

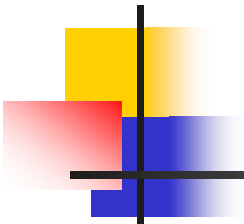
嵌入式Linux学习七步曲

Sailor_forever(扬帆)

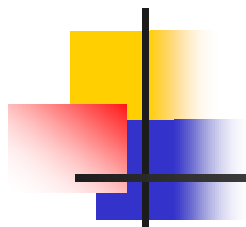
自由传播 版权所有 翻版必究



八一卦-我是who

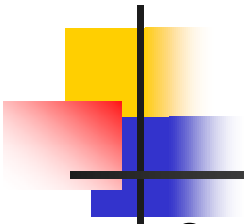
- 
-
- n 目前就职于通信行业某外企研发中心
 - n 参与校园招聘和社会招聘的技术面试工作
 - n 5年嵌入式软件开发经验，擅长嵌入式Linux开发；
 - n 接触的软硬件平台包括ARM，DSP，PowerPC，uC/OS-II，Linux，VxWorks及OSE

八一卦-我是who



- n 嵌入式Linux七步曲 学习群 交流讨论 资源共享
- n 群号 107900817
- n 7steps2linux@gmail.com
- n http://blog.csdn.net/sailor_8318

嵌入式水平小调查



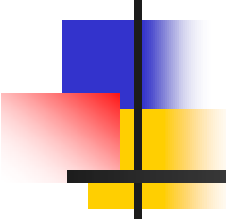
n 0—3个月

n 3—6 个月

n 1年左右

n 2年以上

n 多少人参加过系列交流会？

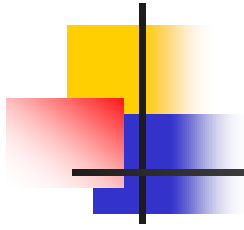


嵌入式Linux学习七步曲

- 
- 1 Linux主机开发环境
 - 2 嵌入式Linux交叉开发环境
 - 3 Linux系统bootloader移植
 - 4 **Linux的内核移植**
 - 5 Linux的内核及驱动编程
 - 6 文件系统制作
 - 7 Linux的高级应用编程

Grammy for Linux





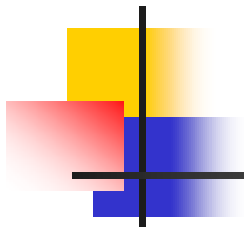
STAR Volunteer

- n Linux系统bootloader移植
 - n Hobby, U-boot如何启动Linux内核
 - n Sailing, U-boot在ARM平台下的启动流程



STAR Volunteer





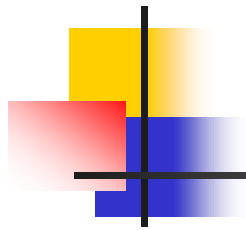
Volunteer Task

n 宗旨

- n 鼓励大家实际的参与嵌入式Linux的开发
- n 自己解决动手解决问题
- n 总结记录、分享
- n 形成知识库
- n 采用统一的模板，争取成为系列交流会的特色项目
- n 扩大BUPT BES的影响力，创造品牌

n 运作

- n 下次交流会之前完成上次的总结文档
- n Share给大家，提建议意见
- n 每次交流会颁奖鼓励
- n 最终将评出 STAR Volunteer



Volunteer Task

n Logo

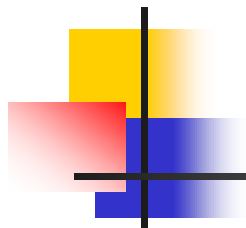
《嵌入式 Linux 学习七步曲》
BUPT BES 系列交流会 Volunteer Task



Key To Success

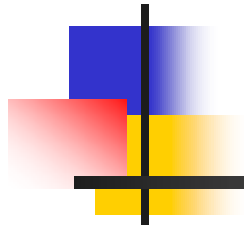
- n Google、Baidu
- n 理论 + 实践（开发板）
- n 勤于思考，善于总结
- n 多上相关技术论坛，他山之石可以攻玉
- n 良好的文档撰写习惯
- n Passion！





CHAPTER

4 Linux的**内核移植**



主要内容



1

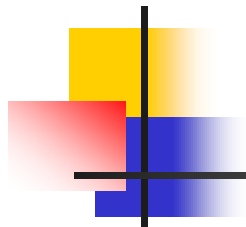
Linux内核的配置编译

2

Linux启动流程

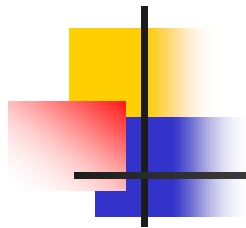
3

Linux内核移植



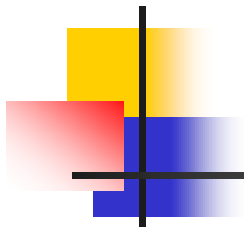
Linux内核的配置编译

- n 配置工具
- n Makefile
- n 链接脚本
- n 编译链接过程



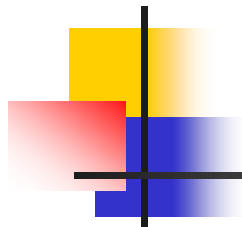
配置工具

- n make xconfig
- n make config
- n make menuconfig
 - n Cpu及板卡选择(优先配置，部分选项依赖于此)
 - n 串口
 - n 网口
 - n 文件系统支持
 - n 网络支持
 - n 调试支持



Makefile

- n 层次化，级级调用
- n 根目录下面的makefile设置基础配置
- n 编译器类型
 - n CROSS_COMPILE
- n CPU类型
 - n ARCH
- n 编译链接参数等
 - n MAKEFLAGS，目录打印
 - n KBUILD_VERBOSE，make V=1，编译选项及文件打印
 - n CONFIG_DEBUG_INFO，gdb符号选项
 - n CONFIG_FRAME_POINTER，防止栈形优化



压缩内核的Makefile

n /arch/arm/boot/Makefile

```
ifneq ($(MACHINE),)
include $(src tree)/$(MACHINE)/Makefile.boot
endif
```

此文件定义了链接基地址对应的物理地址和相应的内核参数物理地址

Note: the following conditions must always be true:

ZRELADDR == virt_to_phys(PAGE_OFFSET + TEXT_OFFSET)

PARAMS_PHYS must be within 4MB of ZRELADDR

INITRD_PHYS must be in RAM

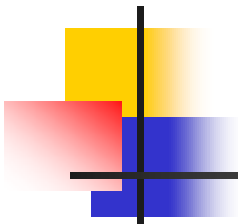
ZRELADDR := \$(zreladdr-y)

PARAMS_PHYS := \$(params_phys-y)

INITRD_PHYS := \$(initrd_phys-y)

export ZRELADDR INITRD_PHYS PARAMS_PHYS

ZRELADDR 和 vm linux 链接的虚拟地址必须是线性映射的，因为内核空间和物理地址空间是线性映射的



压缩内核的Makefile

n /arch/arm/boot/Makefile

n ulmage>zImage>compressed/vmlinux>Image>vmlinux

```
$(obj)/Image: vmlinux FORCE  
    $(call if_changed,objcopy)  
    @echo ' Kernel: $@ is ready'
```

```
$(obj)/compressed/vmlinux: $(obj)/Image FORCE  
    $(Q)$(MAKE) $(build)=$(obj)/compressed $@
```

```
$(obj)/zImage: $(obj)/compressed/vmlinux FORCE  
    $(call if_changed,objcopy)  
    @echo ' Kernel: $@ is ready'
```

```
$(obj)/uImage: $(obj)/zImage FORCE  
    $(call if_changed,uimage)  
    @echo ' Image $@ is ready'
```



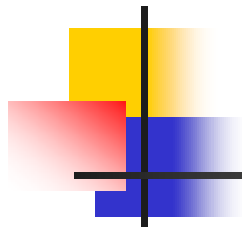
压缩内核的Makefile

n arch/arm/boot/compressed/Makefile

```
# We now have a PIC decompressor implementation. Decompressors running
# from RAM should not define ZTEXTADDR. Decompressors running directly
# from ROM or Flash must define ZTEXTADDR (preferably via the config)
# FIXME: Previous assignment to ztextaddr-y is lost here. See SHARK
ifeq ($(CONFIG_ZBOOT_ROM),y)
ZTEXTADDR:= $(CONFIG_ZBOOT_ROM_TEXT)
ZBSSADDR  := $(CONFIG_ZBOOT_ROM_BSS)
else
ZTEXTADDR:= 0
ZBSSADDR  := ALIGN(4)
endif

SEDFLAGS = s/TEXT_START/$(ZTEXTADDR)/;s/BSS_START/$(ZBSSADDR)/

targets    := vmlinux vmlinux.lds piggy.gz piggy.o font.o font.c \
              head.o misc.o $(OBJS)
EXTRA_CFLAGS := -fpic -fno-builtin
EXTRA_AFLAGS :=
```

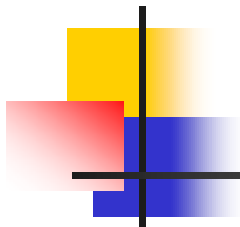


压缩内核的Makefile

n arch/arm/boot/compressed/Makefile

```
# Supply ZRELADDR, INITRD_PHYS and PARAMS_PHYS to the decompressor via
# linker symbols. We only define initrd_phys and params_phys if the
# machine class defined the corresponding makefile variable.
LDFLAGS_vmlinux := --defsym zreladdr=$(ZRELADDR)
ifneq ($(INITRD_PHYS),)
LDFLAGS_vmlinux += --defsym initrd_phys=$(INITRD_PHYS)
endif
ifneq ($(PARAMS_PHYS),)
LDFLAGS_vmlinux += --defsym params_phys=$(PARAMS_PHYS)
Endif
```

ZRELADDR 等宏定义由/arch/arm/boot/Makefile 定义的, zreladdr 为非压缩内核最终运行的地址或者压缩内核解压缩的目的地址



链接脚本

- n 指定程序链接时各个段落的分布
- n 指定程序链接的基地址
- n 非压缩内核自身的链接脚本
- n 压缩内核的链接脚本



非压缩内核链接脚本

n /arch/arm/kernel/vmlinux.lds

n 初始化入口为stext，对应的段为.text.head

n PAGE_OFFSET + TEXT_OFFSET为内核链接的虚拟地址(0xC000 0000 + 0x8000)

```
OUTPUT_ARCH(arm)
ENTRY(stext)

SECTIONS
{
#ifdef CONFIG_XIP_KERNEL
    . = XIP_VIRT_ADDR(CONFIG_XIP_PHYS_ADDR);
#else
    . = PAGE_OFFSET + TEXT_OFFSET;
#endif
    .text.head : {
        _stext = .;
        _sinittext = .;
        *(.text.head)
    }
    .
    .
    .
}
```



压缩内核链接脚本

- n `arch/arm/boot/compressed/vmlinux.lds`

- n 功能

- n 链接基地址为TEXT_START，由makefile指定，对于非XIP内核通常为0

- n 将程序分成.text, .got, .data, .bss, .stack(stack不占据映像大小)

- n 并设置了相关段的起始地址，便于进行代码重定位及内核解压缩

- n 程序入口为
`arch/arm/boot/compressed/head.S`



压缩内核链接脚本

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
  . = TEXT_START;
  _text = .;

  .text : {
    _start = .;
    *(.start)
    *(.text)
    *(.text.*)
    *(.fixup)
    *(.gnu.warning)
    *(.rodata)
    *(.rodata.*)
    *(.glue_7)
    *(.glue_7t)
    *(.piggydata)
    . = ALIGN(4);
  }
```




压缩内核链接脚本

```
etext = .;

got_start = .;
.got                : { *(.got) }
got_end = .;
.got.plt            : { *(.got.plt) }
.data               : { *(.data) }
edata = .;

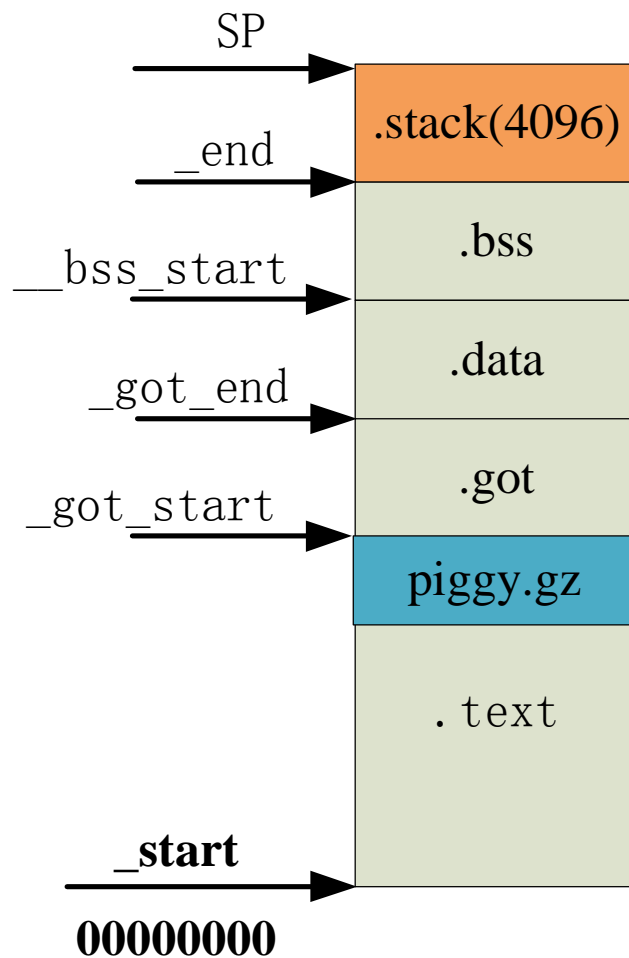
. = BSS_START;
bss_start = .;
.bss                : { *(.bss) }
end = .;

.stack (NOLOAD)    : { *(.stack) }

.stab 0              : { *(.stab) }
.stabstr 0           : { *(.stabstr) }
. . . .
.comment 0          : { *(.comment) }
}
```

压缩内核链接脚本

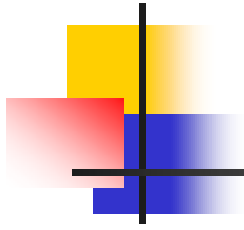
n 映像布局

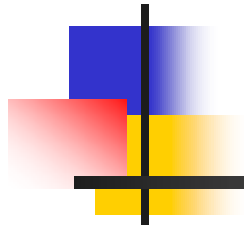




压缩内核的编译链接过程

- n 以0xc0008000为虚拟地址的非压缩内核自身的编译；
- n 对非压缩内核进行gzip压缩形成piggy.gz文件；
- n 将压缩过的内核印象piggy.gz转换成arch/arm/boot/compressed/piggy.o中的.piggydata数据段，并声明其数据域的起始地址；
- n 解压缩代码的编译，链接地址是TEXT_START，以-fpic方式链接，地址无关





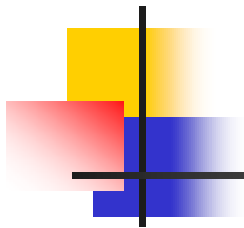
主要内容



① Linux内核的配置编译

② Linux启动流程

③ Linux内核移植



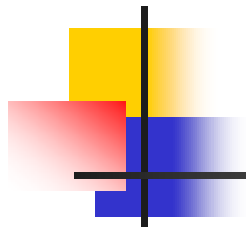
Linux启动流程

- n 压缩内核的启动
 - n 特点是位置无关，对内核启动地址要求低
 - n 内核压缩后比较小，便于存储传输
- n 非压缩内核的启动
 - n 位置相关，必须在0xFFFF 8000的启动地址（即RAM物理基地址0x8000偏移处）
 - n 压缩内核自解压时会自动解压缩到此位置
 - n 非压缩内核需要由bootloader保证此位置
 - n 两个主要阶段
 - n MMU开启前
 - n MMU开启后



压缩内核的启动

- n 相关代码
 - n arch/arm/boot/compressed/
- n 内核入口
 - n arch/arm/boot/compressed/head.S的start
- n 主要阶段
 - n 判断是否需要重定位
 - n 符号表的重定位
 - n 内核解压缩
 - n 非压缩内核的加载



判断是否需要重定位

n 判断标准

n 当前程序的运行地址 =? 压缩内核的链接地址

n 如何实现

n 采用相对寻址指令获得当前程序的运行地址

n 程序的某个标号的链接地址保存在数据段中

判断是否需要重定位

n 汇编代码

.

.text; 重新开始代码段的定义

adr r0, LC0 @LC0 本身是以 0 为基址的偏移, 而 adr 的基于 pc 寻址导致 r0 是相对于当前运行地址的

ldmia r0, {r1, r2, r3, r4, r5, r6, ip, sp} @

subs r0, r0, r1 @ calculate the delta offset

beq not_relocated @ if delta is zero, we are
@ running at the address we
@ were linked at.

.

.type LC0, #object

LC0: .word **LC0** @ r1

.word __bss_start @ r2

.word _end @ r3

.word **zreladdr** @ r4

.word _start @ r5

.word _got_start @ r6

.word _got_end @ ip

.word user_stack+4096 @ sp

判断是否需要重定位

n 反汇编代码

./arch/arm/boot/compressed/vmlinux: file format elf32-littlearm

Disassembly of section .text:

00000000 <start>:

//由符号表可知, 当前程序的链接地址是 0

◦ ◦ ◦ ◦ ◦ ◦ ◦

60:e28f00cc	add	r0, pc , #204 ; 0xcc
64: e890307e	ldmia	r0, {r1, r2, r3, r4, r5, r6, ip, sp}
68: e0500001	subs	r0, r0, r1
6c: 0a00000a	beq	9c <not_relocated>

◦ ◦ ◦

00000134 <LC0>:

134:	00000134	001362f0	0013e728	20008000	4....b..(.....
144:	00000000	00136284	001362e4	0013f728b...b..(...

00000134 = 60 + CC + 8



判断是否需要重定位

n 反汇编代码

- n 8为流水线预取导致的PC值为当前运行指令 + 8
- n `add r0, pc, #204`即获得了当前程序运行地址的CC + 8处的运行地址，存储在r0中
- n `ldmia r0, {r1, r2, r3, r4, r5, r6, ip, sp}`
- n 将r0地址处保存的相关数据顺序加载到r1, r2, r3, r4, r5, r6, ip, sp中
- n `subs r0, r0, r1`
- n r1为LC0的链接地址，r0为LC0的运行地址，求二者差值，得加载域和链接域的差值



符号表的重定位

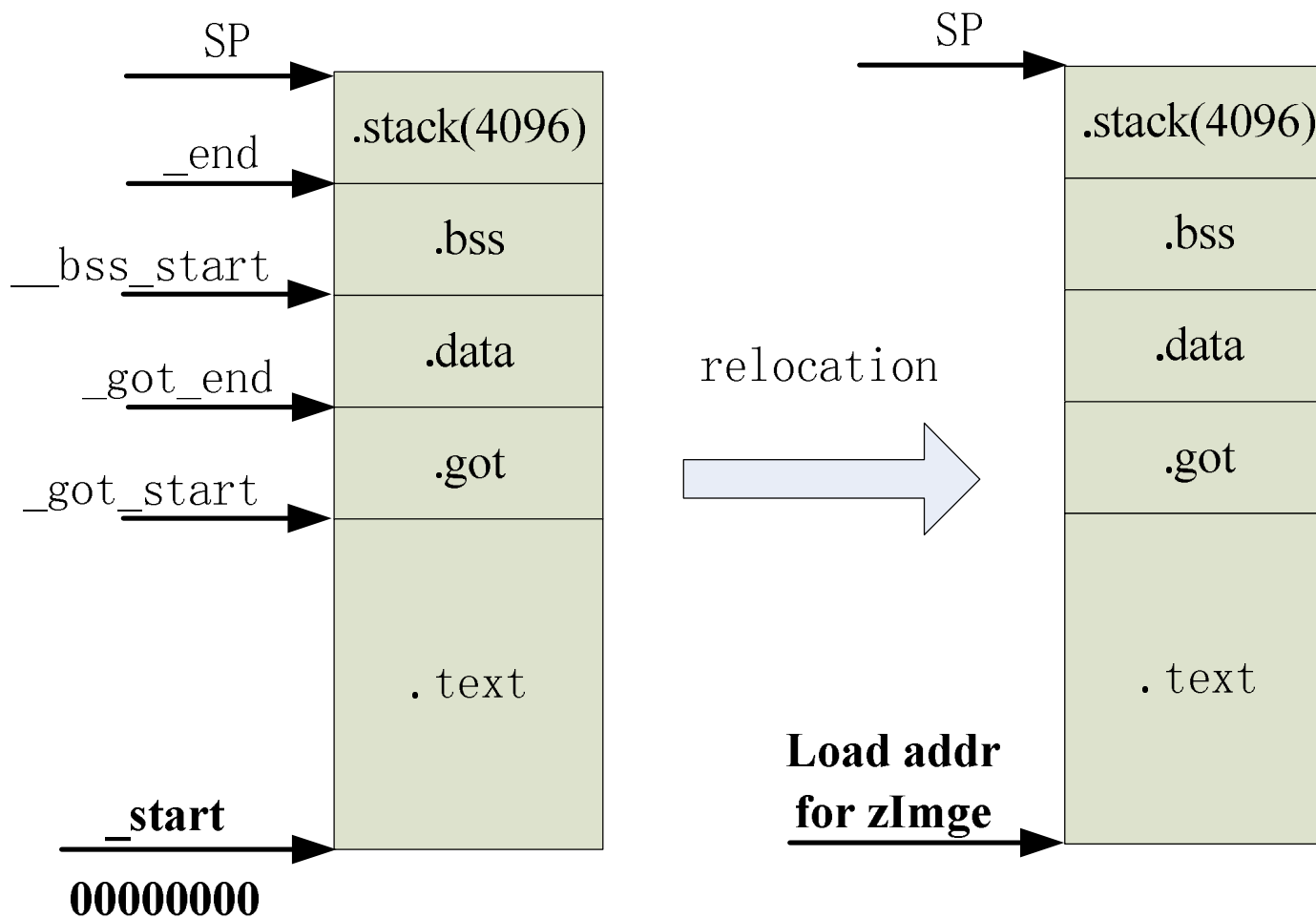
n 为什么要重定位

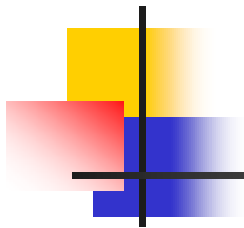
- n 当压缩内核运行在非链接地址上时，需要对全局符号表进行修正，以便C函数中能够正确访问全局变量
- n 内核解压缩时要获得内核印象所处的位置，相关地址需要修正

n 实现

- n 将链接地址加上重定位的偏移量，获得各个符号的加载地址

符号表的重定位





内核解压缩

n 关键因素

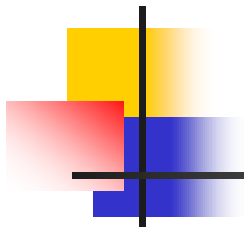
n 覆盖问题

n 解压缩方案

n 内核解压缩的目的地址大于当前程序运行空间的最高地址，直接解压缩

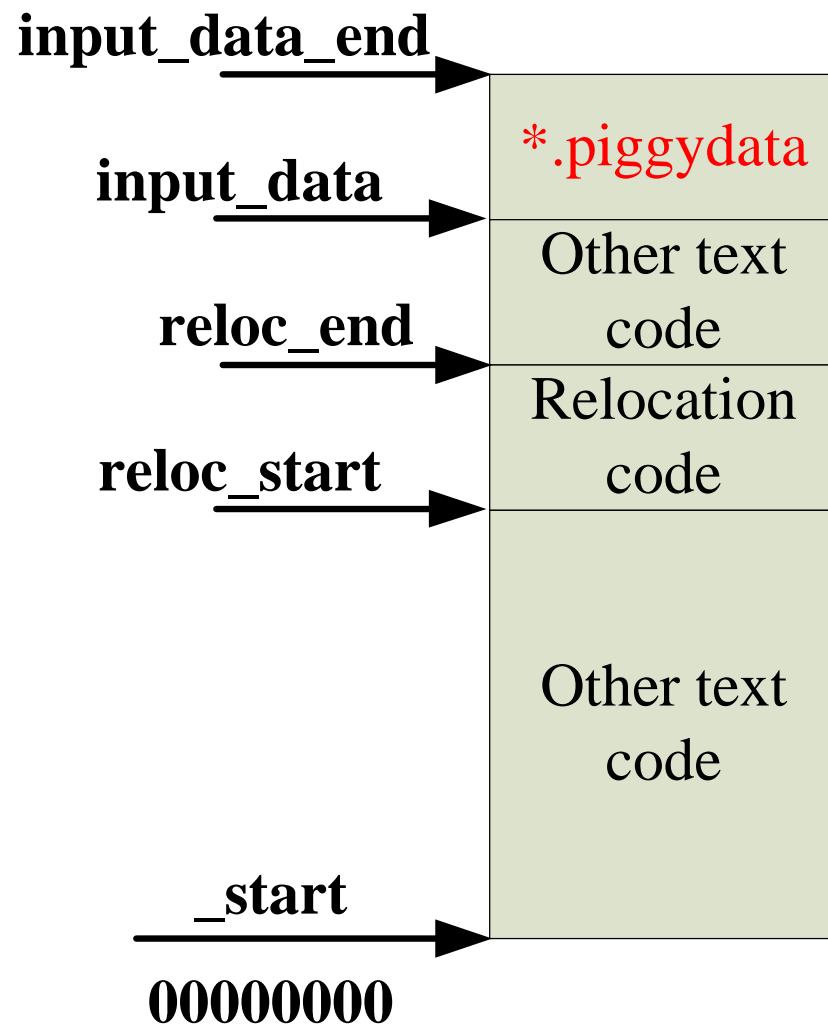
n 内核解压缩的目的地址离当前程序运行的起始地址距离大于内核印象长度，直接解压缩

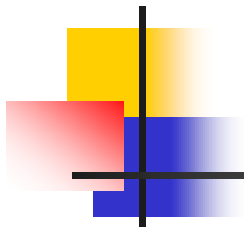
n 介于上述二者之间，需要间接解压缩



内核解压缩

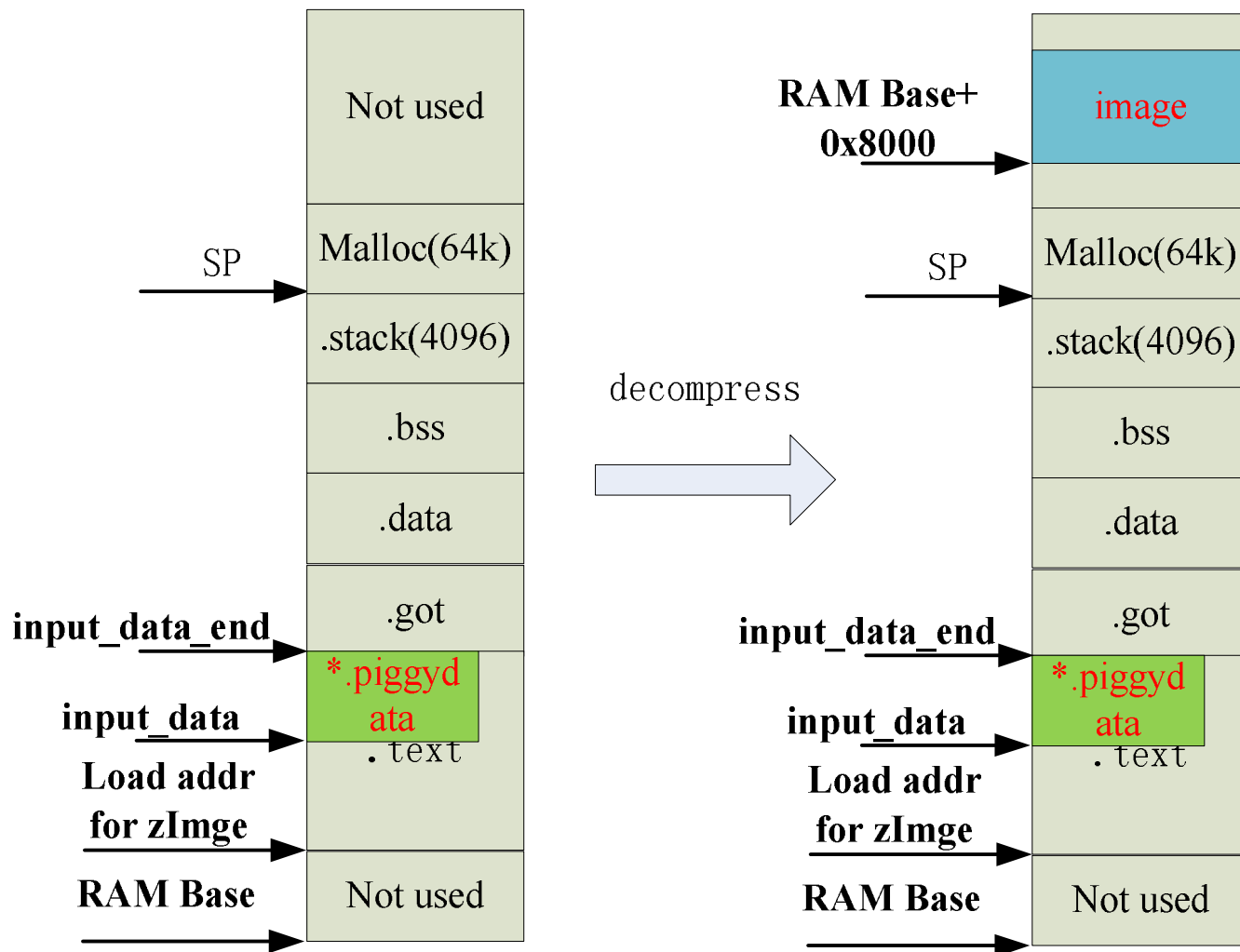
n 内核映像代码段组成

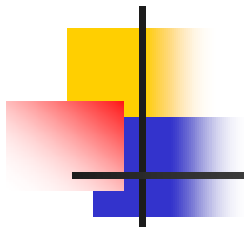




内核解压缩

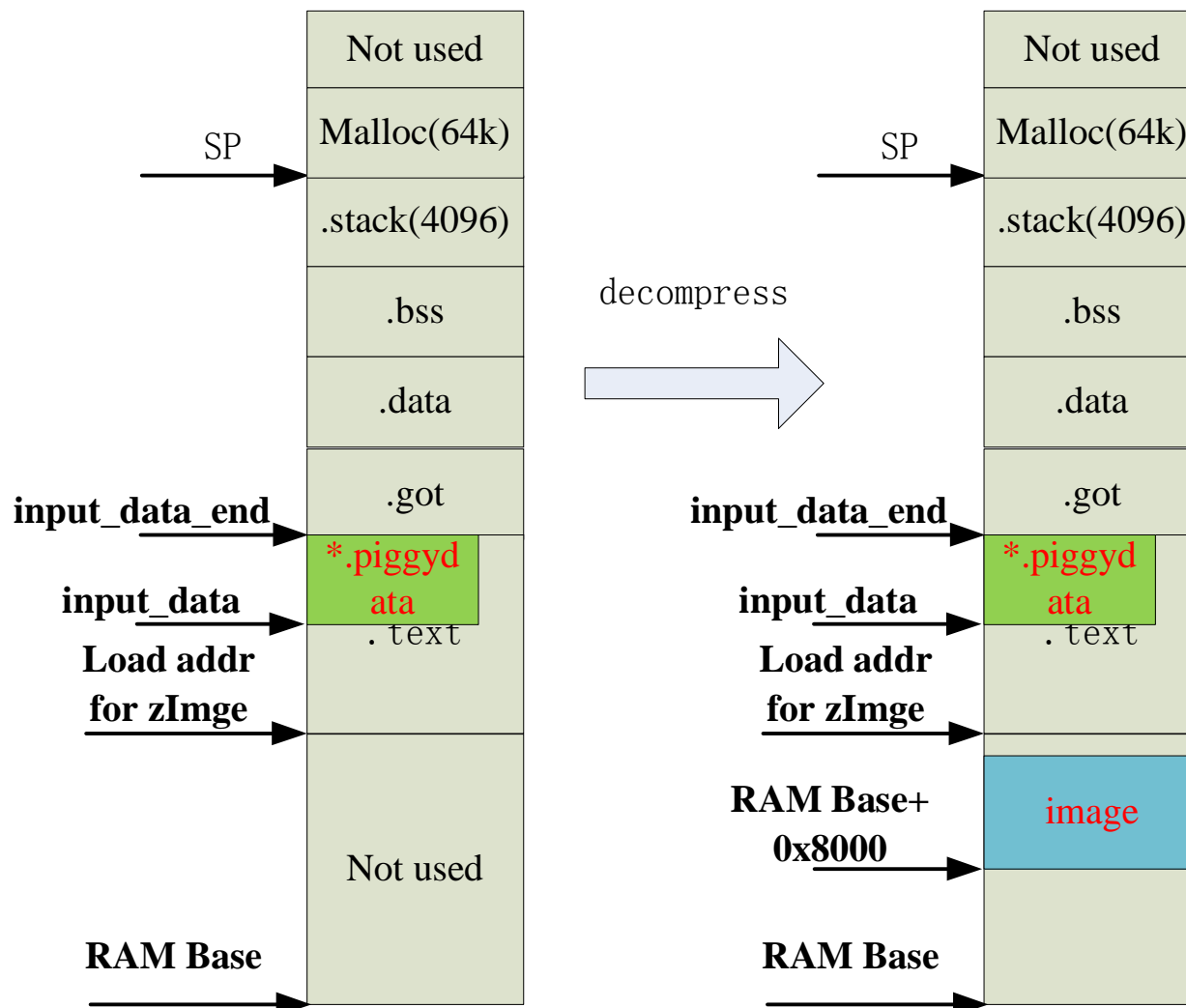
n 直接解压缩到高端地址

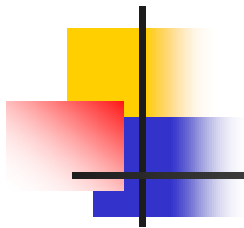




内核解压缩

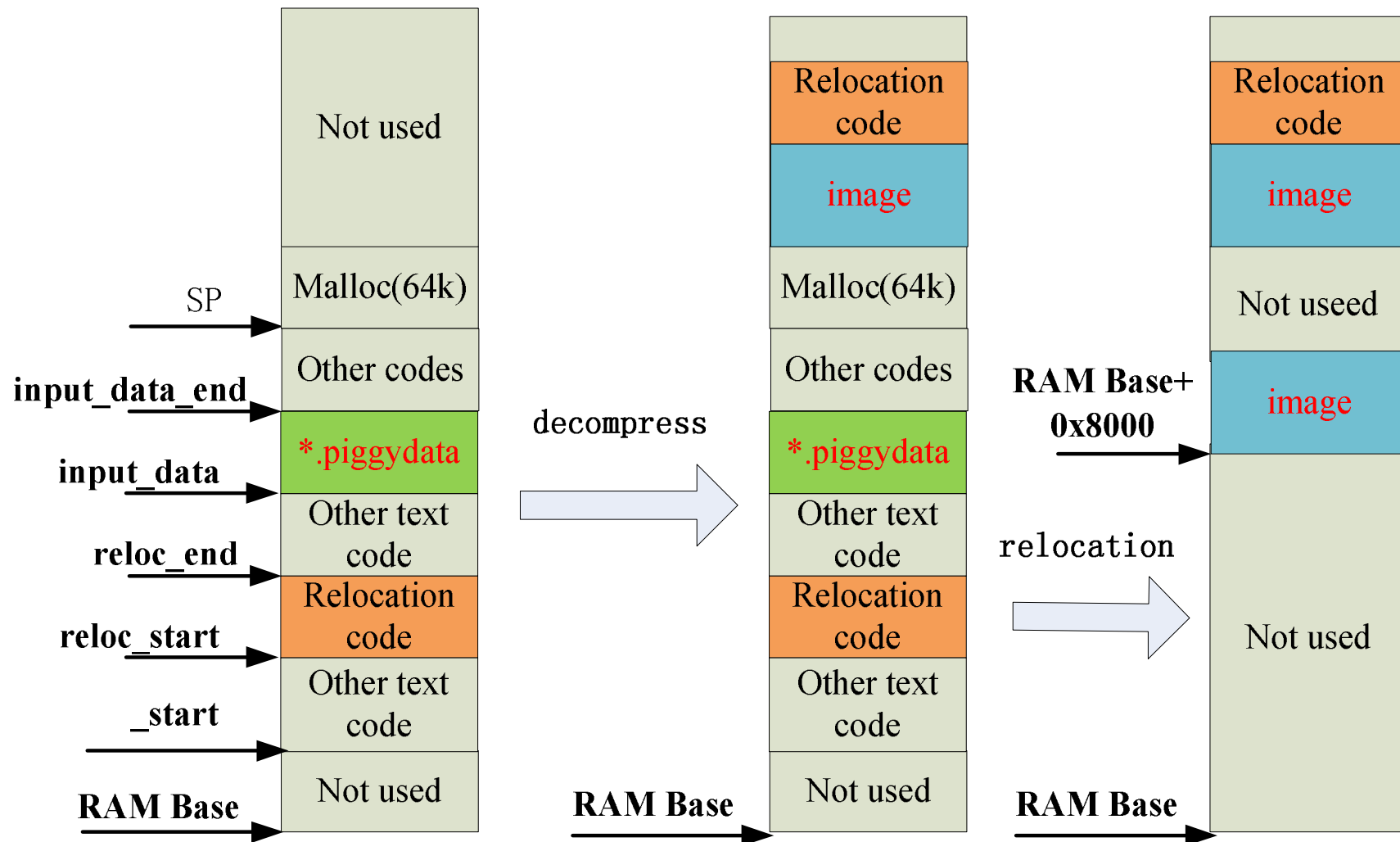
n 直接解压缩到低端地址

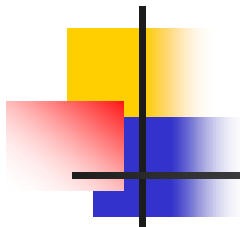




内核解压缩

n 间接解压缩





内核解压缩

n 间接解压缩

- n 确定内核解压缩的临时地址，即位于压缩内核代码之上；
- n 解压缩内核到上述临时地址；
- n 将专门重定位的代码（由`reloc_start`和`reloc_end`标识）拷贝到临时解压缩完毕的内核代码之上；
- n 跳转到重定位代码处运行；
- n 重定位代码将临时解压缩的内核再拷贝到最终的内核运行地址上；



非压缩内核的加载

n 加载条件

- n 非压缩内核已经拷贝到RAM + 0x8000地址
- n 跳转到非压缩内核入口时的状态和从uboot中直接跳转到非压缩内核入口的状态需要一致
- n 关闭cache及MMU，并传递machine id和targs的指针

n 实现

- n 首先刷新cache;
- n 关闭cache;
- n 初始化传递给内核的三个参数;
- n 将内核解压缩后的物理地址赋值给PC，实现绝对跳转



非压缩内核的加载

n 实现

B call_kernel 相对跳转到 call_kernel 处

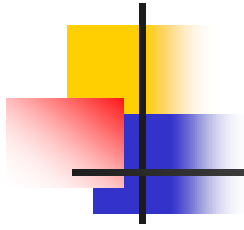
```
call_kernel:  bl      cache_clean_flush  
              bl      cache_off  
              mov     r0, #0  
              mov     r1, r7  
              mov     r2, r8  
              mov     pc, r4
```

@ must be zero

@ restore architecture number

@ restore atags pointer

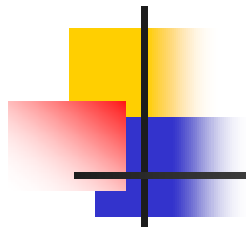
@ call kernel





非压缩内核的启动

- n 内核入口
 - n vmLinux从arch/arm/kernel/head.S
- n MMU开启前
 - n MMU关闭
 - n 运行在物理地址上，链接地址和运行地址不等
 - n 必须相对寻址
- n MMU开启后
 - n 链接地址和运行地址相等，内核最终的运行状态



MMU开启前

- n 检查CPU类型和板卡类型
- n 创建一级页表
 - n 保证内核本身运行所需要的页表
- n 使能MMU

MMU开启前

n 使能MMU—汇编

* Enable the MMU. This completely changes the structure of the visible
* memory space. You will not be able to trace execution through this.
*

* r0 = cp#15 control register

* r13 = *virtual* address to jump to upon completion

*

* other registers depend on the function called upon completion

*/

.align 5

.type __turn_mmu_on, %function

__turn_mmu_on:

mov r0, r0

mcr p15, 0, r0, c1, c0, 0 @ write control reg

mrc p15, 0, r3, c0, c0, 0 @ read id reg

mov r3, r3

mov r3, r3

mov pc, r13

跳转到 r13 所指向的地址，其值为__switch_data 数据域中第一个值
__mmap_switched

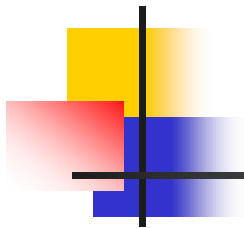


MMU开启前

n 使能MMU—反汇编

```
c0008024: e59fd0c0    ldr    sp, [pc, #192] ; c00080ec <__switch_data>
c0008028: e28fe000    add    lr, pc, #0      ; 0x0
c000802c: e28af010    add    pc, sl, #16     ; 0x10

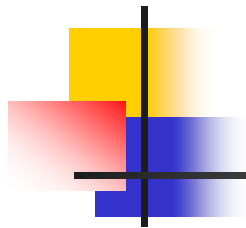
c0008030 <__enable_mmu>:
c0008030: e3800002    orr     r0, r0, #2      ; 0x2
. .
c00080ec <__switch_data>:
c00080ec: c0008110 c0264000 c0264000 c029fa40  ....@&..@&.@.).
```



MMU开启前

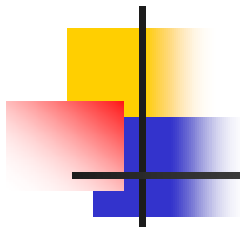
n 使能MMU—反汇编

- n 使能MMU之前，内核运行在物理地址上，而内核编译链接的地址是虚拟地址，此时运行域和加载域不一致，只能采用相对寻址，ldr和adr都是相对于PC寻址的。
- n adr lr, __enable_mmu可实现将__enable_mmu函数的加载域地址存储在LR链接寄存器中
- n add pc, r10, #PROCINFO_INITFUNC直接对PC赋值
- n 上述两句即可模拟一次函数调用，执行完毕处理器相关代码后返回到LR所指定的地址__enable_mmu



MMU开启后

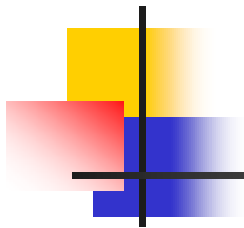
- n 拷贝数据段，准备跳转
 - n b start_kernel
 - n /init/main.c中asmlinkage void __init
start_kernel(void)
- n 设置体系结构相关指针
 - n setup_arch(&command_line)
 - n Arch/arm/kernel/setup.c
 - n 检查CPU类型
 - n 根据machine ID获得板块相关信息
 - n 解析tags形式的命令行参数
 - n 初始化板块相关的函数指针，尤其是板块初始化函数
init_machine



MMU开启后

n 设置体系结构相关指针

```
void __init setup_arch(char **cmdline_p)
{
    setup_processor();
    mdesc = setup_machine(machine_arch_type);
    machine_name = mdesc->name;
    ◦ ◦ ◦
    parse_cmdline(cmdline_p, from);
    ◦ ◦ ◦
    /*
     * Set up various architecture-specific pointers
     */
    init_arch_irq = mdesc->init_irq;
    system_timer = mdesc->timer;
    init_machine = mdesc->init_machine;
    ◦ ◦ ◦ ◦
}
```



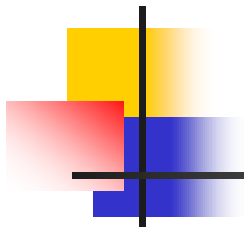
MMU开启后

- n 设置体系结构相关指针
 - n 板卡初始化函数最终将在arch_initcall系列函数调用时被调用

```
static void (*init_machine)(void) __initdata;

static int __init customize_machine(void)
{
    /* customizes platform devices, or adds new ones */
    if (init_machine)
        init_machine();
    return 0;
}

arch_initcall(customize_machine);
```



MMU开启后

n 保存命令参数参数

- n setup_command_line

- n printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);

n 中断定时器等初始化

- n trap_init();

- n init_IRQ();

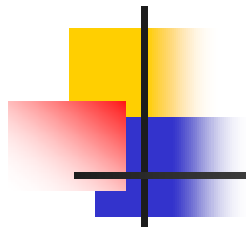
- n init_timers();

- n hrtimers_init();

- n softirq_init();

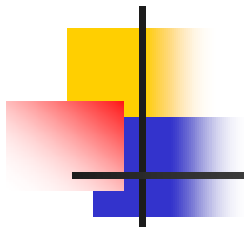
- n timekeeping_init();

- n time_init();



MMU开启后

- n 控制台打印串口初始化
 - n `console_init()`
 - n 从此之后打印输出便可以真正输出到串口终端上
- n 第一个内核线程初始化
 - n `rest_init`
 - n 启用第一个内核线程`init`
 - n 便开始了内核的调度
- n 检查文件系统
 - n `populate_rootfs`
 - n 是否是`ramdisk`，若是，则解压缩



MMU开启后

- n 驱动模块初始化
 - n do_basic_setup
 - n 内核模块初始化
 - n 调用mount_root, 挂接文件系统

其调用 `do_initcalls`, 调用所有的 `init` 修饰的函数

```
extern initcall_t __initcall_start[], __initcall_end[];
```

```
static void __init do_initcalls(void)
{
```

```
    initcall_t *call;
    int count = preempt_count();
```

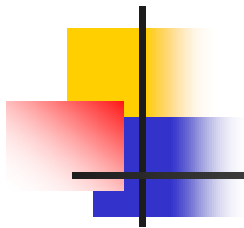
```
    for (call = __initcall_start; call < __initcall_end; call++) {
        char *msg = NULL;
        char msgbuf[40];
        int result;
```

```
        . . .
```

```
        result = (*call)();
```

```
        . . . .
```

```
    }
```



MMU开启后

- n 打开用户空间串口
 - n `sys_open((const char __user *) "/dev/console", O_RDWR, 0)`
 - n 可检查文件系统是否正确
 - n `printk(KERN_WARNING "Warning: unable to open an initial console.\n")`
- n 调用系统启动脚本
 - n `ramdisk_execute_command`为命令行参数提供的初始化脚本init参数
 - n `execute_command`为内核配置的启动脚本
 - n 否则调用`/sbin/init`，其将读取`inittab`脚本，进行相关配置，并最终启动shell

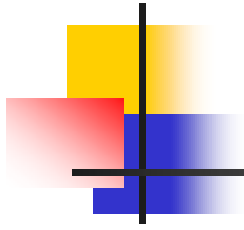


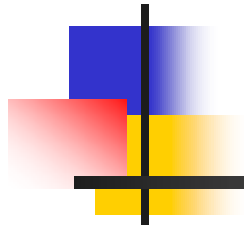
MMU开启后

n 调用系统启动脚本

```
/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
 */
if (execute_command) {
    run_init_process(execute_command);
    printk(KERN_WARNING "Failed to execute %s. Attempting "
        "defaults...\n", execute_command);
}
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found. Try passing init= option to kernel.");
```





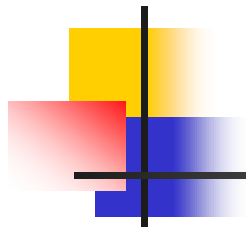
主要内容



① Linux内核的配置编译

② Linux启动流程

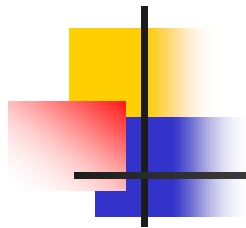
③ Linux内核移植



Linux内核移植

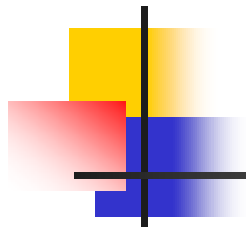
- n 交叉编译器及arch设置
- n 内核配置，CPU及板卡类型选择
- n 驱动配置选择
- n MACHINE结构的定义
 - n arm/mach-xxx

```
MACHINE_START(SMDK2410, "SMDK2410")
/* Maintainer: Jonas Dietsche */
.phys_io = S3C2410_PA_UART,
.io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
.boot_params = S3C2410_SDRAM_PA + 0x100,
.map_io = smdk2410_map_io,
.init_irq = s3c24xx_init_irq,
.init_machine = smdk2410_init,
.timer = &s3c24xx_timer,
MACHINE_END
```



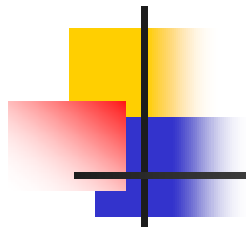
Linux内核移植

- n 外设资源的定义
- n 链接地址设置
 - n `arm/mach-xxx/Makefile.boot`
 - n `zreladdr-y := 0x30008000`
 - n `params_phys-y := 0x30000100`
- n `ulmage`制作
 - n `-a -e`等参数，参见第三篇bootloader移植
- n 启动参数设置
 - n `Bootagrs`中的串口 波特率 root设备 ramdisk 大小 初始化脚本



Linux内核移植

- n Uncompressing Kernel Image ... OK后无输出问题
 - n 内核中串口是否正确配置了，和开发板上目前的调试串口是否一致
 - n 内核使用的波特率是否和当前的调试串口一致
 - n 内核的串口驱动本身有问题
 - n 在串口初始化之前，由于内核本身移植的有问题，系统可能已经crash了
 - n 检查链接地址及uImage制作的相关地址



Linux内核移植

- n Can not mount rootfs
- n Warning: unable to open an initial console
- n No init found. Try passing init= option to kernel

