U-Boot 1.1.6 关键功能模块代码分析

作者：zenf　E-Mail: zenf_zhao@163.com

(2007 年 12 月)

目 录

1.1 U-Boot u-boot.lds 文件分析

```
/*
 * (C) Copyright 2000-2004
 * Wolfgang Denk, DENX Software Engineering, wd@denx.de.
*/

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)  //指定启动时的函数入口地址，_start 在每个 CPU 的 start.S 中定义
               //真正的启动运行地址段由 TEXT_BASE 宏定义在编译时由 config.mk 中定义
SECTIONS
{
   . = 0;                               //指定系统启动从偏移地址 0 开始
   . = ALIGN(4);                        //地址进行 4 字节对其调整，确保低 2bit 地址线为 0
   .text      :                         //定义.text 段空间
   {
     cpu/s3c44b0/start.o (.text)        //指定 start.o 目标文件首先从.text 段分配
     *(.text)                           //后续.text 段内容的分配
   }

   . = ALIGN(4);   //.text 段处理后，进行 4 字节地址对其调整，然后分配.rodata 段空间
   .rodata : { *(.rodata) }

   . = ALIGN(4);    //4 字节地址调整，然后分配.data 段空间
   .data : { *(.data) }

   . = ALIGN(4);    //4 字节地址调整，然后分配.got 段空间
   .got : { *(.got) }

  __u_boot_cmd_start = .;              //定义.u_boot_cmd 的段空间，
  .u_boot_cmd : { *(.u_boot_cmd) }     //并且__u_boot_cmd_start 符号指向段空间开始
  __u_boot_cmd_end = .;                //__u_boot_cmd_end 符号指向该段空间结束

   armboot_end_data = .;               // armboot_end_data 符号指向之前所有分配完段的结束，
                                       //后续将开始.bss 段的分配

   . = ALIGN(4);                       //地址 4 字节调整，开始分配.bss 段空间
   __bss_start = .;                    //.bss 段空间开始地址
   .bss : { *(.bss) }
   _end = .;                           //.bss 段空间结束地址
}
```

说明 1：标准应用程序包括 3 类标准段空间：.text 运行代码段；.data 全局变量等具有初始值的数据空间；.bss 暂态变量，堆栈等数据空间；

说明 2：.rodata，.got，.u_boot_cmd 等段空间由程序员设计需要而自行定义的段空间；

说明 3：本文档采用 ARM7 CPU 进行分析，其指令字长为 4 字节，所以地址调整为 4 字节；

U-BOOT 1.1.6 的一可运行 load，反编译后获取的段空间分配例子：

```
u-boot:     file format elf32-littlearm
u-boot
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0c700000

Program Header:
    LOAD off    0x00008000 vaddr 0x0c700000 paddr 0x0c700000 align 2**15
         filesz 0x00022f28 memsz 0x00057ca4 flags rwx
private flags = 2: [APCS-32] [FPA float format] [has entry point]

Sections:       段长度     段起始地址                          地址调整偏移
Idx Name          Size      VMA       LMA       File off     Algn
  0 .text         0001c1f8  0c700000  0c700000  00008000    2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .glue_7       00000000  0c71c1f8  0c71c1f8  000241f8    2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .glue_7t      00000000  0c71c1f8  0c71c1f8  000241f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00005af8  0c71c1f8  0c71c1f8  000241f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000c18  0c721cf0  0c721cf0  00029cf0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .u_boot_cmd   00000620  0c722908  0c722908  0002a908  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .bss          00034d7c  0c722f28  0c722f28  0002af28  2**2
                  ALLOC
  7 .debug_abbrev 0000800c  00000000  00000000  0002af28  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_info   00081333  00000000  00000000  00032f34  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_line   000210a5  00000000  00000000  000b4267  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_pubnames 0000226b  00000000  00000000  000d530c  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_aranges 000007a0  00000000  00000000  000d7577  2**0
                  CONTENTS, READONLY, DEBUGGING
SYMBOL TABLE:
0c700000 l    d  .text    00000000     //.text 段空间开始 0x0c700000,.text 段结束为 0x0c71c1f8-1
0c71c1f8 l    d  .glue_7 00000000
0c71c1f8 l    d  .glue_7t    00000000
0c71c1f8 l    d  .rodata 00000000 ,    //.rodata 段空间为 0x0c71c1f8，结束为 0c721cf0－1，长度=00005af8
0c721cf0 l    d  .data    00000000      //.data 段空间开始 0x0c721cf0,.data 段空间结束为 0x0c722908-1
```

```
0c722908 l    d  .u_boot_cmd 00000000  //.u_boot_cmd 的段空间开始 0x 0c722908
0c722f28 l    d  .bss    00000000      //.bss 段空间开始 0x0c722f28
```

说明：启动部分汇编代码 start.S 有大量网络文档解释，本文档不赘述。

1.2 U-Boot U_BOOT_CMD 分析

```
struct cmd_tbl_s {
    char   *name;         /* Command Name*/
    int    maxargs;       /* maximum number of arguments */
    int    repeatable;   /* autorepeat allowed? */
    int    (*cmd)(struct cmd_tbl_s *, int, int, char *[]);/* Implementation function */
    char   *usage;       /* Usage message (short) */
#ifdef CFG_LONGHELP
    char       *help;     /* Help  message  (long) */
#endif
#ifdef CONFIG_AUTO_COMPLETE
    /* do auto completion on the arguments */
    int (*complete)(int argc, char *argv[], char last_char, int maxv, char *cmdv[]);
#endif
};  //结构体 total 为 7×4（字长）字节长度，一般使用为 6×4 字节长度，包括 help 项

typedef struct cmd_tbl_s cmd_tbl_t;
extern cmd_tbl_t  __u_boot_cmd_start;  //这两项在 u-boot.lds 文件内定义，见 1.1 中
extern cmd_tbl_t  __u_boot_cmd_end;

typedef   void   command_t (cmd_tbl_t *, int, int, char *[]);

#define Struct_Section  __attribute__ ((unused,section (".u_boot_cmd")))

#ifdef  CFG_LONGHELP
  #define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
  cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage, help}
#else  /* no long help info */
   #define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
   cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage}
#endif /* CFG_LONGHELP */

//宏定义的使用例子 from cmd_boot.c of u-boot 1.1.6
int do_go (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[]);
int do_reset (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[]);
U_BOOT_CMD(
    go, CFG_MAXARGS, 1,  do_go,
    "go     - start application at address 'addr'\n",
    "addr [arg ...]\n    - start application at address 'addr'\n"
```

```
    "      passing 'arg' as arguments\n"
);

U_BOOT_CMD(
    reset, 1, 0,  do_reset,
    "reset   - Perform RESET of the CPU\n",
    NULL
);
```

本章节分析 U_BOOT_CMD 采用 CFG_LONGHELP 已经定义的模式，即 cmd_tbl_s 结构体占据 6 个字长（假设字长标准 4,）空间为 24 字节，内容分别是：
第 1 字长 char* name 保存命令名字字符串的地址指针；而命令名字字符串保存在预初始化的.Data 空间；
第 2 字长 int maxargs 保存参数的个数，比较简单；
第 3 字长 int repeatable 保存是否允许 repeatable 的状态；
第 4 字长 int (*cmd) 保存调用函数入口指针，从 do_go 的案例观察，很 easy;
第 5 字长 char* usage 保存命令帮助字符串地址指针，字符串保存在预初始化的.Data 空间；
第 6 字长 char* help 也仅仅保存字符串地址指针，字符串保存在预初始化的.Data 空间；
很关键的理解是，char*的变量 4 个字节仅仅保存指向某个空间的指针值；

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
  cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage}
```

宏定义 U_BOOT_CMD 理解很关键，首先采用 cmd_tbl_t 定义 24 字节的变量__u_boot_cmd_##name，##name 的意思表示 name 所代表的字符串被拼装，比如 do_go 的定义时，变量为**__u_boot_cmd_go**；

而 Struct_Section =定义指定该变量的保存空间在段空间".u_boot_cmd"分配；

在{#name, maxargs, rep, cmd, usage}中#name 的意思表示 name 所代表的字符生成一个字符串，字符串在.Data 空间分配，结构体中 name 指针指向该字符串的地址，比如在 do_go 的命令中，#name 的结果为”go”字符串；

记住：宏定义生成的字符串的空间从.Data 空间分配，而不是从段".u_boot_cmd"分配，以保证 U_BOOT_CMD 每一项的定义在段".u_boot_cmd"占据固定长度；

1.3 U-Boot 网络功能分析

- U-Boot 的网络功能主要 u-boot/net 目录源代码，以及 u-boot/drivers 目录中的网卡驱动源代码实现。
- U-Boot 的网络功能在 net.c 文件中，而函数 int NetLoop(proto_t protocol)实现总的网络功能处理主循环。 Void NetReceive(volatile uchar * inpkt, int len) 函数被网卡驱动中调用接收处理 IP 报文。
- U-Boot 采用单进程处理方式，所以 NetLoop()每调用一次处理一个网络功能。
- U-Boot 主要的网络功能有：ARP/RARP/PING/TFTP/CDP/NFS，其中常用的为 ARP, PING, TFTP；参考 U-Boot 的网络功能，可以理解掌握基础的 Ethernet 和 IP 协议数据报文处理功能。

代码分析（略）

```
TFTP 正常处理时打印数据：
zenf=> tftp
```

```
TFTP from server 192.168.0.10; our IP address is 192.168.0.30
Filename 'u-boot.bin'.
Load address: 0xc008000
Loading: #############################
done
Bytes transferred = 143684 (23144 hex)
```

TFTP 不输入参数时，采用默认参数配置，加载地址 0x0c008000，加载文件为 u-boot.bin；带参数时的配置格式为：tftp addr "filename"，表示从已经配置默认 server IP 上获取文件 filename，加载到本地的 addr 空间；

TFTP 处理错误时数据：

```
TFTP from server 192.168.0.10; our IP address is 192.168.0.30
Filename 'u-boot.bin'.
Load address: 0xc008000
Loading: T T T T T T T T T T T T T T T T T T T T
Retry count exceeded; starting again
TFTP from server 192.168.0.10; our IP address is 192.168.0.30
Filename 'u-boot.bin'.
Load address: 0xc008000
Loading: T T T T T T T T T T T T T T T T T T T T
Retry count exceeded; starting again
```

1.4 U-Boot 命令行输入功能

u-boot/common/main.c 文件中 void main_loop (void) 为 u-boot 的主循环处理函数入口。主循环中检查串口输入字符，对字符串进行解析后执行相关指令，所有相关的字符串指令均基于 1.2 中分析的 U_BOOT_CMD 存储在相关 FLASH 空间。

解析字符串后，调用 int run_command (const char *cmd, int flag) 执行指令，cmd 为输入的完整字符串，flag 一般使用固定为 0。U-boot 基于字符串指令的第一个字（空格为单位），以字符串比较方式查询 FLASH 中所有存储的 CMD 执行命令（最大可存储的命令个数由编译时宏定义指定），操作过程由如下调用完成：

```
/* Extract arguments，从 cmd 中 提取相关的参数，结果类似 C 函数入口 argc, argv[][]模式*/
if ((argc = parse_line (finaltoken, argv)) == 0) {
    rc = -1;  /* no command at all */
    continue;
}
```

```
/* Look up command in command table，从 FLASH 的 cmd 表中查询对应执行命令，从查找的过程，也显示了 cmd_tbl_s 采用固定长度空间存储命令行指令的原因 */
if ((cmdtp = find_cmd(argv[0])) == NULL) {
    printf ("Unknown command '%s' - try 'help'\n", argv[0]);
    rc = -1;  /* give up after bad command */
    continue;
}
```

```
struct cmd_tbl_s {
    char    *name;          /* Command Name*/
    int     maxargs;        /* maximum number of arguments */
    int     repeatable;    /* autorepeat allowed? */
    int     (*cmd)(struct cmd_tbl_s *, int, int, char *[]);/* Implementation function */
    char    *usage;        /* Usage message (short) */
```

找到命令之后，基于 struct cmd_tbl_s 结构体定义以及其初始化时的函数入口赋值，采用 cmdtp->cmd(…)执行预先定义编译存储在 FLASH 中的函数，即完成 U-Boot 的命令行指令执行。

```
        /* OK - call function to do the command */
        if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {
            rc = -1;
        }
```

说明：关注 U-BOOT 中命令行入口函数的定义模式，以 int do_bootd (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])为例子，其入口参数分别为查询到的 cmd_tbl_s 入口指针，原有的 flag 参数，以及解析后的标准 argc 和 argv 参数。
再以 int do_run (cmd_tbl_t * cmdtp, int flag, int argc, char *argv[])为例子，模式相同；而 do_run 函数在初始化赋值时的定义如下：（基于 1.2 中分析，意义清楚）

```
U_BOOT_CMD(
    run,   CFG_MAXARGS, 1, do_run,
    "run     - run commands in an environment variable\n",
    "var [...]\n"
    "    - run the commands in the environment variable(s) 'var'\n"
);
```

1.5 U-Boot 内存 RAM 和 FLASH 功能分析

（1） CPU 启动 RAM 初始化设置

在 start.S 代码中的指令 bl memsetup 跳转至 memsetup.S 进行 RAM 内存配置初始化，典型的初始化汇编代码如下：（不同的 CPU 初始化配置需求不同，具体参考 CPU 的芯片手册）

```
.globl memsetup
memsetup:
    adr r0, MEMORY_CONFIG
    ldmia r0, {r1-r13}
    ldr    r0, =0x01c80000
    stmia r0, {r1-r13}
    mov pc, lr
```

（2） DRAM 的初始化

对 ARM 板子来说，启动完 start.S 后，跳转至 void start_armboot (void)的 C 代码开始执行，在 void start_armboot (void)中完成一系列的板子初始化，此时的初始化代码均为 C 代码。

说明：而对 PowerPC CPU 来说，如 PPC8260，则基于指令 bl board_init_f，调用 C 函数 void board_init_f (ulong bootflag)初始化板子。

板子初始化时调用一系列静态注册配置的函数，其中关于 RAM 初始化函数如下：

```
int dram_init (void)
{
    DECLARE_GLOBAL_DATA_PTR;
    gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
    gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;
    return (0);
}
```

本例子中笔者的板子仅仅配置有 8M RAM，初始化空间为地址 0x0c000000。

启动时板子 RAM 配置信息显示函数核心部分如下：

```
static int display_dram_config (void)
{
    puts ("RAM Configuration:\n");
    for(i=0; i<CONFIG_NR_DRAM_BANKS; i++) {
        printf ("Bank #%d: %08lx ", i, gd->bd->bi_dram[i].start);
        print_size (gd->bd->bi_dram[i].size, "\n"); }
}
```

（3） FLASH 的初始化 2M

板子初始化时调用 FLASH 初始化接口如下：

```
#ifndef CFG_NO_FLASH
    /* configure available FLASH banks */
    size = flash_init ();
    display_flash_config (size);
#endif /* CFG_NO_FLASH */

unsigned long flash_init (void)
{
#ifdef __DEBUG_START_FROM_SRAM__
    return CFG_DUMMY_FLASH_SIZE;
#else
    unsigned long size_b0;
    int i;

    /* Init: no FLASHes known */
    for (i=0; i<CFG_MAX_FLASH_BANKS; ++i) {
        flash_info[i].flash_id = FLASH_UNKNOWN;
    }

    /* Static FLASH Bank configuration here - FIXME XXX */
    size_b0 = flash_get_size((vu_long *)CFG_FLASH_BASE, &flash_info[0]);

    if (flash_info[0].flash_id == FLASH_UNKNOWN) {
        printf ("## Unknown FLASH on Bank 0 - Size = 0x%08lx = %ld MB\n",
            size_b0, size_b0<<20);}
```

```
    /* Setup offsets */
    flash_get_offsets (0, &flash_info[0]);

    /* Monitor protection ON by default */
    (void)flash_protect(FLAG_PROTECT_SET,
              -CFG_MONITOR_LEN,
              0xffffffff,
              &flash_info[0]);

    flash_info[0].size = size_b0;
    return (size_b0);
#endif
}
```

以下是笔者使用板子的 FLASH 配置时核心配置信息内容：

首先是基于 FLASH 型号获取板子 FLASH 容量和 FLASH 段数量。

```
    case (CFG_FLASH_WORD_SIZE)SST_ID_xF160A:
        info->flash_id += FLASH_SST160A;
        info->sector_count = 32;
        info->size = 0x00200000;
        break;
```

其次对 FLASH 段进行初始化，如下：

```
    /* set up sector start address table */
    if (((info->flash_id & FLASH_VENDMASK) == FLASH_MAN_SST) ||
        ((info->flash_id & FLASH_TYPEMASK) == FLASH_AM640U)) {
        for (i = 0; i < info->sector_count; i++)
        info->start[i] = base + (i * 0x00010000);
```

U-BOOT 使用的 FLASH 配置数据结构如下，最常用的为前五项：

```
/* FLASH Info: contains chip specific data, per FLASH bank*/
typedef struct {
    ulong  size;            /* total bank size in bytes       */
    ushort sector_count;    /* number of erase units       */
    ulong  flash_id;        /* combined device & manufacturer code   */
    ulong  start[CFG_MAX_FLASH_SECT];   /* physical sector start addresses */
    uchar  protect[CFG_MAX_FLASH_SECT]; /* sector protection status */

#ifdef CFG_FLASH_CFI
    uchar  portwidth;       /* the width of the port       */
    uchar  chipwidth;       /* the width of the chip       */
    ushort buffer_size;     /* # of bytes in write buffer     */
    ulong  erase_blk_tout;     /* maximum block erase timeout       */
    ulong  write_tout;      /* maximum write timeout       */
    ulong  buffer_write_tout;  /* maximum buffer write timeout      */
```

```
    ushort vendor;           /* the primary vendor id      */
    ushort cmd_reset;        /* Vendor specific reset command */
    ushort interface;        /* used for x8/x16 adjustments       */
    ushort legacy_unlock;       /* support Intel legacy (un)locking  */
#endif
} flash_info_t;
```

（4） FLASH 2M 的空间分配（段大小为 64K）

```
0X000000 0X03FFFF U-BOOT  64K*4     U-BOOT load 存储空间
0X040000 0X04FFFF PARA    64K*1     U-BOOT 系统参数存储空间
0X050000 0X17FFFF UCLINUX 64K*20    ucLinux load 存储空间
0X180000 0X1FFFFF OTHER   64K*7     其余空闲未用空间为 7 个段
总共 TOTAL 64K*32 为 2M 空间
```

说明：笔者编辑本文档用到的几个典型源文件如下：（基于 hfrks3c44b0 开发板）

说明：笔者保留文档内所有文字和图片版权，没有笔者同意，请勿做商用。

✔ 1 Flash.c (d:\work\uboot116\board\hfrk\hfrks3c44b0)
2 Board.c (d:\work\uboot116\lib_arm)
3 Console.c (d:\work\uboot116\common)
4 Main.c (d:\work\uboot116\common)
5 Hfrks3c44b0.c (d:\work\uboot116\board\hfrk\hfrks3c44b0)
6 memsetup.S (d:\work\uboot116\board\hfrk\hfrks3c44b0)
7 start.S (d:\work\uboot116\cpu\s3c44b0)
8 Net.c (d:\work\uboot116\net)

1.6 附录 1：u-boot 1.1.6 反汇编片断摘录

```
u-boot:     file format elf32-littlearm

Disassembly of section .text:

0c700000 <_start>:
 c700000: ea00000a   b   c700030 <reset>
 c700004: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c700008: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c70000c: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c700010: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c700014: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c700018: e28ff303   add pc, pc, #201326592  ; 0xc000000
 c70001c: e28ff303   add pc, pc, #201326592  ; 0xc000000

0c700020 <_TEXT_BASE>:
 c700020: 0c700000   ldceql 0, cr0, [r0]
```

```
0c700024 <_armboot_start>:
 c700024: 0c700000  ldceql 0, cr0, [r0]

0c700028 <_bss_start>:
 c700028: 0c72314c  ldfeqe f3, [r2], -#304

0c70002c <_bss_end>:
 c70002c: 0c757ec8  ldceql 14, cr7, [r5], -#800

0c700030 <reset>:
 c700030: e10f0000  mrs r0, CPSR
 c700034: e3c0001f  bic r0, r0, #31   ; 0x1f
 c700038: e3800013  orr r0, r0, #19   ; 0x13
 c70003c: e129f000  msr CPSR_fc, r0
 c700040: eb00001a  bl  c7000b0 <cpu_init_crit>
 c700044: eb000056  bl  c7001a4 <memsetup>

0c700048 <relocate>:
 c700048: e24f0050  sub r0, pc, #80   ; 0x50
 c70004c: e51f1034  ldr r1, [pc, #-52]   ; c700020 <_TEXT_BASE>
 c700050: e1500001  cmp r0, r1
 c700054: 0a00000f  beq c700098 <stack_setup>
 c700058: e51f203c  ldr r2, [pc, #-60]   ; c700024 <_armboot_start>
 c70005c: e51f303c  ldr r3, [pc, #-60]   ; c700028 <_bss_start>
 c700060: e0432002  sub r2, r3, r2
 c700064: e0802002  add r2, r0, r2

0c700068 <copy_loop>:
 c700068: e8b007f8  ldmia  r0!, {r3, r4, r5, r6, r7, r8, r9, sl}
 c70006c: e8a107f8  stmia  r1!, {r3, r4, r5, r6, r7, r8, r9, sl}
 c700070: e1500002  cmp r0, r2
 c700074: dafffffb  ble c700068 <copy_loop>
 c700078: e28f007c  add r0, pc, #124  ; 0x7c
 c70007c: e2802b01  add r2, r0, #1024 ; 0x400
 c700080: e3a01303  mov r1, #201326592   ; 0xc000000
 c700084: e2811008  add r1, r1, #8    ; 0x8

0c700088 <vector_copy_loop>:
 c700088: e8b007f8  ldmia  r0!, {r3, r4, r5, r6, r7, r8, r9, sl}
 c70008c: e8a107f8  stmia  r1!, {r3, r4, r5, r6, r7, r8, r9, sl}
 c700090: e1500002  cmp r0, r2
 c700094: dafffffb  ble c700088 <vector_copy_loop>

0c700098 <stack_setup>:
 c700098: e51f0080  ldr r0, [pc, #-128]  ; c700020 <_TEXT_BASE>
```

```
 c70009c: e2400803  sub r0, r0, #196608  ; 0x30000
 c7000a0: e2400080  sub r0, r0, #128 ; 0x80
 c7000a4: e240d00c  sub sp, r0, #12   ; 0xc
 c7000a8: e51ff004  ldr pc, [pc, #-4] ; c7000ac <_start_armboot>

0c7000ac <_start_armboot>:
 c7000ac: 0c70036c  ldceql 3, cr0, [r0], -#432

0c7000b0 <cpu_init_crit>:
 c7000b0: e59f009c  ldr r0, [pc, #156]   ; c700154 <fiq+0x8>
 c7000b4: e3a01000  mov r1, #0 ; 0x0
 c7000b8: e5801000  str r1, [r0]
 c7000bc: e59f1094  ldr r1, [pc, #148]   ; c700158 <fiq+0xc>
 c7000c0: e59f0094  ldr r0, [pc, #148]   ; c70015c <fiq+0x10>
 c7000c4: e5810000  str r0, [r1]
 c7000c8: e3a0161e  mov r1, #31457280 ; 0x1e00000
 c7000cc: e3a00005  mov r0, #5 ; 0x5
 c7000d0: e5810000  str r0, [r1]
 c7000d4: e59f1084  ldr r1, [pc, #132]   ; c700160 <fiq+0x14>
 c7000d8: e3a00e32  mov r0, #800  ; 0x320
 c7000dc: e5c10000  strb  r0, [r1]
 c7000e0: e3a01776  mov r1, #30932992 ; 0x1d80000
 c7000e4: e59f0078  ldr r0, [pc, #120]   ; c700164 <fiq+0x18>
 c7000e8: e5810000  str r0, [r1]
 c7000ec: e59f1074  ldr r1, [pc, #116]   ; c700168 <fiq+0x1c>
 c7000f0: e59f0074  ldr r0, [pc, #116]   ; c70016c <fiq+0x20>
 c7000f4: e5810000  str r0, [r1]
 c7000f8: e1a0f00e  mov pc, lr

0c7000fc <real_vectors>:
 c7000fc: eaffffcb  b  c700030 <reset>
 c700100: ea000005  b  c70011c <undefined_instruction>
 c700104: ea000006  b  c700124 <software_interrupt>
 c700108: ea000007  b  c70012c <prefetch_abort>
 c70010c: ea000008  b  c700134 <data_abort>
 c700110: ea000009  b  c70013c <not_used>
 c700114: ea00000a  b  c700144 <irq>
 c700118: ea00000b  b  c70014c <fiq>

0c70011c <undefined_instruction>:
 c70011c: e3a06003  mov r6, #3 ; 0x3
 c700120: eaffffc2  b  c700030 <reset>

0c700124 <software_interrupt>:
 c700124: e3a06004  mov r6, #4 ; 0x4
 c700128: eaffffc0  b  c700030 <reset>
```

```
0c70012c <prefetch_abort>:
 c70012c: e3a06005  mov r6, #5 ; 0x5
 c700130: eaffffbe  b   c700030 <reset>

0c700134 <data_abort>:
 c700134: e3a06006  mov r6, #6 ; 0x6
 c700138: eaffffbc  b   c700030 <reset>

0c70013c <not_used>:
 c70013c: e3a06007  mov r6, #7 ; 0x7
 c700140: eaffffba  b   c700030 <reset>

0c700144 <irq>:
 c700144: e3a06008  mov r6, #8 ; 0x8
 c700148: eaffffb8  b   c700030 <reset>

0c70014c <fiq>:
 c70014c: e3a06009  mov r6, #9 ; 0x9
 c700150: eaffffb6  b   c700030 <reset>
 c700154: 01d30000  biceqs r0, r3, r0
 c700158: 01e0000c  mvneq  r0, ip
 c70015c: 03fffeff  mvneqs pc, #4080 ; 0xff0
 c700160: 01d8000c  biceqs r0, r8, ip
 c700164: 00088042  andeq  r8, r8, r2, asr #32
 c700168: 01d80004  biceqs r0, r8, r4
 c70016c: 00007ff8  streqd r7, [r0], -r8

0c700170 <MEMORY_CONFIG>:
 c700170: 01001102  tsteq  r0, r2, lsl #2
 c700174: 00007ff4  streqd r7, [r0], -r4
 c700178: 00000a40  andeq  r0, r0, r0, asr #20
 c70017c: 000014bc  streqh r1, [r0], -ip
 c700180: 00007ffc  streqd r7, [r0], -ip
 c700184: 00007ffc  streqd r7, [r0], -ip
 c700188: 00000c40  andeq  r0, r0, r0, asr #24
 c70018c: 00018004  andeq  r8, r1, r4
 c700190: 00018004  andeq  r8, r1, r4
 c700194: 008c060e  addeq  r0, ip, lr, lsl #12
 c700198: 00000010  andeq  r0, r0, r0, lsl r0
 c70019c: 00000020  andeq  r0, r0, r0, lsr #32
 c7001a0: 00000020  andeq  r0, r0, r0, lsr #32

0c7001a4 <memsetup>:
 c7001a4: e24f003c  sub r0, pc, #60   ; 0x3c
 c7001a8: e8903ffe  ldmia  r0, {r1, r2, r3, r4, r5, r6, r7, r8, r9,
```

```
sl, fp, ip, sp}
 c7001ac: e3a00772  mov r0, #29884416 ; 0x1c80000
 c7001b0: e8803ffe  stmia  r0, {r1, r2, r3, r4, r5, r6, r7, r8, r9,
sl, fp, ip, sp}
 c7001b4: e1a0f00e  mov pc, lr

0c7001b8 <mem_malloc_init>:
 c7001b8: e1a0c00d  mov ip, sp
 c7001bc: e92dd800  stmdb  sp!, {fp, ip, lr, pc}
 c7001c0: e59f202c  ldr r2, [pc, #44] ;                    c7001f4
<mem_malloc_init+0x3c>
 c7001c4: e24cb004  sub fp, ip, #4   ; 0x4
 c7001c8: e59f1028  ldr r1, [pc, #40] ;                    c7001f8
<mem_malloc_init+0x40>
 c7001cc: e1a03000  mov r3, r0
 c7001d0: e5823000  str r3, [r2]
 c7001d4: e2832803  add r2, r3, #196608  ; 0x30000
 c7001d8: e5812000  str r2, [r1]
 c7001dc: e59f1018  ldr r1, [pc, #24] ;                    c7001fc
<mem_malloc_init+0x44>
 c7001e0: e0602002  rsb r2, r0, r2
 c7001e4: e5813000  str r3, [r1]
 c7001e8: e3a01000  mov r1, #0 ; 0x0
 c7001ec: eb0040ca  bl  c71051c <memset>
 c7001f0: e91ba800  ldmdb  fp, {fp, sp, pc}
 c7001f4: 0c721ef8  ldceql 14, cr1, [r2], -#992
 c7001f8: 0c721efc  ldceql 14, cr1, [r2], -#1008
 c7001fc: 0c721f00  ldceql 15, cr1, [r2]

0c700200 <sbrk>:
 c700200: e1a0c00d  mov ip, sp
 c700204: e92dd800  stmdb  sp!, {fp, ip, lr, pc}
 c700208: e24cb004  sub fp, ip, #4   ; 0x4
 c70020c: e59fc038  ldr ip, [pc, #56] ; c70024c <sbrk+0x4c>
 c700210: e59f3038  ldr r3, [pc, #56] ; c700250 <sbrk+0x50>
 c700214: e59c1000  ldr r1, [ip]
 c700218: e5933000  ldr r3, [r3]
 c70021c: e0812000  add r2, r1, r0
 c700220: e1520003  cmp r2, r3
 c700224: 3a000003  bcc c700238 <sbrk+0x38>
 c700228: e59f3024  ldr r3, [pc, #36] ; c700254 <sbrk+0x54>
 c70022c: e5933000  ldr r3, [r3]
 c700230: e1520003  cmp r2, r3
 c700234: 9a000001  bls c700240 <sbrk+0x40>
 c700238: e3a00000  mov r0, #0 ; 0x0
 c70023c: e91ba800  ldmdb  fp, {fp, sp, pc}
```

```
 c700240: e1a00001  mov r0, r1
 c700244: e58c2000  str r2, [ip]
 c700248: e91ba800  ldmdb  fp, {fp, sp, pc}
 c70024c: 0c721f00  ldceql 15, cr1, [r2]
 c700250: 0c721ef8  ldceql 14, cr1, [r2], -#992
 c700254: 0c721efc  ldceql 14, cr1, [r2], -#1008

0c700258 <init_baudrate>:
 c700258: e1a0c00d  mov ip, sp
 c70025c: e92dd830  stmdb  sp!, {r4, r5, fp, ip, lr, pc}
 c700260: e24cb004  sub fp, ip, #4    ; 0x4
 c700264: e24b4054  sub r4, fp, #84   ; 0x54
 c700268: e1a01004  mov r1, r4
 c70026c: e3a02040  mov r2, #64   ; 0x40
 c700270: e59f0024  ldr r0, [pc, #36] ; c70029c <init_baudrate+0x44>
 c700274: e24dd040  sub sp, sp, #64   ; 0x40
 c700278: eb002e4a  bl  c70bba8 <getenv_r>
 c70027c: e3500000  cmp r0, #0 ; 0x0
 c700280: e5985000  ldr r5, [r8]
 c700284: da000005  ble c7002a0 <init_baudrate+0x48>
 c700288: e1a00004  mov r0, r4
 c70028c: e3a01000  mov r1, #0 ; 0x0
 c700290: e3a0200a  mov r2, #10   ; 0xa
 c700294: eb004129  bl  c710740 <simple_strtoul>
 c700298: ea000001  b   c7002a4 <init_baudrate+0x4c>
 c70029c: 0c71c238  lfmeq  f4, 3, [r1], -#224
 c7002a0: e59f0010  ldr r0, [pc, #16] ; c7002b8 <init_baudrate+0x60>
 c7002a4: e5880008  str r0, [r8, #8]
 c7002a8: e5983008  ldr r3, [r8, #8]
 c7002ac: e3a00000  mov r0, #0 ; 0x0
 c7002b0: e5853000  str r3, [r5]
 c7002b4: ea000000  b   c7002bc <init_baudrate+0x64>
 c7002b8: 0001c200  andeq  ip, r1, r0, lsl #4
 c7002bc: e91ba830  ldmdb  fp, {r4, r5, fp, sp, pc}

0c7002c0 <display_banner>:
 c7002c0: e1a0c00d  mov ip, sp
 c7002c4: e92dd800  stmdb  sp!, {fp, ip, lr, pc}
 c7002c8: e59f0010  ldr r0, [pc, #16] ;                          c7002e0
<display_banner+0x20>
 c7002cc: e59f1010  ldr r1, [pc, #16] ;                          c7002e4
<display_banner+0x24>
 c7002d0: e24cb004  sub fp, ip, #4    ; 0x4
 c7002d4: eb0031de  bl  c70ca54 <printf>
 c7002d8: e3a00000  mov r0, #0 ; 0x0
 c7002dc: e91ba800  ldmdb  fp, {fp, sp, pc}
```

```
 c7002e0: 0c71c244  lfmeq  f4, 3, [r1], -#272
 c7002e4: 0c71c1f8  ldfeqp f4, [r1], -#992

0c7002e8 <display_dram_config>:
 c7002e8: e1a0c00d  mov ip, sp
 c7002ec: e92dd810  stmdb  sp!, {r4, fp, ip, lr, pc}
 c7002f0: e24cb004  sub fp, ip, #4    ; 0x4
 c7002f4: e3a04000  mov r4, #0 ; 0x0
 c7002f8: e1a02004  mov r2, r4
 c7002fc: e5983000  ldr r3, [r8]
 c700300: e2833020  add r3, r3, #32   ; 0x20
 c700304: e7933182  ldr r3, [r3, r2, lsl #3]
 c700308: e2822001  add r2, r2, #1    ; 0x1
 c70030c: e3520000  cmp r2, #0 ; 0x0
 c700310: e0844003  add r4, r4, r3
 c700314: daffff f8  ble c7002fc <display_dram_config+0x14>
 c700318: e59f0014  ldr r0, [pc, #20] ;                      c700334
<display_dram_config+0x4c>
 c70031c: eb0031bf  bl c70ca20 <puts>
 c700320: e59f1010  ldr r1, [pc, #16] ;                      c700338
<display_dram_config+0x50>
 c700324: e1a00004  mov r0, r4
 c700328: eb003f21  bl c70ffb4 <print_size>
 c70032c: e3a00000  mov r0, #0 ; 0x0
 c700330: e91ba810  ldmdb  fp, {r4, fp, sp, pc}
 c700334: 0c71c24c  lfmeq  f4, 3, [r1], -#304
 c700338: 0c71c254  lfmeq  f4, 3, [r1], -#336

0c70033c <display_flash_config>:
 c70033c: e1a0c00d  mov ip, sp
 c700340: e92dd810  stmdb  sp!, {r4, fp, ip, lr, pc}
 c700344: e1a04000  mov r4, r0
 c700348: e59f0014  ldr r0, [pc, #20] ;                      c700364
<display_flash_config+0x28>
 c70034c: e24cb004  sub fp, ip, #4    ; 0x4
 c700350: eb0031b2  bl c70ca20 <puts>
 c700354: e59f100c  ldr r1, [pc, #12] ;                      c700368
<display_flash_config+0x2c>
 c700358: e1a00004  mov r0, r4
 c70035c: eb003f14  bl c70ffb4 <print_size>
 c700360: e91ba810  ldmdb  fp, {r4, fp, sp, pc}
 c700364: 0c71c258  lfmeq  f4, 3, [r1], -#352
 c700368: 0c71c254  lfmeq  f4, 3, [r1], -#336
```