u-boot-1.1.6/include/asm-arm/global_data.h

只摘选部分关键代码进行分析;

```
/*
 * The following data structure is placed in some memory wich is
 * available very early after boot (like DPRAM on MPC8xx/MPC82xx, or
 * some locked parts of the data cache) to allow for a minimum set of
 * global variables during system initialization (until we have set
 * up the memory controller so that we can use RAM).
 *
 * Keep it *SMALL* and remember to set CFG_GBL_DATA_SIZE > sizeof(gd_t)
 */

typedef	struct  global_data {
	bd_t		*bd;
	unsigned long	flags;
	unsigned long	baudrate;
	unsigned long	have_console;	/* serial_init() was called */
	unsigned long	reloc_off;	/* Relocation Offset */
	unsigned long	env_addr;	/* Address  of Environment struct */
	unsigned long	env_valid;	/* Checksum of Environment valid? */
	unsigned long	fb_base;	/* base address of frame buffer */
#ifdef CONFIG_VFD
	unsigned char	vfd_type;	/* display type */
#endif
#if 0
	unsigned long	cpu_clk;	/* CPU clock in Hz!		*/
	unsigned long	bus_clk;
	unsigned long	ram_size;	/* RAM size */
	unsigned long	reset_status; /* reset status register at boot */
#endif
	void		**jt;		/* jump table */
} gd_t;

/*
 * Global Data Flags
 */
#define	GD_FLG_RELOC	0x00001		/* Code was relocated to RAM
	*/
#define	GD_FLG_DEVINIT	0x00002		/* Devices have been initialized	*/
#define	GD_FLG_SILENT	0x00004		/* Silent mode
	*/

#define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("r8")
```

global_data是在系统早期boot时使用的数据结构;

在需要使用gd指针的时候，只需要加入DECLARE_GLOBAL_DATA_PTR这句话就可以了。

可以知道，gd指针始终是放在r8中的。

u-boot-1.1.6/include/asm-arm/u-boot.h

```
/ ********************************************************************
 * NOTE: This header file defines an interface to U-Boot. Including
 * this (unmodified) header file in another file is considered normal
 * use of U-Boot, and does *not* fall under the heading of "derived
 * work".
 ********************************************************************/

#ifndef _U_BOOT_H_
#define _U_BOOT_H_      1

typedef struct bd_info {
    int                 bi_baudrate;  /* serial console baudrate */
    unsigned long       bi_ip_addr;   /* IP Address */
    unsigned char       bi_enetaddr[6]; /* Ethernet adress */
    struct environment_s            *bi_env;
    ulong               bi_arch_number;       /* unique id for this board */
    ulong               bi_boot_params;       /* where this board expects params */
    struct                                    /* RAM configuration */
    {
        ulong start;
        ulong size;
    }                   bi_dram[CONFIG_NR_DRAM_BANKS];
#ifdef CONFIG_HAS_ETH1
    /* second onboard ethernet port */
    unsigned char   bi_enet1addr[6];
#endif
} bd_t;

#define bi_env_data bi_env->data
#define bi_env_crc  bi_env->crc

#endif /* _U_BOOT_H_ */
```

bd_t中的变量bi_boot_params，表示传递给内核的参数的位置。

然后看看gd和bd的初始化，在lib_arm/board.c中：

```
gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
memset ((void*)gd, 0, sizeof (gd_t));
gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
memset (gd->bd, 0, sizeof (bd_t));
```

说明这两个结构体在内存中的位置是在uboot的代码在往下的地址处，所以进行操作的时候不要覆盖了这个位置！

在board/smdk2410/smdk2410.c中，有如下初始化：

gd->bd->bi_boot_params = 0x30000100; 说明参数位置在0x30000100。

u-boot-1.1.6/include/asm-arm/setup.h

只摘选部分关键代码进行分析;

```
struct tag_header {
        u32 size;
        u32 tag;
};
```

其中 size：表示整个tag 结构体的大小(用字的个数来表示，而不是字节的个数)，等于tag_header的大小加上u联合体的大小，例如，参数结构体 ATAG_CORE 的size=(sizeof(tag->tag_header)+sizeof(tag->u.core))>>2，一般通过函数 tag_size(struct * tag_xxx)来获得每个参数结构体的size。

其中 tag：表示整个tag 结构体的标记，如：ATAG_CORE等。

```
struct tag {
        struct tag_header hdr;
        union {
                struct tag_core                 core;
                struct tag_mem32    mem;
                struct tag_videotext videotext;
                struct tag_ramdisk  ramdisk;
                struct tag_initrd       initrd;
                struct tag_serialnr     serialnr;
                struct tag_revision     revision;
                struct tag_videolfb     videolfb;
                struct tag_cmdline      cmdline;

                /*
                 * Acorn specific
                 */
                struct tag_acorn        acorn;

                /*
                 * DC21285 specific
                 */
                struct tag_memclk       memclk;
        } u;
};
```

联合体 u 包括了所有可选择的内核参数类型，包括：tag_core, tag_mem32，tag_ramdisk等。参数结构体之间的遍历是通过函数 tag_next(struct * tag)来实现的。本系统参数链表包括的结构体有：ATAG_CORE，ATAG_MEM，ATAG_RAMDISK，ATAG_INITRD32，ATAG_CMDLINE，ATAG_END。在整个参数链表中除了参数结构体 ATAG_CORE 和 ATAG_END 的位置固定以外，其他参数结构体的顺序是任意的。本 BootLoader所传递的参数链表如下：第一个内核参数结构体，标记为ATAG_CORE，参数类型为 tag_core。每个参数类型的定义请参考源代码文件。

```
#define tag_next(t)   ((struct tag *)((u32 *)(t) + (t)->hdr.size))
#define tag_size(type)        ((sizeof(struct tag_header) + sizeof(struct type)) >> 2)
```

```
/* The list must start with an ATAG_CORE node */
#define ATAG_CORE       0x54410001

struct tag_core {
        u32 flags;              /* bit 0 = read-only */
        u32 pagesize;
        u32 rootdev;
};

/****************************** 用例示范 ************************************

        tag列表中, 都是以tag_core这个结构体开始的.
        假设参数存放在0x30000100地址, 启动内核时, ARM的R2寄存器要存放该地址.

        params = (struct tag *) 0x30000100;
        params->hdr.tag = ATAG_CORE;
        params->hdr.size = tag_size(tag_core);
        // 宏定义tag_size, 得到tag_header+tag_core共占用的字节.
        // 这里两个结构体都是u32的整数倍, 直接除以4即可得到字节.

        params->u.core.flags = 0;           //这里为了简便, 随便设定的值.
        params->u.core.pagesize = 0;
        params->u.core.rootdev = 0;

        params = tag_next (params);
        // 宏定义tag_next, 得到tag参数列表下一个可以存放的地址.

*******************************************************************************/

/* command line: \0 terminated string */
#define ATAG_CMDLINE    0x54410009

struct tag_cmdline {
        char    cmdline[1];    /* this is the minimum size */
};


/****************************** 用例示范 ************************************

        tag列表中, tag_cmdline这个比较特殊, 因为只有它的大小时可变的, 计算比较麻烦.

        char *p = "root=/dev/mtdblock 2 init=/linuxrc console=ttySAC0";
        params->hdr.tags = ATAG_CMDLINE;
        params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
        //这里大小不是u32的整数倍,所以不能用tag_size宏定义计算大小.
        // strlen(p)+1 是字符串p(包括结束符号'\0')的大小, 最后 + 4 保证除以4后的地址,
        // 不会和cmdline所占用的内存冲突.

        strcpy(params->u.cmdline.cmdlind, p);   //已经计算了足够的内存, 所以直接复制过来
即可
```

```
        params = tag_next (params);
        // 宏定义tag_next, 得到tag参数列表下一个可以存放的地址.

*****************************************************************************/

/* The list ends with an ATAG_NONE node. */
#define ATAG_NONE       0x00000000
/***************************** 用例示范 ************************************
        tag列表中, 都是以标记ATAG_NONE结束的.
        params->hdr.tag = ATAG_NONE;
        params->hdr.size = 0;
*****************************************************************************/
```

内核是如何从gd->bd->bi_boot_params指定的地址上知道参数从哪里开始以及到哪里结束呢?

所以我们在构建各种参数tag时，在开始时先要构建一个参数标记为ATAG_CORE的tag结构标示从这个tag结构开始接下来就是参数。

– –

现来结合代码分析在u-boot中是如何来构建这多个参数的tag结构:

/common/cmd_bootm.c 文件中，bootm 命令对应的do_bootm函数，当分析 uImage 中信息发现 OS 是 Linux 时 ，调用 ./lib_arm/bootm.c 文件中的 do_bootm_linux 函数来启动 Linux kernel 。

```
#if defined (CONFIG_SETUP_MEMORY_TAGS) ll /
    defined (CONFIG_CMDLINE_TAG) ll /
    defined (CONFIG_INITRD_TAG) ll /
    defined (CONFIG_SERIAL_TAG) ll /
    defined (CONFIG_REVISION_TAG) ll /
    defined (CONFIG_LCD) ll /
    defined (CONFIG_VFD)
 setup_start_tag (bd);  //通过bd结构体中参数在内存中的存放地址gd->bd->bi_boot_params来构建初始化的tag结构，表明参数结构的开始
//下面就是构建很多的参数tag结构.
#ifdef CONFIG_SERIAL_TAG
 setup_serial_tag (&params); //构建串口参数的tag结构
#endif
#ifdef CONFIG_REVISION_TAG
 setup_revision_tag (&params);
```

```
#endif
#ifdef CONFIG_SETUP_MEMORY_TAGS
 setup_memory_tags (bd);  //构建内存参数的tag结构
#endif
#ifdef CONFIG_CMDLINE_TAG
 setup_commandline_tag (bd, commandline);  //构建命令行参数的tag结构
#endif
#ifdef CONFIG_INITRD_TAG
 if (initrd_start && initrd_end)
  setup_initrd_tag (bd, initrd_start, initrd_end); //构建ramdisk参数的tag结构
#endif
#if defined (CONFIG_VFD) || defined (CONFIG_LCD)
 setup_videolfb_tag ((gd_t *) gd);
#endif
 setup_end_tag (bd);  //最后是构建参数tag结构结束的tag结构，标示参数已经结束，参数标记为ATAG_NONE
#endif
```

注意上面参数的tag结构的构建是有宏的约束的，再来看看具体是怎样构建每个tag结构的：

```
#if defined (CONFIG_SETUP_MEMORY_TAGS) || /
    defined (CONFIG_CMDLINE_TAG) || /
    defined (CONFIG_INITRD_TAG) || /
    defined (CONFIG_SERIAL_TAG) || /
    defined (CONFIG_REVISION_TAG) || /
    defined (CONFIG_LCD) || /
    defined (CONFIG_VFD)
static void setup_start_tag (bd_t *bd)
{
 params = (struct tag *) bd->bi_boot_params;//将指定的内存中存放参数列表的地址强制转化为struct tag的结构，这样便于内核存取各个参数

 params->hdr.tag = ATAG_CORE; //标示这个tag结构是用来标示参数结构的开始
 params->hdr.size = tag_size (tag_core); //存放整个tag结构的大小

 params->u.core.flags = 0;
 params->u.core.pagesize = 0;
 params->u.core.rootdev = 0;

 params = tag_next (params);
}
```

其中用到了一个重要的指针:params，这是一个指向struct tag的指针，在文件的开始处声明，可以被这个文件中的所有函数访问：static struct tag *params;

到最后可以看到调用： theKernel (0, machid, bd->bi_boot_params);
当然,有很多的宏来选择是否传递相应的tag到linux kenel.实际是这些所以针对于 bd->bi_boot_params 这个变量.这个变量是个整形变量,代表存放所有tag的buffer的地址.
例如,在 smdk2410.c 中的 board_init() 函数中,对于这个变量进行了如下赋值:

```
    gd->bd->bi_boot_params = 0x30000100;
```

0x30000100 这个值可以随意指定, 但是要保证和内核中相应的 mach_type 一致.以smdk2410为例:
在内核中始终这个值的地方是: arch/arm/mach-s3c2410/mach-smdk2410.c的最后

```
MACHINE_START(SMDK2410, "SMDK2410")

    .phys_ram    = S3C2410_SDRAM_PA,
    .phys_io    = S3C2410_PA_UART,
    .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C2410_SDRAM_PA + 0x100,
    .map_io        = smdk2410_map_io,
    .init_irq    = smdk2410_init_irq,
    .timer        = &s3c24xx_timer,
MACHINE_END
```

红色部分的值, 必须等于0x30000100, 否者将会出现无法启动的问题.
内核启动后,会读取0x300000100位置的值, 当然,内核会把这个地址转换成逻辑地址在操作. 因为内核跑起来后,MMU已经工作, 必须要把0x300000100这个物理地址转成逻辑地址然后在操作.对于u- boot传给内核的参数中(tag), 内核比较关系memory的信息,比如memory地址的起始,大小等.如果没有得到,那么内核无法启 动,内核会进入BUG()函数,然后死在那里.
而**memory的信息是由 CONFIG_SETUP_MEMORY_TAGS** 宏决定的. 因此当这个宏没有被定义时,内核跑不起来. 初始化meminfo时会失败. 现象就是:
Starting Kernel ...
死掉.
一般需要定义:

```
#define CONFIG_SETUP_MEMORY_TAGS
#define CONFIG_CMDLINE_TAG
```

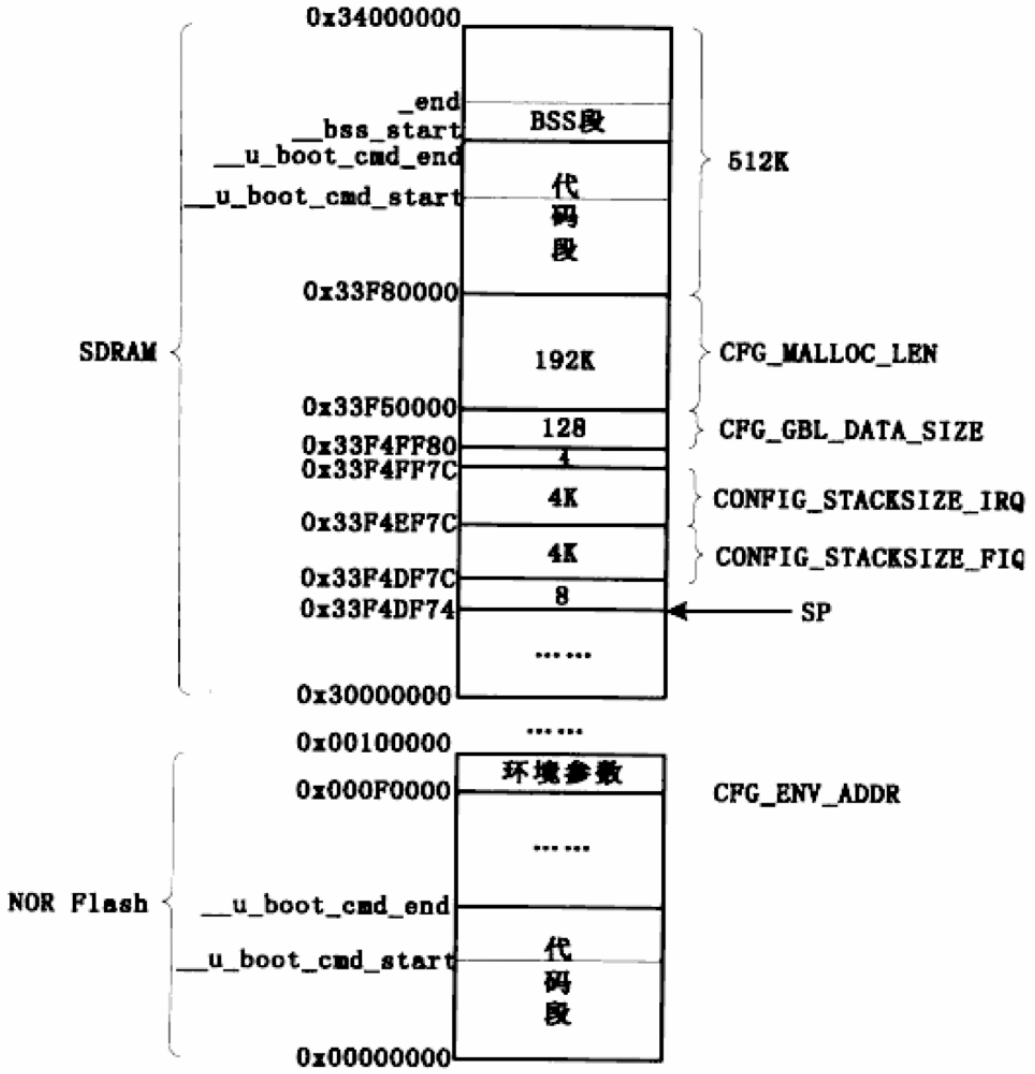

图 15.3　U-Boot 内存使用情况