**Linux内核启动流程 三：内核初始化阶段—>BusyBox初始化阶段**

　　arch/arm/kernel/head-armv.S

　　该文件是内核最先执行的一个文件，包括内核入口ENTRY(stext)到start\_kernel间的初始化代码，

**主要作用**是检查CPU ID， Architecture Type，初始化BSS等操作，并跳到start\_kernel函数。在执行前，处理器应满足以下状态：

r0 - should be 0   
r1 - unique architecture number   
MMU - off   
I-cache - on or off   
D-cache – off

1 /\* 部分源代码分析 \*/

2 /\* 内核入口点 \*/

3 ENTRY(stext)

4 /\* 程序状态，禁止FIQ、IRQ，设定SVC模式 \*/

5 mov r0, #F\_BIT | I\_BIT | MODE\_SVC@ make sure svc mode

6 /\* 置当前程序状态寄存器 \*/

7 msr cpsr\_c, r0 @ and all irqs disabled

8 /\* 判断CPU类型，查找运行的CPU ID值与Linux编译支持的ID值是否支持 \*/

9 bl \_\_lookup\_processor\_type

10 /\* 跳到\_\_error \*/

11 teq r10, #0 @ invalid processor?

12 moveq r0, #'p' @ yes, error 'p'

13 beq \_\_error

14 /\* 判断体系类型，查看R1寄存器的Architecture Type值是否支持 \*/

15 bl \_\_lookup\_architecture\_type

16 /\* 不支持，跳到出错 \*/

17 teq r7, #0 @ invalid architecture?

18 moveq r0, #'a' @ yes, error 'a'

19 beq \_\_error

20 /\* 创建核心页表 \*/

21 bl \_\_create\_page\_tables

22 adr lr, \_\_ret @ return address

23 add pc, r10, #12 @ initialise processor

24 /\* 跳转到start\_kernel函数 \*/

25 b start\_kernel

**1. start\_kernel()函数分析**

　　下面对**start\_kernel()**函数及其相关函数进行分析。

**1.1 lock\_kernel()**

1 /\* Getting the big kernel lock.

2 \* This cannot happen asynchronously,

3 \* so we only need to worry about other

4 \* CPU's.

5 \*/

6 extern \_\_inline\_\_ void lock\_kernel(void)

7 {

8 if (!++current->lock\_depth)

9 spin\_lock(&kernel\_flag);

10 }

**kernel\_flag** 是一个内核大自旋锁，所有进程都通过这个大锁来实现向内核态的迁移。

　　只有获得这个大自旋锁的处理器可以进入内核，如中断处理程序等。在任何一对 lock\_kernel／unlock\_kernel函数里至多可以有一个程序占用CPU。

　　进程的lock\_depth成员初始化为-1，在 kerenl/fork.c文件中设置。在它小于0时 （恒为 -1），进程不拥有内核锁；当大于或等于0时，进程得到内核锁。

**1.2 setup\_arch()**

**setup\_arch()**函数做体系相关的初始化工作，函数的定义在arch/arm/kernel/setup.c文件中，主要涉及下列主要函数及代码。

**setup\_processor()**  
　　该函数主要通过

for (list = &\_\_proc\_info\_begin; list < &\_\_proc\_info\_end ; list++)

if ((processor\_id & list->cpu\_mask) == list->cpu\_val)

break;

　　这样一个循环来在.proc.info段中寻找匹配的processor\_id，processor\_id在head\_armv.S文件中设置。

**1.2.2 setup\_architecture(machine\_arch\_type)**

　　该函数获得体系结构的信息，返回mach-xxx/arch.c 文件中定义的machine结构体的指针，包含以下内容

MACHINE\_START (xxx, “xxx”)

MAINTAINER ("xxx"

BOOT\_MEM (xxx, xxx, xxx)

FIXUP (xxx)

MAPIO (xxx)

INITIRQ (xxx)

MACHINE\_END

**1.2.3内存设置代码**

if (meminfo.nr\_banks == 0)

{

meminfo.nr\_banks = 1;

meminfo.bank[0].start = PHYS\_OFFSET;

meminfo.bank[0].size = MEM\_SIZE;

}

　　meminfo结构表明内存情况，是对物理内存结构meminfo的默认初始化。

　　nr\_banks指定内存块的数量，bank指定每块内存的范围，PHYS \_OFFSET指定某块内存块的开始地址，MEM\_SIZE指定某块内存块长度。 PHYS \_OFFSET和MEM\_SIZE都定义在include/asm-armnommu/arch-XXX/memory.h文件中，其中 PHYS \_OFFSET是内存的开始地址，MEM\_SIZE就是内存的结束地址。

　　这个结构在接下来内存的初始化代码中起重要作用。

**1.2.4 内核内存空间管理**

init\_mm.start\_code = (unsigned long) &\_text; 内核代码段开始

init\_mm.end\_code = (unsigned long) &\_etext; 内核代码段结束

init\_mm.end\_data = (unsigned long) &\_edata; 内核数据段开始

init\_mm.brk = (unsigned long) &\_end; 内核数据段结束

　　每一个任务都有一个**mm\_struct**结构管理其内存空间，init\_mm 是内核的mm\_struct。

　　其中设置成员变量\* mmap指向自己， 意味着内核只有一个内存管理结构，设置 pgd=swapper\_pg\_dir，swapper\_pg\_dir是内核的页目录，ARM体系结构的内核页目录大小定义为16k。init\_mm定义了整个内核的内存空间，内核线程属于内核代码，同样使用内核空间，其访问内存空间的权限与内核一样。

**1.2.5 内存结构初始化**

　　bootmem\_init (&meminfo)函数根据meminfo进行内存结构初始化。

　　bootmem\_init(&meminfo)函数中调用reserve\_node\_zero(bootmap\_pfn, bootmap\_pages) 函数，这个函数的作用是保留一部分内存使之不能被动态分配。

　　这些内存块包括：

reserve\_bootmem\_node(pgdat, \_\_pa(&\_stext), &\_end - &\_stext); /\*内核所占用地址空间\*/

reserve\_bootmem\_node(pgdat, bootmap\_pfn<<PAGE\_SHIFT, bootmap\_pages<<PAGE\_SHIFT) /\*bootmem结构所占用地址空间\*/

**1.2.6 paging\_init(&meminfo, mdesc)**

　　创建内核页表，映射所有物理内存和IO空间，对于不同的处理器，该函数差别比较大。

　　下面简单描述一下**ARM体系结构的存储系统及MMU相关的概念。**   
　　在ARM存储系统中，使用内存管理单元(MMU)实现虚拟地址到实际物理地址的映射。

　　利用MMU，可把SDRAM的地址完全映射到0x0起始的一片连续地址空间，而把原来占据这片空间的FLASH或者ROM映射到其他不相冲突的存储空间位置。

　　例如，FLASH的地址从0x0000 0000～0x00FFFFFF，而SDRAM的地址范围是0x3000 0000～0x3lFFFFFF，则可把SDRAM地址映射为0x0000 0000～0xlFFFFFF，而FLASH的地址可以映射到0x9000 0000～0x90FFFFFF(此处地址空间为空闲，未被占用)。映射完成后，如果处理器发生异常，假设依然为IRQ中断，PC指针指向0xl8处的地址，而这个时候PC实际上是从位于物理地址的0x3000 0018处读取指令。

　　通过MMU的映射，则可实现程序完全运行在SDRAM之中。在实际的应用中．可能会把两片不连续的物理地址空间分配给SDRAM。而在操作系统中，习惯于把SDRAM的空间连续起来，方便内存管理，且应用程序申请大块的内存时，操作系统内核也可方便地分配。通过MMU可实现不连续的物理地址空间映射为连续的虚拟地址空间。操作系统内核或者一些比较关键的代码，一般是不希望被用户应用程序访问。通过MMU可以控制地址空间的访问权限，从而保护这些代码不被破坏。

**MMU的实现过程，实际上就是一个查表映射的过程**。建立页表是实现MMU功能不可缺少的一步。页表位于系统的内存中，页表的每一项对应于一个虚拟地址到物理地址的映射。每一项的长度即是一个字的长度(在ARM中，一个字的长度被定义为4Bytes)。页表项除完成虚拟地址到物理地址的映射功能之外，还定义了访问权限和缓冲特性等。   
　　MMU的映射分为两种，一级页表的变换和二级页表变换。两者的不同之处就是实现的变换地址空间大小不同。   
　　一级页表变换支持1 M大小的存储空间的映射，而二级可以支持64 kB，4 kB和1 kB大小地址空间的映射。

动态表(页表)的大小＝表项数＊每个表项所需的位数，即为整个内存空间建立索引表时，需要多大空间存放索引表本身。   
表项数＝虚拟地址空间/每页大小   
每个表项所需的位数＝Log(实际页表数)+适当控制位数   
实际页表数 ＝物理地址空间/每页大小

**1.3 parse\_options()**

　　分析由内核引导程序发送给内核的启动选项，在初始化过程中按照某些选项运行，并将剩余部分传送给init进程。

　　这些选项可能已经存储在配置文件中，也可能是由用户在系统启动时敲入的。但内核并不关心这些，这些细节都是内核引导程序关注的内容，嵌入式系统更是如此。

**1.4 trap\_init() （/kernel/traps.c do\_trap）**

　　这个函数用来做体系相关的**中断处理**的初始化，在该函数中调用\_\_trap\_init((void \*)vectors\_base())

　　函数将exception vector设置到vectors\_base开始的地址上。 \_\_trap\_init函数位于entry-armv.S文件中，对于ARM处理器，共有复位、未定义指令、SWI、预取终止、数据终止、IRQ和FIQ 几种方式。

**SWI主要用来实现系统调用**，而产生了IRQ之后，通过exception vector进入中断处理过程，执行**do\_IRQ**函数。

　　armnommu的trap\_init（）函数在arch/armnommu/kernel/traps.c文件中。

　　vectors\_base是写中断向量的开始地址，在include/asm-armnommu/proc-armv/system.h文件中设置，地址为0或0XFFFF0000。

ENTRY(\_\_trap\_init)

stmfd sp!, {r4 - r6, lr}

mrs r1, cpsr @ code from 2.0.38

bic r1, r1, #MODE\_MASK @ clear mode bits /\* 设置svc模式，disable IRQ,FIQ \*/

orr r1, r1, #I\_BIT|F\_BIT|MODE\_SVC @ set SVC mode, disable IRQ,FIQ

msr cpsr, r1

adr r1, .LCvectors @ set up the vectors

ldmia r1, {r1, r2, r3, r4, r5, r6, ip, lr}

stmia r0, {r1, r2, r3, r4, r5, r6, ip, lr} /\* 拷贝异常向量 \*/

add r2, r0, #0x200

adr r0, \_\_stubs\_start @ copy stubs to 0x200

adr r1, \_\_stubs\_end

1: ldr r3, [r0], #4

str r3, [r2], #4

cmp r0, r1

blt 1b

LOADREGS(fd, sp!, {r4 - r6, pc})

　　\_\_stubs\_start到\_\_stubs\_end的地址中包含了异常处理的代码，因此拷贝到vectors\_base+0x200的位置上。

**1.5 init\_IRQ()**

1 void \_\_init init\_IRQ(void)

2 {

3 extern void init\_dma(void);

4 int irq;

5

6 for (irq = 0; irq < NR\_IRQS; irq++) {

7 irq\_desc[irq].probe\_ok = 0;

8 irq\_desc[irq].valid = 0;

9 irq\_desc[irq].noautoenable = 0;

10 irq\_desc[irq].mask\_ack = dummy\_mask\_unmask\_irq;

11 irq\_desc[irq].mask = dummy\_mask\_unmask\_irq;

12 irq\_desc[irq].unmask = dummy\_mask\_unmask\_irq;

13 }

14 CSR\_WRITE(AIC\_MDCR, 0x7FFFE); /\* disable all interrupts \*/

15 CSR\_WRITE(CAHCNF,0x0);/\*Close Cache\*/

16 CSR\_WRITE(CAHCON,0x87);/\*Flush Cache\*/

17 while(CSR\_READ(CAHCON)!=0);

18 CSR\_WRITE(CAHCNF,0x7);/\*Open Cache\*/

19

20 init\_arch\_irq();

21 init\_dma();

22 }

　　这个函数用来做体系相关的irq处理的初始化.

**irq\_desc**数组是用来描述IRQ的请求队列，**每一个中断号分配一个irq\_desc结构，组成了一个数组。**

　　NR\_IRQS代表中断数目，这里只是对中断结构irq\_desc进行了初始化。

　　在默认的初始化完成后调用初始化函数init\_arch\_irq，先执行arch/armnommu/kernel/irq-arch.c文件中的函数genarch\_init\_irq()，然后就执行 include/asm-armnommu/arch-xxxx/irq.h中的inline函数**irq\_init\_irq**，在这里对irq\_desc进行了实质的初始化。

　　其中mask用阻塞中断；unmask用来取消阻塞；mask\_ack的作用是阻塞中断，同时还回应ack给硬件表示这个中断已经被处理了，否则硬件将再次发生同一个中断。这里，不是所有硬件需要这个ack回应，所以很多时候mask\_ack与mask用的是同一个函数。

　　接下来执行init\_dma（）函数，如果不支持DMA，可以设置include/asm-armnommu/arch-xxxx/dma.h中的 MAX\_DMA\_CHANNELS为0，这样在arch/armnommu/kernel/dma.c文件中会根据这个定义使用不同的函数。

**1.6 sched\_init()**

　　初始化系统调度进程，主要对定时器机制和时钟中断的Bottom Half的初始化函数进行设置。

　　与时间相关的初始化过程主要有两步：

　　（1）调用 init\_timervecs()函数初始化内核定时器机制；

　　（2）调用init\_bh()函数将BH向量TIMER\_BH、TQUEUE\_BH和 IMMEDIATE\_BH所对应的BH函数分别设置成timer\_bh()、tqueue\_bh()和immediate\_bh()函数

**1.7 softirq\_init()**

　　内核的软中断机制初始化函数。

　　调用tasklet\_init初始化tasklet\_struct结构，软中断的个数为32个。用于bh的 tasklet\_struct结构调用tasklet\_init()以后，它们的函数指针func全都指向bh\_action()。

　　bh\_action就是tasklet实现bh机制的代码，但此时具体的bh函数还没有指定。

　　HI\_SOFTIRQ用于实现bottom half，TASKLET\_SOFTIRQ用于公共的tasklet。

open\_softirq(TASKLET\_SOFTIRQ, tasklet\_action, NULL); /\* 初始化公共的tasklet\_struct要用到的软中断 \*/

open\_softirq(HI\_SOFTIRQ, tasklet\_hi\_action, NULL); /\* 初始化tasklet\_struct实现的bottom half调用 \*/

**1.8 time\_init()**

　　这个函数用来做体系相关的timer的初始化，armnommu的在arch/armnommu/kernel/time.c。

　　这里调用了在 include/asm-armnommu/arch-xxxx/time.h中的inline函数setup\_timer。

**setup\_timer（）**函数的设计与硬件设计紧密相关，主要是根据硬件设计情况设置时钟中断号和时钟频率等。

1 void \_\_inline\_\_ setup\_timer (void)

2 {

3 /\*----- disable timer -----\*/

4 CSR\_WRITE(TCR0, xxx);

5

6 CSR\_WRITE (AIC\_SCR7, xxx); /\* setting priority level to high \*/

7 /\* timer 0: 100 ticks/sec \*/

8 CSR\_WRITE(TICR0, xxx);

9

10 timer\_irq.handler = xxxxxx\_timer\_interrupt;

11 setup\_arm\_irq(IRQ\_TIMER, &timer\_irq); /\* IRQ\_TIMER is the interrupt number \*/

12

13 INT\_ENABLE(IRQ\_TIMER);

14 /\* Clear interrupt flag \*/

15 CSR\_WRITE(TISR, xxx);

16

17 /\* enable timer \*/

18 CSR\_WRITE(TCR0, xxx);

19 }

**1.9 console\_init()**

　　控制台初始化。控制台也是一种驱动程序，由于其特殊性，提前到该处完成初始化，主要是为了提前看到输出信息，据此判断内核运行情况。

　　很多嵌入式Linux操作系统由于没有在/dev目录下正确配置console设备，造成启动时发生诸如unable to open an initial console的错误。

**1.10 init\_modules()**

　　模块初始化。如果编译内核时使能该选项，则内核支持模块化加载/卸载功能

**1.11 kmem\_cache\_init()**

　　内核Cache初始化

**1.12 sti()**

**使能中断，这里开始，中断系统开始正常工作。**

**1.13 calibrate\_delay()**

　　近似计算BogoMIPS数字的内核函数。作为第一次估算，calibrate\_delay计算出在每一秒内执行多少次\_\_delay循环，也就是每个定时器滴答（timer tick）―百分之一秒内延时循环可以执行多少次。这种计算只是一种估算，结果并不能精确到纳秒，但这个数字供内核使用已经足够精确了。

　　BogoMIPS的数字由内核计算并在系统初始化的时候打印。它近似的给出了每秒钟CPU可以执行一个短延迟循环的次数。在内核中，这个结果主要用于需要等待非常短周期的设备驱动程序――例如，等待几微秒并查看设备的某些信息是否已经可用。

　　计算一个定时器滴答内可以执行多少次循环需要在滴答开始时就开始计数，或者应该尽可能与它接近。全局变量jiffies中存储了从内核开始保持跟踪时间开始到现在已经经过的定时器滴答数， jiffies保持异步更新，在一个中断内——每秒一百次，内核暂时挂起正在处理的内容，更新变量，然后继续刚才的工作。

**1.15 kmem\_cache\_sizes\_init()**

　　内核内存管理器的初始化，也就是初始化cache和SLAB分配机制。

**1.16 pgtable\_cache\_init()**

　　页表cache初始化。

**1.17 fork\_init()**

　　这里根据硬件的内存情况，如果计算出的max\_threads数量太大，可以自行定义。

**1.18 proc\_caches\_init();**

　　为proc文件系统创建高速缓冲

**1.19 vfs\_caches\_init(num\_physpages);**

　　为VFS创建SLAB高速缓冲

**1.20 buffer\_init(num\_physpages);**

　　初始化buffer

**1.21 page\_cache\_init(num\_physpages);**

　　页缓冲初始化

**1.22 signals\_init();**

　　创建信号队列高速缓冲

**1.23 proc\_root\_init();**

**在内存中创建包括根结点在内的所有节点**

**1.24 check\_bugs();**

　　检查与处理器相关的bug

**1.25 smp\_init();**

**1.26 rest\_init();**

　　此函数调用**kernel\_thread**(init, NULL, CLONE\_FS | CLONE\_FILES | CLONE\_SIGNAL)函数。

**1.26.1 kernel\_thread()函数分析**

　　这里调用了arch/armnommu/kernel/process.c中的函数kernel\_thread，kernel\_thread函数中通过**\_\_syscall(clone)**创建新线程。

　　\_\_syscall(clone)函数参见armnommu/kernel目录下的entry- common.S文件。

**1.26.2 init()完成下列功能：**

**init()**函数通过kernel\_thread(init, NULL, CLONE\_FS | CLONE\_FILES | CLONE\_SIGNAL)的回调函数执行，完成下列功能。

**do\_basic\_setup()**在该函数里，sock\_init()函数进行网络相关的初始化，占用相当多的内存，如果所开发系统不支持网络功能，可以把该函数的执行注释掉。

**do\_initcalls()**实现驱动的初始化, 这里需要与**vmlinux.lds**联系起来看才能明白其中奥妙。

static void \_\_init do\_initcalls(void)

{

　　initcall\_t \*call;

　　call = &\_\_initcall\_start;

　　do {

　　　(\*call)();

　　　call++;

　　} while (call < &\_\_initcall\_end);

　　/\* Make sure there is no pending stuff from the initcall sequence \*/

　　flush\_scheduled\_tasks();

}

　　查看 /arch/i386/vmlinux.lds，其中有一段代码

　\_\_initcall\_start = .;

　.initcall.init : { \*(.initcall.init) }

　\_\_initcall\_end = .;

　　其含义是\_\_initcall\_start指向代码节.initcall.init的节首，而\_\_initcall\_end指向.initcall.init的节尾。

**do\_initcalls**所作的是系统中有关驱动部分的初始化工作，那么**这些函数指针数据是怎样放到了.initcall.init节呢？**

在include/linux/init.h文件中有如下3个定义：

1. #define \_\_init\_call　　 \_\_attribute\_\_ ((unused,\_\_section\_\_ (".initcall.init" ))

\_\_attribute\_\_的含义就是构建一个在.initcall.init节的指向初始函数的指针。

2. #define \_\_initcall(fn) static initcall\_t \_\_initcall\_##fn \_\_init\_call = fn   
##意思就是在可变参数使用宏定义的时候构建一个变量名称为所指向的函数的名称，并且在前面加上\_\_initcall\_   
3. #define **module\_init**(x) \_\_initcall(x);   
**很多驱动中都有类似module\_init(usb\_init)的代码，通过该宏定义逐层解释存放到.initcall.int节中。**



**1.26.3 init执行过程**

　　在内核引导结束并启动init之后，系统就转入用户态的运行，在这之后创建的一切进程，都是在用户态进行。

　　这里先要清楚一个**概念**：

　　就是init进程虽然是从内核开始的，即在前面所讲的init/main.c中的init()函数在启动后就已经是一个核心线程，但在转到执行init程序（如 /sbin/init）之后，内核中的init()就变成了/sbin/init程序，状态也转变成了用户态，也就是说**核心线程变成了一个普通的进程。**

　　这样一来，内核中的init函数实际上只是用户态init进程的入口，它在执行execve("/sbin/init",argv\_init, envp\_init)时改变成为一个普通的用户进程。这也就是exec函数的乾坤大挪移法，在exec函数调用其他程序时，当前进程被其他进程“灵魂附体”。

　　除此之外，它们的代码来源也有差别，内核中的init()函数的源代码在/init/main.c中，是内核的一部分。而/sbin/init程序的源代码是应用程序。

　　init程序启动之后，要完成以下任务：检查文件系统，启动各种后台服务进程，最后为每个终端和虚拟控制台启动一个getty进程供用户登录。由于所有其它用户进程都是由init派生的，因此它又是其它一切用户进程的父进程。

　　init进程启动后，按照/etc/inittab的内容进程系统设置。很多嵌入式系统用的是BusyBox的init，它与一般所使用的init不一样，会先执行/etc/init.d/rcS而非/etc/rc.d/rc.sysinit。

