(1条消息) u-boot-2021.01（imx6ull）启动流程分析之六：以bootz命令为例追踪u-boot启动内核过程___ASDFGH的博客-CSDN博客

4、以bootz为例追踪u-boot启动内核过程

bootz命令的定义可以在`cmd/bootz.c`文件中找到，它的声明如下：

```
U_BOOT_CMD(
	bootz,	CONFIG_SYS_MAXARGS,	1,	do_bootz,
	"boot Linux zImage image from memory", bootz_help_text
);
```

- 1
- 2
- 3
- 4
- 5

根据前面分析命令组织形式，可以知道执行`bootz`命令会调用到`do_bootz`函数，所以必须从`do_bootz`函数入手。

先剧透函数的调用关系：

```
do_bootz
	|_ bootz_start
	|	|_ do_bootm_states（start阶段）
	|	|_ images->ep = image_load_addr
	|	|_ bootz_setup
	|	|_ bootm_find_images
	|_ bootm_disable_interrupts
	|_ images.os.os = IH_OS_LINUX
	|_ do_bootm_states（启动内核阶段）
		|_ boot_fn = bootm_os_get_boot_func(images->os.os)
		|_ boot_fn（do_bootm_linux函数的BOOTM_STATE_OS_PREP阶段）
		|	|_ boot_prep_linux
		|_ boot_selected_os
			|_ boot_fn（do_bootm_linux函数的BOOTM_STATE_OS_GO阶段）
				|_ boot_jump_linux
					|_ announce_and_cleanup
					|_ kernel_entry
```

- 1
- 2
- 3
- 4
- 5

4.1 do_bootz

```
int do_bootz(struct cmd_tbl *cmdtp, int flag, int argc, char *const argv[])
{
        int ret;


        argc--; argv++;

        if (bootz_start(cmdtp, flag, argc, argv, &images))
                return 1;


        bootm_disable_interrupts();

        images.os.os = IH_OS_LINUX;
        ret = do_bootm_states(cmdtp, flag, argc, argv,
#ifdef CONFIG_SYS_BOOT_RAMDISK_HIGH
                              BOOTM_STATE_RAMDISK |
#endif
                              BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
                              BOOTM_STATE_OS_GO,
                              &images, 1);

        return ret;
}
```

- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28

其中，函数整体可以大概分为以下四部分：

```
bootz_start
bootm_disable_interrupts
images.os.os = IH_OS_LINUX
do_bootm_states
```

- 1
- 2
- 3
- 4

所以接下来就是一个一个看它们的内部实现以及作用。

4.1.1 bootz_start

```
/* file: cmd/bootz.c */
static int bootz_start(struct cmd_tbl *cmdtp, int flag, int argc,
                       char *const argv[], bootm_headers_t *images)
{
        int ret;
        ulong zi_start, zi_end;

        ret = do_bootm_states(cmdtp, flag, argc, argv, BOOTM_STATE_START,
                              images, 1);

        /* Setup Linux kernel zImage entry point */
        if (!argc) {
                images->ep = image_load_addr;
                debug("*  kernel: default image load address = 0x%08lx\n",
                                image_load_addr);
        } else {
                images->ep = simple_strtoul(argv[0], NULL, 16);
                debug("*  kernel: cmdline image address = 0x%08lx\n",
                        images->ep);
        }

        ret = bootz_setup(images->ep, &zi_start, &zi_end);
```

```
        if (ret != 0)
                return 1;

        lmb_reserve(&images->lmb, images->ep, zi_end - zi_start);

        /*
         * Handle the BOOTM_STATE_FINDOTHER state ourselves as we do not
         * have a header that provide this informaiton.
         */
        if (bootm_find_images(flag, argc, argv, images->ep, zi_end - zi_start))
                return 1;

        return 0;
}
```

这里插播一下，要想继续往下分析，不得不先了解`images`这个全局变量，它是`bootm_headers`类型结构体，定义如下：

```
bootm_headers_t images;


typedef struct bootm_headers {
        image_header_t  *legacy_hdr_os;
        image_header_t  legacy_hdr_os_copy;
        ulong           legacy_hdr_valid;
        ...

#ifndef USE_HOSTCC
        image_info_t    os;
        ulong           ep;

        ulong           rd_start, rd_end;

        char            *ft_addr;
        ulong           ft_len;

        ulong           initrd_start;
        ulong           initrd_end;
        ulong           cmdline_start;
        ulong           cmdline_end;
        struct bd_info          *kbd;
#endif

        int             verify;

#define BOOTM_STATE_START       (0x00000001)
#define BOOTM_STATE_FINDOS      (0x00000002)
#define BOOTM_STATE_FINDOTHER   (0x00000004)
#define BOOTM_STATE_LOADOS      (0x00000008)
#define BOOTM_STATE_RAMDISK     (0x00000010)
#define BOOTM_STATE_FDT         (0x00000020)
#define BOOTM_STATE_OS_CMDLINE  (0x00000040)
#define BOOTM_STATE_OS_BD_T     (0x00000080)
#define BOOTM_STATE_OS_PREP     (0x00000100)
#define BOOTM_STATE_OS_FAKE_GO  (0x00000200)
#define BOOTM_STATE_OS_GO       (0x00000400)
        int             state;

#ifdef CONFIG_LMB
        struct lmb      lmb;
#endif
} bootm_headers_t;
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45

回归正题，`bootz_start`函数首先就是调用`do_bootm_states`来执行`BOOTM_STATE_START`阶段。

4.1.1.1 do_bootm_states（START阶段主要就调用bootm_start函数）

```
static int bootm_start(struct cmd_tbl *cmdtp, int flag, int argc,
                       char *const argv[])
{
        memset((void *)&images, 0, sizeof(images));
        images.verify = env_get_yesno("verify");

        boot_start_lmb(&images);

        bootstage_mark_name(BOOTSTAGE_ID_BOOTM_START, "bootm_start");
        images.state = BOOTM_STATE_START;
```

```
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

函数主要是将全局变量image清零并且设置一下它的几个成员就返回了。

4.1.1.2 设置 images->ep = image_load_addr

回到`bootz_start`函数之后，就设置`images->ep = image_load_addr`，这个参数比较关键，从前面“插播”的images结构体定义可以知道它保存kernel的入口地址。跟踪下`image_load_addr`变量的定义：

```
ulong image_load_addr = CONFIG_SYS_LOAD_ADDR;


#define CONFIG_SYS_LOAD_ADDR            CONFIG_LOADADDR


#if defined(CONFIG_MX6SL) || defined(CONFIG_MX6SLL) || \
        defined(CONFIG_MX6SX) || \
        defined(CONFIG_MX6UL) || defined(CONFIG_MX6ULL)
#define CONFIG_LOADADDR         0x82000000
#else
#define CONFIG_LOADADDR         0x12000000
#endif
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

- 12
- 13
- 14

可以看到，内核镜像的入口地址就在0x82000000。但是这个地址不是固定的，因为在前面分析`board_init_r`函数的时候就已经看到过`image_load_addr`变量被设置了，它是在`initr_env`函数里面通过获取环境变量来设置的，如果该环境变量没有被设置则使用原本默认的地址：

```
image_load_addr = env_get_ulong("loadaddr", 16, image_load_addr);
```

- 1

那么，知道内核入口地址之后，接着就调用`bootz_setup`函数来设置`zi_start`和`zi_end`两个值。

4.1.1.3 bootz_setup

```
int bootz_setup(ulong image, ulong *start, ulong *end)
{
        struct arm_z_header *zi = (struct arm_z_header *)image;

        if (zi->zi_magic != LINUX_ARM_ZIMAGE_MAGIC &&
            zi->zi_magic != BAREBOX_IMAGE_MAGIC) {
#ifndef CONFIG_SPL_FRAMEWORK
                puts("zimage: Bad magic!\n");
#endif
                return 1;
        }

        *start = zi->zi_start;
        *end = zi->zi_end;
#ifndef CONFIG_SPL_FRAMEWORK
        printf("Kernel image @ %#08lx [ %#08lx - %#08lx ]\n",
               image, *start, *end);
#endif

        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

设置zi_start和zi_end两个值可以用于后续的bootm_find_images函数来找到相关的启动镜像文件。

4.1.1.4 bootm_find_images

```
int bootm_find_images(int flag, int argc, char *const argv[], ulong start,
                      ulong size)
{
        int ret;


        ret = boot_get_ramdisk(argc, argv, &images, IH_INITRD_ARCH,
                               &images.rd_start, &images.rd_end);
        if (ret) {
                puts("Ramdisk image is corrupt or invalid\n");
                return 1;
        }
        ...

#if IMAGE_ENABLE_OF_LIBFDT

        ret = boot_get_fdt(flag, argc, argv, IH_ARCH_DEFAULT, &images,
                           &images.ft_addr, &images.ft_len);
        if (ret) {
                puts("Could not find a valid device tree\n");
                return 1;
        }
        ...
#endif
        ...

        return 0;
}
```

- 1
- 2
- 3
- 4

- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29

函数里面主要还是在找ramdisk和dtb文件。至此，`bootz_start`函数基本结束了，剩下的就是`do_bootm_states`函数，它刚才也在`bootz_start`函数里被调用去处理`BOOTM_STATE_START`阶段的工作。其实这个函数根据参数states来处理不同阶段的事情，接下来就是通过`BOOTM_STATE_OS_FAKE_GO`等宏定义来决定去启动内核了。

4.1.2 bootm_disable_interrupts

bootm启动内核之前，先关闭中断，说是这么说，但找到该函数定义发现它什么也没干，因为早就在reset复位后不久设置cpsr寄存器把FIQ快速中断和IRQ中断关闭了（见3.1章节部分）：

```
int disable_interrupts(void)
{
	return 0;
}
```

- 1
- 2
- 3
- 4
- 5

4.1.3 images.os.os = IH_OS_LINUX

设置镜像的操作系统类型为Linux，后面`do_bootm_states`函数启动内核时会根据它来找到对应的启动函数。

4.1.4 do_bootm_states（启动内核阶段）

```
int do_bootm_states(struct cmd_tbl *cmdtp, int flag, int argc,
                    char *const argv[], int states, bootm_headers_t *images,
                    int boot_progress)
{
        boot_os_fn *boot_fn;
        ulong iflag = 0;
        int ret = 0, need_boot_fn;

        images->state |= states;

        if (states & BOOTM_STATE_START)
                ret = bootm_start(cmdtp, flag, argc, argv);
        ...
#if IMAGE_ENABLE_OF_LIBFDT && defined(CONFIG_LMB)
        if (!ret && (states & BOOTM_STATE_FDT)) {
                boot_fdt_add_mem_rsv_regions(&images->lmb, images->ft_addr);
                ret = boot_relocate_fdt(&images->lmb, &images->ft_addr,
                                        &images->ft_len);
        }
#endif


        if (ret)
                return ret;
        boot_fn = bootm_os_get_boot_func(images->os.os);
        need_boot_fn = states & (BOOTM_STATE_OS_CMDLINE |
                        BOOTM_STATE_OS_BD_T | BOOTM_STATE_OS_PREP |
                        BOOTM_STATE_OS_FAKE_GO | BOOTM_STATE_OS_GO);
        if (boot_fn == NULL && need_boot_fn) {
                if (iflag)
                        enable_interrupts();
                printf("ERROR: booting os '%s' (%d) is not supported\n",
                       genimg_get_os_name(images->os.os), images->os.os);
                bootstage_error(BOOTSTAGE_ID_CHECK_BOOT_OS);
                return 1;
        }



        if (!ret && (states & BOOTM_STATE_OS_CMDLINE))
                ret = boot_fn(BOOTM_STATE_OS_CMDLINE, argc, argv, images);
        if (!ret && (states & BOOTM_STATE_OS_BD_T))
                ret = boot_fn(BOOTM_STATE_OS_BD_T, argc, argv, images);
```

```
        if (!ret && (states & BOOTM_STATE_OS_PREP)) {
                ret = boot_fn(BOOTM_STATE_OS_PREP, argc, argv, images);
        }
        ...


        if (ret) {
                puts("subcommand not supported\n");
                return ret;
        }


        if (!ret && (states & BOOTM_STATE_OS_GO))
                ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_GO,
                                images, boot_fn);
        ...

        return ret;
}
```

- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63

这个函数在前面`bootz_start`里已经被调用过一次，但是当时处理的是宏定义`BOOTM_STATE_START`部分内容。然而在`do_bootz`函数里调用的时候参数states则是`BOOTM_STATE_OS_PREP`、`BOOTM_STATE_OS_FAKE_GO`和`BOOTM_STATE_OS_GO`（imx6ull没有定义`CONFIG_SYS_BOOT_RAMDISK_HIGH`），所以函数主要还是执行了后半部分的启动kernel，这一阶段主要有3个比较重要的函数：`bootm_os_get_boot_func`、`boot_fn`和`boot_selected_os`。

4.1.4.1 bootm_os_get_boot_func

先看下`bootm_os_get_boot_func`函数，从定义可以看得出来它是根据os获取相应的启动函数：

```
boot_os_fn *bootm_os_get_boot_func(int os)
{
        ...
        return boot_os[os];
}
```

- 1
- 2
- 3
- 4
- 5

- 6

继续看下数组的boot_os的实现：

```
static boot_os_fn *boot_os[] = {
        [IH_OS_U_BOOT] = do_bootm_standalone,
#ifdef CONFIG_BOOTM_LINUX
        [IH_OS_LINUX] = do_bootm_linux,
#endif
#ifdef CONFIG_BOOTM_NETBSD
        [IH_OS_NETBSD] = do_bootm_netbsd,
#endif
...
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

还记得前面do_bootz函数里面设置images.os.os = IH_OS_LINUX，所以就是启动Linux内核的函数为do_bootm_linux，将它作为返回值传递给了boot_fn函数。接着就是根据BOOTM_STATE_OS_PREP定义调用了boot_fn函数。

4.1.4.2 boot_fn（BOOTM_STATE_OS_PREP阶段）

在得到启动函数之后，就会根据states标志来疯狂地调用。这里先看一眼boot_fn实际指向的do_bootm_linux函数：

```
int do_bootm_linux(int flag, int argc, char *const argv[],
                   bootm_headers_t *images)
{

        if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
                return -1;

        if (flag & BOOTM_STATE_OS_PREP) {
                boot_prep_linux(images);
                return 0;
        }

        if (flag & (BOOTM_STATE_OS_GO | BOOTM_STATE_OS_FAKE_GO)) {
```

```
                boot_jump_linux(images, flag);
                return 0;
        }

        boot_prep_linux(images);
        boot_jump_linux(images, flag);
        return 0;
}
```

对于ARM结构的CPU来说，函数里主要还是调用了`boot_prep_linux`和`boot_jump_linux`两个函数。根据启动流程来讲，函数会在调用`boot_prep_linux`函数后就返回了（`boot_jump_linux`函数在后面启动内核的时候再研究）。通过函数的名字也可以知道它就是启动Linux之前的一些准备、设置的一些工作：

```
static void boot_prep_linux(bootm_headers_t *images)
{
        char *commandline = env_get("bootargs");

        if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len) {
#ifdef CONFIG_OF_LIBFDT
                debug("using: FDT\n");
                if (image_setup_linux(images)) {
                        printf("FDT creation failed! hanging...");
                        hang();
                }
#endif
        } else if (BOOTM_ENABLE_TAGS) {
                debug("using: ATAGS\n");
                setup_start_tag(gd->bd);
                if (BOOTM_ENABLE_SERIAL_TAG)
                        setup_serial_tag(&params);
```

```
                ...
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

准备工作结束之后，往下就是启动内核了。

4.1.4.3 boot_selected_os

由于没有定义`CONFIG_TRACE`，所以是通过`BOOTM_STATE_OS_GO`标志来启动，但是最终也都是调用了`boot_selected_os`函数：

```
int boot_selected_os(int argc, char *const argv[], int state,
                     bootm_headers_t *images, boot_os_fn *boot_fn)
{
        arch_preboot_os();
        board_preboot_os();
        boot_fn(state, argc, argv, images);

        ...
        return BOOTM_ERR_RESET;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

- 10
- 11

从函数内容也可以知道，最终也还是调用了boot_fn（do_bootm_linux）来启动，对比BOOTM_STATE_OS_GO标志来说，do_bootm_linux函数里调用的就是boot_jump_linux函数，现在就可以研究它的实现了，经过宏定义的简化之后函数就是：

```
static void boot_jump_linux(bootm_headers_t *images, int flag)
{
	unsigned long machid = gd->bd->bi_arch_number;
	char *s;
	void (*kernel_entry)(int zero, int arch, uint params);
	unsigned long r2;
	int fake = (flag & BOOTM_STATE_OS_FAKE_GO);

	kernel_entry = (void (*)(int, int, uint))images->ep;

	s = env_get("machid");
	if (s) {
		if (strict_strtoul(s, 16, &machid) < 0) {
			debug("strict_strtoul failed!\n");
			return;
		}
		printf("Using machid 0x%lx from environment\n", machid);
	}

	debug("## Transferring control to Linux (at address %08lx)" \
		"...\n", (ulong) kernel_entry);
	bootstage_mark(BOOTSTAGE_ID_RUN_OS);
	announce_and_cleanup(fake);

	if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
		r2 = (unsigned long)images->ft_addr;
	else
		r2 = gd->bd->bi_boot_params;

	if (!fake) {
			kernel_entry(0, machid, r2);
	}
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34

简化之后的函数目的比较明确，设置好函数的入口之后，通过获取环境变量`machid`来标志单板，但是如果使用了设备树形式的启动，则不需要理会。紧接着设置启动标志为`BOOTSTAGE_ID_RUN_OS`之后调用`announce_and_cleanup`函数宣布一下“Starting kernel …”，并且进行启动前的一些清理工作，简化一下宏定义后内容如下：

```
static void announce_and_cleanup(int fake)
{
        bootstage_mark_name(BOOTSTAGE_ID_BOOTM_HANDOFF, "start_kernel");

        board_quiesce_devices();

        printf("\nStarting kernel ...%s\n\n", fake ?
                "(fake run for tracing)" : "");

        dm_remove_devices_flags(DM_REMOVE_ACTIVE_ALL);

        cleanup_before_linux();
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

- 11
- 12
- 13
- 14

函数`announce_and_cleanup`调用结束后回到`boot_jump_linux`函数之后就是判断是否使用设备树，如果使用则设置r2为设备树的地址，然后作为`kernel_entry`函数的参数真正地进入内核。

一旦启动了内核，u-boot程序从此不再使用了。

结束！