# UBOOT启动第一阶

## 前景概要

由于工作原因这里只能做大概的分析，以三星s32440a来做分析！！！

根据makefile文件可以得到的结果是板子最开始的运行的代码文件是start.S的汇编文件，目录为cpu/arm920t/start.S，uboot的启动分为三个阶段，前两个阶段分别为硬件相关的阶段，软件相关的阶段。第三阶段还没有接触到，所以这里只做第一阶段的概述。为了以后方便再次编辑该文档时更为详细，留下一些未写的结构。

## 进入reset

程序一上电执行如下的代码：

.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

```
.balignl 16,0xdeadbeef
```

程序就直接进入reset

## reset之进入SVC32模式

代码如下：

```
/*
 * set the cpu to SVC32 mode
 */
mrs r0,cpsr
bic r0,r0,#0x1f
orr r0,r0,#0xd3
msr cpsr,r0
```

先说说bic和oor指令的用法：

BIC————位清除指令 指令格式： BIC{cond}{S} Rd,Rn,operand2 BIC指令将Rn 的值与操作数operand2 的反码按位逻辑"与"，结果存放到目的寄存器Rd 中。指令示例：BIC R0,R0,#0x0F ；将R0最低4位清零，其余位不变。

OOR ORR指令的格式为： ORR{条件}{S} 目的寄存器，操作数1，操作数2 ORR指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数1应该是一 个寄存器，操作数2可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操 作数1的某些位。 指令示例： ORR R0，R0，＃3 ； 该指令设置R0的0、1位，其余位保持不变。

orr r0,r0,#0xd3 0xd3=1101 0111 将r0与0xd3作算数或运算，然后将结果返还给r0,即把r0的bit[7:6]和bit[4]和bit[2:0]置为1。

这里作简要分析，cpsr是状态寄存器，mrs r0,cpsr为获取cpsr寄存器的值，并存放得到r0，然后bic和orr指令来修改r0的值，最后一句把修改后的值重新赋予cpsr。具体为什么要这样修改，可以参见三星s3c2440a的数据手册。

**那后面的分析可能就没这么详细了，主要是汇编没怎么学习，只能说它是干什么的，以后来补充。**

# reset之进入关闭看门狗

这里列出执行了的代码没有执行的代码就不列出来了，如下：

#define pWTCON 0x53000000 /*看门狗地址*/

#define INTMOD 0X4A000004 /*中断模式地址*/

#define INTMSK 0x4A000008 /* Interupt-Controller base addresses */

#define INTSUBMSK 0x4A00001C /*中断子掩码*/

#define CLKDIVN 0x4C000014 /* clock divisor register */

#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) /*关闭看门狗*/ ldr r0, =pWTCON mov r1, #0x0 str r1, [r0]

```
/*
 * mask all IRQs by setting all bits in the INTMR - default
 */
mov r1, #0xffffffff.    /*屏蔽所有中断*/
ldr r0, =INTMSK
str r1, [r0]
```

if defined(CONFIG_S3C2410)

```
ldr r1, =0x3ff
ldr r0, =INTSUBMSK
str r1, [r0]
```

#endif

#endif

这里关闭所有的中断INTMSK，INTSUBMSK表示掩码寄存器，意思就是某一种类型的中断来了，若是在掩码寄存器对该中断相应的位置1，那么该中断就不会被执行。mov r1, #0xffffffff.就是用来屏蔽所有相关的中断的。这里还有个中断子掩码，具体就要看数据手册了，总之就是把所有的中断全部屏蔽掉了。

# reset之代码的位置

代码如下：

*/查看程序是否被拷贝到SDRAM,因为ldr r1, _TEXT_BASE 0x33f80000/* #ifndef CONFIG_SKIP_LOWLEVEL_INIT adr r0, _start /* r0 <- current position of code */ *ldr r1, _TEXT_BASE /* test if we run from flash or RAM */ *cmp r0, r1 /* don't reloc during debug */ *blne cpu_init_crit /*如果cpu还是在flash或者step stone的内存中运行的话，跳转到cpu_init_crit运行*/ #endif

这段代码是来测试程序是在FLASH里面还是在SDRAM里面，首先*TEXT_BASE这个标识符号，标识了.word TEXT_BASE这么一个变量，这个值就是0x33f80000，为什么是这个值，因为在makefile里面有一个-Ttext的参数指定了代码段的位置应该放在0x33f8000的位置（**0x33f80000指的是SDRAM的地址，不是flash的地址***）那么r1 = 0x33f800，而r0的值是这样计算的pc + offset（*start相对于代码段的偏移量），若是程序被放在了flash，pc = 0,

*start相对于代码段的偏移量就是0，所以r0 = 0,；若是代码放在SDRAM，由于makefile脚本指示了代码段的位置放在0x33f80000的位置，那么pc = 0x33f8000，偏移量同样为0，那么r0 = 0x33f80000，而ldr r1, _TEXT_BASE使得r1 = 0x33f80000，因为*TEXT_BASE定义了TEXT_BASE，TEXT_BASE固定死了为0x33f80000，所以通过cmp来比较是不是相等的，若是相等的代码就被放在了SDRAM里面了，若是不相等，说明SDRAM还没有初始化，代码还存放在FLASH里面，因为只有SDRAM初始化了，SDRAM才能被使用。那么cpu_init_crit 就是用来初始化SDRAM，具体代码就不分析了，那么这里留下一个结构分析的过程。

## cpu_init_crit 初始化SDRAM

以后作分析！！！！

# reset之设置栈

## 设置malloc和全局变量分区

stack_setup:

ldr r0, _TEXT_BASE /**upper 128 KiB: relocated uboot*/*

sub r0, r0, #CFG_MALLOC_LEN /**malloc area*/*

sub r0, r0, #CFG_GBL_DATA_SIZE /**bdinfo*/*

**为什么要设置栈，因为要调用c函数就必须设置栈，具体是为什么，我目前还没搞清楚。**

这段代码比较简单，那么我们可以从名字看到的是从代码段开始CFG_MALLOC_LEN用来设置malloc分区

CFG_GBL_DATA_SIZE 用来设置全局变量的分区，**那么我们可以学习到变量名的命名。**那么我们可以得到内存现在分为四个区，*TEXT_BASE以上的内存，以*TEXT_BASE为基础大小为CFG_MALLOC_LEN的分区，以*TEXT_BASE+CFG_MALLOC_LEN为基础大小为CFG_GBL_DATA_SIZ的分区，最后是以*TEXT_BASE+CFG_MALLOC_LEN+CFG_GBL_DATA_SIZE为基础的下面的分区。后面还有更多的分区，慢慢的看吧。

## 设置快速中断和普通中断

#ifdef CONFIG_USE_IRQ

sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)

#endif

sub sp, r0, #12 /**leave 3 words for abort-stack*/*

所谓的快速中断，指的是不可以中断来了立马执行，而普通的中断可以理解为用一个队列来维护解决完一个中断接着执行下一个中断，那么这里我们又多了两个分区，快速中断+普通中断的分区外加3个字的停止栈，也就是栈的分区到这里就结束了。

# reset之初始化时钟

代码如下：

#ifndef CONFIG_SKIP_LOWLEVEL_INIT bl clock_init #endif

具体是如何初始化的留下一个结构以后再写

## 初始化时钟

以后方便在写！！！！

# reset之清除bss段

代码如下：

#ifndef CONFIG_SKIP_RELOCATE_UBOOT

relocate: /**relocate U-Boot to RAM*\ r0 <- current position of code*/

ldr r1, _TEXT_BASE /*test if we run from flash or RAM/*

cmp r0, r1 /*don't reloc during debug/*

beq clear_bss

同样的方法来判断代码是否在SDRAM，若是不在，则清除bss段，clear_bss代码如下：

clear_bss:

ldr r0, _bss_start /**find start of bss segment*/*

ldr r1, _bss_end /**stop here **/

mov r2, #0x00000000 /**clear **/

那么*bss_start和*bss_end又是多少，来自于一个叫uboot.lds的脚本来定义的，如下代码：

SECTIONS { . = 0x00000000;

```
. = ALIGN(4);
.text      :
{
/*(.text)的意思是.text包括cpu/arm920t/start.o这个目标文件里的(.text)*/
  cpu/arm920t/start.o   (.text)
      board/100ask24x0/boot_init.o (.text)
  *(.text)
}

. = ALIGN(4);
.rodata : { *(.rodata) }

. = ALIGN(4);
.data : { *(.data) }

. = ALIGN(4);
.got : { *(.got) }

. = .;
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;

. = ALIGN(4);
__bss_start = .;
.bss : { *(.bss) }
_end = .;
```

}

上面的代码

. = ALIGN(4);

__bss_start = .;

.bss : { *(.bss) }

_end = .;

这个文件告诉各个段是怎么分配在内存的。

**那么从上面我们得出.bss存储的是未初始化全局变量和静态变量以及初始化的全局和静态变量，若是不初始化，那么代码就会清楚该内存使得未初始化全局变量和静态变量默认值是0，这就是为什么全局变量不初始化，打印结果为0的原因。**

# reset之cpoy代码到RAM

代码如下：

#if 1

bl CopyCode2Ram /r0: source, r1: dest, r2: size = _bss_start - _armboot_start(代码段的开始)/

#else

add r2, r0, r2 /r2 <- source end address/

CopyCode2Ram就是把代码拷贝到SDRAM，正好前面初始化了SDRAM，那么SDRAM就能用了，CopyCode2Ram代码如下：

int CopyCode2Ram(unsigned long start_addr, unsigned char *buf, int size)

{

/r0: source, r1: dest, r2: size = _bss_start - _armboot_start(代码段的开始)/

/r0 : _start r1 : _TEXT_BASE/

unsigned int *pdwDest;

unsigned int *pdwSrc;

int i;

```
if (bBootFrmNORFlash())
{
    pdwDest = (unsigned int *)buf;
    pdwSrc  = (unsigned int *)start_addr;
    /* 从 NOR Flash启动 */
    for (i = 0; i < size / 4; i++)
    {
        pdwDest[i] = pdwSrc[i];
    }
    return 0;
}
else
{
    /* 初始化NAND Flash */
    nand_init_ll();
    /* 从 NAND Flash启动 */
    nand_read_ll_lp(buf, start_addr, (size + NAND_BLOCK_MASK_LP)&~(NAND_BLOCK_MASK_LP));
    return 0;
}
```

}

这里我们分析bBootFrmNORFlash()函数这个函数很有趣代码如下：

int bBootFrmNORFlash(void) { volatile unsigned int *pdw = (volatile unsigned int *)0; unsigned int dwVal;

```
/*
 * 无论是从NOR Flash还是从NAND Flash启动，
 * 地址0处为指令"b    Reset"，机器码为0xEA00000B，
 * 对于从NAND Flash启动的情况，其开始4KB的代码会复制到CPU内部4K内存中，
 * 对于从NOR Flash启动的情况，NOR Flash的开始地址即为0。
 * 对于NOR Flash，必须通过一定的命令序列才能写数据，
 * 所以可以根据这点差别来分辨是从NAND Flash还是NOR Flash启动:
 * 向地址0写入一个数据，然后读出来，如果没有改变的话就是NOR Flash
 */
```

```
dwVal = *pdw;
*pdw = 0x12345678;
if (*pdw != 0x12345678)
{
    return 1;
}
else
{
    *pdw = dwVal;
    return 0;
}
```

}

由于NORFLASH对应的地址就是执行的地址，但是FLASH是不能写的，而NAND FLASH的前4k对应片内的SRAM是可以写的，所以在0地址处写入0x12345678，若是可以写，那么就是nandflash，若是不可以写那就是norflash。所以用bBootFrmNORFlash函数来判断是在norflash启动还是在nandflash启动的，用该函数来做判断。

CopyCode2Ram其他的代码就不做具体分析了，所以留下一个结构

## CopyCode2Ram函数分析

以后做具体分析！！！！

# 总结

那么上面就是uboot第一阶段做的事情，我们来总结一下：

SVC32模式->关闭看门狗->初始化SDRAM->设置栈->初始化时钟->清除bss段->拷贝代码到内存