2014.4新版uboot启动流程分析

最近开始接触uboot，现在需要将2014.4版本uboot移植到公司armv7开发板。

在网上搜索讲uboot启动过程的文章，大多都是比较老版本的uboot，于是决定将新版uboot启动过程记录下来，和大家共享。

对于uboot，我写了一个专栏来记录我的一些理解，感兴趣的朋友可以点击以下链接：

u-boot学习笔记

辛苦之作，大家共享，转载还请注明出处！

Author : kerneler

Email ：karse0104@163.com

[cpp] view plain copy print?

```
#
#  (C)  Copyright  2000-2013
#  Wolfgang  Denk,  DENX  Software  Engineering,  wd@denx.de.
#
#  SPDX-License-Identifier:    GPL-2.0+
#

VERSION  =  2014
PATCHLEVEL  =  04
SUBLEVEL  =
EXTRAVERSION  =
NAME  =
```

```
#
# (C) Copyright 2000-2013
# Wolfgang Denk, DENX Software Engineering, wd@denx.de.
#
# SPDX-License-Identifier:  GPL-2.0+
#

VERSION = 2014
PATCHLEVEL = 04
SUBLEVEL =
EXTRAVERSION =
NAME =
```

到我写这篇文章之时，这个版本的uboot是最新版本。

2014.4版本uboot启动至命令行几个重要函数为：_start，_main，board_init_f，relocate_code，board_init_r。

一 _start

对于任何程序，入口函数是在链接时决定的，uboot的入口是由链接脚本决定的。uboot下armv7链接脚本默认目录为arch/arm/cpu/u-boot.lds。这个可以在配置文件中与CONFIG_SYS_LDSCRIPT来指定。

入口地址也是由连接器决定的，在配置文件中可以由CONFIG_SYS_TEXT_BASE指定。这个会在编译时加在ld连接器的选项-Ttext中

uboot的配置编译原理也非常值得学习，我想在另外写一篇文章来记录，这里不详细说了。

查看u-boot.lds

[cpp] view plain copy print?

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        .  =  0x00000000;

        .  =  ALIGN(4);
        .text  :
        {
                *(.__image_copy_start)
                CPUDIR/start.o  (.text*)
                *(.text*)
        }

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

        .  =  ALIGN(4);
        .data  :  {
                *(.data*)
        }
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :
```

```
    {
        *(.__image_copy_start)
        CPUDIR/start.o (.text*)
        *(.text*)
    }

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

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

链接脚本中这些宏的定义在linkage.h中，看字面意思也明白，程序的入口是在_start.，后面是text段，data段等。

_start在arch/arm/cpu/armv7/start.S中，一段一段的分析，如下：

[cpp] view plain copy print?

```
.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
#ifdef CONFIG_SPL_BUILD
_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
_pad:                     .word 0x12345678 /* now 16*4=64 */
#else
.globl _undefined_instruction
_undefined_instruction: .word undefined_instruction
```

```
.globl  _software_interrupt
_software_interrupt:        .word  software_interrupt
.globl  _prefetch_abort
_prefetch_abort:        .word  prefetch_abort
.globl  _data_abort
_data_abort:                .word  data_abort
.globl  _not_used
_not_used:            .word  not_used
.globl  _irq
_irq:                       .word  irq
.globl  _fiq
_fiq:                       .word  fiq
_pad:                       .word  0x12345678  /*  now  16*4=64  */</span>
<span  style="font-size:14px;">#endif    /*  CONFIG_SPL_BUILD  */

.global  _end_vect
_end_vect:

        .balignl  16,0xdeadbeef
.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
#ifdef CONFIG_SPL_BUILD
_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
_pad:           .word 0x12345678 /* now 16*4=64 */
#else
.globl _undefined_instruction
_undefined_instruction: .word undefined_instruction
```

```
.globl _software_interrupt
_software_interrupt:    .word software_interrupt
.globl _prefetch_abort
_prefetch_abort:    .word prefetch_abort
.globl _data_abort
_data_abort:        .word data_abort
.globl _not_used
_not_used:      .word not_used
.globl _irq
_irq:           .word irq
.globl _fiq
_fiq:           .word fiq
_pad:           .word 0x12345678 /* now 16*4=64 */</span>
<span style="font-size:14px;">#endif  /* CONFIG_SPL_BUILD */

.global _end_vect
_end_vect:

    .balignl 16,0xdeadbeef
```

.global声明_start为全局符号，_start就会被连接器链接到，也就是链接脚本中的入口地址了。

以上代码是设置arm的异常向量表，arm异常向量表如下：

地址	异常	进入模式	描述
0x00000000	复位	管理模式	复位电平有效时，产生复位异常，程序跳转到复位处理程序处执行
0x00000004	未定义指令	未定义模式	遇到不能处理的指令时，产生未定义指令异常
0x00000008	软件中断	管理模式	执行SWI指令产生，用于用户模式下的程序调用特权操作指令
0x0000000c	预存指令	中止模式	处理器预取指令的地址不存在，或该地址不允许当前指令访问，产生指令预取中止异常
0x00000010	数据操作	中止模式	处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常
0x00000014	未使用	未使用	未使用
0x00000018	IRQ	IRQ	外部中断请求有效，且CPSR中的I位为0时，产生IRQ异常
0x0000001c	FIQ	FIQ	快速中断请求引脚有效，且CPSR中的F位为0时，产生FIQ异常

8种异常分别占用4个字节，因此每种异常入口处都填写一条跳转指令，直接跳转到相应的异常处理函数中，reset异常是直接跳转到reset函数，其他7种异常是用ldr将处理函数入口地址加载到pc中。

后面汇编是定义了7种异常的入口函数，这里没有定义CONFIG_SPL_BUILD，所以走后面一个。

接下来定义的_end_vect中用.balignl来指定接下来的代码要16字节对齐，空缺的用0xdeadbeef，方便更加高效的访问内存。接着分析下面一段代码

[cpp] view plain copy print?

```
#ifdef  CONFIG_USE_IRQ
```

```
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
        .word      0x0badc0de

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
        .word  0x0badc0de
#endif

/* IRQ stack memory (calculated at run-time) + 8 bytes */
.globl IRQ_STACK_START_IN
IRQ_STACK_START_IN:
        .word      0x0badc0de
#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
    .word   0x0badc0de

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
    .word 0x0badc0de
#endif

/* IRQ stack memory (calculated at run-time) + 8 bytes */
.globl IRQ_STACK_START_IN
IRQ_STACK_START_IN:
    .word   0x0badc0de
```

如果uboot中使用中断，这里声明中断处理函数栈起始地址，这里给出的值是0x0badc0de，是一个非法值，注释也说明了，这个值会在运行时重新计算，我查找了一下代码是在interrupt_init中。

[cpp] view plain copy print?

```
reset:
        bl     save_boot_params
        /*
          * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
```

```
         * except if in HYP mode already
         */
        mrs  r0,  cpsr
        and  r1,  r0,  #0x1f            @ mask mode bits
        teq  r1,  #0x1a            @ test for HYP mode
        bicne      r0,  r0,  #0x1f            @ clear all mode bits
        orrne      r0,  r0,  #0x13            @ set SVC mode
        orr  r0,  r0,  #0xc0            @ disable FIQ and IRQ
        msr  cpsr,r0
reset:
    bl  save_boot_params
    /*
     * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
     * except if in HYP mode already
     */
    mrs r0, cpsr
    and r1, r0, #0x1f       @ mask mode bits
    teq r1, #0x1a       @ test for HYP mode
    bicne   r0, r0, #0x1f       @ clear all mode bits
    orrne   r0, r0, #0x13       @ set SVC mode
    orr r0, r0, #0xc0       @ disable FIQ and IRQ
    msr cpsr,r0
```

在上电或者重启后，处理器取得第一条指令就是b reset，所以会直接跳转到reset函数处。reset首先是跳转到save_boot_params中，如下：

[cpp] view plain copy print?

```
/*************************************************************************
  *
  * void save_boot_params(u32 r0, u32 r1, u32 r2, u32 r3)
  *    __attribute__((weak));
  *
  * Stack pointer is not yet initialized at this moment
  * Don't save anything to stack even if compiled with -O0
  *
  *************************************************************************/
ENTRY(save_boot_params)
        bx    lr                    @ back to my caller
ENDPROC(save_boot_params)
        .weak       save_boot_params
/*************************************************************************
 *
```

```
 * void save_boot_params(u32 r0, u32 r1, u32 r2, u32 r3)
 *  __attribute__((weak));
 *
 * Stack pointer is not yet initialized at this moment
 * Don't save anything to stack even if compiled with -O0
 *
 *************************************************************************/
ENTRY(save_boot_params)
    bx  lr          @ back to my caller
ENDPROC(save_boot_params)
    .weak   save_boot_params
```

这里save_boot_params函数中没做什么直接跳回，注释也说明了，栈没有初始化，最好不要再函数中做操作。

这里值得注意的是.weak关键字，在网上找了到的解释，我的理解是.weak相当于声明一个函数，如果该函数在其他地方没有定义，则为空函数，有定义则调用该定义的函数。

具体解释可以看这位大神的详解：

http://blog.csdn.net/norains/article/details/5954459

接下来reset执行7条指令，修改cpsr寄存器，设置处理器进入svc模式，并且关掉irq和fiq。

[cpp] view plain copy print?

```
/*
  *  Setup  vector:
  *  (OMAP4  spl  TEXT_BASE  is  not  32  byte  aligned.
  *  Continue  to  use  ROM  code  vector  only  in  OMAP4  spl)
  */
#if  !(defined(CONFIG_OMAP44XX)  &&  defined(CONFIG_SPL_BUILD))
        /*  Set  V=0  in  CP15  SCTRL  register  -  for  VBAR  to  point  to  vector  */
        mrc  p15,  0,  r0,  c1,  c0,  0      @  Read  CP15  SCTRL  Register
        bic  r0,  #CR_V              @  V  =  0
        mcr  p15,  0,  r0,  c1,  c0,  0      @  Write  CP15  SCTRL  Register

        /*  Set  vector  address  in  CP15  VBAR  register  */
        ldr  r0,  =_start
        mcr  p15,  0,  r0,  c12,  c0,  0    @Set  VBAR
#endif

        /*  the  mask  ROM  code  should  have  PLL  and  others  stable  */
#ifndef  CONFIG_SKIP_LOWLEVEL_INIT
        bl    cpu_init_cp15
        bl    cpu_init_crit
#endif
```

```
        bl      _main
/*
 * Setup vector:
 * (OMAP4 spl TEXT_BASE is not 32 byte aligned.
 * Continue to use ROM code vector only in OMAP4 spl)
 */
#if !(defined(CONFIG_OMAP44XX) && defined(CONFIG_SPL_BUILD))
    /* Set V=0 in CP15 SCTRL register - for VBAR to point to vector */
    mrc p15, 0, r0, c1, c0, 0   @ Read CP15 SCTRL Register
    bic r0, #CR_V       @ V = 0
    mcr p15, 0, r0, c1, c0, 0   @ Write CP15 SCTRL Register

    /* Set vector address in CP15 VBAR register */
    ldr r0, =_start
    mcr p15, 0, r0, c12, c0, 0  @Set VBAR
#endif

    /* the mask ROM code should have PLL and others stable */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl  cpu_init_cp15
    bl  cpu_init_crit
#endif

    bl  _main
```

前面6条汇编指令是对协处理器cp15进行操作，设置了处理器的异常向量入口地址为_start，
这里需要注意，ARM默认的异常向量表入口在0x0地址，uboot的运行介质（norflash nandflash sram等）映射地址可能不在0x0起始的地址，所以需要修改异常向量表入口。
但是我在网上没有找到cp15协处理器的c12寄存器的说明，可能是armv7新添加的
协处理器cp15的说明可以看我转的一篇文章：
http://blog.csdn.net/skyflying2012/article/details/25823967
接下来如果没有定义宏CONFIG_SKIP_LOWLEVEL_INIT，则会分别跳转执行cpu_init_cp15以及cpu_init_crit。
在分析这2个函数之前先总结一下上面分析的这一段_start中汇编的作用：
1 初始化异常向量表　　2 设置cpu svc模式，关中断　　　3 配置cp15，设置异常向量入口
都是跟异常有关的部分。
接下来先分析cpu_init_cp15
[cpp] view plain copy print?

```
/*************************************************************************
 *
 * cpu_init_cp15
 *
 * Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on unless
 * CONFIG_SYS_ICACHE_OFF is defined.
 *
 *************************************************************************/
ENTRY(cpu_init_cp15)
	/*
	 * Invalidate L1 I/D
	 */
	mov r0, #0			@ set up for MCR
	mcr p15, 0, r0, c8, c7, 0	@ invalidate TLBs
	mcr p15, 0, r0, c7, c5, 0	@ invalidate icache
	mcr p15, 0, r0, c7, c5, 6	@ invalidate BP array
	mcr	p15, 0, r0, c7, c10, 4	@ DSB
	mcr	p15, 0, r0, c7, c5, 4	@ ISB

	/*
	 * disable MMU stuff and caches
	 */
	mrc p15, 0, r0, c1, c0, 0
	bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
	bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
	orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
	orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
	bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
#else
	orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
#endif
	mcr p15, 0, r0, c1, c0, 0
#ifdef CONFIG_ARM_ERRATA_716044
	mrc p15, 0, r0, c1, c0, 0	@ read system control register
	orr r0, r0, #1 << 11		@ set bit #11
	mcr p15, 0, r0, c1, c0, 0	@ write system control register
#endif

#if (defined(CONFIG_ARM_ERRATA_742230) || defined(CONFIG_ARM_ERRATA_794072))
```

```
        mrc  p15,  0,  r0,  c15,  c0,  1    @  read  diagnostic  register
        orr  r0,  r0,  #1  <<  4          @  set  bit  #4
        mcr  p15,  0,  r0,  c15,  c0,  1    @  write  diagnostic  register
#endif

#ifdef  CONFIG_ARM_ERRATA_743622
        mrc  p15,  0,  r0,  c15,  c0,  1    @  read  diagnostic  register
        orr  r0,  r0,  #1  <<  6          @  set  bit  #6
        mcr  p15,  0,  r0,  c15,  c0,  1    @  write  diagnostic  register
#endif

#ifdef  CONFIG_ARM_ERRATA_751472
        mrc  p15,  0,  r0,  c15,  c0,  1    @  read  diagnostic  register
        orr  r0,  r0,  #1  <<  11        @  set  bit  #11
        mcr  p15,  0,  r0,  c15,  c0,  1    @  write  diagnostic  register
#endif
#ifdef  CONFIG_ARM_ERRATA_761320
        mrc  p15,  0,  r0,  c15,  c0,  1    @  read  diagnostic  register
        orr  r0,  r0,  #1  <<  21        @  set  bit  #21
        mcr  p15,  0,  r0,  c15,  c0,  1    @  write  diagnostic  register
#endif

        mov  pc,  lr                    @  back  to  my  caller
ENDPROC(cpu_init_cp15)
/*************************************************************************
 *
 * cpu_init_cp15
 *
 * Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on unless
 * CONFIG_SYS_ICACHE_OFF is defined.
 *
 *************************************************************************/
ENTRY(cpu_init_cp15)
    /*
     * Invalidate L1 I/D
     */
    mov r0, #0          @ set up for MCR
    mcr p15, 0, r0, c8, c7, 0   @ invalidate TLBs
    mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
    mcr p15, 0, r0, c7, c5, 6   @ invalidate BP array
```

```
    mcr     p15, 0, r0, c7, c10, 4  @ DSB
    mcr     p15, 0, r0, c7, c5, 4   @ ISB

    /*
     * disable MMU stuff and caches
     */
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
    bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
    orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
    orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
    bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
#else
    orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
#endif
    mcr p15, 0, r0, c1, c0, 0
#ifdef CONFIG_ARM_ERRATA_716044
    mrc p15, 0, r0, c1, c0, 0   @ read system control register
    orr r0, r0, #1 << 11    @ set bit #11
    mcr p15, 0, r0, c1, c0, 0   @ write system control register
#endif

#if (defined(CONFIG_ARM_ERRATA_742230) || defined(CONFIG_ARM_ERRATA_794072))
    mrc p15, 0, r0, c15, c0, 1  @ read diagnostic register
    orr r0, r0, #1 << 4     @ set bit #4
    mcr p15, 0, r0, c15, c0, 1  @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_743622
    mrc p15, 0, r0, c15, c0, 1  @ read diagnostic register
    orr r0, r0, #1 << 6     @ set bit #6
    mcr p15, 0, r0, c15, c0, 1  @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_751472
    mrc p15, 0, r0, c15, c0, 1  @ read diagnostic register
    orr r0, r0, #1 << 11    @ set bit #11
    mcr p15, 0, r0, c15, c0, 1  @ write diagnostic register
```

```
#endif
#ifdef CONFIG_ARM_ERRATA_761320
    mrc p15, 0, r0, c15, c0, 1  @ read diagnostic register
    orr r0, r0, #1 << 21    @ set bit #21
    mcr p15, 0, r0, c15, c0, 1  @ write diagnostic register
#endif

    mov pc, lr          @ back to my caller
ENDPROC(cpu_init_cp15)
```

cpu_init_cp15函数是配置cp15协处理器相关寄存器来设置处理器的MMU，cache以及tlb。如果没有定义CONFIG_SYS_ICACHE_OFF则会打开icache。关掉mmu以及tlb。

具体配置过程可以对照cp15寄存器来看，这里不详细说了

接下来看cpu_init_crit

[cpp] view plain copy print?

```
/*************************************************************************
  *
  *  CPU_init_critical  registers
  *
  *  setup  important  registers
  *  setup  memory  timing
  *
  *************************************************************************/
ENTRY(cpu_init_crit)
        /*
          *  Jump  to  board  specific  initialization...
          *  The  Mask  ROM  will  have  already  initialized
          *  basic  memory.  Go  here  to  bump  up  clock  rate  and  handle
          *  wake  up  conditions.
          */
        b       lowlevel_init               @  go  setup  pll,mux,memory
ENDPROC(cpu_init_crit)
/*************************************************************************
 *
 * CPU_init_critical registers
 *
 * setup important registers
 * setup memory timing
 *
```

```
 ***************************************************************************/
ENTRY(cpu_init_crit)
    /*
     * Jump to board specific initialization...
     * The Mask ROM will have already initialized
     * basic memory. Go here to bump up clock rate and handle
     * wake up conditions.
     */
    b   lowlevel_init       @ go setup pll,mux,memory
ENDPROC(cpu_init_crit)
```

看注释可以明白，cpu_init_crit调用的lowlevel_init函数是与特定开发板相关的初始化函数，在这个函数里会做一些pll初始化，如果不是从mem启动，则会做memory初始化，方便后续拷贝到mem中运行。

lowlevel_init函数则是需要移植来实现，做clk初始化以及ddr初始化

从cpu_init_crit返回后，_start的工作就完成了，接下来就要调用_main，总结一下_start工作：

1 前面总结过的部分，初始化异常向量表，设置svc模式，关中断

2 配置cp15，初始化mmu cache tlb

3 板级初始化，pll memory初始化

二 _main

_main函数在arch/arm/lib/crt0.S中，mian函数的作用在注释中有详细的说明，我们分段来分析一下

[cpp] view plain copy print?

```
ENTRY(_main)

/*
 * Set up initial C runtime environment and call board_init_f(0).
 */

#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
        ldr sp, =(CONFIG_SPL_STACK)
#else
        ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
        bic sp, sp, #7    /* 8-byte alignment for ABI compliance */
        sub sp, sp, #GD_SIZE        /* allocate one GD above SP */
        bic sp, sp, #7    /* 8-byte alignment for ABI compliance */
        mov r9, sp             /* GD is above SP */
        mov r0, #0
        bl   board_init_f
ENTRY(_main)
```

```
/*
 * Set up initial C runtime environment and call board_init_f(0).
 */

#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
    ldr sp, =(CONFIG_SPL_STACK)
#else
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
    bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
    sub sp, sp, #GD_SIZE    /* allocate one GD above SP */
    bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
    mov r9, sp      /* GD is above SP */
    mov r0, #0
    bl  board_init_f
```

首先将CONFIG_SYS_INIT_SP_ADDR定义的值加载到栈指针sp中，这个宏定义在配置头文件中指定。

这段代码是为board_init_f C函数调用提供环境，也就是栈指针sp初始化

8字节对齐，然后减掉GD_SIZE,这个宏定义是指的全局结构体gd的大小，是160字节在此处，这个结构体用来保存uboot一些全局信息，需要一块单独的内存。

最后将sp保存在r9寄存器中。因此r9寄存器中的地址就是gd结构体的首地址。

在后面所有code中如果要使用gd结构体，必须在文件中加入DECLARE_GLOBAL_DATA_PTR宏定义，定义如下：

[cpp] view plain copy print?

```
#define  DECLARE_GLOBAL_DATA_PTR         register  volatile  gd_t  *gd  asm  ("r9")
```

```
#define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("r9")
```

gd结构体首地址就是r9中的值。

C语言函数栈是向下生长，这里sp为malloc空间顶端减去gd bd空间开始的，起初很纳闷，sp设在这里，以后的C函数调用不都会在malloc空间了吗，堆栈空间不就重合了嘛，不用急，看完board_init_f就明白了。

接着说_main上面一段代码，接着r0赋为0，也就是参数0为0，调用board_init_f

三 board_init_f

移植uboot先做一个最精简版本，很多配置选项都没有打开，比如fb mmc等硬件都默认不打开，只配置基本的ddr serial，这样先保证uboot能正常启动进入命令行，然后再去添加其他。

我们这里分析就是按最精简版本来，这样可以更加简洁的说明uboot的启动流程。

board_init_f函数主要是根据配置对全局信息结构体gd进行初始化。

gd结构体中有个别成员意义我也不是很理解，这里我只说我理解并且在后面起到作用的成员。

[cpp] view plain copy print?

```
gd->mon_len  =  (ulong)&__bss_end  -  (ulong)_start;
```

```
    gd->mon_len = (ulong)&__bss_end - (ulong)_start;
```

初始化mon_len，代表uboot code的大小。

[cpp] view plain copy print?

```
for  (init_fnc_ptr  =  init_sequence;  *init_fnc_ptr;  ++init_fnc_ptr)  {
        if  ((*init_fnc_ptr)()  !=  0)  {
                hang  ();
        }
}
```

```
    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }
```

遍历调用init_sequence所有函数，init_sequence定义如下：

[cpp] view plain copy print?

```
init_fnc_t  *init_sequence[]  =  {
        arch_cpu_init,            /*  basic  arch  cpu  dependent  setup  */
        mark_bootstage,
#ifdef  CONFIG_OF_CONTROL
        fdtdec_check_fdt,
#endif
#if  defined(CONFIG_BOARD_EARLY_INIT_F)
        board_early_init_f,
#endif
        timer_init,          /*  initialize  timer  */
#ifdef  CONFIG_BOARD_POSTCLK_INIT
        board_postclk_init,
#endif
#ifdef  CONFIG_FSL_ESDHC
        get_clocks,
#endif
        env_init,              /*  initialize  environment  */
        init_baudrate,           /*  initialze  baudrate  settings  */
        serial_init,               /*  serial  communications  setup  */
        console_init_f,          /*  stage  1  init  of  console  */
        display_banner,          /*  say  that  we  are  here  */
        print_cpuinfo,           /*  display  cpu  info  (and  speed)  */
#if  defined(CONFIG_DISPLAY_BOARDINFO)
        checkboard,          /*  display  board  info  */
#endif
```

```
#if  defined(CONFIG_HARD_I2C)  ||  defined(CONFIG_SYS_I2C)
        init_func_i2c,
#endif
        dram_init,            /*  configure  available  RAM  banks  */
        NULL,
};
init_fnc_t *init_sequence[] = {
    arch_cpu_init,      /* basic arch cpu dependent setup */
    mark_bootstage,
#ifdef CONFIG_OF_CONTROL
    fdtdec_check_fdt,
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
    board_early_init_f,
#endif
    timer_init,     /* initialize timer */
#ifdef CONFIG_BOARD_POSTCLK_INIT
    board_postclk_init,
#endif
#ifdef CONFIG_FSL_ESDHC
    get_clocks,
#endif
    env_init,       /* initialize environment */
    init_baudrate,      /* initialze baudrate settings */
    serial_init,        /* serial communications setup */
    console_init_f,     /* stage 1 init of console */
    display_banner,     /* say that we are here */
    print_cpuinfo,      /* display cpu info (and speed) */
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,     /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SYS_I2C)
    init_func_i2c,
#endif
    dram_init,      /* configure available RAM banks */
    NULL,
};
```

arch_cpu_init需要实现，要先启动uboot这里可以先写一个空函数。

timer_init在lib/time.c中有实现，也是空函数，但是有__WEAK关键字，如果自己实现，则会调用自己实现的这个函数

对最精简uboot，需要做好就是ddr和serial，所以我们最关心是serial_init,console_init_f以及dram_init.

先看serial_init

[cpp] view plain copy print?

```
int  serial_init(void)
{
        return  get_current()->start();
}
static  struct  serial_device  *get_current(void)
{
        struct  serial_device  *dev;

        if  (!(gd->flags  &  GD_FLG_RELOC))
                dev  =  default_serial_console();
        else  if  (!serial_current)
                dev  =  default_serial_console();
        else
                dev  =  serial_current;

        /*  We  must  have  a  console  device  */
        if  (!dev)  {
#ifdef  CONFIG_SPL_BUILD
                puts("Cannot  find  console\n");
                hang();
#else
                panic("Cannot  find  console\n");
#endif
        }

        return  dev;
}
int serial_init(void)
{
    return get_current()->start();
}
static struct serial_device *get_current(void)
{
    struct serial_device *dev;
```

```
    if (!(gd->flags & GD_FLG_RELOC))
        dev = default_serial_console();
    else if (!serial_current)
        dev = default_serial_console();
    else
        dev = serial_current;

    /* We must have a console device */
    if (!dev) {
#ifdef CONFIG_SPL_BUILD
        puts("Cannot find console\n");
        hang();
#else
        panic("Cannot find console\n");
#endif
    }

    return dev;
}
```

gd->flags还没做初始化，serial_current用来存放我们当前要使用的serial，这里也还没做初始化，所以最终serial_device就是default_serial_console(),这个在serial驱动中有实现，来返回一个默认的调试串口。

serial_device结构体代表了一个串口设备，其中的成员都需要在自己的serial驱动中实现。

这样在serial_init中get_current获取就是串口驱动中给出的默认调试串口结构体，执行start，做一些特定串口初始化。

console_init_f将gd中have_console置1，这个函数不详细说了。

display_banner,print_cpuinfo利用现在的调试串口打印了uboot的信息。

接下来就是dram_init。

dram_init对gd->ram_size初始化，以便board_init_f后面代码对dram空间进行规划。

dram_init实现可以通过配置文件定义宏定义来实现，也可以通过对ddrc控制器读获取dram信息。

继续分析board_init_f，剩余代码将会对sdram空间进行规划！

[cpp] view plain copy print?

```
#if  defined(CONFIG_SYS_MEM_TOP_HIDE)
        /*
         * Subtract specified amount of memory to hide so that it won't
         * get "touched" at all by U-Boot. By fixing up gd->ram_size
         * the Linux kernel should now get passed the now "corrected"
         * memory size and won't touch it either. This should work
```

```
        * for arch/ppc and arch/powerpc. Only Linux board ports in
        * arch/powerpc with bootwrapper support, that recalculate the
        * memory size from the SDRAM controller setup will have to
        * get fixed.
        */
       gd->ram_size -= CONFIG_SYS_MEM_TOP_HIDE;
#endif

       addr = CONFIG_SYS_SDRAM_BASE + get_effective_memsize();
#if defined(CONFIG_SYS_MEM_TOP_HIDE)
    /*
     * Subtract specified amount of memory to hide so that it won't
     * get "touched" at all by U-Boot. By fixing up gd->ram_size
     * the Linux kernel should now get passed the now "corrected"
     * memory size and won't touch it either. This should work
     * for arch/ppc and arch/powerpc. Only Linux board ports in
     * arch/powerpc with bootwrapper support, that recalculate the
     * memory size from the SDRAM controller setup will have to
     * get fixed.
     */
    gd->ram_size -= CONFIG_SYS_MEM_TOP_HIDE;
#endif

    addr = CONFIG_SYS_SDRAM_BASE + get_effective_memsize();
```

CONFIG_SYS_MEM_TOP_HIDE宏定义是将一部分内存空间隐藏，注释说明对于ppc处理器在内核中有接口来实现使用uboot提供的值，这里咱们不考虑。

addr的值由CONFIG_SYS_SDRAM_BASE加上ram_size。也就是到了可用sdram的顶端。

[cpp] view plain copy print?

```
#if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF))
       /* reserve TLB table */
       gd->arch.tlb_size = PGTABLE_SIZE;
       addr -= gd->arch.tlb_size;

       /* round down to next 64 kB limit */
       addr &= ~(0x10000 - 1);

       gd->arch.tlb_addr = addr;
            debug("TLB table from %08lx to %08lx\n", addr, addr + gd->arch.tlb_size);
```

```
#endif

        /*  round  down  to  next  4  kB  limit  */
        addr  &=  ~(4096  -  1);
        debug("Top  of  RAM  usable  for  U-Boot  at:  %08lx\n",  addr);
#if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF))
    /* reserve TLB table */
    gd->arch.tlb_size = PGTABLE_SIZE;
    addr -= gd->arch.tlb_size;

    /* round down to next 64 kB limit */
    addr &= ~(0x10000 - 1);

    gd->arch.tlb_addr = addr;
    debug("TLB table from %08lx to %08lx\n", addr, addr + gd->arch.tlb_size);
#endif

    /* round down to next 4 kB limit */
    addr &= ~(4096 - 1);
    debug("Top of RAM usable for U-Boot at: %08lx\n", addr);
```

如果打开了icache以及dcache，则预留出PATABLE_SIZE大小的tlb空间，tlb存放首地址赋值给gd->arch.tlb_addr。

最后addr此时值就是tlb的地址，4kB对齐。

[cpp] view plain copy print?

```
#ifdef  CONFIG_LCD
#ifdef  CONFIG_FB_ADDR
        gd->fb_base  =  CONFIG_FB_ADDR;
#else
        /*  reserve  memory  for  LCD  display  (always  full  pages)  */
        addr  =  lcd_setmem(addr);
        gd->fb_base  =  addr;
#endif  /*  CONFIG_FB_ADDR  */
#endif  /*  CONFIG_LCD  */

        /*
          *  reserve  memory  for  U-Boot  code,  data  &  bss
          *  round  down  to  next  4  kB  limit
          */
        addr  -=  gd->mon_len;
```

```
        addr  &=  ~(4096  -  1);

                debug("Reserving     %ldk     for     U-Boot     at:     %08lx\n",     gd-
>mon_len  >>  10,  addr);
#ifdef CONFIG_LCD
#ifdef CONFIG_FB_ADDR
    gd->fb_base = CONFIG_FB_ADDR;
#else
    /* reserve memory for LCD display (always full pages) */
    addr = lcd_setmem(addr);
    gd->fb_base = addr;
#endif /* CONFIG_FB_ADDR */
#endif /* CONFIG_LCD */

    /*
     * reserve memory for U-Boot code, data & bss
     * round down to next 4 kB limit
     */
    addr -= gd->mon_len;
    addr &= ~(4096 - 1);

    debug("Reserving %ldk for U-Boot at: %08lx\n", gd->mon_len >> 10, addr);
```
如果需要使用frambuffer，使用配置fb首地址CONFIG_FB_ADDR或者调用lcd_setmem获取fb大小，这里面有板级相关函数需要实现，不过为了先能启动uboot，没有打开fb选项。addr值就是fb首地址。
gd->fb_base保存fb首地址。
接着-gd->mon_len为uboot的code留出空间，到这里addr的值就确定，addr作为uboot relocate的目标addr。
到这里，可以看出uboot现在空间划分是从顶端往下进行的。
先总结一下addr之上sdram空间的划分：
由高到低 ： top-->hide mem-->tlb space(16K)-->framebuffer space-->uboot code space-->addr
接下来要确定addr_sp的值。
[cpp] view plain copy print?
```
#ifndef  CONFIG_SPL_BUILD
        /*
          *  reserve  memory  for  malloc()  arena
          */
        addr_sp  =  addr  -  TOTAL_MALLOC_LEN;
        debug("Reserving  %dk  for  malloc()  at:  %08lx\n",
                        TOTAL_MALLOC_LEN  >>  10,  addr_sp);
```

```
	/*
	 * (permanently) allocate a Board Info struct
	 * and a permanent copy of the "global" data
	 */
	addr_sp -= sizeof (bd_t);
	bd = (bd_t *) addr_sp;
	gd->bd = bd;
	debug("Reserving %zu Bytes for Board Info at: %08lx\n",
			sizeof (bd_t), addr_sp);
#ifdef CONFIG_MACH_TYPE
	gd->bd->bi_arch_number = CONFIG_MACH_TYPE; /* board id for Linux */
#endif

	addr_sp -= sizeof (gd_t);
	id = (gd_t *) addr_sp;
	debug("Reserving %zu Bytes for Global Data at: %08lx\n",
			sizeof (gd_t), addr_sp);
#ifndef CONFIG_ARM64
	/* setup stackpointer for exeptions */
	gd->irq_sp = addr_sp;
#ifdef CONFIG_USE_IRQ
	addr_sp -= (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ);
	debug("Reserving %zu Bytes for IRQ stack at: %08lx\n",
		CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ, addr_sp);
#endif
	/* leave 3 words for abort-stack        */
	addr_sp -= 12;

	/* 8-byte alignment for ABI compliance */
	addr_sp &= ~0x07;
#else	/* CONFIG_ARM64 */
	/* 16-byte alignment for ABI compliance */
	addr_sp &= ~0x0f;
#endif	/* CONFIG_ARM64 */
#ifndef CONFIG_SPL_BUILD
    /*
     * reserve memory for malloc() arena
     */
    addr_sp = addr - TOTAL_MALLOC_LEN;
    debug("Reserving %dk for malloc() at: %08lx\n",
```

```
            TOTAL_MALLOC_LEN >> 10, addr_sp);
    /*
     * (permanently) allocate a Board Info struct
     * and a permanent copy of the "global" data
     */
    addr_sp -= sizeof (bd_t);
    bd = (bd_t *) addr_sp;
    gd->bd = bd;
    debug("Reserving %zu Bytes for Board Info at: %08lx\n",
            sizeof (bd_t), addr_sp);
#ifdef CONFIG_MACH_TYPE
    gd->bd->bi_arch_number = CONFIG_MACH_TYPE; /* board id for Linux */
#endif

    addr_sp -= sizeof (gd_t);
    id = (gd_t *) addr_sp;
    debug("Reserving %zu Bytes for Global Data at: %08lx\n",
            sizeof (gd_t), addr_sp);
#ifndef CONFIG_ARM64
    /* setup stackpointer for exeptions */
    gd->irq_sp = addr_sp;
#ifdef CONFIG_USE_IRQ
    addr_sp -= (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ);
    debug("Reserving %zu Bytes for IRQ stack at: %08lx\n",
        CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ, addr_sp);
#endif
    /* leave 3 words for abort-stack    */
    addr_sp -= 12;

    /* 8-byte alignment for ABI compliance */
    addr_sp &= ~0x07;
#else   /* CONFIG_ARM64 */
    /* 16-byte alignment for ABI compliance */
    addr_sp &= ~0x0f;
#endif  /* CONFIG_ARM64 */
```

首先预留malloc len，这里我定义的是0x400000.

注释中说明，为bd，gd做一个永久的copy。

留出了全局信息bd_t结构体的空间，首地址存在gd->bd。

留出gd_t结构体的空间。首地址存在id中。

将此地址保存在gd->irq_sp中作为异常栈指针。uboot中我们没有用到中断。

最后留出12字节，for abort stack，这个没看懂。

到这里addr_sp值确定，总结一下addr_sp之上空间分配。

由高到低 ： addr-->malloc len(0x400000)-->bd len-->gd len-->12 byte-->addr_sp（栈往下增长，addr_sp之下空间作为栈空间）

[cpp] view plain copy print?

```
gd->bd->bi_baudrate  =  gd->baudrate;
/*  Ram  ist  board  specific,  so  move  it  to  board  code  ...  */
dram_init_banksize();
display_dram_config();    /*  and  display  it  */

gd->relocaddr  =  addr;
gd->start_addr_sp  =  addr_sp;
gd->reloc_off  =  addr  -  (ulong)&_start;
debug("relocation  Offset  is:  %08lx\n",  gd->reloc_off);
if  (new_fdt)  {
        memcpy(new_fdt,  gd->fdt_blob,  fdt_size);
        gd->fdt_blob  =  new_fdt;
}
memcpy(id,  (void  *)gd,  sizeof(gd_t));
    gd->bd->bi_baudrate = gd->baudrate;
    /* Ram ist board specific, so move it to board code ... */
    dram_init_banksize();
    display_dram_config();  /* and display it */

    gd->relocaddr = addr;
    gd->start_addr_sp = addr_sp;
    gd->reloc_off = addr - (ulong)&_start;
    debug("relocation Offset is: %08lx\n", gd->reloc_off);
    if (new_fdt) {
        memcpy(new_fdt, gd->fdt_blob, fdt_size);
        gd->fdt_blob = new_fdt;
    }
    memcpy(id, (void *)gd, sizeof(gd_t));
```

给bd->bi_baudrate赋值gd->baudrate，gd->baudrate是在前面baudrate_init中初始化。

dram_init_banksize()是需要实现的板级函数。根据板上ddrc获取ddr的bank信息。填充在gd->bd->bi_dram[CONFIG_NR_DRAM_BANKS]。

gd->relocaaddr为目标addr，gd->start_addr_sp为目标addr_sp,gd->reloc_off为目标addr和现在实际code起始地址的偏移。reloc_off非常重要，会作为后面relocate_code函数的参数，来实现code的拷贝。

最后将gd结构体的数据拷贝到新的地址id上。

board_init_f函数将sdram空间重新进行了划分，可以看出栈空间和堆空间是分开的，就不存在_main调用board_init_f之前的那个问题啦。

并且在重新规划空间完成之前并没有出现初始化堆，以及使用堆空间的问题，比如malloc函数，所以之前的堆栈空间重合的问题是过虑了。

至此，board_init_f结束，回到_main

四 _main

board_init_f结束后，代码如下：

[cpp] view plain copy print?

```
#if  !  defined(CONFIG_SPL_BUILD)

/*
 *  Set  up  intermediate  environment  (new  sp  and  gd)  and  call
 *  relocate_code(addr_moni).  Trick  here  is  that  we'll  return
 *  'here'  but  relocated.
 */

        ldr  sp,  [r9,  #GD_START_ADDR_SP]  /*  sp  =  gd->start_addr_sp  */
        bic  sp,  sp,  #7    /*  8-byte  alignment  for  ABI  compliance  */
        ldr  r9,  [r9,  #GD_BD]               /*  r9  =  gd->bd  */
        sub  r9,  r9,  #GD_SIZE               /*  new  GD  is  below  bd  */

        adr  lr,  here
        ldr  r0,  [r9,  #GD_RELOC_OFF]          /*  r0  =  gd->reloc_off  */
        add  lr,  lr,  r0
        ldr  r0,  [r9,  #GD_RELOCADDR]          /*  r0  =  gd->relocaddr  */
        b      relocate_code
here:
#if ! defined(CONFIG_SPL_BUILD)

/*
 * Set up intermediate environment (new sp and gd) and call
 * relocate_code(addr_moni). Trick here is that we'll return
 * 'here' but relocated.
 */
```

```
    ldr sp, [r9, #GD_START_ADDR_SP] /* sp = gd->start_addr_sp */
    bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
    ldr r9, [r9, #GD_BD]        /* r9 = gd->bd */
    sub r9, r9, #GD_SIZE        /* new GD is below bd */

    adr lr, here
    ldr r0, [r9, #GD_RELOC_OFF]     /* r0 = gd->reloc_off */
    add lr, lr, r0
    ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr */
    b   relocate_code
here:
```

这段汇编很有意思，前4条汇编实现了新gd结构体的更新。

首先更新sp，并且将sp 8字节对齐，方便后面函数开辟栈能对齐，

然后获取gd->bd地址到r9中，需要注意，在board_init_f中gd->bd已经更新为新分配的bd了，下一条汇编将r9减掉bd的size，这样就获取到了board_init_f中新分配的gd了！

后面汇编则是为relocate_code做准备，首先加载here地址，然后加上新地址偏移量给lr，则是code relocate后的新here了，relocate_code返回条转到lr，则是新位置的here！

最后在r0中保存code的新地址，跳转到relocate_code

五 relocate_code

relocate_code函数在arch/arm/lib/relocate.S中，这个函数实现了将uboot code拷贝到relocaddr。

这部分算是整个uboot中最核心也是最难理解的代码，我单独写了一篇文章来介绍这一部分的工作原理，感兴趣的朋友可以看下面这个链接

http://blog.csdn.net/skyflying2012/article/details/37660265

这里就不再详说了。

到这里需要总结一下，经过上面的分析可以看出，

新版uboot在sdram空间分配上，是自顶向下，

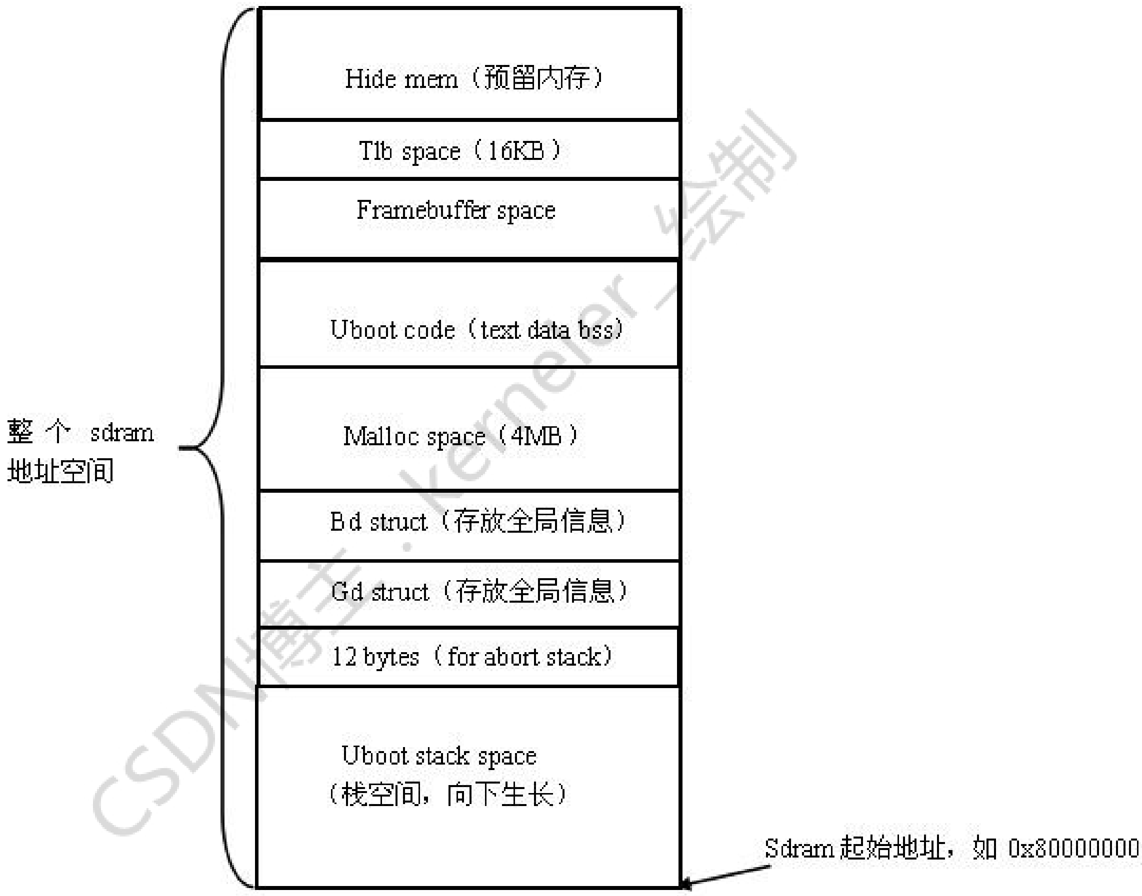
不管uboot是从哪里启动，spiflash，nandflash，sram等跑到这里code都会被从新定位到sdram上部的一个位置，继续运行。

我找了一个2010.6版本的uboot大体看了一下启动代码，是通过判断_start和TEXT_BASE（链接地址）是否相等来确定是否需要relocate。如果uboot是从sdram启动则不需要relocate。

新版uboot在这方面还是有较大变动。

这样变动我考虑好处可能有二，一是不用考虑启动方式，all relocate code。二是不用考虑uboot链接地址，因为都要重新relocate。

uboot sdram空间规划图：



六 _main

从relocate_code回到_main中，接下来是main最后一段代码，如下：

[cpp] view plain copy print?

```
/*  Set  up  final  (full)  environment  */

        bl    c_runtime_cpu_setup  /*  we  still  call  old  routine  here  */

        ldr  r0,  =__bss_start        /*  this  is  auto-relocated!  */
        ldr  r1,  =__bss_end            /*  this  is  auto-relocated!  */

        mov  r2,  #0x00000000          /*  prepare  zero  to  clear  BSS  */

clbss_l:cmp  r0,  r1                    /*  while  not  at  end  of  BSS  */
        strlo      r2,  [r0]                /*  clear  32-bit  BSS  word  */
        addlo      r0,  r0,  #4              /*  move  to  next  */
        blo  clbss_l

        bl  coloured_LED_init
        bl  red_led_on

        /*  call  board_init_r(gd_t  *id,  ulong  dest_addr)  */
```

```
        mov         r0,  r9                               /*  gd_t  */
        ldr  r1,  [r9,  #GD_RELOCADDR]  /*  dest_addr  */
        /*  call  board_init_r  */
        ldr  pc,  =board_init_r      /*  this  is  auto-relocated!  */

        /*  we  should  not  return  here.  */
/* Set up final (full) environment */

    bl  c_runtime_cpu_setup /* we still call old routine here */

    ldr r0, =__bss_start    /* this is auto-relocated! */
    ldr r1, =__bss_end      /* this is auto-relocated! */

    mov r2, #0x00000000     /* prepare zero to clear BSS */

clbss_l:cmp r0, r1          /* while not at end of BSS */
    strlo   r2, [r0]        /* clear 32-bit BSS word */
    addlo   r0, r0, #4      /* move to next */
    blo clbss_l

    bl coloured_LED_init
    bl red_led_on

    /* call board_init_r(gd_t *id, ulong dest_addr) */
    mov     r0, r9                  /* gd_t */
    ldr r1, [r9, #GD_RELOCADDR] /* dest_addr */
    /* call board_init_r */
    ldr pc, =board_init_r   /* this is auto-relocated! */

    /* we should not return here. */
```

首先跳转到c_runtime_cpu_setup，如下：

[cpp] view plain copy print?

```
ENTRY(c_runtime_cpu_setup)
/*
  *  If  I-cache  is  enabled  invalidate  it
  */
#ifndef  CONFIG_SYS_ICACHE_OFF
        mcr  p15,  0,  r0,  c7,  c5,  0      @  invalidate  icache
```

```
        mcr          p15,   0,   r0,   c7,   c10,   4     @  DSB
        mcr          p15,   0,   r0,   c7,   c5,   4      @  ISB
#endif
/*
  *  Move  vector  table
  */
        /*  Set  vector  address  in  CP15  VBAR  register  */
        ldr          r0,   =_start
        mcr          p15,   0,   r0,   c12,   c0,   0    @Set  VBAR

        bx    lr

ENDPROC(c_runtime_cpu_setup)
ENTRY(c_runtime_cpu_setup)
/*
 * If I-cache is enabled invalidate it
 */
#ifndef CONFIG_SYS_ICACHE_OFF
    mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
    mcr     p15, 0, r0, c7, c10, 4  @ DSB
    mcr     p15, 0, r0, c7, c5, 4   @ ISB
#endif
/*
 * Move vector table
 */
    /* Set vector address in CP15 VBAR register */
    ldr     r0, =_start
    mcr     p15, 0, r0, c12, c0, 0  @Set VBAR

    bx  lr

ENDPROC(c_runtime_cpu_setup)
```

如果icache是enable，则无效掉icache，保证从sdram中更新指令到cache中。

接着更新异常向量表首地址，因为code被relocate，所以异常向量表也被relocate。

从c_runtime_cpu_setup返回，下面一段汇编是将bss段清空。

接下来分别调用了coloured_LED_init以及red_led_on,很多开发板都会有led指示灯，这里可以实现上电指示灯亮，有调试作用。

最后r0赋值gd指针，r1赋值relocaddr，进入最后的board_init_r！

七 board_init_r

参数1是新gd指针，参数2是relocate addr，也就是新code地址

[cpp] view plain copy print?

```
gd->flags  |=  GD_FLG_RELOC;    /*  tell  others:  relocation  done  */
bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_R,  "board_init_r");

monitor_flash_len  =  (ulong)&__rel_dyn_end  -  (ulong)_start;

/*  Enable  caches  */
enable_caches();

debug("monitor  flash  len:  %08lX\n",  monitor_flash_len);
board_init();      /*  Setup  chipselects  */
```

```
    gd->flags |= GD_FLG_RELOC;  /* tell others: relocation done */
    bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_R, "board_init_r");

    monitor_flash_len = (ulong)&__rel_dyn_end - (ulong)_start;

    /* Enable caches */
    enable_caches();

    debug("monitor flash len: %08lX\n", monitor_flash_len);
    board_init();   /* Setup chipselects */
```

置位gd->flags，标志已经relocate。monitor_flash_len这个变量作用没看懂。使能cache，最后board_init是需要实现的板级支持函数。做开发板的基本初始化。

[cpp] view plain copy print?

```
#ifdef  CONFIG_CLOCKS
        set_cpu_clk_info();  /*  Setup  clock  information  */
#endif

        serial_initialize();

        debug("Now  running  in  RAM  -  U-Boot  at:  %08lx\n",  dest_addr);
```

```
#ifdef CONFIG_CLOCKS
    set_cpu_clk_info(); /* Setup clock information */
#endif

    serial_initialize();
```

```
    debug("Now running in RAM - U-Boot at: %08lx\n", dest_addr);
```

如果打开CONFIG_CLOCKS,set_cpu_clk_info也是需要实现的板级支持函数。

重点来说一些serial_initialize，对于最精简能正常启动的uboot，serial和ddr是必须正常工作的。

实现在drivers/serial/serial.c中，如下：

[cpp] view plain copy print?

```
void  serial_initialize(void)
{
        mpc8xx_serial_initialize();
        ns16550_serial_initialize();
        pxa_serial_initialize();
        s3c24xx_serial_initialize();
        s5p_serial_initialize();
        mpc512x_serial_initialize();。。。。
      mxs_auart_initialize();
        arc_serial_initialize();
        vc0718_serial_initialize();

        serial_assign(default_serial_console()->name);
}
void serial_initialize(void)
{
    mpc8xx_serial_initialize();
    ns16550_serial_initialize();
    pxa_serial_initialize();
    s3c24xx_serial_initialize();
    s5p_serial_initialize();
    mpc512x_serial_initialize();。。。。
   mxs_auart_initialize();
    arc_serial_initialize();
    vc0718_serial_initialize();

    serial_assign(default_serial_console()->name);
}
```

所有串口驱动都会实现一个xxxx_serial_initialize函数，并且添加到serial_initialize中，xxxx_serial_initialize函数中是将所有需要的串口（用结构体struct serial_device表示，其中实现了基本的收 发 配置）调用serial_register注册，serial_register如下：

[cpp] view plain copy print?

```
void  serial_register(struct  serial_device  *dev)
{
```

```
#ifdef  CONFIG_NEEDS_MANUAL_RELOC
        if  (dev->start)
                dev->start  +=  gd->reloc_off;
        if  (dev->stop)
                dev->stop  +=  gd->reloc_off;
        if  (dev->setbrg)
                dev->setbrg  +=  gd->reloc_off;
        if  (dev->getc)
                dev->getc  +=  gd->reloc_off;
        if  (dev->tstc)
                dev->tstc  +=  gd->reloc_off;
        if  (dev->putc)
                dev->putc  +=  gd->reloc_off;
        if  (dev->puts)
                dev->puts  +=  gd->reloc_off;
#endif

        dev->next  =  serial_devices;
        serial_devices  =  dev;
}
void serial_register(struct serial_device *dev)
{
#ifdef CONFIG_NEEDS_MANUAL_RELOC
    if (dev->start)
        dev->start += gd->reloc_off;
    if (dev->stop)
        dev->stop += gd->reloc_off;
    if (dev->setbrg)
        dev->setbrg += gd->reloc_off;
    if (dev->getc)
        dev->getc += gd->reloc_off;
    if (dev->tstc)
        dev->tstc += gd->reloc_off;
    if (dev->putc)
        dev->putc += gd->reloc_off;
    if (dev->puts)
        dev->puts += gd->reloc_off;
#endif

    dev->next = serial_devices;
```

```
    serial_devices = dev;
}
```

就是将你的serial_dev加到全局链表serial_devices中。

可以想象，如果你有4个串口，则再你的串口驱动中分别定义4个serial device，并实现对应的收发配置，然后serial_register注册者4个串口。

回到serial-initialize,最后调用serial_assign，default_serial_console我们之前说过，就是你在串口驱动给出一个默认调试串口，serial_assign如下：

[cpp] view plain copy print?

```
int  serial_assign(const  char  *name)
{
        struct  serial_device  *s;

        for  (s  =  serial_devices;  s;  s  =  s->next)  {
                if  (strcmp(s->name,  name))
                        continue;
                serial_current  =  s;
                return  0;
        }

        return  -EINVAL;
}
int serial_assign(const char *name)
{
    struct serial_device *s;

    for (s = serial_devices; s; s = s->next) {
        if (strcmp(s->name, name))
            continue;
        serial_current = s;
        return 0;
    }

    return -EINVAL;
}
```

serial_assign就是从serial_devices链表中找到指定的默认调试串口，条件就是串口的name，最后serial_current就是当前的默认串口了。

总结一下，serial_initialize工作是将所有serial驱动中所有串口注册到serial_devices链表中，然后找到指定的默认串口。

[cpp] view plain copy print?

```
/*  The  Malloc  area  is  immediately  below  the  monitor  copy  in  DRAM  */
        malloc_start  =  dest_addr  -  TOTAL_MALLOC_LEN;
        mem_malloc_init  (malloc_start,  TOTAL_MALLOC_LEN);
```

```
/* The Malloc area is immediately below the monitor copy in DRAM */
    malloc_start = dest_addr - TOTAL_MALLOC_LEN;
    mem_malloc_init (malloc_start, TOTAL_MALLOC_LEN);
```

根据咱们之前board_init_f中的分析，relocate addr之下的部分就是malloc的预留空间了。这里获取malloc首地址malloc_start.

[cpp] view plain copy print?

```
void  mem_malloc_init(ulong  start,  ulong  size)
{
        mem_malloc_start  =  start;
        mem_malloc_end  =  start  +  size;
        mem_malloc_brk  =  start;

        memset((void  *)mem_malloc_start,  0,  size);

        malloc_bin_reloc();
}
```

```
void mem_malloc_init(ulong start, ulong size)
{
    mem_malloc_start = start;
    mem_malloc_end = start + size;
    mem_malloc_brk = start;

    memset((void *)mem_malloc_start, 0, size);

    malloc_bin_reloc();
}
```

mem_malloc_init中就是对malloc预留的空间初始化，起始地址，结束地址，清空。咱们已经relocate，malloc_bin_reloc中无操作了。

board_init_r接下来的代码是做一些外设的初始化，比如mmc flash eth，环境变量的设置，还有中断的使能等，这里需要说一下是关于串口的2个函数，stdio_init和console_init_r.

看stdio_init代码，我们只定义了serial，会调到serial_stdio_init，如下:

[cpp] view plain copy print?

```
void  serial_stdio_init(void)
{
        struct  stdio_dev  dev;
        struct  serial_device  *s  =  serial_devices;

        while  (s)  {
                memset(&dev,  0,  sizeof(dev));

                strcpy(dev.name,  s->name);
                dev.flags  =  DEV_FLAGS_OUTPUT  |  DEV_FLAGS_INPUT;

                dev.start  =  s->start;
                dev.stop  =  s->stop;
                dev.putc  =  s->putc;
                dev.puts  =  s->puts;
                dev.getc  =  s->getc;
                dev.tstc  =  s->tstc;

                stdio_register(&dev);

                s  =  s->next;
        }
}
void serial_stdio_init(void)
{
    struct stdio_dev dev;
    struct serial_device *s = serial_devices;

    while (s) {
        memset(&dev, 0, sizeof(dev));

        strcpy(dev.name, s->name);
        dev.flags = DEV_FLAGS_OUTPUT | DEV_FLAGS_INPUT;

        dev.start = s->start;
        dev.stop = s->stop;
        dev.putc = s->putc;
        dev.puts = s->puts;
        dev.getc = s->getc;
```

```
        dev.tstc = s->tstc;

        stdio_register(&dev);

        s = s->next;
    }
}
```

将serial_devices链表上所有serial device同样初始化一个stdio_dev，flag为output input，调用stdio-register，将stdio_dev添加到全局devs链表中。

可以想象，serial_stdio_init是在drivers/serial/serial.c中实现，uboot在这里是利用的内核分层思想，drivers/serial下是特定serial驱动，分别调用serial_register注册到serial_devices中，这可以说是通用的serial驱动层，

通用serial层调用serial-stdio-init将所有serial注册到stdio device中，这就是通用的stdio层。

看来分层思想还是非常重要的！

board_init_r中调用完stdio_init后又调用了console_init_r，如下

[cpp] view plain copy print?

```cpp
int  console_init_r(void)
{
        struct  stdio_dev  *inputdev  =  NULL,  *outputdev  =  NULL;
        int  i;
        struct  list_head  *list  =  stdio_get_list();
        struct  list_head  *pos;
        struct  stdio_dev  *dev;

        /*  Scan  devices  looking  for  input  and  output  devices  */
        list_for_each(pos,  list)  {
                dev  =  list_entry(pos,  struct  stdio_dev,  list);

                if  ((dev->flags  &  DEV_FLAGS_INPUT)  &&  (inputdev  ==  NULL))  {
                        inputdev  =  dev;
                }
                if  ((dev->flags  &  DEV_FLAGS_OUTPUT)  &&  (outputdev  ==  NULL))  {
                        outputdev  =  dev;
                }
                if(inputdev  &&  outputdev)
```

```
                        break;
        }

      if  (outputdev  !=  NULL)  {
                console_setfile(stdout,  outputdev);
                console_setfile(stderr,  outputdev);
        }

        /*  Initializes  input  console  */
        if  (inputdev  !=  NULL)  {
                console_setfile(stdin,  inputdev);
        }

#ifndef  CONFIG_SYS_CONSOLE_INFO_QUIET
        stdio_print_current_devices();
#endif  /*  CONFIG_SYS_CONSOLE_INFO_QUIET  */

        /*  Setting  environment  variables  */
        for  (i  =  0;  i  <  3;  i++)  {
                setenv(stdio_names[i],  stdio_devices[i]->name);
        }

                                                                                                    gd-
>flags  |=  GD_FLG_DEVINIT;        /*  device  initialization  completed  */

        return  0;
}
int console_init_r(void)
{
    struct stdio_dev *inputdev = NULL, *outputdev = NULL;
    int i;
    struct list_head *list = stdio_get_list();
    struct list_head *pos;
    struct stdio_dev *dev;

    /* Scan devices looking for input and output devices */
    list_for_each(pos, list) {
        dev = list_entry(pos, struct stdio_dev, list);
```

```
        if ((dev->flags & DEV_FLAGS_INPUT) && (inputdev == NULL)) {
            inputdev = dev;
        }
        if ((dev->flags & DEV_FLAGS_OUTPUT) && (outputdev == NULL)) {
            outputdev = dev;
        }
        if(inputdev && outputdev)
            break;
    }

   if (outputdev != NULL) {
        console_setfile(stdout, outputdev);
        console_setfile(stderr, outputdev);
    }

    /* Initializes input console */
    if (inputdev != NULL) {
        console_setfile(stdin, inputdev);
    }

#ifndef CONFIG_SYS_CONSOLE_INFO_QUIET
    stdio_print_current_devices();
#endif /* CONFIG_SYS_CONSOLE_INFO_QUIET */

    /* Setting environment variables */
    for (i = 0; i < 3; i++) {
        setenv(stdio_names[i], stdio_devices[i]->name);
    }

    gd->flags |= GD_FLG_DEVINIT;    /* device initialization completed */

    return 0;
}
```

console_init_r前半部分很清楚了，从devs.list链表中查找flag为output或者input的dev，如果只有serial之前注册了stdio_dev，则outputdev inputdev都是咱们注册的第一个serial。

之后调用console_setfile，如下：

[cpp] view plain copy print?

```
static int console_setfile(int file, struct stdio_dev * dev)
```

```
{
        int error = 0;

        if (dev == NULL)
                return -1;

        switch (file) {
        case stdin:
        case stdout:
        case stderr:
                /* Start new device */
                if (dev->start) {
                        error = dev->start();
                        /* If it's not started dont use it */
                        if (error < 0)
                                break;
                }

                /* Assign the new device (leaving the existing one started) */
                stdio_devices[file] = dev;

                /*
                 * Update monitor functions
                 * (to use the console stuff by other applications)
                 */
                switch (file) {
                case stdin:
                        gd->jt[XF_getc] = dev->getc;
                        gd->jt[XF_tstc] = dev->tstc;
                        break;
                case stdout:
                        gd->jt[XF_putc] = dev->putc;
                        gd->jt[XF_puts] = dev->puts;
                        gd->jt[XF_printf] = printf;
                        break;
                }

              break;

        default:                /* Invalid file ID */
```

```
                error  =  -1;
        }
        return  error;
}
static int console_setfile(int file, struct stdio_dev * dev)
{
    int error = 0;

    if (dev == NULL)
        return -1;

    switch (file) {
    case stdin:
    case stdout:
    case stderr:
        /* Start new device */
        if (dev->start) {
            error = dev->start();
            /* If it's not started dont use it */
            if (error < 0)
                break;
        }

        /* Assign the new device (leaving the existing one started) */
        stdio_devices[file] = dev;

        /*
         * Update monitor functions
         * (to use the console stuff by other applications)
         */
        switch (file) {
        case stdin:
            gd->jt[XF_getc] = dev->getc;
            gd->jt[XF_tstc] = dev->tstc;
            break;
        case stdout:
            gd->jt[XF_putc] = dev->putc;
            gd->jt[XF_puts] = dev->puts;
            gd->jt[XF_printf] = printf;
```

```
            break;
        }

        break;

    default:        /* Invalid file ID */
        error = -1;
    }
    return error;
}
```

首先运行设备的start，就是特定serial实现的start函数。然后将stdio_device放到stdio_devices全局数组中，这个数组3个成员，stdout，stderr，stdin。最后还会在gd中设一下操作函数。

在console_init_r中最后会改变gd中flag状态，为GD_FLG_DEVINIT。表示设备初始化完成。

board_init_r进行完板级初始化后最后进入死循环，打印命令行，等待命令输入和解析。到这里uboot的启动过程就全部结束了！

上面用很大篇幅自下往上解释uboot下serial到console的架构，那来看一下实际使用时由printf到最后serial输出这个自上到下的流程吧。

首先来看printf，实现在common/console.c中如下：

[cpp] view plain copy print?

```
int printf(const char *fmt, ...)
{
        va_list args;
        uint i;
        char printbuffer[CONFIG_SYS_PBSIZE];

#if !defined(CONFIG_SANDBOX) && !defined(CONFIG_PRE_CONSOLE_BUFFER)
        if (!gd->have_console)
                return 0;
#endif

        va_start(args, fmt);

        /* For this to work, printbuffer must be larger than
         * anything we ever want to print.
         */
```

```
        i  =  vscnprintf(printbuffer,  sizeof(printbuffer),  fmt,  args);
        va_end(args);

        /*  Print  the  string  */
        puts(printbuffer);
        return  i;
}
int printf(const char *fmt, ...)
{
    va_list args;
    uint i;
    char printbuffer[CONFIG_SYS_PBSIZE];

#if !defined(CONFIG_SANDBOX) && !defined(CONFIG_PRE_CONSOLE_BUFFER)
    if (!gd->have_console)
        return 0;
#endif

    va_start(args, fmt);

    /* For this to work, printbuffer must be larger than
     * anything we ever want to print.
     */
    i = vscnprintf(printbuffer, sizeof(printbuffer), fmt, args);
    va_end(args);

    /* Print the string */
    puts(printbuffer);
    return i;
}
```

字符串的拼接跟一般printf实现一样，最后调用puts，puts实现也在console.c中，如下：

[cpp] view plain copy print?

```
void  puts(const  char  *s)
{
#ifdef  CONFIG_SANDBOX
        if  (!gd)  {
                os_puts(s);
                return;
        }
#endif
```

```
#ifdef  CONFIG_SILENT_CONSOLE
        if  (gd->flags  &  GD_FLG_SILENT)
                return;
#endif

#ifdef  CONFIG_DISABLE_CONSOLE
        if  (gd->flags  &  GD_FLG_DISABLE_CONSOLE)
                return;
#endif

        if  (!gd->have_console)
                return  pre_console_puts(s);

        if  (gd->flags  &  GD_FLG_DEVINIT)  {
                /*  Send  to  the  standard  output  */
                fputs(stdout,  s);
        }  else  {
                /*  Send  directly  to  the  handler  */
                serial_puts(s);
        }
}
void puts(const char *s)
{
#ifdef CONFIG_SANDBOX
    if (!gd) {
        os_puts(s);
        return;
    }
#endif

#ifdef CONFIG_SILENT_CONSOLE
    if (gd->flags & GD_FLG_SILENT)
        return;
#endif

#ifdef CONFIG_DISABLE_CONSOLE
    if (gd->flags & GD_FLG_DISABLE_CONSOLE)
        return;
```

```
#endif

    if (!gd->have_console)
        return pre_console_puts(s);

    if (gd->flags & GD_FLG_DEVINIT) {
        /* Send to the standard output */
        fputs(stdout, s);
    } else {
        /* Send directly to the handler */
        serial_puts(s);
    }
}
```

gd->have_console 在 board_init_f 的 console_init_f 中置位，flag 的 GD_FLG_DEVINIT 则是在刚才 board_init_r中console_init_r最后置位。
如果GD_FLG_DEVINIT没有置位，表明console没有注册，是在board_init_f之后，board_init_r执行完成之前，这时调用serial_puts,如下：
[cpp] view plain copy print?

```
void  serial_puts(const  char  *s)
{
        get_current()->puts(s);
}
void serial_puts(const char *s)
{
    get_current()->puts(s);
}
```

直接调到serial.c中的函数，完全符合board_init_f中serial_init的配置，仅仅找到一个默认串口来使用，其他串口暂且不管。
如果GD_FLG_DEVINIT置位，表明console注册完成。调用fputs，如下：
[cpp] view plain copy print?

```
void  fputs(int  file,  const  char  *s)
{
        if  (file  <  MAX_FILES)
                console_puts(file,  s);
}

static  inline  void  console_puts(int  file,  const  char  *s)
```

```
{
        stdio_devices[file]->puts(s);
}
void fputs(int file, const char *s)
{
    if (file < MAX_FILES)
        console_puts(file, s);
}

static inline void console_puts(int file, const char *s)
{
    stdio_devices[file]->puts(s);
}
```

fputs调console_puts从全局stdio_devices中找到对应stdout对应的成员stdio_device，调用puts，最终也是会调用到特定serial的puts函数。

分析后总结一下：

可以看出，对于serial，uboot实现了一个2级初始化：

stage 1，仅初始化default console serial，printf到puts后会直接调用特定串口的puts函数，实现打印

stage 2，将所有serial注册为stdio_device，并挑出指定调试串口作为stdio_devices的stdout stdin stderr。printf到puts后再到全局stdio_devices中找到对应stdio_device，调用stdio-device的puts，最终调用特定serial的puts，实现打印。

区分这2个stage，是利用gd的flag，GD_FLG_DEVINIT。