6.3.2 软件跟踪

假如U-Boot没有任何串口打印信息，手头又没有硬件调试工具，那样怎么知道U-Boot执行到什么地方了呢？可以通过开发板上的LED指示灯判断。

开发板上最好设计安装八段数码管等LED，可以用来显示数字或者数字位。

U-Boot可以定义函数show\_boot\_progress (int status)，用来指示当前启动进度。在include/common.h头文件中声明这个函数。

#ifdef CONFIG\_SHOW\_BOOT\_PROGRESS

void    show\_boot\_progress (int status);

#endif

CONFIG\_SHOW\_BOOT\_PROGRESS是需要定义的。这个在板子配置的头文件中定义。CSB226开发板对这项功能有完整实现，可以参考。在头文件include/configs/csb226.h中，有下列一行。

#define CONFIG\_SHOW\_BOOT\_PROGRESS       1

函数show\_boot\_progress (int status)的实现跟开发板关系密切，所以一般在board目录下的文件中实现。看一下CSB226在board/csb226/csb226.c中的实现函数。

/\*\* 设置CSB226板的0、1、2三个指示灯的开关状态

 \* csb226\_set\_led: - switch LEDs on or off

 \* @param led:   LED to switch (0,1,2)

 \* @param state: switch on (1) or off (0)

 \*/

void csb226\_set\_led(int led, int state)

{

      switch(led) {

             case 0: if (state==1) {

                              GPCR0 |= CSB226\_USER\_LED0;

                    } else if (state==0) {

                            GPSR0 |= CSB226\_USER\_LED0;

                    }

                    break;

             case 1: if (state==1) {

                              GPCR0 |= CSB226\_USER\_LED1;

                    } else if (state==0) {

                              GPSR0 |= CSB226\_USER\_LED1;

                    }

                    break;

             case 2: if (state==1) {

                              GPCR0 |= CSB226\_USER\_LED2;

                  } else if (state==0) {

                          GPSR0 |= CSB226\_USER\_LED2;

                  }

                  break;

      }

      return;

}

/\*\* 显示启动进度函数，在比较重要的阶段，设置三个灯为亮的状态（1, 5, 15）\*/

void show\_boot\_progress (int status)

{

      switch(status) {

            case  1: csb226\_set\_led(0,1); break;

            case  5: csb226\_set\_led(1,1); break;

            case 15: csb226\_set\_led(2,1); break;

      }

      return;

}

这样，在U-Boot启动过程中就可以通过show\_boot\_progresss指示执行进度。比如hang()函数是系统出错时调用的函数，这里需要根据特定的开发板给定显示的参数值。

void hang (void)

{

      puts ("### ERROR ### Please RESET the board ###\n");

#ifdef CONFIG\_SHOW\_BOOT\_PROGRESS

      show\_boot\_progress(-30);

#endif

      for (;;);

}

6.3.3  U-Boot启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行U-Boot的第一条指令，然后顺序执行U-Boot启动函数。函数调用顺序如图6.3所示。

看一下board/smsk2410/u-boot.lds这个链接脚本，可以知道目标程序的各部分链接顺序。第一个要链接的是cpu/arm920t/start.o，那么U-Boot的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

1．cpu/arm920t/start.S

这个汇编程序是U-Boot的入口程序，开头就是复位向量的代码。

图6.3  U-Boot启动代码流程图

\_start: b       reset        //复位向量

       ldr   pc, \_undefined\_instruction

       ldr   pc, \_software\_interrupt

       ldr   pc, \_prefetch\_abort

       ldr   pc, \_data\_abort

       ldr   pc, \_not\_used

       ldr   pc, \_irq      //中断向量

       ldr   pc, \_fiq      //中断向量

…

 /\* the actual reset code  \*/

reset:          //复位启动子程序

       /\* 设置CPU为SVC32模式 \*/

       mrs   r0,cpsr

       bic   r0,r0,#0x1f

       orr   r0,r0,#0xd3

       msr   cpsr,r0

/\* 关闭看门狗 \*/

/\* 这些初始化代码在系统重起的时候执行，运行时热复位从RAM中启动不执行 \*/

#ifdef CONFIG\_INIT\_CRITICAL

       bl    cpu\_init\_crit

#endif

relocate:                       /\* 把U-Boot重新定位到RAM \*/

       adr   r0, \_start          /\* r0是代码的当前位置 \*/

       ldr   r1, \_TEXT\_BASE      /\* 测试判断是从Flash启动，还是RAM \*/

       cmp     r0, r1          /\* 比较r0和r1，调试的时候不要执行重定位 \*/

       beq     stack\_setup    /\* 如果r0等于r1，跳过重定位代码 \*/

       /\* 准备重新定位代码 \*/

       ldr   r2, \_armboot\_start

       ldr   r3, \_bss\_start

       sub   r2, r3, r2          /\* r2 得到armboot的大小   \*/

       add   r2, r0, r2          /\* r2 得到要复制代码的末尾地址 \*/

copy\_loop: /\* 重新定位代码 \*/

       ldmia r0!, {r3-r10}   /\*从源地址[r0]复制 \*/

       stmia r1!, {r3-r10}   /\* 复制到目的地址[r1] \*/

       cmp   r0, r2          /\* 复制数据块直到源数据末尾地址[r2] \*/

       ble   copy\_loop

       /\* 初始化堆栈等    \*/

stack\_setup:

       ldr   r0, \_TEXT\_BASE              /\* 上面是128 KiB重定位的u-boot \*/

       sub   r0, r0, #CFG\_MALLOC\_LEN     /\* 向下是内存分配空间 \*/

       sub   r0, r0, #CFG\_GBL\_DATA\_SIZE /\* 然后是bdinfo结构体地址空间  \*/

#ifdef CONFIG\_USE\_IRQ

       sub   r0, r0, #(CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ)

#endif

       sub   sp, r0, #12     /\* 为abort-stack预留3个字 \*/

clear\_bss:

       ldr   r0, \_bss\_start      /\* 找到bss段起始地址 \*/

       ldr   r1, \_bss\_end        /\*  bss段末尾地址   \*/

       mov   r2, #0x00000000     /\* 清零 \*/

clbss\_l:str r2, [r0]        /\* bss段地址空间清零循环...  \*/

       add   r0, r0, #4

       cmp   r0, r1

       bne   clbss\_l

       /\* 跳转到start\_armboot函数入口，\_start\_armboot字保存函数入口指针 \*/

       ldr   pc, \_start\_armboot

\_start\_armboot: .word start\_armboot     //start\_armboot函数在lib\_arm/board.c中实现

/\* 关键的初始化子程序 \*/

cpu\_init\_crit:

……  //初始化CACHE，关闭MMU等操作指令

       /\* 初始化RAM时钟。

       \* 因为内存时钟是依赖开发板硬件的，所以在board的相应目录下可以找到memsetup.S文件。

       \*/

       mov   ip, lr

       bl    memsetup        //memsetup子程序在board/smdk2410/memsetup.S中实现

       mov   lr, ip

       mov   pc, lr

2．lib\_arm/board.c

start\_armboot是U-Boot执行的第一个C语言函数，完成系统初始化工作，进入主循环，处理用户输入的命令。

void start\_armboot (void)

{

       DECLARE\_GLOBAL\_DATA\_PTR;

       ulong size;

       init\_fnc\_t \*\*init\_fnc\_ptr;

       char \*s;

       /\* Pointer is writable since we allocated a register for it \*/

       gd = (gd\_t\*)(\_armboot\_start - CFG\_MALLOC\_LEN - sizeof(gd\_t));

       /\* compiler optimization barrier needed for GCC >= 3.4 \*/

       \_\_asm\_\_ \_\_volatile\_\_("": : :"memory");

       memset ((void\*)gd, 0, sizeof (gd\_t));

       gd->bd = (bd\_t\*)((char\*)gd - sizeof(bd\_t));

       memset (gd->bd, 0, sizeof (bd\_t));

       monitor\_flash\_len = \_bss\_start - \_armboot\_start;

       /\* 顺序执行init\_sequence数组中的初始化函数 \*/

       for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {

              if ((\*init\_fnc\_ptr)() != 0) {

                      hang ();

              }

       }

       /\*配置可用的Flash \*/

       size = flash\_init ();

       display\_flash\_config (size);

       /\* \_armboot\_start 在u-boot.lds链接脚本中定义 \*/

       mem\_malloc\_init (\_armboot\_start - CFG\_MALLOC\_LEN);

       /\* 配置环境变量，重新定位 \*/

       env\_relocate ();

       /\* 从环境变量中获取IP地址 \*/

       gd->bd->bi\_ip\_addr = getenv\_IPaddr ("ipaddr");

       /\* 以太网接口MAC 地址 \*/

       ……

       devices\_init ();      /\* 获取列表中的设备 \*/

       jumptable\_init ();

       console\_init\_r ();    /\* 完整地初始化控制台设备 \*/

       enable\_interrupts (); /\* 使能例外处理 \*/

       /\* 通过环境变量初始化 \*/

       if ((s = getenv ("loadaddr")) != NULL) {

               load\_addr = simple\_strtoul (s, NULL, 16);

       }

       /\* main\_loop()总是试图自动启动，循环不断执行 \*/

       for (;;) {

               main\_loop ();      /\* 主循环函数处理执行用户命令 -- common/main.c \*/

       }

       /\* NOTREACHED - no way out of command loop except booting \*/

}

3．init\_sequence[]

init\_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

init\_fnc\_t \*init\_sequence[] = {

       cpu\_init,             /\* 基本的处理器相关配置 -- cpu/arm920t/cpu.c \*/

       board\_init,           /\* 基本的板级相关配置 -- board/smdk2410/smdk2410.c \*/

       interrupt\_init,       /\* 初始化例外处理 -- cpu/arm920t/s3c24x0/interrupt.c \*/

       env\_init,             /\* 初始化环境变量 -- common/cmd\_flash.c \*/

       init\_baudrate,        /\* 初始化波特率设置 -- lib\_arm/board.c \*/

       serial\_init,          /\* 串口通讯设置 -- cpu/arm920t/s3c24x0/serial.c \*/

       console\_init\_f,       /\* 控制台初始化阶段1 -- common/console.c \*/

       display\_banner,       /\* 打印u-boot信息 -- lib\_arm/board.c \*/

       dram\_init,            /\* 配置可用的RAM -- board/smdk2410/smdk2410.c \*/

       display\_dram\_config,  /\* 显示RAM的配置大小 -- lib\_arm/board.c \*/

       NULL,

};

6.3.4  U-Boot与内核的关系

U-Boot作为Bootloader，具备多种引导内核启动的方式。常用的go和bootm命令可以直接引导内核映像启动。U-Boot与内核的关系主要是内核启动过程中参数的传递。

1．go命令的实现

/\* common/cmd\_boot.c  \*/

int do\_go (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

       ulong addr, rc;

       int     rcode = 0;

       if (argc < 2) {

              printf ("Usage:\n%s\n", cmdtp->usage);

              return 1;

       }

       addr = simple\_strtoul(argv[1], NULL, 16);

       printf ("## Starting application at 0x%08lX ...\n", addr);

       /\*

        \* pass address parameter as argv[0] (aka command name),

        \* and all remaining args

        \*/

       rc = ((ulong (\*)(int, char \*[]))addr) (--argc, &argv[1]);

       if (rc != 0) rcode = 1;

       printf ("## Application terminated, rc = 0x%lX\n", rc);

       return rcode;

}

go命令调用do\_go()函数，跳转到某个地址执行的。如果在这个地址准备好了自引导的内核映像，就可以启动了。尽管go命令可以带变参，实际使用时一般不用来传递参数。

2．bootm命令的实现

/\* common/cmd\_bootm.c \*/

int do\_bootm (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

       ulong iflag;

       ulong addr;

       ulong data, len, checksum;

       ulong  \*len\_ptr;

       uint  unc\_len = 0x400000;

       int   i, verify;

       char  \*name, \*s;

       int   (\*appl)(int, char \*[]);

       image\_header\_t \*hdr = &header;

       s = getenv ("verify");

       verify = (s && (\*s == 'n')) ? 0 : 1;

       if (argc < 2) {

              addr = load\_addr;

       } else {

              addr = simple\_strtoul(argv[1], NULL, 16);

       }

       SHOW\_BOOT\_PROGRESS (1);

       printf ("## Booting image at %08lx ...\n", addr);

       /\* Copy header so we can blank CRC field for re-calculation \*/

       memmove (&header, (char \*)addr, sizeof(image\_header\_t));

       if (ntohl(hdr->ih\_magic) != IH\_MAGIC)

       {

              puts ("Bad Magic Number\n");

              SHOW\_BOOT\_PROGRESS (-1);

              return 1;

       }

       SHOW\_BOOT\_PROGRESS (2);

       data = (ulong)&header;

       len  = sizeof(image\_header\_t);

       checksum = ntohl(hdr->ih\_hcrc);

       hdr->ih\_hcrc = 0;

       if(crc32 (0, (char \*)data, len) != checksum) {

              puts ("Bad Header Checksum\n");

              SHOW\_BOOT\_PROGRESS (-2);

              return 1;

       }

       SHOW\_BOOT\_PROGRESS (3);

       /\* for multi-file images we need the data part, too \*/

       print\_image\_hdr ((image\_header\_t \*)addr);

       data = addr + sizeof(image\_header\_t);

       len  = ntohl(hdr->ih\_size);

       if(verify) {

              puts ("   Verifying Checksum ... ");

              if(crc32 (0, (char \*)data, len) != ntohl(hdr->ih\_dcrc)) {

                     printf ("Bad Data CRC\n");

                     SHOW\_BOOT\_PROGRESS (-3);

                     return 1;

              }

              puts ("OK\n");

       }

       SHOW\_BOOT\_PROGRESS (4);

       len\_ptr = (ulong \*)data;

……

       switch (hdr->ih\_os) {

       default:                /\* handled by (original) Linux case \*/

       case IH\_OS\_LINUX:

             do\_bootm\_linux  (cmdtp, flag, argc, argv,

                         addr, len\_ptr, verify);

             break;

       ……

}

bootm命令调用do\_bootm函数。这个函数专门用来引导各种操作系统映像，可以支持引导Linux、vxWorks、QNX等操作系统。引导Linux的时候，调用do\_bootm\_linux()函数。

3．do\_bootm\_linux函数的实现

/\* lib\_arm/armlinux.c \*/

void do\_bootm\_linux (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[],

                   ulong addr, ulong \*len\_ptr, int verify)

{

       DECLARE\_GLOBAL\_DATA\_PTR;

       ulong len = 0, checksum;

       ulong initrd\_start, initrd\_end;

       ulong data;

       void (\*theKernel)(int zero, int arch, uint params);

       image\_header\_t \*hdr = &header;

       bd\_t \*bd = gd->bd;

#ifdef CONFIG\_CMDLINE\_TAG

       char \*commandline = getenv ("bootargs");

#endif

       theKernel = (void (\*)(int, int, uint))ntohl(hdr->ih\_ep);

       /\* Check if there is an initrd image \*/

       if(argc >= 3) {

              SHOW\_BOOT\_PROGRESS (9);

              addr = simple\_strtoul (argv[2], NULL, 16);

              printf ("## Loading Ramdisk Image at %08lx ...\n", addr);

              /\* Copy header so we can blank CRC field for re-calculation \*/

              memcpy (&header, (char \*) addr, sizeof (image\_header\_t));

              if (ntohl (hdr->ih\_magic) != IH\_MAGIC) {

                      printf ("Bad Magic Number\n");

                      SHOW\_BOOT\_PROGRESS (-10);

                      do\_reset (cmdtp, flag, argc, argv);

              }

              data = (ulong) & header;

              len = sizeof (image\_header\_t);

              checksum = ntohl (hdr->ih\_hcrc);

              hdr->ih\_hcrc = 0;

              if(crc32 (0, (char \*) data, len) != checksum) {

                     printf ("Bad Header Checksum\n");

                     SHOW\_BOOT\_PROGRESS (-11);

                     do\_reset (cmdtp, flag, argc, argv);

              }

              SHOW\_BOOT\_PROGRESS (10);

              print\_image\_hdr (hdr);

              data = addr + sizeof (image\_header\_t);

              len = ntohl (hdr->ih\_size);

              if(verify) {

                     ulong csum = 0;

                     printf ("   Verifying Checksum ... ");

                     csum = crc32 (0, (char \*) data, len);

                     if (csum != ntohl (hdr->ih\_dcrc)) {

                            printf ("Bad Data CRC\n");

                            SHOW\_BOOT\_PROGRESS (-12);

                            do\_reset (cmdtp, flag, argc, argv);

                     }

                     printf ("OK\n");

              }

              SHOW\_BOOT\_PROGRESS (11);

              if ((hdr->ih\_os != IH\_OS\_LINUX) ||

                     (hdr->ih\_arch != IH\_CPU\_ARM) ||

                     (hdr->ih\_type != IH\_TYPE\_RAMDISK)) {

                     printf ("No Linux ARM Ramdisk Image\n");

                     SHOW\_BOOT\_PROGRESS (-13);

                     do\_reset (cmdtp, flag, argc, argv);

              }

              /\* Now check if we have a multifile image \*/

       } else if ((hdr->ih\_type == IH\_TYPE\_MULTI) && (len\_ptr[1])) {

               ulong tail = ntohl (len\_ptr[0]) % 4;

               int i;

               SHOW\_BOOT\_PROGRESS (13);

               /\* skip kernel length and terminator \*/

               data = (ulong) (&len\_ptr[2]);

               /\* skip any additional image length fields \*/

               for (i = 1; len\_ptr[i]; ++i)

                       data += 4;

              /\* add kernel length, and align \*/

              data += ntohl (len\_ptr[0]);

              if (tail) {

                       data += 4 - tail;

              }

              len = ntohl (len\_ptr[1]);

       } else {

               /\* no initrd image \*/

              SHOW\_BOOT\_PROGRESS (14);

              len = data = 0;

       }

       if (data) {

               initrd\_start = data;

               initrd\_end = initrd\_start + len;

       } else {

               initrd\_start = 0;

               initrd\_end = 0;

       }

       SHOW\_BOOT\_PROGRESS (15);

       debug ("## Transferring control to Linux (at address %08lx) ...\n",

               (ulong) theKernel);

#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)

      setup\_start\_tag (bd);

#ifdef CONFIG\_SERIAL\_TAG

      setup\_serial\_tag (¶ms);

#endif

#ifdef CONFIG\_REVISION\_TAG

      setup\_revision\_tag (¶ms);

#endif

#ifdef CONFIG\_SETUP\_MEMORY\_TAGS

      setup\_memory\_tags (bd);

#endif

#ifdef CONFIG\_CMDLINE\_TAG

      setup\_commandline\_tag (bd, commandline);

#endif

#ifdef CONFIG\_INITRD\_TAG

      if (initrd\_start && initrd\_end)

               setup\_initrd\_tag (bd, initrd\_start, initrd\_end);

#endif

      setup\_end\_tag (bd);

#endif

      /\* we assume that the kernel is in place \*/

      printf ("\nStarting kernel ...\n\n");

      cleanup\_before\_linux ();

      theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);

}