非常熟悉ARM64的异常，那么可以忽略这个章节。 第三章描述了ARM64上各种形形色色的异常，第四章描述了ARM64上硬件提供的协助，最后一章描述了代码过程。

为了简化，本文对所描述的异常进行了限制：

1、所有的exception level的运行状态都是AArch64，不考虑异常发生在AArch32 excution state的时候

2、不考虑支持security extension，也就是说EL3状态的异常处理也不在本文描述

3、不考虑virtualization的支持，也就是说EL2的异常处理不会在本文描述

一句话总结，本文主要描述EL0和EL1这两个exception level下的异常向量表的设定。

## 二、 异常（exception）的基础知识

1、什么是异常（exception）？

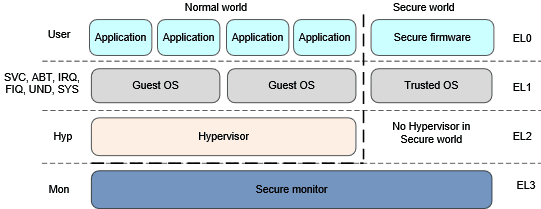
对于ARM64而言，exception是指cpu的某些异常状态或者一些系统的事件（可能来自外部，也可能来自内部），这些状态或者事件可以导致cpu去执行一些预先设定的，具有更高执行权利的软件（也叫exception handler）。执行exception handler可以进行异常的处理，从而让系统平滑的运行。exception handler执行完毕之后，需要返回发生异常的现场。上面这段话非常的拗口，但是不要紧，当进入细节的时候一切会慢慢清晰起来，下面我们逐一介绍各种异常：中断（interrupt），abort，reset，异常指令……

2、exception level

上一节中，我们在定义异常的时候说道：一旦异常发生，系统（包括硬件和软件）将切换到具有更高执行权利的状态，对于cpu而言，就是exception level了，ARM64最大支持EL0～EL3四个exception level，EL0的execution privilege最低，EL3的execution privilege最高。当发生异常的时候，系统的exception会迁移到更高的exception level或者维持不变，但是绝不会降低。此外，不会有任何的异常会去到EL0。

对比是一个不错的学习方法，我们先看看ARM32的情况。对于ARM32而言，cpu可以处在各种processor mode下，例如User、FIQ、IRQ、Abort、Undefined、System，这些不同的mode又对应两种privilege level，non-privilege（user processor mode）和privilege（其他processor mode）。

来到ARM64，processor mode这个概念已经消失了，取而代之的是exception level，如果没有支持两个security state（但是支持虚拟化），那么ARM64有3个exception level，分别是：EL0（对应user mode的application），EL1（guest OS）和EL2（Hypervisor）。如果支持两个security state（但是不支持虚拟化），ARM64还是有3个exception level，分别是：EL0（对应trusted service），EL1（trusted OS kernel）和EL3（Secure monitor）。如果支持了虚拟化并且同时支持两种security state，那么ARM64的处理器可以处于4种exception level，具体如下（摘自wowo同学的文章里面的图片，^\_^）：



由于EL3是和安全相关的，目前linux kernel并不涉及这部分的内容，因此本文将不考虑EL3这个exception level。

**2、异步异常（asynchronous exception）和同步异常（synchronous exception）**

虽然异常各具形态，但是基本可以分成两类，一类是asynchronous exception，另外一类是synchronous exception。

asynchronous exception基本上可以类似大家平常说的中断，它是毫无预警的，丝毫不考虑cpu core感受的外部事件（需要注意的是：外部并不是表示外设，这里的外部是针对cpu core而言，有些中断是来自SOC的其他HW block，例如GIC，这时候，对于processor或者cpu（指soc）而言，这些事件是内部的），这些事件打断了cpu core对当前软件的执行，因此称之interrupt。interrupt或者说asynchronous exception有下面的特点：

（1）异常和CPU执行的指令无关。

（2）返回地址是硬件保存下来并提供给handler，以便进行异常返回现场的处理。这个返回地址并非产生异常时的指令

根据这个定义IRQ、FIQ和SError interrupt属于asynchronous exception。

synchronous exception和asynchronous exception相反，其特点是：

（1）异常的产生是和cpu core执行的指令或者试图执行执行相关

（2）硬件提供给handler的返回地址就是产生异常的那一条指令所在的地址

synchronous exception又可以细分成两个类别，一种我们称之为synchronous abort，例如未定义的指令、data abort、prefetch instruction abort、SP未对齐异常，debug exception等等。还有一种是正常指令执行造成的，包括SVC/HVC/SMC指令，这些指令的使命就是产生异常。

3、precise exception

什么是precise exception呢？现代的cpu设计越来越复杂，各种pipe技术，各种cache技术，分支预测，multi-issue，out-of-order-execution等等，这些都让cpu core执行指令处于一个混沌的状态（主要是指不象sequential cpu 那么清晰），因此，当发生一个exception的时候，需要一个快刀来斩断那些处于各种纠缠中的乱麻一样的各种cpu以及memory的状态。这时候，我们需要选定一条指令，该指令之前的指令（不包括该指令）都已经全部执行完毕，修改了硬件上下文（cpu状态寄存器，各种通用寄存器，系统寄存器等等，memory hierarchy状态）。而该指令之后（包括该指令）的指令都没有执行（各种HW状态保持该指令之前状态），如果有执行的需要回退（exception的检测发生在流水线的write back stage，比较靠后，我们选定的那条指令应该在流水线的execute stage，是不是需要回退我也不知道，反正HW设计的时候需要考虑如何处理execute stage之后那些在流水线中的指令，确保硬件状态的一致性）。OK，快刀已经斩在了两条指令之间，让hardware context定格在这一刻，并把控制权转交给exception handler。

为何要定义precise exception呢？有什么用呢？无它，让生活变得简单一些（当然这里的生活不是去超市买东西，是指软件debug之类的程序员生活）。例如：这样exception handler返回现场就比较容易了。

在ARM64中，除了SError interrupt这种exception，其他的exception都是precise exception。

## 三、异常的分类

1、中断

中断主要有两种，physical interrupt和virtual interrupt（不在本文中描述）。physical interrupt是来自cpu core（或者叫做PE）外部一种信号，它包括下面三种类型：

（1）IRQ

（2）FIQ

（3）System error，简称SError

IRQ和FIQ是广大ARM嵌入式工程师的老朋友了，大家常说的中断实际上特指IRQ和FIQ，当然，实际上SError也是一种中断类型。IRQ和FIQ分别和cpu core的nIRQ和nFIQ这两根实际的信号线相关，interrupt controller收集各种来自外设的（或者来自其他CPU core的）中断请求，通过nIRQ和nFIQ的电平信号（active low）来通知cpu core某些外设的异步事件（或者来自其他CPU core的异步事件）。其中IRQ是普通中断，而FIQ是快速中断。由于中断来自cpu core的外部，可以在任何的时间点插入，和cpu core上执行的指令没有任何的关系，因此中断这种exception被归入asynchronous exception类别。

要想理解SError interrupt这个概念，我们需要先看看external abort这个术语。external abort来自memory system，当然不是所有的来自memory system的abort都是external abort，例如来自MMU的abort就不是external abort（这里的external是针对processor（而非cpu core）而言，因此MMU实际上是internal的）。external abort发生在processor通过bus访问memory的时候（可能是直接对某个地址的读或者写，也可能是取指令导致的memory access），processor在bus上发起一次transaction，在这个过程中发生了abort，abort来自processor之外的memory block、device block或者interconnection block，用来告知processor，搞不定了，你自己看着办吧。external abort可以被实现成synchronous exception（precise exception），也可以实现成asynchronous exception（imprecise exception）。如果external abort是asynchronous的，那么它可以通过SError interrupt来通知cpu core。

我的习惯是搞清楚一个术语的定义之后，下一个问题就是why，不过大部分的资料都不会讲why，因此在回答why的时候，往往没有那么权威，多半是自己的思考，不保证是正确的。为何要定义SError interrupt？当一个abort发生的时候，如果HW如果能够通过寄存器提供一些信息的话，那么事情还没有那么糟糕，例如return address给出了发生exception的具体的位置。从这个角度看，软件工程师更喜欢precise exception，毕竟知道导致异常的指令的位置，从而找到一些蛛丝马迹，让系统恢复。从这个角度看，根本就不应该存在SError interrupt这样的imprecise exception的鬼东西，HW没有提供有意义的信息，就只是两手一摊，反正在memory access的时候发生了system error，而且我也不知道是哪一条指令干的，这让软件工程师情何以堪呐。氮素，这只是事情的一个方面，从设计CPU的IC工程师角度看，他们更喜欢设计最快的处理器来完成自己的价值。SError interrupt是imprecise exception，允许更大的指令并发，因此性能更好。

2、reset

reset是一种优先级最高的异常，无法mask。系统首次上电，watchdog以及软件控制都可以让cpu core历经一次reset。reset有两种，一种是cold reset，另外一种是warm reset，它们之间唯一的不同是是否reset cpu core build-in的debug HW block。

3、abort

abort有两种，一种是和指令的执行有关，进入异常状态时候保持的返回地址精准的反应了发生异常的那一条指令，我们称之synchronoud abort。有同步就有异步，asynchronous abort（也就是上面描述的SError interrupt）和执行的指令无关，返回地址也不会提供abort时候的执行指令相关的精准信息。asynchronous abort和中断类似，只不过中断多半是来自外部（外设或者其他cpu core），而asynchronous abort来自外部memory system，源于bus上的一些错误，例如不可恢复的ECC error。

synchronoud abort有可能在cpu执行指令过程中的任何一步发生。例如在取指阶段失败，在译码阶段失败，在指令执行阶段等等。synchronoud abort和指令的执行过程有关，abort有可能在很早的阶段就被感知到，例如cpu core在将保存在memory系统中的指令读取到cpu core内部准备译码执行的时候就发现了错误，怎么办呢？由于指令还没有进入执行阶段（正在执行的是pipeline中的其他指令），因此不能触发exception，而是仅仅mark这个abort，一旦该指令在pipeline中到底execute stage，一次synchronoud abort被触发了。如果还没有执行cpu core就flush了这个pipeline（例如该指令的上一条指令是跳转指令），那么这次abort不会触发。

4、异常指令

对于ARM而言，system call被运行在non-privilege的软件用来向运行在privilege mode（例如linux kernel）的软件申请服务。来到ARM64，privilege包括了EL1/EL2/EL3，随之而来的是system call这个概念的扩展。从low privilege level都可以通过系统调用来申请higer privilege level的服务，因此，在ARM64中，能正常产生异常（不是abort）并申请拥有更高权利的软件服务的指令有三条：

（1）Supervisor Call (SVC)指令。类似于ARM时代的SWI指令，主要被EL0（user mode）的软件用来申请EL1（OS service）软件的服务。

（2）Hypervisor Call (HVC) 指令。主要被guest OS用来请求hypervisor的服务。

（3）Secure monitor Call (SMC) 指令，用来切换不同的世界，normal world或是secure world。

毫无疑问，SVC/HVC/SMC指令产生的异常属于synchronous exception。

## 四、ARM64为了异常处理提供了哪些帮助？

1、一旦发生了异常，处理器会切换到哪一个exception level？

对于ARM处理器，发生了不同的异常会进入不同的processor mode，例如发生了IRQ中断，处理器进入IRQ mode（privilege mode的一种）。对于ARM64，这个问题变成了处理器会切换到哪一个exception level？我们首先看reset和异常指令，因为这两个是最简单的。我们先看最简单的reset exception，当发生reset之后，处理处于那个EL呢？这个是和实现相关（实现的最高的EL是哪一个），一般而言，ARM64处理在reset的时候缺省会进入最高的那个exception level，例如如果处理器最高支持的EL是EL2，那么reset之后，系统就是处于EL2。对于那些正常通过system call产生的异常，处理器会切换到哪一个exception level这个问题也很好回答，SVC、HVC和SMC都有自己target exception level。

对于中断和abort，稍微有些复杂，这是通过各种寄存器可以进行配置的，具体参考ARM ARM文档。对于本文的场景，我们不支持EL2和EL3，因此exception的路由配置比较简单，大部分的异常都被路由到EL1来处理。

**2、为了方便软件工程师处理各个exception level 的exception handler，PE提供了哪些寄存器的支持？**

不同的exception level使用相同的general purpose registers，按64bit访问的话，寄存器是x0～x30共计31个寄存器，如果按照32bit访问的话，寄存器是w0～w30。

每个exception level都有自己的stack pointer register，名字是SP\_ELx，当然，前提是处理器支持该EL。对于EL0，只能使用SP\_EL0，对于其他的exception level，除了可以使用自己的SP\_ELx，还可以选择使用SP\_EL0。

当发生异常的时候，PE总是把当前cpu的状态保存在SPSR寄存器中，该寄存器有三个，分别是SPSR\_EL1，SPSR\_EL2，SPSR\_EL3，异常迁移到哪一个exception level就使用哪一个SPSR。由于不会有异常把系统状态迁移到EL0，因此也就不存在SPSR\_EL0了。在返回异常现场的时候，可以使用SPSR\_ELx来恢复PE的状态。

当发生异常的时候，PE把一个适当的返回地址保存在ELR（Exception Link Register）寄存器中，该寄存器有三个，分别是ELR\_EL1，ELR\_EL2，ELR\_EL3，异常迁移到哪一个exception level就使用哪一个ELR。同样的，由于不会有异常把系统状态迁移到EL0，因此也就不存在ELR\_EL0了。在返回异常现场的时候，可以使用ELR\_ELx来恢复PC值。

对于abort（包括synchronous exception和SError interrupt），ESR寄存器（ Exception Syndrome Register）保存了更详细的异常信息。ESR寄存器有三个，分别是ESR\_EL1，ESR\_EL2，ESR\_EL3。

3、如何定义exception handler？

系统有那么多异常，不同的异常有可以将处理器状态迁移到不同的exception level中，如何组织这些exception handler呢？第一阶是各个exception level的Vector Base Address Register (VBAR)寄存器，该寄存器保存了各个exception level的异常向量表的基地址。该寄存器有三个，分别是VBAR\_EL1，VBAR\_EL2，VBAR\_EL3。

具体的exception handler是通过vector base address ＋ offset得到，offset的定义如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| exception level迁移情况 | Synchronous exception的offset值 | IRQ和vIRQ exception的offset值 | FIQ和vFIQ exception的offset值 | SError和vSError exception的offset值 |
| 同级exception level迁移，使用SP\_EL0。例如EL1迁移到EL1 | 0x000 | 0x080 | 0x100 | 0x180 |
| 同级exception level迁移，使用SP\_ELx。例如EL1迁移到EL1 | 0x200 | 0x280 | 0x300 | 0x380 |
| ELx迁移到ELy，其中y>x并且ELx处于AArch64状态 | 0x400 | 0x480 | 0x500 | 0x580 |
| ELx迁移到ELy，其中y>x并且ELx处于AArch32状态 | 0x600 | 0x680 | 0x700 | 0x780 |

## 五、代码分析

对于linux kernel而言，它可以有两种情况：一种是做Hypervisor，另外一种是做Guest OS。linux kernel一般不会做Trusted OS，一般Trusted OS都是size比较小，比较轻盈的OS。因此，对于linux kernel而言，我们只要设定两个异常向量表：一个是做为guest OS（或者普通OS）的EL1 EL0的异常向量表，另外一个是for Hypervisor的EL2的异常向量表。

1、EL1的异常向量表。当发生一个异常，并且处理器迁移到了EL1这个exception level，那么该异常由EL1的异常向量表来决定如果跳转到相应的exception handler。具体代码如下：

    .align    11－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（1）   
ENTRY(vectors)－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－（2）   
    ventry    el1\_sync\_invalid        // Synchronous EL1t－－－－－－－－－－－－－－（3）   
    ventry    el1\_irq\_invalid            // IRQ EL1t   
    ventry    el1\_fiq\_invalid            // FIQ EL1t   
    ventry    el1\_error\_invalid        // Error EL1t

    ventry    el1\_sync            // Synchronous EL1h －－－－－－－－－－－－－－－－（4）   
    ventry    el1\_irq                // IRQ EL1h   
    ventry    el1\_fiq\_invalid            // FIQ EL1h   
    ventry    el1\_error\_invalid        // Error EL1h

    ventry    el0\_sync            // Synchronous 64-bit EL0 －－－－－－－－－－－－－－（5）   
    ventry    el0\_irq                // IRQ 64-bit EL0   
    ventry    el0\_fiq\_invalid            // FIQ 64-bit EL0   
    ventry    el0\_error\_invalid        // Error 64-bit EL0    
    ventry    el0\_sync\_invalid        // Synchronous 32-bit EL0 －－－－－－－－－－－－（6）   
    ventry    el0\_irq\_invalid            // IRQ 32-bit EL0   
    ventry    el0\_fiq\_invalid            // FIQ 32-bit EL0   
    ventry    el0\_error\_invalid        // Error 32-bit EL0   
END(vectors)

（1）EL1的异常向量表保存在VBAR\_EL1寄存器中（Vector Base Address Register (EL1)），该寄存器的低11bit是reserve的，11～63表示了Vector Base Address，因此这里的异常向量表是2K对齐的。

（2）exception vector table中有很多entry（否则也不会叫做vector table了），发生了异常具体选择哪一个entry需要考虑下面的这些因素：该exception从何而来？（对于本场景，exception只能来自EL0或者EL1），使用哪一个stack pointer？（SP\_EL0还是SP\_EL1），哪一种类型的异常？（synchronous exception、IRQ、FIQ还是SError interrupt）。

（3）异常向量表是分组的，每一组都包括四种类型的exception，分别对应synchronous exception（elx\_sync or elx\_sync\_invalid），irq中断（elx\_irq or elx\_irq\_invalid），fiq中断（elx\_fiq or elx\_fiq\_invalid）以及SError中断（elx\_error or elx\_error\_invalid）。这一组异常对应异常状态的迁移是EL1到EL1的迁移，并且选择使用了SP\_EL0。对于linux kernel而言，这类exception vector实际上就是Invalid mode handlers。

（4）如果用大家熟悉的语言，其实这一段exception vectors可以这样表述：这些是异常发生在内核态（EL1）并且系统配置为内核处理这些异常（这些异常导致PE迁移到EL1）时候的异常向量。目前版本的ARM64代码还没有fiq和SError这两种异常的支持。fiq在ARM ARM文档中建议是在secure state世界中处理，因此没有出现在linux中是合理的，但是为何SError为何不处理呢？估计是因为SError是一种异步异常，硬件没有提供足够的信息恢复，因此linux只能是按照invalid mode来处理。

（5）异常发生在了用户态（EL0）并且该异常需要在内核态（EL1）中处理。

（6）同（5）不过发生异常的现场处于AArch32状态。

2、EL2的异常向量表。在本文的场景下（不支持虚拟化），实际上EL2的异常向量表没有存在的意义，如果硬件支持EL2，这时候，linux kernel可以做为hypervisor存在。启动的时候，在el2\_setup函数中会设定一个dummy的EL2异常向量表（\_\_hyp\_stub\_vectors）。当然，真正的EL2的异常向量表会在kvm初始化的时候完成设定。

## 六、参考文献

1、ARM ARM

2、Computer Architecture, A Quantitative Approach, 5th

3、DEN0024A\_v8\_architecture\_PG.pdf