vivi源代码最为详细分析（一）

通过vivi研究bootloader有一段时间了，基本是在与之相关的基础方面做工作，还没有真正深入研究vivi。以后的学习重心就要放到研究vivi源代码上面了。我想，真正细致地弄清楚vivi实现的细节，对C语言水平的提高，对ARM体系结构的认识，对S3C2410的熟悉，对嵌入式bootloader相关技术，都能有很大的好处。学习的进度会慢一些，但是务求深入，并且打好相关的基础。

一、写在前面的话

嵌入式系统软件开发主要包括五个方面：bootloader编写（移植）、驱动程序编写（移植）、操作系统裁减（移植）、文件系统制作、应用程序编写（移植）。嵌入式开发流程我已经熟悉，但是仅限于完成最为基本的工作，大部分是借助网络资料，自己独立解决的问题很有限。学习嵌入式系统已经一年了，算是入门了。然而，入门之后如何继续深入学习嵌入式系统开发，如何提高自身的能力？

我想，这也许是独立摸索的学习者都会遇到的问题吧。思考之后有所得，核心就是一句话：务实，理论与实践结合！具体说来，就是要不断的认识自己，去了解自己最适合做什么。这是最为重要的，如果不知道做什么，就无法安排学习的重点。嵌入式开发的领域太广，要想在方方面面都深入不太容易（当然牛人除外）。现在对自己的认识如下：本科有硬件、通信背景，但是没有太多机会进行硬件设计。而硬件设计最为重要的就是经验，动手能力，所以不打算把硬件设计作为学习的重点。底层软件开发既需要对硬件熟悉，又需要软件设计能力，正适合我。所以以后的学习，以底层软件开发（bootloader设计、驱动程序设计）为重点，同时也要加强硬件学习。学习有重点，但是嵌入式开发的其他领域也要涉及，了解广博才能更有助于设计。进展慢不要紧，关键是要深入，深入，再深入。真正地去理解这些技术，并且能够熟练的应用。这半年的核心就是bootloader技术研究，打算先看vivi，然后看uboot。手头上的板子有s3c2410、at91rm9200，这些都可以拿来训练，争取能够通过bootloader技术的掌握，同时熟悉了ARM体系结构、ARM汇编、开发工具等等一系列相关内容，总结学习的方法，提高学习能力。

二、准备工作

在分析vivi源代码的时候，不打算完全按照vivi的代码来进行。我的思路是，以从nand flash启动为主线，分析从上电到引导内核的过程。以自己的理解去实现vivi的源代码，要自己手动编写，即使与vivi的代码相同。只有这样，才能从整体上理解vivi的设计方法，理解vivi各个功能的实现细节。这份文档算是自己的学习笔记。

三、bootloader stage1：【arch/s3c2410/head.S】

首先解决一个问题，就是为什么使用head.S而不是用head.s？有了GNU AS和GNU GCC的基础，不难理解主要原因就是为了使用C预处理器的宏替换和文件包含功能（GNU AS的预处理无法完成此项功能）。可以参考前面的总结部分。这样的好处就是可以使用C预处理器的功能来提高ARM汇编的程序设计环境，更加方便。但是因为ARM汇编和C在宏替换的细节上有所不同，为了区分，引入了__ASSEMBLY__这个变量，这是通过Makefile中AFLAGS来引入的（一般在顶层Makefile中定义），具体如下：

```
AFLAGS := -D__ASSEMBLY__ $(CPPFLAGS)
```

在后面的头文件中，会看到很多#ifdef __ASSEMBLY__等的操作，就是用来区分这个细节的。在编译汇编文件时，加入AFLAGS选项，所以__ASSEMBLY__传入，也就是定义了__ASSEMBLY__；在编译C文件时，没有用AFLAGS选项，自然也就没有定义__ASSEMBLY__。由此相应的问题就比较清晰了。这个小技巧也是值得学习和借鉴的。

1 首先关注一下开始的三个头文件。

```
#include "config.h"
#include "linkage.h"
#include "machine.h"
```

（1）利用source insight来查看【include/config.h】。

```
#ifndef _CONFIG_H_
#define _CONFIG_H_

#include "autoconf.h"

#endif /* _CONFIG_H_ */
```

可见，config.h只是包含一个autoconf.h。而关于autoconf.h的生成，在vivi配置文件分析的时候也解释的很清楚了，在这里就不用再细分析了。需要解释的一点是，如果写一个专用的bootloader，不采用vivi的配置机制，那么配置部分就没有这么复杂了，只需要在include文件夹中包含一个配置头文件即可。现在bootloader的设计有两种趋势，一种是针对特定应用，有特殊要求，也就是“专用”。那么设计时，不需要过多的配置，只需要简单的完成引导内核的功能就可以了。二是普通应用，一般是对基本“通用”的bootloader，比如uboot等，然后根据相应的模

版进行移植。这就需要了解uboot等的架构，可以进行定制和功能的增加。uboot完成的不仅仅是一个bootloader的功能，还可以提供调试等功能，所以其角色还包含驻留程序这个功能，也就是uboot真正的角色是monitor。当然，可以不加区分，统称为bootloader。而分析vivi源代码的实现，对这两个方向都有帮助。

（2）【include/linkage.h】就是实现了ENTRY宏的封装。其实这个头文件也仅仅为head.S提供了服务，实际上没有必要写的这么复杂，可以简化一些。比如，我修改了这个头文件，如下：

```
[armlinux@lqm include]$ cat linkage.h
#ifndef _VIVI_LINKAGE_H
#define _VIVI_LINKAGE_H

#define SYMBOL_NAME(X) X
#ifdef __STDC__
    #define SYMBOL_NAME_LABEL(X) X##:
#else
    #define SYMBOL_NAME_LABEL(X) X/**/:
#endif

#ifdef __ASSEMBLY__

#define ALIGN .align 0

#define ENTRY(name) /
    .globl SYMBOL_NAME(name); /
    ALIGN; /
    SYMBOL_NAME_LABEL(name)

#endif

#endif
```

在这里，要加强一下C语言宏的设计和分析能力。下面就几个点简单的分析一下，后面专门就C宏部分做个总结。

关于__STDC__这个宏，是编译器自动添加的，含义就是支持标准C。如果支持标准C，那么##的作用就是“连接”，所以SYMBOL_NAME_LABEL(_start)宏展开为_start:（依我的意思理解##也就相当于分号，表示后面可以继续连接其他的一些定义），如果不支持标准C，则利用了C预处理器对注释的处理方式，就是把/**/替换为一个空格（也就是相当于展开的字符后带个空格），可以测试一下。

另外，关于ENTRY宏的封装，利用了GNU AS在ARM上的相关特点。首先，利用了分号作为三条语句的连接符(包括最后一个由##展开来的分号)，而分号是GNU AS汇编注释符的一种（另外一种是@）。另外，关于ALIGN为什么用.align 0。这可以参考GNU AS手册，上面讲解的比较清晰，主要是为了兼容ARM本身的编译器。理解了这个也就不难得出ENTRY(_start)宏展开后的形式了。有一个技巧就是可以通过下面的命令来检测宏展开后的结果，比如：

```
[root@lqm vivi_myboard]# gcc -E -D__ASSEMBLY__ -I./include arch/s3c2410/head.S >aaa
```

可以查看aaa文件的显示结果，做了一些注释：

```
# 1 "arch/s3c2410/head.S"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "arch/s3c2410/head.S"
# 35 "arch/s3c2410/head.S"
# 1 "include/config.h" 1
# 14 "include/config.h"
# 1 "include/autoconf.h" 1
# 15 "include/config.h" 2
# 36 "arch/s3c2410/head.S" 2
# 1 "include/linkage.h" 1
# 37 "arch/s3c2410/head.S" 2
# 1 "include/machine.h" 1
# 19 "include/machine.h"
# 1 "include/platform/smdk2410.h" 1
```

```
# 1 "include/s3c2410.h" 1
# 22 "include/s3c2410.h"
# 1 "include/hardware.h" 1
# 23 "include/s3c2410.h" 2
# 1 "include/bitfield.h" 1
# 24 "include/s3c2410.h" 2
# 3 "include/platform/smdk2410.h" 2

# 1 "include/sizes.h" 1
# 8 "include/platform/smdk2410.h" 2
# 74 "include/platform/smdk2410.h"
# 1 "include/architecture.h" 1
# 75 "include/platform/smdk2410.h" 2
# 20 "include/machine.h" 2
# 38 "arch/s3c2410/head.S" 2

@ Start of executable code
```

宏定义展开

```
.globl _start; .align 0; _start:
.globl ResetEntryPoint; .align 0; ResetEntryPoint:
```

下面是装载中断向量表，ARM规定，在起始必须有8条跳转指令，你可以用b 标号也可以用ldr pc 标号。这样的8条规则的标志被arm定义为bootloader的识别标志，检测到这样的标志后，就可以从该位置启动。这样的做法是因为开始的时候不一定有bootloader，必须有一种识别机制，如果识别到bootloader，那么就从bootloader启动。

```
@
@ Exception vector table (physical address = 0x00000000)
@

@ 0x00: Reset
    b Reset

@ 0x04: Undefined instruction exception
UndefEntryPoint:
    b HandleUndef

@ 0x08: Software interrupt exception
SWIEntryPoint:
    b HandleSWI
```

```
@ 0x0c: Prefetch Abort (Instruction Fetch Memory Abort)
PrefetchAbortEnteryPoint:
    b HandlePrefetchAbort

@ 0x10: Data Access Memory Abort
DataAbortEntryPoint:
    b HandleDataAbort

@ 0x14: Not used
NotUsedEntryPoint:
    b HandleNotUsed

@ 0x18: IRQ(Interrupt Request) exception
IRQEntryPoint:
    b HandleIRQ

@ 0x1c: FIQ(Fast Interrupt Request) exception
FIQEntryPoint:
    b HandleFIQ

下面是固定位置存放环境变量
@
@ VIVI magics
@

@ 0x20: magic number so we can verify that we only put
    .long 0
@ 0x24:
    .long 0
@ 0x28: where this vivi was linked, so we can put it in memory in the right place
    .long _start  //_start用来指定链接后的起始装载地址装载到内存中的地址
@ 0x2C: this contains the platform, cpu and machine id
    .long ((1 << 24) | (6 << 16) | 193)
@ 0x30: vivi capabilities
    .long 0

@ 0x34:
    b SleepRamProc

@
@ Start VIVI head
```

```
@
Reset: //上电后cpu会从0x0地址读取指令执行，也就是b Reset

    @ disable watch dog timer
    mov r1, #0x53000000
    mov r2, #0x0
    str r2, [r1]
# 121 "arch/s3c2410/head.S"
    @ disable all interrupts
    mov r1, #0x4A000000
    mov r2, #0xffffffff
    str r2, [r1, #0x08]  //0x4A000008为中断屏蔽寄存器，将里面赋全1，表示屏蔽这32个中断源
    ldr r2, =0x7ff
    str r2, [r1, #0x1C]  //0x4A00001C为中断子屏蔽寄存器，里面低11位也用来表示屏蔽掉这11个中断源

    @ initialise system clocks
    mov r1, #0x4C000000
    mvn r2, #0xff000000 //MVN指令可完成从另一个寄存器、被移位的寄存器、或将一个立即数加载到目的寄存器。与MOV指令不同之处是在传送之前按位被取反了，即把一个被取反的值传送到目的寄存器中。也就是r2=0x00ffffff
    str r2, [r1, #0x00] //我们可以在程序开头启动MPLL，在设置MPLL的几个寄存器后，需要等待一段 时间(Lock Time)， MPLL的输出才稳定。在这段时间(Lock Time)内，FCLK停振，CPU停止工作。Lock Time的长短由寄存器LOCKTIME设定，Lock Time之后，MPLL输出正常，CPU工作在新的FCLK下，前面说过，MPLL启动后需要等待一段时间(Lock Time)，使得其输出稳定。
位[23:12]用于UPLL，位[11:0]用于MPLL。本实验使用确省值0x00ffffff。

    @ldr r2, mpll_50mhz
    @str r2, [r1, #0x04]

    @ 1:2:4
    mov r1, #0x4C000000
    mov r2, #0x3
    str r2, [r1, #0x14] //0x4C000014为分频寄存器，用于设置FCLK、HCLK、PCLK三者的比例
    bit[2] — — HDIVN1，若为1，则 bit[1:0] 必 须 设 为 0b00， 此 时FCLK:HCLK:PCLK=1:1/4:1/4；若为0，三者比例由bit[1:0]确定 bit[1]——HDIVN，0：HCLK=FCLK；1：HCLK=FCLK/2 bit[0]——PDIVN，0：PCLK=HCLK；1：PCLK=HCLK/2
 本实验设为0x03，则FCLK:HCLK:PCLK=1:1/2:1/4

    mrc p15, 0, r1, c1, c0, 0 @ read ctrl register
    orr r1, r1, #0xc0000000 @ Asynchronous
```

45:25

```
    mcr p15, 0, r1, c1, c0, 0 @ write ctrl register
```

上面这三句代码的意思是切换模式：If HDIVN = 1, the CPU bus mode has to be changed from the fast bus mode to the asynchronous bus mode using following instructions：

```
MMU_SetAsyncBusMode
mrc  p15, 0, r0, c1, c0, 0
orr  r0, r0, #R1_nF:OR:R1_iA
mcr  p15,0, r0, c1, c0, 0
```

其中的“ R1_nF:OR:R1_iA ”等于0xc0000000。意思就是说，当HDIVN = 1时，CPU bus mode需要从原来的“ fast bus mode ”改为“ asynchronous bus mode ”。

```
    @ now, CPU clock is 200 Mhz
    mov r1, #0x4C000000
    ldr r2, mpll_200mhz
    str r2, [r1, #0x04] //0x4C000004为MPLLCON寄存器， 对于MPLLCON寄存器，[19:12]为MDIV，[9:4]为PDIV，[1:0]为SDIV。有如下计算公式：
  MPLL(FCLK) = (m * Fin)/(p * 2^s) 其中: m = MDIV + 8, p = PDIV + 2 对于本开发板，Fin =  12MHz, MPLLCON设为0x5c0040，可以计算出FCLK=200MHz，再由CLKDIVN的设置可知：HCLK=100MHz, PCLK=50MHz。
# 164 "arch/s3c2410/head.S"
    bl memsetup

    @ Check if this is a wake-up from sleep
    ldr r1, PMST_ADDR  //0x560000B4为GSTATUS2寄存器，里面[0:2]三位有效
    ldr r0, [r1] //将该寄存器中的值取出来保存到r0中
    tst r0, #((1 << 1)) //测试r0的第一位。这位是Power_OFF reset. The reset after the wakeup from Power_OFF mode.The setting is cleared by writing "1" to this bit.TST指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的与运算，并根据运算结果更新CPSR中条件标志位的值。
    bne WakeupStart

    @ All LED on 这里就需要对GPIO口进行控制
    mov r1, #0x56000000
    add r1, r1, #0x50 //0x56000050是GPFCON用来配置port F端口
```

```
    ldr r2,=0x55aa
    str r2, [r1, #0x0] //设置为0101010110101010因为每两位用来控制一个引脚，也就是将GPF4-GPF7设置为输出，将GPF0-GPF3设置为中断
    mov r2, #0xff
    str r2, [r1, #0x8]  //0x56000058为GPFUP为port F的上拉寄存器，全设置为1表示禁止上拉功能
    mov r2, #0x00
    str r2, [r1, #0x4]  //0x56000054是GPFDAT，总共8位有效，每位控制一个引脚，主要是将GPF4-GPF7数据位全设置为0而这四个引脚是用来控制板子上4个LED，置0则表示亮。
# 230 "arch/s3c2410/head.S"
    @ set GPIO for UART
    mov r1, #0x56000000
    add r1, r1, #0x70 //0x56000070为GPHCON 用来配置port H 而port H主要就是来控制UART的各个引脚如：UART中接收和发送端口RXDn和TXDn，当然还有自动流控制模式的nRTS0和nCTS0端口。
    ldr r2, gpio_con_uart  //我们可以看到 gpio_con_uart: .long vGPHCON  gpio_up_uart: .long vGPHUP 而在platform中的smdk2410.h中定义了这两个值#define vGPHCON   0x0016faaa 表示GPHCON控制11个引脚，如GPH2若设置为10则表示TXD0.类似，具体的查看数据手册
#define vGPHUP   0x000007ff //同样将这11位的引脚上拉禁止
    str r2, [r1, #0x0]
    ldr r2, gpio_up_uart
    str r2, [r1, #0x8] //上面也是来配置串口所用到的GPIO口，因为串口的输入输出口都是利用到GPIO
    bl InitUART
# 259 "arch/s3c2410/head.S"
    bl copy_myself

    @ jump to ram
    ldr r1, =on_the_ram
    add pc, r1, #0
    nop
    nop
1: b 1b @ infinite loop

on_the_ram:
# 279 "arch/s3c2410/head.S"
    @ get read to call C functions开始调用C函数之前就需要将一些参数准备好，如堆栈要准备好函数调用时需要进出栈
    ldr sp, DW_STACK_START @ setup stack pointer
    mov fp, #0 @ no previous frame, so fp=0
    mov a2, #0 @ set argv to NULL

    bl main @ call main
```

:45:25

```
    mov pc, #0x00000000 @ otherwise, reboot

@
@ End VIVI head
@

下面是子例程
@
@ Wake-up codes
@

WakeupStart:
    @ Clear sleep reset bit
    ldr r0, PMST_ADDR
    mov r1, #(1 << 1)
    str r1, [r0]

    @ Release the SDRAM signal protections
    ldr r0, PMCTL1_ADDR
    ldr r1, [r0]
    bic r1, r1, #((1 << 19) | (1 << 18) | (1 << 17))
    str r1, [r0]

    @ Go...
    ldr r0, PMSR0_ADDR @ read a return address
    ldr r1, [r0]
    mov pc, r1
    nop
    nop
1: b 1b @ infinite loop

SleepRamProc:
    @ SDRAM is in the self-refresh mode */
    ldr r0, REFR_ADDR
    ldr r1, [r0]
    orr r1, r1, #(1 << 22)
    str r1, [r0]

    @ wait until SDRAM into self-refresh
    mov r1, #16
```

```
1: subs r1, r1, #1
    bne 1b

    @ Set the SDRAM singal protections
    ldr r0, PMCTL1_ADDR
    ldr r1, [r0]
    orr r1, r1, #((1 << 19) | (1 << 18) | (1 << 17))
    str r1, [r0]


    ldr r0, PMCTL0_ADDR
    ldr r1, [r0]
    orr r1, r1, #(1 << 3)
    str r1, [r0]
1: b 1b
# 379 "arch/s3c2410/head.S"
.globl memsetup; .align 0; memsetup:  //对ENTRY(memsetup)宏的展开
    @ initialise the static memory
    //同样在这里是对内存控制中用到的13个寄存器进行初始化
    @ set memory control registers
    mov r1, #0x48000000
    adrl r2, mem_cfg_val
    add r3, r1, #52
1: ldr r4, [r2], #4
    str r4, [r1], #4
    cmp r1, r3
    bne 1b

    mov pc, lr



@
@ copy_myself: copy vivi to ram
@下面的将vivi从flash中搬移到sdram中来执行，很重要
copy_myself:
    mov r10, lr  //将返回地址保存到r10，为了执行完后返回到主程序

    @ reset NAND
    mov r1, #0x4E000000
    ldr r2, =0xf830 @ initial value 初始化NFCONF寄存器，至于为什么设置为0xf830前面在NAND里面讲过
```

```
    str r2, [r1, #0x00]
    ldr r2, [r1, #0x00]
    bic r2, r2, #0x800 @ enable chip 也就是相当于NFCONF &= ~0x800 将第15为置为1使能NAND FLASH
    str r2, [r1, #0x00]
    mov r2, #0xff @ RESET command
    strb r2, [r1, #0x04] //向NFCMD寄存器中置0xff也就是reset命令
    mov r3, #0 @ wait //下面几句是一个延时程序用来等待几秒，等到NAND 准备好
1: add r3, r3, #0x1
    cmp r3, #0xa
    blt 1b
2: ldr r2, [r1, #0x10] @ wait ready
    tst r2, #0x1 //检查NFSTAT寄存器的第0位是否为1也就是是否准备好
    beq 2b //没有准备好则继续等待
    ldr r2, [r1, #0x00]
    orr r2, r2, #0x800 @ disable chip 相当于NFCONF |= 0x800 禁止掉NAND FLASH等到要使用FLASH时再使能
    str r2, [r1, #0x00]

    @ get read to call C functions (for nand_read())
    ldr sp, DW_STACK_START @ setup stack pointer //每次需要从汇编调到C函数时 都需要设置好堆栈
    mov fp, #0 @ no previous frame, so fp=0

    @ copy vivi to RAM 之前都是一些初始化，下面才开始利用C函数来真正开始拷贝
    ldr r0, =(0x30000000 + 0x04000000 - 0x00100000)//DRAM_BASE + DRAM_SIZE - VIVI_RAM_SIZE,为什么这里不是将vivi拷贝到sdram的起始地址而是拷贝到64MB的sdram的最后1M的区域，可能是这里的sdram采用高端模式，将映象从高地址向低地址存放
    mov r1, #0x0 //r1则是vivi的起始地址，也就是flash上的0x0地址
    mov r2, #0x20000 //上面三句都是用来为调用nand_read_ll函数准备好参数，r2表示拷贝大小
    bl nand_read_ll //这个c函数在arch/s3c2410/nand_read.c中 nand_read_ll就不具体分析了在nand里面有讲过

    tst r0, #0x0  //为什么要比较r0与上0，等于0的话 则去执行ok_nand_read,在这里r0是nand_read_ll函数的返回值，拷贝成功则返回0,所以这就是为什么将r0和0比较
    beq ok_nand_read //ok_nand_read子程序用来比较拷贝到sdram最后1MB的vivi和原始的存放在flash上的vivi是否相同，检查拷贝是否成功，vivi在sdram中的位置也就是离0x34000000地址前1MB的位置也就是0x33f00000
# 441 "arch/s3c2410/head.S"
ok_nand_read:

    @ verify
    mov r0, #0  //r0这里为flash上vivi的起始地址
```

```
    ldr r1, =0x33f00000 //r1这里为拷贝到sdram上vivi的起始地址
    mov r2, #0x400 @ 4 bytes * 1024 = 4K-bytes //要比较多少个字节数
go_next:
    ldr r3, [r0], #4 //将r0对应地址的值取出来保存到r3中，然后r0自加4个字节
    ldr r4, [r1], #4
    teq r3, r4  //测试r3和r4是否相等
    bne notmatch //若不相等，则跳到notmarch处
    subs r2, r2, #4 //将比较总字节数减去4个字节，已经比较了4个字节
    beq done_nand_read //若r2为0,则表示已经比较完毕，跳转到done_nand_read处
    bne go_next //r2若还不等于0则继续取值比较
notmatch:
# 469 "arch/s3c2410/head.S"
1: b 1b
done_nand_read:

    mov pc, r10 //比较完了 就退出子程序，返回主程序执行

@ clear memory
@ r0: start address
@ r1: length
mem_clear:
    mov r2, #0
    mov r3, r2
    mov r4, r2
    mov r5, r2
    mov r6, r2
    mov r7, r2
    mov r8, r2
    mov r9, r2

clear_loop:
    stmia r0!,{r2-r9} //将r2-r9也就是0赋值给从r0为内存起始地址的连续的8*4个字节中
    subs r1, r1, #(8 * 4) //每次清除32个字节
    bne clear_loop



    mov pc, lr
# 613 "arch/s3c2410/head.S"
@ Initialize UART
```

```
@
@ r0 = number of UART port
InitUART: //这里也不细讲了，在UART章节中已经详细的讲解了每个寄存器的设置
	ldr r1, SerBase
	mov r2, #0x0
	str r2, [r1, #0x08]
	str r2, [r1, #0x0C]
	mov r2, #0x3
	str r2, [r1, #0x00]
	ldr r2, =0x245
	str r2, [r1, #0x04]

	mov r2, #((50000000 / (115200 * 16)) - 1)
	str r2, [r1, #0x28]

	mov r3, #100
	mov r2, #0x0
1: sub r3, r3, #0x1
	tst r2, r3
	bne 1b
# 653 "arch/s3c2410/head.S"
	mov pc, lr


@
@ Exception handling functions
@
HandleUndef:
1: b 1b @ infinite loop

HandleSWI:
1: b 1b @ infinite loop

HandlePrefetchAbort:
1: b 1b @ infinite loop

HandleDataAbort:
1: b 1b @ infinite loop

HandleIRQ:
1: b 1b @ infinite loop
```

```
HandleFIQ:
1: b 1b @ infinite loop

HandleNotUsed:
1: b 1b @ infinite loop

@
@ Low Level Debug
@
# 838 "arch/s3c2410/head.S"
@
@ Data Area
@
@ Memory configuration values
.align 4
mem_cfg_val: //这些变量都是内存控制寄存器的初始值，因为寄存器比较多，所以将初始值制作成表的形式，然后分别读表来初始化各个寄存器
    .long 0x22111110
    .long 0x00000700
    .long 0x00000700
    .long 0x00000700
    .long 0x00000700
    .long 0x00000700
    .long 0x00000700
    .long 0x00018005
    .long 0x00018005
    .long 0x008e0459
    .long 0xb2
    .long 0x30
    .long 0x30

@ Processor clock values
.align 4
clock_locktime:
    .long 0x00ffffff
mpll_50mhz:
    .long ((0x5c << 12) | (0x4 << 4) | (0x2))

mpll_200mhz:
```

```
        .long ((0x5c << 12) | (0x4 << 4) | (0x0))
clock_clkcon:
        .long 0x0000fff8
clock_clkdivn:
        .long 0x3
@ initial values for serial
uart_ulcon:
        .long 0x3
uart_ucon:
        .long 0x245
uart_ufcon:
        .long 0x0
uart_umcon:
        .long 0x0
@ inital values for GPIO
gpio_con_uart:
        .long 0x0016faaa
gpio_up_uart:
        .long 0x000007ff

        .align 2
DW_STACK_START:
        .word (((((0x30000000 + 0x04000000 - 0x00100000) - 0x00100000) - 0x00004000) - (0x00004000 + 0x00004000 +

0x00004000)) - 0x00008000)+0x00008000 -4
# 922 "arch/s3c2410/head.S"
.align 4
SerBase:

        .long 0x50000000
# 935 "arch/s3c2410/head.S"
.align 4
PMCTL0_ADDR:
        .long 0x4c00000c
PMCTL1_ADDR:
        .long 0x56000080
PMST_ADDR:
        .long 0x560000B4
PMSR0_ADDR:
        .long 0x560000B8
```

```
REFR_ADDR:
    .long 0x48000024
[root@lqm vivi_myboard]#
```

【include/machine.h】则是利用条件编译来选择适合自己开发板的头文件，如：

```
#ifdef CONFIG_S3C2410_SMDK
#include "platform/smdk2410.h"
#endif
```

本开发板的头文件是**【include/platform/smdk2410.h】**，主要是一些寄存器的初始值（以**v**开头）和一些相关的地址等等的定义。一般开发板不同，都是修改此文件相应的部分。这个头文件很重要，因为我们一般要将**vivi**移植到某个类似于**smdk2410**的开发板时都只是在这个头文件中修改一些寄存器的初始值和一些寄存器的地址值还有内存分配的地址值等等

2、关于中断向量表

开始对中断向量表很疑惑。现在的理解比较清晰了，在硬件实现上，会支持中断机制，这个可以参考微机接口原理部分详细理解。现在的中断机制处理的比较智能，对每一类中断会固定一个中断向量，比如说，发生IRQ中断，中断向量地址为0x00000018（当然，这还与ARM9TDMI core有关，其中一个引脚可以把中断向量表配置为高端启动，或者低端启动。你可以通过CP15的register 1的bit 13的V bit来设置，可以查看Datasheet TABLE 2-10来解决。），那么PC要执行的指令就是0x00000018。如果在这个地址放上一个跳转指令（只能使用b或者ldr），那么就可以跳到实际功能代码的实现区了。ARM体系结构规定在上电复位后的起始位置，必须有8条连续的跳转指令，这是bootloader的识别入口，通过硬件实现。看一下vivi的中断向量表：

```
@ 0x00
    b Reset
@ 0x04
HandleUndef:
    b HandleUndef
@ 0x08
HandleSWI:
    b HandleSWI
@ 0x0c
HandlePrefetchAbort:
    b HandlePrefetchAbort
@ 0x10
HandleDataAbort:
    b HandleDataAbort
@ 0x14
HandleNotUsed:
    b HandleNotUsed
@ 0x18
HandleIRQ:
    b HandleIRQ
@ 0x1c
HandleFIQ:
    b HandleFIQ
```

注意，中断向量表可以修改，也可以通过**MMU**实现地址重映射，来改变对应的物理介质**(**在**MMU**一章中我们提到过由于**flash**介质上执行速度比较慢，所以我们一般利用**MMU**将**0x0**开始的一段代码映射到**SDRAM**从**0x30000000**开始的区域，这样代码的执行就比较快**)**。如果不对每种中断进行测试，可以采用下面的书写方式。

```
@ 0x00
    b Reset
@ 0x04
HandleUndef:
    b .
@ 0x08
HandleSWI:
    b .
@ 0x0c
HandlePrefetchAbort:
    b .
@ 0x10
HandleDataAbort:
    b .
@ 0x14
HandleNotUsed:
    b .
@ 0x18
HandleIRQ:
    b .
@ 0x1c
HandleFIQ:
    b .
```

其中，“.”表示当前地址，那么很明显，“b .”，就表示了死循环了。

如果增加中断处理，则需要考虑使用b还是使用ldr。主要的区别在于b指令跳转范围有限，仅仅是+-32M。所以如果超出32M，那么就要采用ldr了，这也就是为什么下面中断表中reset是采用b跳转指令，而其他中断向量都是采用ldr,基本的模式是：

```
.globl _start
_start: b reset
        ldr pc, _undefined_instruction
        ldr pc, _software_interrupt
        ldr pc, _prefetch_abort
        ldr pc, _data_abort
        ldr pc, _not_used
        ldr pc, _irq
        ldr pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort: .word data_abort
_not_used: .word not_used
_irq: .word irq
_fiq: .word fiq

...
/*
 * exception handlers
 */
        .align  5
undefined_instruction:
        get_bad_stack
        bad_save_user_regs
        bl      do_undefined_instruction

...
```

这就是uboot采用的方式。这些技术是相当成熟的。现在在加速启动上，就会考虑对中断向量表作出一些相应的处理。比如，如果从nor flash启动，启动后把kernel等加载到sdram里面运行。无疑，sdram里面运行速度比nor flash里面快。但是如果不对中断向量表进行处理，那么发生中断时，首先还是得访问nor flash里面的中断向量表。然后跳转到sdram相应的执行部分。所以，提高速度就是要解决不经过nor flash，完全在sdram里面运行。采用的技术手段是通过重定向机制，有些SoC提供了这样的硬件手段，但是有些没有，比如S3C2410就没有。但是S3C2410有MMU，可以通过MMU来改变映射关系实现虽然中断向量还是0x00000018，但是实际的物理介质是

sdram，具体的方法可以参考MMU部分的总结来完成。如果采用**nand flash**启动的话，就不必作出这样的处理。本质上是因为**nand flash**并非内存映射，而且中断向量表占用的是开始**4K**的**steppingstone**，是**sram**，其速度比**sdram**还要快，如果作出上面的处理，速度反而会下降了。

中断向量表的出现也是具有历史原因的，是解决特定问题采用的一种技术手段。为了了解中断向量表的一些特殊用途，还应该理解加载域和运行域的不同。这样就能从全局上把握处理的原则，相应的解决机制也比较容易理解了。

3、关于vivi magic number

接下来，vivi设置了自己的一些magic number。理解什么是magic number，需要通过google来查找。本来wiki上有关于magic number（programming）的解释，不过最近打不开了，找了一个替代网站，网址是http://www.answers.com/，上面的内容是wiki的备份吧（好像可以这样理解）。关于magic number（programming）的文章为http://www.answers.com/topic/magic-number-programming。关于magic number（programming）的总结和解释，专门拿出一篇来总结（基本上是上述英文文章的翻译吧，不过讲解的确实非常清晰，读完之后对magic number就很明白了，而且在程序设计中，如何恰当的使用magic number，也会有一定的认识）。

```
@ 0x20: magic number so we can verify that we only put
        .long 0
@ 0x24:
        .long 0
@ 0x28: where this vivi was linked, so we can put it in memory in the right place
        .long _start
@ 0x2C: this contains the platform, cpu and machine id
        .long ARCHITECTURE_MAGIC
@ 0x30: vivi capabilities
        .long 0
```

对vivi的这些magic number，虽然设计在这里，不过大部分还是没有使用的。其中0x20和0x24没有使用，在0x2C处，倒是设计了一个magic number，组成的格式如下：bit[31:24]为platform，bit[23:16]为cpu type，bit[15:0]为machine id。关于ARCHITECTURE_MAGIC的定义，在【include/platform/smdk2410.h】，如下：

```
/*
 * Architecture magic and machine type
 */
#include "architecture.h"
#define MACH_TYPE      193
#define ARCHITECTURE_MAGIC    ((ARM_PLATFORM << 24) | (ARM_S3C2410_CPU << 16) | /
            MACH_TYPE)
```

具体的值的定义则在【include/architecture.h】里面。ARM_PLATFORM为1，ARM_S3C2410_CPU为6，很简单的推理，可以计算出s3c2410的值0x010600c1。不过你可以尝试把此部分去掉，编译仍然没有问题（在我的配置的前提下，因为还没有分析完，所以还不好确定，在vivi的打印信息里面，只有一个MACH_TYPE，实现的手段是通过上述宏，还是通过读取内存提取字段，等分析到那个部分的时候再具体解决，总之，在这里，这个并不成为问题。你可以有多种猜想，也可以按照自己的想法定制，本来magic number就是“幻数”，你当然也可以玩一下了。）

4、实际完成的主要任务：

·关闭看门狗。上电复位后，看门狗默认是开启的，vivi是不需要使用看门狗的，所以首先要关闭。

·关闭所有中断。上电复位后，所有中断默认是关闭的，所以可以不需要代码实现。当然，为了保险和理解上的方便，可以增加此部分代码（vivi就是如此）。

·初始化系统时钟。参考clock部分。

·内存设置。主要就是完成13个相关寄存器的设置，当然，正常情况下，加入内存检测是必要的，如果内存不可用，或者设置有问题，那么后续工作都是无法完成的。关于内存检测算法，可以参考詹荣开的《嵌入式

bootloader技术内幕》，主要分析了blob的内存检测算法，这也具有一定的通用性。vivi采取了类似的算法。这里需要理解的问题就是测试的时候为什么要采用0xaa和0x55。写成二进制就比较清晰了，它们正好是1和0交替，结合sdram的硬件特点，采用这样的值能检测出多种问题。我在用C8051F020写sdram检测时，就是采用了这样的内存检测算法。可以展开来分析研究。

·初始化调试指示灯（可选）

·初始化UART，作为调试口（可选择，这部分工作是可以放到stage 2来完成的，不过，如果对stage 1来进行调试，可以采用调试灯的方法，也可以初始化UART来完成调试信息的打印，可以参考ARM调试总结部分）。

·复制代码到sdram中。

·跳转到main，进入stage 2。在这里，詹荣开提到一种弹簧床的技术，实际实现比较简单。在vivi中，可以用如下方法：

```
@ jump to ram
    @ a technology about trampoline
    ldr r1, =on_the_ram
    add pc, r1, #0
    nop
    nop
1:
    b 1b

on_the_ram:
    @ setup by APCS
    ldr sp, DW_STACK_START @ setup stack pointer
    mov fp, #0 @ no previous frame, so fp=0
    mov a2, #0 @ set argv to NULL

    bl main @ call main

    mov pc, #FLASH_BASE @ otherwise, reboot
```

这里有几点需要说明，一是开始的几处nop，实际上是考虑到流水线断流问题而设计的。这部分我也理解的不是太清晰，可以参考毛德操的书来分析。但是可以按照自己想法做简单的修改，也就是不考虑流水线问题，直接

采用简单的ldr加载（至于为什么用ldr，而不是使用b，主要还是加载域和运行域的问题，参考前面总结），经过测试也没有问题。二是关于fp等的设置，这里主要是利用了APCS（ARM过程调用标准），具体可以参考相应的标准，同样的道理，如果把这三条去掉，运行也是没有问题的。加入这三条语句和去除这三条语句，在程序设计上，到底有什么不同，可能造成的微影响是什么，还有待研究。三是，如果调用出现问题，直接软复位，这个那个弹簧床技术本质一致，但是处理手段不同。当然，你也可以用b on_the_ram。

具体的每个阶段的工作在前面的基础实验中都做了详细的分析，具体可以参考那些总结部分，就不罗列在这里了。

至此，bootloader（vivi）第一阶段的分析就完成了。

vivi源代码最为详细分析(二)

现在进入bootloader之vivi分析的第二阶段，这部分使用C语言实现，部分代码采取内嵌汇编的方式。这里需要用到GNU GCC内嵌汇编的知识，这部分基础还没有具备，需要学习。

下面先按照流程进行分析。需要注意的是，此部分内容并非完全按照原版的vivi源代码，而是加入了自己的理解。另外，对非常简单、google出一片而且有分析正确的部分，在这里就简化了，不做详细分析，只是对网上没有分析到位而又影响理解的部分进行深入分析。我想，这部分内容应该是对《s3c2410完全开发》中vivi源代码分析部分的补充和完善。

stage 2：【init/main.c】

第二阶段的入口就是init/main.c，按照源代码的组织流程，根据模块化划分的原则，应该分为**8**个功能模块，源

代码注释以step区分，非常清晰。现在首先解决一个问题，就是关于main的形参。vivi源代码中对main的原型使用了：int main(int argc, char *argv[])的标准形式，在第一阶段的【arch/s3c2410/head.S】中，利用APCS设定了相应的入口参数，如下：

```
@ get read to call C functions
    ldr sp, DW_STACK_START @ setup stack pointer
    mov fp, #0 @ no previous frame, so fp=0
    mov a2, #0 @ set argv to NULL

    bl main @ call main
```

这里的**sp**、**fp**、**a2**都是**APCS**中名字，与之对应的寄存器分别为**R13**、**R11**、**R2**。这里理解的重点在于fp（frame pointer，帧指针），也就是栈帧指针。这是比较复杂的一个地方，对栈需要有深入的分析。搜集了一些资料，看了APCS标准，后续会把关于fp和APCS的部分单独拿出，总结成文。不管怎样，通过其入口地址的设置也可以看出，main是不需要入口地址的。那么，为了理解上的方便，不妨把main原型改为int main(void)，这样，相应的入口地址就不需要设置了。更改后的head.S对应部分如下：

```
@ jump to ram
    @ a technology about trampoline
    ldr pc, =on_the_ram

on_the_ram:
    bl main
    @ if main ever returns, reboot
    mov pc, #FLASH_BASE
```

清晰而且符合规则，前提是init/main.c中main原型修改为int main(void)。当然，对main还动了一些手术，现在把main的主流程部分放在这里，后面会对为什么如此改动详细说明。

```
int main(void)
{
	int ret;

	/*
	 * Step 1:
	 * Print Vivi version information
	 */
	putstr("/r/n");
	putstr(vivi_banner);
	reset_handler();
	/*
	 * Step 2:
	 * initialize board environment
	 */
	ret = board_init();
	if (ret) {
		putstr("Failed a board_init() procedure/r/n");
		error();
	}

	/*
	 * Step 3:
	 * MMU management
	 * When it's done, vivi is running on the ram and MMU is enabled.
	 */
	mem_map_init();
	mmu_init();
	putstr("Succeed memory mapping./r/n");

	/*
	 * Step 4:
	 * initialize the heap area
	 */
	ret = heap_init();
	if (ret) {
		putstr("Failed initailizing heap region/r/n");
		error();
```

```
    }

    /*
     * Step 5:
     * initialize the MTD device
     */
    ret = mtd_dev_init();

    /*
     * Step 6:
     * initialize the private data
     */
    init_priv_data();

    /*
     * Step 7:
     * initialize the humanmachine environment
     */
    misc();
    init_builtin_cmds();

    /*
     * Step 8:
     * boot kernel or step into vivi
     */
    boot_or_vivi();

    return 0;
}
```

（1）step 1：打印版本信息

这一部分其实是作为调试和增强人机交互行而用的，如果不用，对vivi的主要功能也不会产生影响。本来是最为简单的一个部分，但是实际上却是我理解上问题最多的一个部分，对这块动的手术也最多。事实上，这个部分的reset_handler存在bug。具体分析一下。

源代码step 1部分如下：

```
putstr("/r/n");
putstr(vivi_banner);

reset_handler();
```

打印的vivi_banner在【init/version.c】中，如下：

```
#include "version.h"
#include "compile.h"

const char *vivi_banner =
                "VIVI version " VIVI_RELEASE " (" VIVI_COMPILE_BY "@"
                VIVI_COMPILE_HOST ") (" VIVI_COMPILER ") " UTS_VERSION "/r/n";
```

vivi_banner就是字符串，中间有五个未知的宏，这是非常明显的。显然就是头文件中给出的。但是find，没有version.h。compile.h，所以现在就要知道version.h,compile.h是怎样生成的，在compile.h中一次定义了VIVI_COMPILE_BY，VIVI_COMPILE_HOST，VIVI_COMPILER ，UTS_VERSION 这四个宏，这在前面Makefile分析时也详细讲解过了，我们可以在顶层的Makefile中找到在编译时是如何自动生成version.h以及compile.h这两个头文件的：

include/version.h:

@echo /#define VIVI_RELEASE /"$(VIVIRELEASE)/" > .ver

@echo /#define VIVI_VERSION_CODE `expr $(VERSION) //* 65536 + $(PATCHLEVEL) //* 256 + $(SUBLEVEL)` >> .ver

@echo '#define VIVI_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))' >>.ver

@mv -f .ver $@

可以看到在version.h中定义了VIVI_RELEASE，VIVI_VERSION_CODE以及VIVI_VERSION（a,b,c）三个宏，同样在顶层Makefile中也定义了compile.h的生成规则，而且还把version.h合并入compile.h里面了。可以just for

fun，增加一些个性化的打印信息，比如，我的version.c修改如下：

```
#include "compile.h"

const char *vivi_banner =
                "/r/n/t^_^ Well done, boy! Go on -->/r/n"
                "VIVI version " VIVI_RELEASE " (" VIVI_COMPILE_BY "@"
                VIVI_COMPILE_HOST ") (" VIVI_COMPILER ") " UTS_VERSION "/r/n";
```

下面进入疑惑的reset_handler功能部分。作者的本意是利用**reset_handler()**实现软复位跟硬复位的处理。

```
【lib/reset_handle.c】
void
reset_handler(void)
{
  int pressed;

  pressed = is_pressed_pw_btn();

  if (pressed == PWBT_PRESS_LEVEL) {
    DPRINTK("HARD RESET/r/n");
    hard_reset_handle();
  } else {
    DPRINTK("SOFT RESET/r/n");
    soft_reset_handle();
  }
}
```

首先看一下is_pressed_pw_btn，按照函数字面意思，应该是判断电源复位键是否按下，如果按下，则证明是硬复位；如果没有检测到键按下，那么就是软复位。具体代码如下：

```
static int
is_pressed_pw_btn(void)
{
   return read_bt_status();
}

--> read_bt_status

static int
read_bt_status(void)
{
   ulong status;

   //status = ((GPLR & (1 << GPIO_PWBT)) >> GPIO_PWBT);

   status = ((PWBT_REG & (1 << PWBT_GPIO_NUM)) >> PWBT_GPIO_NUM);

   if (status)
      return HIGH;
   else
      return LOW;
}
```

可是，PWBT_REG是没有定义的，PWBT_GPIO_NUM也没有定义，也就是说，这个函数实际上是不可能编译通过的。从表面上分析，如同网上大部分讨论一样，我可以知道作者是什么意图，但是这段代码真正有效吗？从上面分析看，答案显然是这个reset_handle.c根本就是无效的。那么为什么vivi移植成功都没有注意这个问题，还只是按照表面意思分析代码呢？这个可以看reset_handle.h(在include文件夹中)的头文件。

```
#ifdef CONFIG_RESET_HANDLING
void reset_handler(void);
#else
#define reset_handler()    (void)(0)
#endif
```

很显然，在配置的时候，CONFIG_RESET_HANDLING是没有定义的，那么reset_handler()为空，也就是说这部分根本就是空代码，并没有实际执行功能。如果还不放心，那就做测试，如果把CONFIG_RESET_HANDLING选中（具体是把General setup部分的support reset handler选中），那么就会出现错误：

```
reset_handle.c: In function `read_bt_status':
reset_handle.c:31: `PWBT_REG' undeclared (first use in this function)
reset_handle.c:31: (Each undeclared identifier is reported only once
reset_handle.c:31: for each function it appears in.)
reset_handle.c:31: `PWBT_GPIO_NUM' undeclared (first use in this function)
reset_handle.c:28: warning: `status' might be used uninitialized in this function
reset_handle.c: In function `hard_reset_handle':
reset_handle.c:52: `USER_RAM_BASE' undeclared (first use in this function)
reset_handle.c:52: `USER_RAM_SIZE' undeclared (first use in this function)
reset_handle.c: In function `reset_handler':
reset_handle.c:68: `PWBT_PRESS_LEVEL' undeclared (first use in this function)
make[2]: *** [reset_handle.o] Error 1
make[2]: Leaving directory `/home/armlinux/embedded_Linux/s3c2410/bootloader/m-boot-1.0.0/lib'
make[1]: *** [first_rule] Error 2
make[1]: Leaving directory `/home/armlinux/embedded_Linux/s3c2410/bootloader/m-boot-1.0.0/lib
```

可见这部分功能是多余的。可以选择把这部分设置完全去掉，方法如下：

·【init/main.c】去掉行reset_handler();，去掉#include <reset_handle.h>。

·删除【lib/reset_handle.c】，删除【include/reset_handle.h】

·【arch/config.in】，删除行bool 'support reset handler' CONFIG_RESET_HANDLING，这样就彻底把此项配置部分也删除了。如果还有原来的默认配置文件，可以把# CONFIG_RESET_HANDLING is not set删除。

经过上面三步，就可以把reset handler功能去掉了。这些在你了解了vivi的配置机制后是很容易操作的，它们之间的关系并不复杂，就是一条链，顺着找就可以了。

我现在第一步想做的是把vivi进行“瘦身”，只需要完成在EDUKIT-III上从nand flash启动引导内核的功能就可以，从中也可以了解核心技术和主要流程。但是，在整个的软件架构上是保持不变的。如果我想增加功能，因为对这个软件架构熟悉了，所以很容易扩展，而且也容易自己重新做一个功能更好一些的bootloader。

（2）step 2：

主要是初始化GPIO。这个在前面实验中做过了，基本的思路和方法就是在把握好整个系统硬件资源的前提下，根据datasheet把所有的初始值设定，在这里利用这个函数就可以完成初始化了。我们在这里稍微分析下board_init()函数(arch/s3c2410/smdk.c)

```
int board_init(void)
{
 init_time();   //arch/s3c2410/proc.c中
 set_gpios(); //arch/s3c2410/smdk.c
 return 0;
}
```

先来看看init_time():

```
void init_time(void)
{
 TCFG0 = (TCFG0_DZONE(0) | TCFG0_PRE1(15) | TCFG0_PRE0(0));
}
```

再来看看set_gpios()

```
void set_gpios(void)
{
 GPACON  = vGPACON;
 GPBCON  = vGPBCON;
```

GPCCON = vGPCCON;

GPCUP   = vGPCUP;

GPDCON = vGPDCON;

GPDUP   = vGPDUP;

GPECON = vGPECON;

GPEUP   = vGPEUP;

GPFCON = vGPFCON;

GPFUP   = vGPFUP;

GPGCON = vGPGCON;

GPGUP   = vGPGUP;

GPHCON = vGPHCON;

GPHUP   = vGPHUP;

EXTINT0 = vEXTINT0;

EXTINT1 = vEXTINT1;

EXTINT2 = vEXTINT2;

}也就是设置各个GPIO口的控制寄存器，和上拉寄存器

（3）step 3：

MMU初始化。这部分在MMU基础实验中完成了。关于GNU GCC内嵌汇编部分还不是太清晰，还有待于在后续工作中加强。

mem_map_init(); //arch/s3c2410/mmu.c

mmu_init();

（4）step 4：

ret = heap_init(); //lib/heap.c

heap——堆，内存动态分配函数mmalloc就是从heap中划出一块空闲内存的，mfree则将动态分配的某块内存释放回heap中。

heap_init函数在SDRAM中指定了一块1M大小的内存作为heap(起始地址HEAP_BASE = 0x33e00000)，并在heap的开头定义了一个数据结构blockhead——事实上，heap就是使用一系列的blockhead数据结构来描述和操作的。每个blockhead数据结构对应着一块heap内存， 假设一个**blockhead**数据结构的存放位置为**A**，则它对应的可分配内存地址为**“A + sizeof(blockhead)”**到**“A + sizeof(blockhead) + size - 1”**。**(**因为**A**地址处存放了一个**blockhead**结构所以不能分配，从**A+sizeof (blockhead)**处开始，而**blockhead**结构体中的**size**定义了该结构控制的区域的大小，所以这块区域的尾地址也就是**A+sizeof**（**blockhead**）**+size -1,**后面的**blockhead**结构就要从这块的尾地址开始存放，这样依次存放，每个**blockhead**结构通过指针练成一个链表，这样利于分配**)**

堆初始化。堆与栈的区别已经比较清晰了，在动手分析vivi的过程中，更为明确了。在这里，实际上就是实现动态内存分配策略。具体实现部分在【lib/heap.c】。因为以前自己没有写过动态内存分配，所以要仔细分析这部分是如何实现的。这部分的工作主要有两个：一是分析封装调试宏的技巧和**printk**的实现方法，这部分在这里还是挺重要的。二是**heap**基本的原理是什么？具体如何实现？我们暂且来分析下**heap_init**函数：

```
int heap_init(void)
{
 return mmalloc_init((unsigned char *)(HEAP_BASE), HEAP_SIZE);
}
```

我们在smdk2410.h中发现上面两个常量的定义：

```
#define HEAP_SIZE  SZ_1M //堆的分配大小为1M
#define HEAP_BASE  (VIVI_RAM_BASE - HEAP_SIZE)//我们可以看到一是高端顺序存放二是堆空间的分配挨着sdram中VIVI的分区
typedef struct blockhead_t {
```

```
 Int32 signature;  //固定为BLOCKHEAD_SIGNATURE
 Bool allocated;  //此区域是否已经分配出去：0-N，1-Y
 unsigned long size;  //此区域大小
 struct blockhead_t *next;  //链表指针
 struct blockhead_t *prev;  //链表指针
} blockhead;
```

vivi对heap的操作比较简单，vivi中有一个全局变量static blockhead *gHeapBase，它是heap的链表头指针，通过它可以遍历所有blockhead数据结构。假设需要动态申请一块sizeA大小的内存，则mmalloc函数从gHeapBase开始搜索

blockhead数据结构，如果发现某个blockhead满足：

a. allocated = 0 //表示未分配

b. size > sizeA,

则找到了合适的blockhead，于是进行如下操作：

a. allocated设为1

b. 如果size – sizeA > sizeof(blockhead)，则将剩下的内存组织成一个新的blockhead，放入链表中

c. 返回分配的内存的首地址

分析分析动态分配内存函数mmalloc函数：

```
void *mmalloc(unsigned long size)
{
 blockhead *blockptr = gHeapBase; //将heap的链表头地址赋值，从头开始查找
 blockhead *newblock;
 Bool compacted = FALSE;
 size = (size+7)&~7; /* unsigned long align the size */
 DPRINTK("malloc(): size = 0x%08lx/n", size);
```

```
while (blockptr != NULL) {
 if (blockptr->allocated == FALSE) { //判断是否被分配出去
  if (blockptr->size >= size) { //判断这个结构控制的区域够不够分配的大小
   blockptr->allocated=TRUE; //将这个区域定义为已经分出去的标志
   if ((blockptr->size - size) > sizeof(blockhead)) { //如果该区域除开分配的内存大小，余下的比结构体大小还要大，就重新创建一个新的blockhead结构体链接到该结构体后面
    newblock = (blockhead *)((unsigned char *)(blockptr) + sizeof(blockhead) + size);
    newblock->signature = BLOCKHEAD_SIGNATURE;
    newblock->prev = blockptr;
    newblock->next = blockptr->next; //链表操作
    newblock->size = blockptr->size - size - sizeof(blockhead);//这时候新建的结构体可分配的大小就是余下的内存减去新建结构体大小
    newblock->allocated = FALSE;
    blockptr->next = newblock;
    blockptr->size = size;
   } else {
   }
   break;
  } else {
   if ((blockptr->next == NULL) && (compacted == FALSE)) {
    if (compact_heap()) {
     compacted=TRUE;
     blockptr = gHeapBase;
     continue;
    }
   }
```

```
  }
 }
  blockptr = blockptr->next;
 }
 DPRINTK("malloc(): returning blockptr = 0x%08lx/n", blockptr);
 if (blockptr == NULL)
  printk("Error: malloc(), out of storage. size = 0x%x/n", size);
 return (blockptr != NULL) ? ((unsigned char *)(blockptr)+sizeof(blockhead)) : NULL;
}
```

再来看看动态内存的释放mfree函数：

```
void mfree(void *block) {
 blockhead *blockptr;
 if (block == NULL) return;
 blockptr = (blockhead *)((unsigned char *)(block) - sizeof(blockhead));
 if (blockptr->signature != BLOCKHEAD_SIGNATURE) return;
 blockptr->allocated=FALSE;//主要就是这句，将该区域的分配标志置为没分出去，就表示回收啦
 return;
}
```

释放内存的操作更简单，直接将要释放的内存对应的blockhead数据结构的allocated设为0即可。

```
static blockhead *gHeapBase = NULL; //分配堆空间的起始地址结构
static inline int mmalloc_init(unsigned char *heap, unsigned long size)
{
 if (gHeapBase != NULL) return -1;
 DPRINTK("malloc_init(): initialize heap area at 0x%08lx, size = 0x%08lx/n", heap, size);//#define DPRINTK
(args...) printk(##args)
```

```
 gHeapBase = (blockhead *)(heap); //在堆的分配起始地址处定义一个blockhead的结构体
 gHeapBase->allocated=FALSE;
 gHeapBase->signature=BLOCKHEAD_SIGNATURE;
 gHeapBase->next=NULL;
 gHeapBase->prev=NULL;
 gHeapBase->size = size - sizeof(blockhead);
 return 0;
}
```

分配heap区域后，内存划分情况如下：

0x34000000-0x33f00000这1MB的空间存放的是vivi的stage1和stage2,0x33f00000-0x33e00000这1MB的空间就是堆空间。

下面首先进行第一个重点分析。关于调试手段，在分析ARM的基本调试手段时也提到过，使用串口打印调试信息是一个非常有效且常用的手段，vivi中采取的也是这种方式。当然，如果你只是实现最为简单的打印字符串等，那么初始化串口后，封装一个基本的输出函数就可以了。但是，这个基本函数的功能是非常有限的。我们在Linux用的printk则要强大好用的多。vivi的思想就是把Linux kernel的printk拿过来，稍微裁减一下（因为vivi不需要打印级别，但是需要打印手段的多样化）。这样，自己的工作量并不大，但是调试手段则要完善得多了。在这里，关于printk的代码细节不作为重点，vivi也只是借用了Linux kernel的printk的实现，并做了简单的修改，把console映射到了串口0上。

手头上暂时有linux-2.4.18的内核，暂时以这个为依据来探讨vivi中printk的实现。

·复制【lib/vsprintf.c】到vivi的lib目录下，更改名称为printk.c。然后只保留vsnprintf，及其用到的number函数、skip_atoi函数。skip_atoi中用到了isdigit，所以把【include/linux/ctype.h】复制到vivi的include目录下。另外，还要用到do_div和strnlen两个函数。其中do_div是宏，在【include/asm-arm/div64.h】中实现，直接复制到vivi的include文件夹中。strnlen应该在string.c中实现，可以从【lib/string.c】复制然后添加到vivi的lib下的string.c文件中，最后把声明加到include下的vivi_string.h中。这样，printk需要的基础部分就具备了。

·复制【kernel/printk.c】，然后把printk的实现部分摘出来，去掉打印等级等功能，参考vivi的就可以封装起来了。

可见，vivi中的printk只是把Linux kernel中的代码拿过来，做了及其少量的修改。我现在已经重现了这个过程，并且对整个vivi工程文件做出一些修改，编译下载，测试功能稳定。

实现了printk，往往需要封装一个调试宏。在【lib/heap.c】中和其他一些文件中，调试宏都是这样的形式：

```
#ifdef DEBUG_HEAP
#define DPRINTK(args...)    printk(##args)
#else
#define DPRINTK(args...)
#endif
```

分析这样是不妥的。原因就在于stdarg.h（注意，这里只需要定义头文件<stdarg.h>就可以把变长参数表的功能引入了。该头文件的实现因为不同的机器而不同，但是提供的接口是一致的。具体可以看《The C Programming Language》。但是一个细节就是你如果find，会找不到stdarg.h这个头文件，原因就是gcc直接把stdarg.h放到编译器里。）规定的变长参数表必须至少包括一个有名参数，**va_start**会将最后一个有名参数作为起点。这里封装的printk缺少了有名参数，这里可以做测试。测试工程如下：

通过这个测试手段，发现如果选择方式**#define DPRINTK(fmt, args...) printk(fmt, ##args)**，那么结果如下：

```
[armlinux@lqm printk_test]$ make
gcc -Wall -g -O2 -c -o printk.o printk.c
gcc -Wall -g -O2 -c -o test.o test.c
gcc -Wall -g -O2 printk.o test.o -o test
[armlinux@lqm printk_test]$ ls
Makefile printk.c printk.h printk.o test test.c test.o
[armlinux@lqm printk_test]$ ./test
test: i = 5, j = 10
```

如果选择**#define DPRINTK(args...)　　printk(##args)**，那么结果如下：

```
[armlinux@lqm printk_test]$ make
gcc -Wall -g -O2 -c -o printk.o printk.c
gcc -Wall -g -O2 -c -o test.o test.c
test.c:23:47: warning: pasting "(" and ""test: i = %d, j = %d/n"" does not give a valid preprocessing token
gcc -Wall -g -O2 printk.o test.o -o test
[armlinux@lqm printk_test]$ ls
Makefile printk.c printk.h printk.o test test.c test.o
[armlinux@lqm printk_test]$ ./test
test: i = 5, j = 10
```

可见第二种方式是不合适的。于是修改如下：

```
#ifdef DEBUG_HEAP
#define DPRINTK(fmt, args...) printk(fmt, ##args)
#else
#define DPRINTK(fmt, args...)
#endif
```

这样的调试宏就没有问题了。也算是宏的一个小技巧吧，在Linux内核中查看，可以看到不少的printk的宏封装都是这样的。

vivi源代码最为详细分析(三)

step 5：

MTD设备初始化。

关于什么是MTD，为什么要使用MTD，MTD技术的架构是什么，等等，可以参考《Linux MTD源代码分析》（作者：Jim Zeus，2002-04-29）。这份文档的参考价值比较大，猜想作者在当时可能研究了很长时间，毕竟2002年的时候资料还比较缺乏。当然，因为完全分析透彻，方方面面都点透，这份文档还是没有做到。

在分析代码前先介绍一下MTD(Memory Technology Device)相关的技术。在inux系统中，我们通常会用到不同的存储设备，特别是FLASH设备。为了在使用新的存储设备时，我们能更简便地提供它的驱动程序，在上层应用和硬件驱动的中间，抽象出MTD设备层。驱动层不必关心存储的数据格式如何，比如是FAT32、ETX2还是FFS2或其它。它仅仅提供一些简单的接口，比如读写、擦除及查询。如何组织数据，则是上层应用的事情。

MTD层将驱动层提供的函数封装起来，向上层提供统一的接口。这样，上层即可专注于文件系统的实现，而不必关心存储设备的具体操作。

在我们即将看到的代码中，使用mtd_info数据结构表示一个MTD设备，使用nand_chip数据结构表示一个nand flash芯片。在mtd_info结构中，对nand_flash结构作了封装，向上层提供统一的接口。比如，它根据nand_flash提供的read_data(读一个字节)、read_addr(发送要读的扇区的地址)等函数，构造了一个通用的读函数read，将此函数的指针作为自己的一个成员。而上层要读写flash时，执行mtd_info中的read、write函数即可。

vivi采用Linux kernel的架构，所以把Linux kernel的MTD子系统借用过来了，做了一些裁减。可以简单地看成：Flash硬件驱动层和MTD设备层。这样，最终以抽象的统一的接口向vivi提供。

还是以nand flash启动这个情景为主线，对MTD初始化流程进行分析。下面先从入口开始。

```
ret = mtd_dev_init();
```

利用source insight跟踪，看一下此函数的接口定义部分：

```
/*
 * VIVI Interfaces
 */
#ifdef CONFIG_MTD
int write_to_flash(loff_t ofs, size_t len, const u_char *buf, int flag);
int mtd_dev_init(void);
#else
#define write_to_flash(a, b, c, d) (int)(1)
#define mtd_dev_init() (int)(1)
#endif
```

可见，vivi在配置的时候是必须配置MTD功能部分的。如果不配置MTD，那么CONFIG_MTD就不存在定义。由

此导致写flash的动作实际上是没有的。也就是说，无法完成写flash的动作。当然，在这里可以做测试，就是使用MTD子系统的vivi把分区等都设置好。然后重新编译一下vivi，把mtd功能去除，做简单的修改（把bon_cmd部分从【lib/command.c】中去掉，否则编译不通过），生成大小为35152字节。给开发板重新上电，利用老的vivi烧写nand flash的vivi分区，完成后做一下reset，于是没有MTD功能的vivi就跑起来了。但是，这样的bootloader仅仅适合于最终的产品阶段，不适合开发，没什么太大的价值。有兴趣倒是可以据此研究一下配置部分，整个引导时间相应的缩短。

下面【drivers/mtd/mtdcore.c】，看看mtd_dev_init函数，核心部分就是调用mtd_init函数（【drivers/mtd/maps/s3c2410_flash.c】）。

```
int mtd_init(void)
{
   int ret;

#ifdef CONFIG_MTD_CFI
   ret = cfi_init();
#endif
#ifdef CONFIG_MTD_SMC
   ret = smc_init();
#endif
#ifdef CONFIG_S3C2410_AMD_BOOT
   ret = amd_init();
#endif
```

可见，vivi现在支持三种类型的存储接口，一种是CFI，也就是Intel发起的一个flash的接口标准，主要就是intel的nor flash系列；一种是smc，智能卡系列接口，nand flash就是通过这个接口实现读写的；一种是AMD的flash系列。选择什么启动方式，就要选择相应的配置项。

核心部分根据配置应该调用smc_init函数。-->【drivers/mtd/maps/s3c2410_flash.c】。这里最为核心的就是两

个数据结构，一个是mtd_info，位于【include/mtd/mtd.h】，如下：

```
struct mtd_info {
 u_char type;
 u_int32_t flags;
 u_int32_t size;  // Total size of the MTD
 /* "Major" erase size for the device. Na鬯e users may take this
  * to be the only erase size available, or may use the more detailed
  * information below if they desire
  */
 u_int32_t erasesize;
 u_int32_t oobblock;  // Size of OOB blocks (e.g. 512)
 u_int32_t oobsize;   // Amount of OOB data per block (e.g. 16)
 u_int32_t ecctype;
 u_int32_t eccsize;
 // Kernel-only stuff starts here.
 char *name;
 int index;
 /* Data for variable erase regions. If numeraseregions is zero,
  * it means that the whole device has erasesize as given above.
  */
 int numeraseregions;
 struct mtd_erase_region_info *eraseregions;
 /* This really shouldn't be here. It can go away in 2.5 */
 u_int32_t bank_size;
 struct module *module;
```

```
    int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
    /* This stuff for eXecute-In-Place */
    int (*point) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char **mtdbuf);
    /* We probably shouldn't allow XIP if the unpoint isn't a NULL */
    void (*unpoint) (struct mtd_info *mtd, u_char * addr);

    int (*read) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*write) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const u_char *buf);
    int (*read_ecc) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf, u_char *eccbuf);
    int (*write_ecc) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const u_char *buf, u_char *eccbuf);
    int (*read_oob) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*write_oob) (struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const u_char *buf);
    /*
     * Methods to access the protection register area, present in some
     * flash devices. The user data is one time programmable but the
     * factory data is read only.
     */
    int (*read_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    int (*read_fact_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);
    /* This function is not yet implemented */
    int (*write_user_prot_reg) (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf);

    /* Chip-supported device locking */
    int (*lock) (struct mtd_info *mtd, loff_t ofs, size_t len);
    int (*unlock) (struct mtd_info *mtd, loff_t ofs, size_t len);
    void *priv;
};
```

mtd_info是表示MTD设备的结构，每个分区也被表示为一个mtd_info，如果有两个MTD设备，每个设备有三个分区，那么在系统中就一共有6个mtd_info结构。关于mtd_info，在《Linux MTD源代码分析》中讲解非常透彻，不过需要注意的是，在vivi的实现中没有使用mtd_table，另外priv指向的是nand_chip，这些都是与Linux下不同的地方，主要是为了简化。另一个是**nand_chip**，这个结构则包含了**nand flash**的所有信息。

```
struct nand_chip {
#ifdef CONFIG_MTD_NANDY
 void (*hwcontrol)(int cmd);
 void (*write_cmd)(u_char val);
 void (*write_addr)(u_char val);
 u_char (*read_data)(void);
 void (*write_data)(u_char val);
 void (*wait_for_ready)(void);
 /*spinlock_t chip_lock;*/
 /*wait_queue_head_t wq;*/
 /*nand_state_t state;*/
 int page_shift;
 u_char *data_buf;
 u_char *data_cache;
 int cache_page;
 struct nand_smc_dev *dev;
 u_char spare[SMC_OOB_SIZE];
#else /* CONFIG_MTD_NANDY */
 unsigned long IO_ADDR_R;
 unsigned long IO_ADDR_W;
```

```
 void (*hwcontrol)(int cmd);
 int (*dev_ready)(void);
 int chip_delay;
 /*spinlock_t chip_lock;*/
 /*wait_queue_head_t wq;*/
 /*nand_state_t state;*/
 int page_shift;
 u_char *data_buf;
 u_char *data_cache;
 int cache_page;
#ifdef CONFIG_MTD_NAND_ECC
 u_char ecc_code_buf[6];
 u_char reserved[2];
#endif
#endif /* CONFIG_MTD_NANDY */
};
```

所谓的初始化，其实就是填充处理上述两个数据结构的过程。填充完毕之后，后续的工作都会基于此展开。下面开始看smc_init的代码。

```
mymtd = mmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip));
this = (struct nand_chip *)(&mymtd[1]);
```

在这里，第一句参考前面**heap**的实现代码，重点看第二句代码。这句代码是有一定的技巧性，但是也存在着很大的风险。其中，**mymtd**是指向**struct mtd_info**的指针，那么**mymtd[1]**实际上是等效于***(mymtd + 1)**的数学计算模式，注意**mymtd**并非数组，这里仅仅利用了编译器翻译的特点。对于指针而言，加**1**实际上增加的指针对应

类型的值，在这里地址实际上增加了**sizeof(struct mtd_info)**，因为前面分配了两块连续的地址空间，所以**&(*(mymtd + 1))**实际上就是**mtd_info**数据结构结束的下一个地址，然后实现强制转换，于是**this**就成为了**nand_chip**的入口指针了。但是，这里必须要把握好，因为这个地方是不会进行内存的检查的，也就是说，如果你使用了**mymtd[2]**，那么仍然按照上述公式解析，虽然可以运算，可是就是明显的指针泄漏了，可能会出现意料不到的结果。写了一个测试程序，对这点进行了探讨，要小心内存问题。

了解清楚了，mymtd指向mtd_info的入口，this指向nand_chip的入口。

```
memset((char *)mymtd, 0, sizeof(struct mtd_info));
memset((char *)this, 0, sizeof(struct nand_chip));
```

```
mymtd->priv = this;
```

上述代码首先初始化这两个结构体，即均为0.然后利用priv把二者联系起来，也就是mymtd通过其成员priv指向this，那么mymtd中的抽闲操作函数，比如read、write等，真正的是通过this来实现的。很明显，this的实现部分属于**flash**硬件驱动层，而mymtd部分则属于MTD设备层，二者的联系就是通过成员priv实现的。

接下来首先是初始化nand flash设备，这跟前面的基础实验一致。

```
/* set NAND Flash controller */
nfconf = NFCONF;
/* NAND Flash controller enable */
nfconf |= NFCONF_FCTRL_EN;

/* Set flash memory timing */
nfconf &= ~NFCONF_TWRPH1;    /* 0x0 */
nfconf |= NFCONF_TWRPH0_3;   /* 0x3 */
nfconf &= ~NFCONF_TACLS;     /* 0x0 */
```

```
NFCONF = nfconf;
```

然后填充nand flash的数据结构的一个实例this，分成了两个部分，nand flash基本操作函数成员的初始化、其余信息的填写。

```
/* Set address of NAND IO lines */
this->hwcontrol = smc_hwcontrol;
this->write_cmd = write_cmd;
this->write_addr = write_addr;
this->read_data = read_data;
this->write_data = write_data;
this->wait_for_ready = wait_for_ready;

/* Chip Enable -> RESET -> Wait for Ready -> Chip Disable */
this->hwcontrol(NAND_CTL_SETNCE);
this->write_cmd(NAND_CMD_RESET);
this->wait_for_ready();
this->hwcontrol(NAND_CTL_CLRNCE);

smc_insert(this);
```

上面这些都不难理解，感觉在结构体设计上还是比较出色的，把成员和相应的操作封装起来，面向对象的一种方法。下面看smc_insert，根据刚才填充的nand_flash结构，构造mtd_info结构无非还是按照结构体填写相应的信息，细节部分就不深入探讨了。

```
inline int
smc_insert(struct nand_chip *this) {
   /* Scan to find existance of the device */
   if (smc_scan(mymtd)) {
      return -ENXIO;
   }
   /* Allocate memory for internal data buffer */
   this->data_buf = mmalloc(sizeof(u_char) *
         (mymtd->oobblock + mymtd->oobsize));

   if (!this->data_buf) {
      printk("Unable to allocate NAND data buffer for S3C2410./n");
      this->data_buf = NULL;
      return -ENOMEM;
   }

   return 0;
}
```

第一部分扫描填充mymtd数据结构。后面主要用于nand flash的oob缓冲处理。具体部分可以参考《s3c2410完全开发》。我们先来看看smc_scan函数的执行(drivers/mtd/nand/smc_core.c这个文件中包含的是nand flash中绝大多数真正进行操作的函数)：

```
int smc_scan(struct mtd_info *mtd)
{
 int i, nand_maf_id, nand_dev_id;  //定义flash的厂家ID和设备ID
 struct nand_chip *this = mtd->priv; //获得与mtd设备相联系的真正的flash设备结构
 /* Select the device */
 nand_select(); //#define nand_select() this->hwcontrol(NAND_CTL_SETNCE);  nand_command(mtd,
NAND_CMD_RESET, -1, -1);   udelay(10);这三句代码和我们之前在nand章节中讲解的片选flash是一个意思,
先将使能nand的那位置1也就是片选nand，然后想nand发送reset命令，然后等待一段时间。
 /* Send the command for reading device ID */
```

```
 nand_command(mtd, NAND_CMD_READID, 0x00, -1);//向nand发送读ID的命令
 this->wait_for_ready(); //等待nand结果数据准备好
 /* Read manufacturer and device IDs */
 nand_maf_id = this->read_data();//nand数据准备好后，通过read_data可以相继的读出厂家ID和设备ID
 nand_dev_id = this->read_data();
 /* Print and sotre flash device information */
 for (i = 0; nand_flash_ids[i].name != NULL; i++) { //在数组nand_flash_ids中查找与ID相符合的项，可以看到下面对数组说明
  if (nand_maf_id == nand_flash_ids[i].manufacture_id &&
      nand_dev_id == nand_flash_ids[i].model_id) {
#ifdef USE_256BYTE_NAND_FLASH
   if (!mtd->size) {      //下面都是根据在nand_flash_ids数组中找到相符的项，然后从对应的nand_flash_dev结构体中取出对应的属性值来填充mtd_info结构
    mtd->name = nand_flash_ids[i].name;
    mtd->erasesize = nand_flash_ids[i].erasesize;
    mtd->size = (1 << nand_flash_ids[i].chipshift); //
    mtd->eccsize = 256;
    if (nand_flash_ids[i].page256) {
     mtd->oobblock = 256;
     mtd->oobsize = 8;
     this->page_shift = 8;
    } else {
     mtd->oobblock = 512;
     mtd->oobsize = 16;
     this->page_shift = 9;
```

```
    }
    this->dev = &nand_smc_info[GET_DI_NUM(nand_flash_ids[i].chipshift)];
   }
#else
   if (!(mtd->size) && !(nand_flash_ids[i].page256)) {
    mtd->name = nand_flash_ids[i].name;
    mtd->erasesize = nand_flash_ids[i].erasesize;
    mtd->size = (1 << nand_flash_ids[i].chipshift);
    mtd->eccsize = 256;
    mtd->oobblock = 512;
    mtd->oobsize = 16;
    this->page_shift = 9;
    this->dev = &nand_smc_info[GET_DI_NUM(nand_flash_ids[i].chipshift)];
   }
#endif
   printk("NAND device: Manufacture ID:" /
    " 0x%02x, Chip ID: 0x%02x (%s)/n",
    nand_maf_id, nand_dev_id, mtd->name);
   break;
  }
 }
 /* De-select the device */
 nand_deselect();  //在对nand flash操作完后，需要禁止nand flash#define nand_deselect() this->hwcontrol
(NAND_CTL_CLRNCE);也就是将对应的位置0
 /* Print warning message for no device */
 if (!mtd->size) {
```

```
  printk("No NAND device found!!!/n");
  return 1;
 }
 /* Fill in remaining MTD driver data */
 mtd->type = MTD_NANDFLASH;
 mtd->flags = MTD_CAP_NANDFLASH | MTD_ECC; //填充mtd_info结构体中的函数指针，这些函数大多都是在
smc_core.c定义
 mtd->module = NULL;
 mtd->ecctype = MTD_ECC_SW;
 mtd->erase = nand_erase;
 mtd->point = NULL;
 mtd->unpoint = NULL;
 mtd->read = nand_read;
 mtd->write = nand_write;
 mtd->read_ecc = nand_read_ecc;
 mtd->write_ecc = nand_write_ecc;
 mtd->read_oob = nand_read_oob;
 mtd->write_oob = nand_write_oob;
 mtd->lock = NULL;
 mtd->unlock = NULL;
 /* Return happy */
 return 0;
}
```

这里重点是学习一种结构体的构造技巧。

首先构造一级数据结构，表示抽象实体。例如：

```
struct nand_flash_dev {
   char * name; //设备名
   int manufacture_id; //flash的厂家ID
   int model_id; //flash的设备ID
   int chipshift; //片选移位数，用来构建flash的大小
   char page256; //
   char pageadrlen;
   unsigned long erasesize;
};
```

然后构造实例集合，表现形式就是一个大的数组,nand_flash_ids是nand_flash_dev结构的数组，里面存放的是世界上比较常用的nand flash型号的一些特性。

```
static struct nand_flash_dev nand_flash_ids[] = {
   {"Toshiba TC5816BDC", NAND_MFR_TOSHIBA, 0x64, 21, 1, 2, 0x1000},   // 2Mb 5V
   ... ....
   {"Samsung K9D1G08V0M", NAND_MFR_SAMSUNG, 0x79, 27, 0, 3, 0x4000},   // 128Mb

   {NULL,}
};
```

执行完**mtd_dev_init**后，我们得到了一个**mtd_info**结构的全局变量**(mymtd**指向它**)**，以后对**nand flash**的操作，直接通过**mymtd**提供的接口进行。

这样修改扩展等等后续的操作就简便多了。抽象的能力及其训练在读代码的时候是可以很好的学习的，在vivi中，多处都采用了这种设计原则，应该掌握并利用。

step 6：

此部分的功能是把vivi可能用到的所有私有参数都放在预先规划的内存区域，大小为48K，基地址为0x33df0000。在内存的分配示意图方面，《s3c2410完全开发》已经比较详尽，就不放在这里了。到此为止，vivi作为bootloader的三大核心任务：initialise various devices, and eventually call the Linux kernel,passing information to the kernel.，现在只是完成第一方面的工作，设备初始化基本完成，实际上step 6是为启动**Linux**内核和传递参数做准备的，把**vivi**的私有信息，内核启动参数，**mtd**分区信息等都放到特定的内存区域，等待后面两个重要工作使用（在step 8完成，后面的step 7也是为step 8服务的）。这**48K**区域分为三个组成部分：**MTD**参数、**vivi parameter**、**Linux**启动命令。每块的具体内容框架一致，以vivi param tlb这个情景为主线进行分析：

入口：

```
init_priv_data();
```

进入【lib/priv_data/rw.c】--init_priv_data()

```
int
init_priv_data(void)
{
   int ret_def;
#ifdef CONFIG_PARSE_PRIV_DATA
   int ret_saved;
#endif
   ret_def = get_default_priv_data();
#ifdef CONFIG_PARSE_PRIV_DATA
   ret_saved = load_saved_priv_data();
   if (ret_def && ret_saved) {
      printk("Could not found vivi parameters./n");
      return -1;
   } else if (ret_saved && !ret_def) {
```

```
        printk("Could not found stored vivi parameters.");
        printk(" Use default vivi parameters./n");
    } else {
        printk("Found saved vivi parameters./n");
    }
#else
    if (ret_def) {
        printk("Could not found vivi parameters/n");
        return -1;
    } else {
        printk("Found default vivi parameters/n");
    }
#endif
    return 0;
```

此函数将启动内核的命令参数取出，存放在内存特定的位置中。这些参数来源有两个：vivi预设的默认参数，用户设置的参数(用户设置的参数存放在nand flash上，也就是我们进入到vivi控制端界面利用setenv来进行设置的参数，主要是以这个为主)。**init_priv_data**先读出默认参数，存放在**“VIVI_PRIV_RAM_BASE”**开始的内存上；然后读取用户参数，若成功则用用户参数覆盖默认参数，否则使用默认参数。

init_priv_data函数分别调用 **get_default_priv_data** 函 数 和**load_saved_priv_data**函数来读取默认参数和用户参数。这些参数分为3类：

a. vivi自身使用的一些参数，比如传输文件时的使用的协议等

b. linux启动命令

c. nand flash的分区参数

下面分为两步：首先读取默认设置到特定的内存区域，然后读取nand flash的param区域的信息，如果读取成功，就覆盖掉前面的默认设置。首先看第一步，get_default_priv_data--get_default_param_tlb-->

```
static inline int get_default_priv_data(void)
{
 if (get_default_param_tlb()) //获取默认的参数表
  return NO_DEFAULT_PARAM;
 if (get_default_linux_cmd()) //获取到在vivi设置的linux启动命令
  return NO_DEFAULT_LINUXCMD;
 if (get_default_mtd_partition()) //这步很重要，主要用来获取到nand flash分区表
  return NO_DEFAULT_MTDPART;
 return 0;
}
int get_default_param_tlb(void)
{
   char *src = (char *)&default_vivi_parameters;//这个默认的参数表也就是在程序中自定义的vivi_parameter_t数组，每个参数都用这样一个vivi_parameter_t来表示，default_vivi_parameters则为默认参数数组名，可以参考下面
   char *dst = (char *)(VIVI_PRIV_RAM_BASE + PARAMETER_TLB_OFFSET); //48KB的参数区域从下到上依次是16KB的mtd参数信息，16KB的参数表，16KB的linux启动命令，所以这里PARAMETER_TLB_OFFSET=16KB
   int num = default_nb_params;

   if (src == NULL) return -1;

   /*printk("number of vivi parameters = %d/n", num); */
   *(nb_params) = num;

   //参数表的长度不可以超过预设内存的大小，可以看到这里每个参数都是由vivi_parameter_t结构体来定义的，这里总共有num个参数，也就对应有num个结构体
   if ((sizeof(vivi_parameter_t)*num) > PARAMETER_TLB_SIZE) {
      printk("Error: too large partition table/n");
      return -1;
   }
   //首先复制magic number
   memcpy(dst, vivi_param_magic, 8);
   //预留下8个字节作为扩展
   dst += 16;
   //复制真正的parameter
   memcpy(dst, src, (sizeof(vivi_parameter_t)*num));
   return 0;
```

```
}
```

内存的入口地址为**VIVI_PRIV_RAM_BASE+PARAMETER_TLB_OFFSET**，开始的**8**个字节放**magic number**，这里**vivi**定义为**“VIVIPARA”**，后面空下**8**个字节，留作扩展，从第17个字节开始放置真正的param。这里用到了多处技巧，第一处就是上面刚刚介绍过的数据结构构造技巧，这里的vivi_parameter_t就是一级数据结构：

```
typedef struct parameter {
   char name[MAX_PARAM_NAME];
   param_value_t value;
   void (*update_func)(param_value_t value);
} vivi_parameter_t;
```

利用其构造了默认的成员表(下面就是在程序中定义的默认参数表，传递给内核启动)：

```
vivi_parameter_t default_vivi_parameters[] = {
   { "mach_type",          MACH_TYPE,    NULL },
   { "media_type",         MT_S3C2410,   NULL },
   { "boot_mem_base",      0x30000000,   NULL },
   { "baudrate",           UART_BAUD_RATE,   NULL },
   { "xmodem_one_nak",     0,      NULL },
   { "xmodem_initial_timeout",   300000,     NULL },
   { "xmodem_timeout",     1000000,    NULL },
   { "ymodem_initial_timeout",   1500000,    NULL },
   { "boot_delay",         0x1000000,    NULL }
};
```

我们这时就可以很清楚的看到param show列出的配置参数了。

另外一个技巧就是利用宏计算数组长度。

```
int default_nb_params = ARRAY_SIZE(default_vivi_parameters);
```

其中ARRAY_SIZE为：

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

这是从Linux kernel中拿来的，也是值得学习和利用的地方。

我们再来看看get_default_linux_cmd()用来获取vivi中设置的linux启动命令：

```
int get_default_linux_cmd(void)
{
 char *src = linux_cmd;
 char *dst = (char *)(VIVI_PRIV_RAM_BASE + LINUX_CMD_OFFSET);
 if (src == NULL) return -1;
 memcpy((char *)dst, (char *)linux_cmd_magic, 8);
 dst += 8;
 memcpy(dst, src, (strlen(src) + 1));
 return 0;
}
```

其实这三个函数大部分都是一样的，只不过赋值的源地址指向不同而已，在这里指向的是linux_cmd，也就是定义启动命令指针，然后将其拷贝到VIVI_PRIV_RAM_BASE + LINUX_CMD_OFFSET处，在这里linux_CMD_OFFSET为32KB，同样也是先拷贝magic number，然后留下8个字节扩展。可以看看linux_cmd的定义（这个定义是在arch/s3c2410/smdk.c中定义）：

char linux_cmd[] = "noinitrd root=/dev/bon/2 init=/linuxrc console=ttyS0";这个说白了也不是什么启动命令，也是传递给**linux**内核启动时候的参数，也就相当于**vivi**中的**bootargs**参数。

主要来看看很重要的**nand flash**分区信息的获取**get_default_mtd_partition()**：

具体的函数就不用分析啦，因为跟上面分析的两个函数一样，只是默认分区信息的源地址不同而已

char *src_parts = (char *)&default_mtd_partitions;这里看到将分区的信息存放在了**default_mtd_partitions**数组中，同样在**smdk.c**中可以找到这个数组的定义：

```
#ifdef CONFIG_S3C2410_NAND_BOOT
mtd_partition_t default_mtd_partitions[] = {
 {
  name:  "vivi",
  offset:  0,
  size:  0x00020000,
  flag:  0
 }, {
  name:  "param",
  offset:  0x00020000,
  size:  0x00010000,
  flag:  0
 }, {
  name:  "kernel",
  offset:  0x00030000,
  size:  0x001d0000,  // 2M sector
  flag:  0
 }, {
  name:  "root",
  offset:  0x00200000,
  size:  0x03000000,
```

```
  flag:  MF_BONFS
 }
};
```

在这里在flash分为vivi区，参数区，内核区，文件系统区，我们在vivi期间会将对应的内核，文件系统都会放到指定的区域，然后将信息传递给内核，这样内核在启动时会知道从哪地方加载内核，和文件系统。

同样我们也可以来看看，vivi是怎样去nand上读取用户设置的参数信息的:（下面的宏变量都是在smdk2410.h中定义）

```
int load_saved_priv_data(void)
{
 char *buf = (char *)(DRAM_BASE);
 char *dst = (char *)(VIVI_PRIV_RAM_BASE);
 if (read_saved_priv_data_blk(buf)) {
  printk("invalid (saved) parameter block/n");
  return -1;
 }
 /* load parameter table */
 if (strncmp((buf + PARAMETER_TLB_OFFSET), vivi_param_magic, 8) != 0)
  return WRONG_MAGIC_PARAM;
 memcpy(dst + PARAMETER_TLB_OFFSET, buf + PARAMETER_TLB_OFFSET,
  PARAMETER_TLB_SIZE);
 /* load linux command line */
 if (strncmp((buf + LINUX_CMD_OFFSET), linux_cmd_magic, 8) != 0)
  return WRONG_MAGIC_LINUXCMD;
 memcpy((dst + LINUX_CMD_OFFSET), buf + LINUX_CMD_OFFSET, LINUX_CMD_SIZE);
 /* load mtd partition table */
 if (strncmp(buf + MTD_PART_OFFSET, mtd_part_magic, 8) != 0)
```

```
 return WRONG_MAGIC_MTDPART;
 memcpy(dst + MTD_PART_OFFSET, buf + MTD_PART_OFFSET, MTD_PART_SIZE);
 return 0;
}
```

接着来看看read_saved_priv_data_blk函数：

```
int read_saved_priv_data_blk(char *buf)
{
 char *src = (char *)(VIVI_PRIV_ROM_BASE); //VIVI_PRIV_ROM_BASE为flash上参数区域的起始地址
 size_t size = (size_t)(VIVI_PRIV_SIZE); //VIVI_PRIV_SIZE也就是这三种类型的参数大小也就是48KB
#ifdef CONFIG_USE_PARAM_BLK
 {    //如果是在nand flash上定义了param分区，我们则直接去得到param分区的mtd_partition_t结构体
  mtd_partition_t *part = get_mtd_partition("param");
  if (part == NULL) {
   printk("Could not found 'param' partition/n");
   return -1;
  }
  src = (char *)part->offset; //然后获取到该结构体中定义的该分区的偏移地址，也真正就是存放参数的地址
 }
#endif
 return read_mem(buf, src, size); //然后从参数的源地址中读取参数先存放到sdram的起始地址也就是
0x30000000,然后再拷贝到sdram的参数区，也就是load_saved_priv_data函数中的操作。read_mem函数里面也
就是nand_read_ll函数，读nand flash之前已经讲过。
}
```

在load阶段内无非就是找到param分区，然后根据配置，找到相应的flash硬件驱动（这就是MTD层的作用所在，不过可以看出nand chip的databuf确实没有起到作用，现在也未看出这部分究竟用在何处）。然后就是读操作。当然，读取出来的信息先放到临时缓冲区，判断头部的magic number，如果符合则说明是正确的分区信息，

然后把信息从临时缓冲区复制到对应的默认配置区，这样就完成了真正的配置。

其实这个地方可以改进。首先看看param分区是否有合适的分区信息，如果有，直接读取到vivi parameter区域，不需要再读取默认的配置信息；如果没有合适的分区信息，然后读取默认的配置信息。这样在用户修正了分区信息时，不必再读取默认的配置信息，这也算是一处优化。

step 7：

调用add_command()函数，增加vivi作为终端时命令的相应处理函数。其实，这种机制还是比较简单的，就是利用了链表。

整个命令处理机制及其初始化的实现是在【lib/command.c】中完成的，包括添加命令、查找命令、执行命令、解析命令行等等。具体的命令函数则在相应的模块里面，这样形成了一个2层的软件架构：顶部管理层+底部执行层。维护的核心就是一个数据结构user_command：

```
typedef struct user_command {
    const char *name;
    void (*cmdfunc)(int argc, const char **);
    struct user_command *next_cmd;
    const char *helpstr;
} user_command_t;
```

第一个成员是指向name字符串的指针表示命令的名字，第二个成员就是命令的处理函数，第三个成员是指向下一个命令，第四个成员是帮助信息。如果你想添加一个命令，那么首先需要构造一个数据结构**user_command**的实例，比如：

```
user_command_t help_cmd = {
   "help",
   command_help,
   NULL,
   "help [{cmds}] /t/t/t-- Help about help?"
};
```

然后实现命令的真正处理函数command_help。

```
void command_help(int argc, const char **argv)
{
   user_command_t *curr;

   /* help <command>. invoke <command> with 'help' as an argument */
   if (argc == 2) {
      if (strncmp(argv[1], "help", strlen(argv[1])) == 0) {
         printk("Are you kidding?/n");
         return;
      }
      argv[0] = argv[1];
      argv[1] = "help";
      execcmd(argc, argv);
      return;
   }

   printk("Usage:/n");
   curr = head_cmd;
   while(curr != NULL) {
      printk(" %s/n", curr->helpstr);
      curr = curr->next_cmd;
   }
}
```

构造好之后，需要把它加入链表，也就是在init_builtin_cmds中增加add_command(&help_cmd);，其中add_command的实现如下：

```
void add_command(user_command_t *cmd)
{
  if (head_cmd == NULL) { //对链表的操作，表示如果当前vivi中还没有任何命令，则将新加进来的
user_command_t作为链表头指针
     head_cmd = tail_cmd = cmd;
  } else { //如果之前就存在，则将新加进来的user_command_t结构加载链表尾端就行
     tail_cmd->next_cmd = cmd;
     tail_cmd = cmd;
  }
  /*printk("Registered '%s' command/n", cmd->name);*/
}
```

这样，自己如果增加新的程序，就按照如上的步骤添加即可。

其余具体命令的实现暂时不做解释。

step 8：

根据情况，要么进入vivi的命令行交互界面，要么直接启动内核。关于此部分的流程分析，有了前面的基础和经验，是不难理解的。很容易通过vivi的打印信息得知进行到了第几步，《s3c2410完全开发》在过程上讲解的也很清楚。所以不打算具体分析了。还是来稍微分析下：

```
void boot_or_vivi(void)
{
 char c;
 int ret;
 ulong boot_delay;
 boot_delay = get_param_value("boot_delay", &ret);    //从vivi的环境参数boot_delay中获取到等待的时间
```

```
 if (ret) boot_delay = DEFAULT_BOOT_DELAY;  //若没有设置该参数的值，则利用默认的等待时间
if (boot_delay == 0) vivi_shell();/* If a value of boot_delay is zero,
 * unconditionally call vivi shell */

/*
 * wait for a keystroke (or a button press if you want.)
 */
printk("Press Return to start the LINUX now, any other key for vivi/n");
c = awaitkey(boot_delay, NULL);  //这句话也很重要，在等待时间内获取到用户按下去的键盘码值
if (((c != '/r') && (c != '/n') && (c != '/0'))) {
 printk("type /"help/" for help./n");
 vivi_shell();
}
run_autoboot(); //启动内核
return;
}
void vivi_shell(void)
{
#ifdef CONFIG_SERIAL_TERM
 serial_term();
#else
#error there is no terminal.
#endif
}
//drivers/serial/term.c
void serial_term(void)
```

```
{
 char cmd_buf[MAX_CMDBUF_SIZE];
 for (;;) {
  printk("%s> ", prompt);
  getcmd(cmd_buf, MAX_CMDBUF_SIZE);
  /* execute a user command */
  if (cmd_buf[0])
   exec_string(cmd_buf);//最终还是调用exec_string来进行命令的处理函数分配
 }
}
```

而对于启动内核的函数：

```
void run_autoboot(void)
{
 while (1) {
  exec_string("boot");
  printk("Failed 'boot' command. reentering vivi shell/n");
  /* if default boot fails, drop into the shell */
  vivi_shell();
 }
}
```

其实上也是调用exec_string来执行boot命令，真正启动内核的是在boot命令的处理函数中执行，则要来看看exec_string函数的实现：

```
void exec_string(char *buf)
{
 int argc;
```

```
 char *argv[128];
 char *resid;
 while (*buf) {
  memset(argv, 0, sizeof(argv));
  parseargs(buf, &argc, argv, &resid); //从得到的命令字符串解析出命令
  if (argc > 0)
   execcmd(argc, (const char **)argv);//然后执行命令，在这里其实也是查表，找到命令名和argc相同的
user_command_t结构体，然后调用user_command_t结构体中对应的命令处理函数来执行。
  buf = resid;
 }
}
```

现在翻阅网上资料，有一个问题实际上模模糊糊，如下：

vivi作为bootloader的一个重要的功能就是向Linux kernel传递启动参数，这个情景究竟是如何完成的呢？虽然网上讨论很多，但是因为vivi具有一点特殊性，所以使得理解上有一定的困难。现在已经比较清晰了，算是回答网友的一个问题，也算是总结，就bootloader如何向kernel传递参数，作为一个情景进行详尽的分析。事先需要说明的是，我们假定vivi为A，Linux kernel为B，A要传给B东西，这就是一个通信的过程。要想通信，至少我们得有一个约定，那就是协议。现在存在的协议有两种，一种是基于struct param_struct，不过这种因为其局限性即将作废；一种是基于tags技术。基本的情景框架就是A必须按照协议设置好参数，B呢，就需要来读取解析这些参数。它们之间必须配合好，如果配合不好，那么，kernel是无法引导成功的。现在嵌入式系统的移植，很多时候kernel引导不起来，部分原因就直接来自于参数传递问题。但是设计到这个问题，不能不分析Linux kernel的引导过程。现在还不想细致到代码层，只是根据部分代码把Linux kernel启动至获取引导参数的过程从整体上了解清楚，必要的时候辅助相应的代码。这部分内容的详细分析，专门在下篇总结中完成。

学习总结：

学习一种技术，采用历史的观点是很好的方法。我们现在学习的技术并非最新的理论研究，所以有大量前人的工作经验可以借鉴。站在巨人的肩上，不做无谓的工作，是好的学习方法。我现在的学习观点就是事先要分析阅读前人的相关经验，包括经典书籍、网上资料、网友的经验等等，然后呢，需要对这些知识理解消化，深入，深入再深入，形成自己的认识，转化成自己的经验。正像网友所说，这些都是现成的技术，只要静下心来肯学，就一定能够学好。

另外，一定要多思考，多动手，多给自己提出问题。没有问题说明你根本就没有深入，有问题才能在解决的过程中提升自己！学习首先从整体上把握流程，然后呢，需要具体的细节。只看整体，不看细节，容易“眼高手低”；只看细节，不看整体，容易“只见树木，不见森林”，提高不到一定的层次。

这些都是学习过程中的经验总结。欢迎交流！