(1条消息) u-boot-2021.01（imx6ull）启动流程分析之一：分析启动流程之前的准备知识和工作___ASDFGH的博客-CSDN博客

前言

本文章主要以基于ARM® Cortex-A7内核的Freescale i.MX6ULL这款SoC探索u-boot-2021.01启动linux内核的流程。在了解启动流程之后，如果需要移植不同版本的u-boot或移植到其他平台的SoC，相信也能按照同样的思路去排查定位问题所在。

在查看文章的过程中，如果遇到描述有错误的地方，还希望大家悉心指出。

1、关于Freescale i.MX6ULL

i.mx6ull这款SoC到底是如何一款芯片，这里不过多地描述，随便一个搜索引擎都能找到比较详细的描述，这里主要还是以分析u-boot启动出发，了解这款SoC与u-boot相关的一些内容，或者说与一般芯片的差异部分。

- **u-boot**程序的格式：与其他厂家芯片（如s3c24x0）不一样，烧录到i.mx6ull的u-boot程序不是bin格式，而是imx格式，这是因为芯片的启动方式有所不同。i.mx6ull的u-boot程序是通过使用imxdownload程序在u-boot程序的基础上添加了**“头部”**，这个“头部”就是由”IVT+BootData+DCD“组成，假设生成的bin文件为u-boot.bin，那么用于烧录的u-boot.imx就等于“IVT+BootData+DCD+u-boot.bin”。
- **“头部”**的内容：（了解即可，一般不用太深入探究。参考：《IMX6ULL参考手册.pdf》）

① **IVT**（**Image vector table**）：包含IVT头部、程序入口地址、DCD地址、Boot Data地址等信息，如表格展示：

Table 8-26. IVT format

header
entry: Absolute address of the first instruction to execute from the image
reserved1: Reserved and should be zero
dcd: Absolute address of the image DCD. The DCD is optional so this field may be set to NULL if no DCD is required. See Device Configuration Data (DCD) for further details on the DCD.
boot data: Absolute address of the boot data
self: Absolute address of the IVT. Used internally by the ROM.
csf: Absolute address of the Command Sequence File (CSF) used by the HAB library. See High-Assurance Boot (HAB) for details on the secure boot using HAB. This field must be set to NULL when not performing a secure boot
reserved2: Reserved and should be zero

其中header部分内容：Tag固定为0xD1、Length为2个字节大端保存IVT长度、Version为0x40或0x41，结构如图：

② **BootData**：包含镜像要拷贝到的目的地址和大小（与**plugin**标志位均用32bit空间表示），格式如图：

Table 8-27. Boot data format

start	Absolute address of the image
length	Size of the program image
plugin	Plugin flag (see Plugin image)

③ **DCD**（**Device configuration data**）：MMDC 控制器， DDR等外设的寄存器初始化数据。

- **“头部”**如何被使用：芯片内部的boot rom程序会读取解析imx格式的文件头部来初始化外设，然后将imx文件从Flash中拷贝到外部RAM中运行。也正因为在**imx6ull**中，**u-boot**程序一开始就是在外部**RAM**中运行，所以它的**“代码重定位”**是RAM->RAM的代码**“搬运”**，而不是Flash->RAM。

2、先编译一遍u-boot

2.1 为什么要先编译

编译后会在u-boot根目录生成一些比较重要的文件。比如我们就可以通过`u-boot.lds`文件获取程序各个段的分布、通过`u-boot.map`文件查看各个段的内容分布（尤其是text代码段，可以查看某个函数是在哪个目录下编译）、更有各个子目录下编译生成的.o中间文件、甚至还可以通过命令`xxx-objdump -D -m arm u-boot > u-boot.dis`生成反汇编文件来定位运行异常的问题等等。所以，先编译一遍u-boot就显得很有必要。

2.2 编译流程

查看configs目录下关于i.mx6ull的默认配置，发现有2个：

```
mx6ull_14x14_evk_defconfig
mx6ull_14x14_evk_plugin_defconfig
```

- 1
- 2

我们选择前者就行，因为后者通过《IMX6ULL参考手册.pdf》可以知道这是为了支持以太网等方式启动，原文如下：

8.8 Plugin image

> The ROM supports a limited number of boot devices. When using other devices as a boot source (for example, Ethernet, CDROM, or USB), the supported boot device must be used (typically serial ROM) as a firmware to provide the missing boot drivers.

故编译步骤如下：

```
tar xjf u-boot-2021.01.tar.bz2
cd u-boot-2021.01/
make mx6ull_14x14_evk_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

- 1
- 2
- 3
- 4

编译后生成以下u-boot相关文件：

```
-rwxrwxr-x   1 book book 3571160 2月  14 11:52 u-boot
-rw-rw-r--   1 book book  371855 2月  14 11:52 u-boot.bin
-rw-rw-r--   1 book book   12569 2月  14 11:51 u-boot.cfg
-rw-rw-r--   1 book book    6288 2月  14 11:52 u-boot.cfg.configs
-rw-rw-r--   1 book book   28571 2月  14 11:52 u-boot.dtb
-rw-rw-r--   1 book book  371855 2月  14 11:52 u-boot-dtb.bin
-rw-rw-r--   1 book book    3244 2月  14 11:52 u-boot-dtb.cfgout
-rw-rw-r--   1 book book  375808 2月  14 11:52 u-boot-dtb.imx
-rw-rw-r--   1 book book     194 2月  14 11:52 u-boot-dtb.imx.log
-rw-rw-r--   1 book book    1719 2月  14 11:52 u-boot.lds
-rw-rw-r--   1 book book  609654 2月  14 11:52 u-boot.map
-rwxrwxr-x   1 book book  343284 2月  14 11:52 u-boot-nodtb.bin
-rwxrwxr-x   1 book book 1029978 2月  14 11:52 u-boot.srec
-rw-rw-r--   1 book book  154428 2月  14 11:52 u-boot.sym
```

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

编译后生成的可以直接烧录的`u-boot-dtb.imx`，从名字里可以知道imx6ull的u-boot程序里使用了dts设备树。直接执行命令`grep "CONFIG_OF_" .config | grep "=y"`搜索一下：

```
CONFIG_OF_CONTROL=y
CONFIG_OF_SEPARATE=y
CONFIG_OF_TRANSLATE=y
CONFIG_OF_LIBFDT=y
```

- 1
- 2
- 3
- 4

除了表示支持linux使用设备树的`CONFIG_OF_LIBFDT`外，还有几个定义。在根目录下的README文件就有关于它们的说明：

README (~/work/imx6ull/system/u-boot-2021.01) - VIM

```
699
700 - Device tree:
701     CONFIG_OF_CONTROL
702     If this variable is defined, U-Boot will use a device tree
703     to configure its devices, instead of relying on statically
704     compiled #defines in the board file. This option is
705     experimental and only available on a few boards. The device
706     tree is available in the global data as gd->fdt_blob.
707
708     U-Boot needs to get its device tree from somewhere. This can
709     be done using one of the three options below:
710
711     CONFIG_OF_EMBED
712     If this variable is defined, U-Boot will embed a device tree
713     binary in its image. This device tree file should be in the
714     board directory and called <soc>-<board>.dts. The binary file
715     is then picked up in board_init_f() and made available through
716     the global data structure as gd->fdt_blob.
717
718     CONFIG_OF_SEPARATE
719     If this variable is defined, U-Boot will build a device tree
720     binary. It will be called u-boot.dtb. Architecture-specific
721     code will locate it at run-time. Generally this works by:
722
723       cat u-boot.bin u-boot.dtb >image.bin
724
725     and in fact, U-Boot does this for you, creating a file called
726     u-boot-dtb.bin which is useful in the common case. You can
727     still use the individual files if you need something more
728     exotic.
729
730     CONFIG_OF_BOARD
731     If this variable is defined, U-Boot will use the device tree
732     provided by the board at runtime instead of embedding one with
733     the image. Only boards defining board_fdt_blob_setup() support
734     this option (see include/fdtdec.h file).
735
```


由于imx6ull默认配置中定义了`CONFIG_OF_SEPARATE`，所以实际上u-boot-dtb.bin就是u-boot.bin+u-boot.dtb，由于上面是直接使用cat命令将它们合并在一起，所以u-boot.dtb的存储位置其实是

紧挨着u-boot.bin尾部的。而对于imx6ull的u-boot程序来说，用于烧录的镜像文件就是u-boot-dtb.imx=IVT+BootData+DCD+u-boot.bin+u-boot.dtb。

注意，u-boot程序中的设备树和用于启动linux内核时用的设备树并不冲突，它们只是同时都使用了设备树来代替大量的板级细节信息描述而已。

此外，并非所有的SoC都支持u-boot程序中使用设备树，但它们的启动流程基本差不多，只是获取一些硬件信息描述时有些差别，整体启动流程还是一样的。

未完待续...

(1条消息) u-boot-2021.01（imx6ull）启动流程分析之二：从执行第1句u-boot代码开始分析 ___ASDFGH的博客-CSDN博客

3、启动流程分析

先预览函数的大概调用框图，后面对每个函数进行分析：

```
_start
        |_ cpu_init_cp15
        |_ cpu_init_crit
        |       |_ lowlevel_init
        |               |_ s_init
        |_ _main
                |_ board_init_f_alloc_reserve
                |_ board_init_f_init_reserve
                |_ board_init_f
                |_ relocate_code
                |_ relocate_vectors
                |_ board_init_r
```

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

3.1 _start

查看程序的入口可以从编译生成的u-boot.lds入手，因为lds文件是指定程序各个段的存储位置，查看它的内容：

```
/* file: u-boot.lds */
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x00000000;
 . = ALIGN(4);
 .text :
 {
  *(.__image_copy_start)
  *(.vectors)
  arch/arm/cpu/armv7/start.o (.text*)
 }
...
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

通过以上ENTRY(_start)定义，我们可以找到arm架构的arch/arm/lib/vectors.S文件，摘取部分内容如下：

```
/* file: arch/arm/lib/vectors.S */
        .macro ARM_VECTORS
#ifdef CONFIG_ARCH_K3
        ldr     pc, _reset
#else
        b       reset
#endif
        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
        .endm
...

_start:
#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
  .word CONFIG_SYS_DV_NOR_BOOT_CFG
#endif
```

```
  ARM_VECTORS
#endif /* !defined(CONFIG_ENABLE_ARM_SOC_BOOT0_HOOK) */
...

_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
...

#ifdef CONFIG_SPL_BUILD
...
#else   /* !CONFIG_SPL_BUILD */
...
undefined_instruction:
        get_bad_stack
        bad_save_user_regs
        bl      do_undefined_instruction
...
#endif  /* CONFIG_SPL_BUILD */
```

根据文件名字也可以看到，这些都是向量表相关的一些跳转和异常处理。在文件开始部分使用汇编宏`.macro ARM_VECTORS ... .endm`定义了`ARM_VECTORS`，然后在程序的入口`_start`标志处引用了这个宏。

这个宏涉及了异常处理，看下如图Cortex A7的异常向量表:

Table 11-1 Summary of exception behavior

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
0x0	0xFFFF0000	Not used	Reset	Reset	Not used
0x4	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
0x8	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
0xC	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort
0x10	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
0x14	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
0x18	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

举个例子，当发生FIQ快速中断时，硬件决定直接跳到如表格展示的0x1C地址去，但是该地址处无法处理过多的指令，因为这些向量表的地址都是连续的。所以干脆在对应的地址处使用`ldr pc, xxx`再次跳转到其他地址去处理。

但在分析启动流程时，我们只关心**u-boot**的第一句代码——**`b reset`**跳转到复位语句，reset标志就在对应的cpu架构目录`arch/arm/cpu/armv7/start.S`中定义，摘取部分内容如下：

```
/* file: arch/arm/cpu/armv7/start.S */
reset:
        /* Allow the board to save important registers */
        b       save_boot_params
save_boot_params_ret:
#ifdef CONFIG_ARMV7_LPAE
/*
 * check for Hypervisor support
 */
        mrc     p15, 0, r0, c0, c1, 1           @ read ID_PFR1
        and     r0, r0, #CPUID_ARM_VIRT_MASK    @ mask virtualization bits
        cmp     r0, #(1 << CPUID_ARM_VIRT_SHIFT)
        beq     switch_to_hypervisor
switch_to_hypervisor_ret:
#endif
        /*
         * 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

/*
 * 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 SCTLR register - for VBAR to point to vector */
        mrc     p15, 0, r0, c1, c0, 0   @ Read CP15 SCTLR Register
        bic     r0, #CR_V               @ V = 0
        mcr     p15, 0, r0, c1, c0, 0   @ Write CP15 SCTLR Register

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

        /* the mask ROM code should have PLL and others stable */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
#ifdef CONFIG_CPU_V7A
```

```
        bl      cpu_init_cp15
#endif
#ifndef CONFIG_SKIP_LOWLEVEL_INIT_ONLY
        bl      cpu_init_crit
#endif
#endif

        bl      _main
```

首先就是先跳转到同文件中的`save_boot_params`:

```
/* file: arch/arm/cpu/armv7/start.S */
ENTRY(save_boot_params)
        b       save_boot_params_ret            @ back to my caller
ENDPROC(save_boot_params)
        .weak   save_boot_params
```

然而发现它并没有作任何处理，又继续跳转回来。接着就是根据宏`CONFIG_ARMV7_LPAE`判断是否需要支持Hypervisor模式，但没有定义所以不需要关心；然后就是设置cpsr寄存器让CPU进入SVC32管理模式并且关闭FIQ快速中断和IRQ中断；

往下就是使用CR_V的值（在`arch/arm/include/asm/system.h`中定义）来设置CP15协处理器，目的是支持向量表的重定位，接着就是设置新的向量表地址了。

再往下就是跳转执行`cpu_init_cp15`、`cpu_init_crit`和`_main`，下面就逐个分析。

3.2 cpu_init_cp15

它在同文件中定义，主要是设置CP15协处理器，目的是初始化并设置“数据缓存dcache”和“指令缓存icache”。其中，dcache是务必关闭的，因为u-boot程序更多时候是和硬件在打交道，所以尽可能少用dcache中的缓存数据，目的是实时读写硬件，而icache则根据“SYS_ICACHE_OFF”是否配置来决定。其次函数工作内容还有关闭MMU等等，代码如下:

```
/* file: arch/arm/cpu/armv7/start.S */
```

```
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
#if CONFIG_IS_ENABLED(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
...
        mov     r5, lr                  @ Store my Caller
...
        mov     pc, r5                  @ back to my caller
ENDPROC(cpu_init_cp15)
```

- 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

执行完`cpu_init_cp15`函数接着继续跳转到`cpu_init_crit`函数。

3.3 cpu_init_crit

它也是在同文件中定义，根据官方注释也可以清楚地看到它的目的是设置重要的寄存器等等，往里看看就知道了：

```
/* file: arch/arm/cpu/armv7/start.S */
/*************************************************************************
 *
 * 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)
```

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

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18

看到它就只是一句跳转到`lowlevel_init`函数。

3.3.1 lowlevel_init

```
/* file: arch/arm/cpu/armv7/lowlevel_init.S */
WEAK(lowlevel_init)
        /*
         * Setup a temporary stack. Global data is not available yet.
         */
#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 */
#ifdef CONFIG_SPL_DM
        mov     r9, #0
#else
        /*
         * Set up global data for boards that still need it. This will be
         * removed soon.
         */
#ifdef CONFIG_SPL_BUILD
        ldr     r9, =gdata
#else
        sub     sp, sp, #GD_SIZE
        bic     sp, sp, #7
        mov     r9, sp
#endif
#endif
        /*
         * Save the old lr(passed in ip) and the current lr to stack
         */
        push    {ip, lr}

        /*
         * Call the very early init function. This should do only the
         * absolute bare minimum to get started. It should not:
         *
         * - set up DRAM
         * - use global_data
         * - clear BSS
         * - try to start a console
         *
         * For boards with SPL this should be empty since SPL can do all of
         * this init in the SPL board_init_f() function which is called
         * immediately after this.
         */
        bl      s_init
        pop     {ip, pc}
ENDPROC(lowlevel_init)
```

- 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
- 46
- 47

可以看到，函数里面就是先设置sp栈地址并进行8字节对齐，目的是后面跳转到s_init函数。

3.3.1.1 s_init

```
void s_init(void)
{
        struct anatop_regs *anatop = (struct anatop_regs *)ANATOP_BASE_ADDR;
        struct mxc_ccm_reg *ccm = (struct mxc_ccm_reg *)CCM_BASE_ADDR;
        u32 mask480;
        u32 mask528;
        u32 reg, periph1, periph2;

        if (is_mx6sx() || is_mx6ul() || is_mx6ull() || is_mx6sll())
                return;
...
}
```

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

函数里面会先判断芯片的型号，发现mx6ull不作任何处理就直接返回了，往上返回之后就到了_main函数。

3.4 _main

这个函数非常重要，从函数命名也可以知道它负责主要的工作，它会直接或间接地调用一些函数来初始化硬件和启动内核:

```
/* file: arch/arm/lib/crt0.S */
ENTRY(_main)

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

#if defined(CONFIG_TPL_BUILD) && defined(CONFIG_TPL_NEEDS_SEPARATE_STACK)
        ldr     r0, =(CONFIG_TPL_STACK)
#elif defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
        ldr     r0, =(CONFIG_SPL_STACK)
#else
        ldr     r0, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
        bic     r0, r0, #7      /* 8-byte alignment for ABI compliance */
        mov     sp, r0
        bl      board_init_f_alloc_reserve
        mov     sp, r0
        /* set up gd here, outside any C code */
        mov     r9, r0
        bl      board_init_f_init_reserve

#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_EARLY_BSS)
        CLEAR_BSS
#endif

        mov     r0, #0
        bl      board_init_f

#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     r0, [r9, #GD_START_ADDR_SP]     /* sp = gd->start_addr_sp */
        bic     r0, r0, #7      /* 8-byte alignment for ABI compliance */
        mov     sp, r0
        ldr     r9, [r9, #GD_NEW_GD]            /* r9 <- gd->new_gd */

        adr     lr, here
        ldr     r0, [r9, #GD_RELOC_OFF]         /* r0 = gd->reloc_off */
```

```
        add     lr, lr, r0
#if defined(CONFIG_CPU_V7M)
        orr     lr, #1                          /* As required by Thumb-only */
#endif
        ldr     r0, [r9, #GD_RELOCADDR]         /* r0 = gd->relocaddr */
        b       relocate_code
here:
/*
 * now relocate vectors
 */

        bl      relocate_vectors

/* Set up final (full) environment */

        bl      c_runtime_cpu_setup     /* we still call old routine here */
#endif
#if !defined(CONFIG_SPL_BUILD) || CONFIG_IS_ENABLED(FRAMEWORK)

#if !defined(CONFIG_SPL_BUILD) || !defined(CONFIG_SPL_EARLY_BSS)
        CLEAR_BSS
#endif

# ifdef CONFIG_SPL_BUILD
        /* Use a DRAM stack for the rest of SPL, if requested */
        bl      spl_relocate_stack_gd
        cmp     r0, #0
        movne   sp, r0
        movne   r9, r0
# endif

#if ! defined(CONFIG_SPL_BUILD)
        bl coloured_LED_init
        bl red_led_on
#endif
        /* 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 */
#if CONFIG_IS_ENABLED(SYS_THUMB_BUILD)
        ldr     lr, =board_init_r       /* this is auto-relocated! */
        bx      lr
#else
        ldr     pc, =board_init_r       /* this is auto-relocated! */
#endif
        /* we should not return here. */
#endif

ENDPROC(_main)
```

函数内容有点多，但我们比较感兴趣的主要有以下几句：

```
bl      board_init_f_alloc_reserve
bl      board_init_f_init_reserve
bl      board_init_f
b       relocate_code
bl      relocate_vectors
ldr     pc, =board_init_r
```

前两个函数都是和全局变量gd结构体相关，后四个函数内容更多也比较重要，所以需要重点研究。废话不多说，接下来就开始研究这几个函数。

3.4.1 board_init_f_alloc_reserve、board_init_f_init_reserve

其中，board_init_f_alloc_reserve是为了预留全局变量gd结构体的内存空间，而board_init_f_init_reserve是将gd结构体清0，初始化预留内存空间等等。不妨看下它们的内容，验证下我们的说法：

```
ulong board_init_f_alloc_reserve(ulong top)
{

#if CONFIG_VAL(SYS_MALLOC_F_LEN)
        top -= CONFIG_VAL(SYS_MALLOC_F_LEN);
#endif

        top = rounddown(top-sizeof(struct global_data), 16);

        return top;
}

void board_init_f_init_reserve(ulong base)
{
        struct global_data *gd_ptr;



        gd_ptr = (struct global_data *)base;

        memset(gd_ptr, '\0', sizeof(*gd));


    ...

        if (CONFIG_IS_ENABLED(SYS_REPORT_STACK_F_USAGE))
                board_init_f_init_stack_protection();
}
```

- 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

其实不只是`board_init_f_init_reserve`函数，接下来很多操作都是设置gd这个全局变量，我们不会探讨它每一个成员设置过程，主要看比较重要的几个即可。

未完待续...

(1条消息) u-boot-2021.01（imx6ull）启动流程分析之三：board_init_f函数分析___ASDFGH的博客-CSDN博客

3.4.2 board_init_f

顾名思义，函数主要工作是早期的一些硬件初始化和设置全局变量gd结构体的成员：

```
void board_init_f(ulong boot_flags)
{
        gd->flags = boot_flags;
        gd->have_console = 0;

        if (initcall_run_list(init_sequence_f))
                hang();

#if !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX) && \
                !defined(CONFIG_EFI_APP) && !CONFIG_IS_ENABLED(X86_64) && \
                !defined(CONFIG_ARC)

        hang();
#endif
}
```

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

函数内容不多，但是`init_sequence_f`这个函数指针数组内容可不简单，由于数组元素较多，所以将相关宏定义简化之后就是：

```
static const init_fnc_t init_sequence_f[] = {
        setup_mon_len,
    fdtdec_setup,
    trace_early_init,
    initf_malloc,
        log_init,
        initf_bootstage,
    setup_spl_handoff,
        initf_console_record,
    arch_cpu_init,
        mach_cpu_init,
        initf_dm,
        arch_cpu_init_dm,
    board_early_init_f,
    get_clocks,
        timer_init,
    board_postclk_init,
        env_init,
        init_baud_rate,
        serial_init,
        console_init_f,
        display_options,
        display_text_info,
        checkcpu,
    print_cpuinfo,
    show_board_info,
    INIT_FUNC_WATCHDOG_INIT
    misc_init_f,
    INIT_FUNC_WATCHDOG_RESET
    init_func_i2c,
    init_func_vid,
    announce_dram_init,
        dram_init,
    post_init_f,
    init_post,
    ...
        reserve_uboot,
        reserve_malloc,
        reserve_board,
        setup_machine,
        reserve_global_data,
        reserve_fdt,
    ...
        dram_init_banksize,
        show_dram_config,
        setup_bdinfo,
        display_new_sp,
        ...
        reloc_fdt,
        reloc_bootstage,
        reloc_bloblist,
        setup_reloc,
        ...
        NULL,
};
```

- 1
- 2
- 3
- 4

这个数组最终作为函数initcall_run_list的参数，遍历执行这个函数指针数组的每一个元素（函数），所以重点可以放在函数指针数组里的各个元素。篇章问题，这里就不一一探究，挑出个别容易在移植过程中出现的函数即可。

3.4.2.1 setup_mon_len：设置**gd**的**mon_len**成员（代码长度）

去掉宏定义之后的函数定义：

```
static int setup_mon_len(void)
{
        gd->mon_len = (ulong)&__bss_end - CONFIG_SYS_MONITOR_BASE;
        return 0;
}
```

3.4.2.2 fdtdec_setup：如果**u-boot**中使用设备树，则需处理一些相关工作

去掉宏定义之后的函数定义：

```
int fdtdec_setup(void)
{
        int ret;


        gd->fdt_blob = board_fdt_blob_setup();


        gd->fdt_blob = map_sysmem
                (env_get_ulong("fdtcontroladdr", 16,
                               (unsigned long)map_to_sysmem(gd->fdt_blob)), 0);
        ret = fdtdec_prepare_fdt();
        if (!ret)
                ret = fdtdec_board_setup(gd->fdt_blob);
        return ret;
}
```

- 11
- 12
- 13
- 14
- 15
- 16
- 17

函数先是调用board_fdt_blob_setup来设置gd结构体的fdt_blob成员，简化宏定义之后函数如下：

```
__weak void *board_fdt_blob_setup(void)
{
        void *fdt_blob = NULL;

        fdt_blob = (ulong *)&_end;
        return fdt_blob;
}
```

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

由前面的镜像组成可以知道设备树文件是放在u-boot.bin的末尾，所以取它的地址返回即可。 后面继续调用env_get_ulong从环境变量中获取设备树的地址，如果该环境变量没有被设置则返回原本的默认值。最终继续调用fdtdec_prepare_fdt函数来打印一些信息：

```
int fdtdec_prepare_fdt(void)
{
        if (!gd->fdt_blob || ((uintptr_t)gd->fdt_blob & 3) ||
            fdt_check_header(gd->fdt_blob)) {
#ifdef CONFIG_SPL_BUILD
                puts("Missing DTB\n");
#else
                puts("No valid device tree binary found - please append one to U-Boot binary, use u-boot-dtb.bin or define CONFIG_OF_EMBED. For sandbox, use -d <file.dtb>\n");
# ifdef DEBUG
                if (gd->fdt_blob) {
                        printf("fdt_blob=%p\n", gd->fdt_blob);
                        print_buffer((ulong)gd->fdt_blob, gd->fdt_blob, 4,
                                     32, 0);
                }
# endif
#endif
                return -1;
        }
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

3.4.2.3 board_early_init_f：配置串口引脚

```
int board_early_init_f(void)
{
        setup_iomux_uart();

        return 0;
}
```

```
#define UART_PAD_CTRL  (PAD_CTL_PKE | PAD_CTL_PUE |             \
        PAD_CTL_PUS_100K_UP | PAD_CTL_SPEED_MED |               \
        PAD_CTL_DSE_40ohm   | PAD_CTL_SRE_FAST  | PAD_CTL_HYS)

static iomux_v3_cfg_t const uart1_pads[] = {
        MX6_PAD_UART1_TX_DATA__UART1_DCE_TX | MUX_PAD_CTRL(UART_PAD_CTRL),
        MX6_PAD_UART1_RX_DATA__UART1_DCE_RX | MUX_PAD_CTRL(UART_PAD_CTRL),
};

static void setup_iomux_uart(void)
{
        imx_iomux_v3_setup_multiple_pads(uart1_pads, ARRAY_SIZE(uart1_pads));
}
```

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

```
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15
    • 16
    • 17
    • 18
    • 19
    • 20
    • 21
    • 22
```

发现函数目的是串口uart1引脚的复用配置，至于函数imx_iomux_v3_setup_multiple_pads就不继续往下追踪了，这里只需要知道函数的功能即可，函数一般不会有太大问题，如果程序运行不了再继续往下追，否则会篇幅太长。

3.4.2.4 init_baud_rate：设置波特率

```
static int init_baud_rate(void)
{
        gd->baudrate = env_get_ulong("baudrate", 10, CONFIG_BAUDRATE);
        return 0;
}
```

```
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
```

从env_get_ulong函数定义可以知道它是从环境变量里获取十进制的baudrate值，如果没有则使用默认的CONFIG_BAUDRATE。

3.4.2.5 serial_init：初始化串口

（如果不知道函数在哪个文件中定义，可以通过u-boot.map和各个子目录下的.o文件快速定位。）

```
int serial_init(void)
{
        gd->flags |= GD_FLG_SERIAL_READY;
        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;


        if (!dev) {
#ifdef CONFIG_SPL_BUILD
                puts("Cannot find console\n");
                hang();
#else
                panic("Cannot find console\n");
#endif
        }

        return dev;
}
```

```
    • 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

get_current函数的一进来就是判断是否已经代码重定位了，很明显，执行到这里还没有，所以条件为真，于是通过default_serial_console来获取默认的串口设备，按照u-boot.map文件和各个子目录下的.o文件也不难发现它的定义：

```
__weak struct serial_device *default_serial_console(void)
{
        return &mxc_serial_drv;
}


static struct serial_device mxc_serial_drv = {
        .name   = "mxc_serial",
        .start  = mxc_serial_init,
        .stop   = NULL,
        .setbrg = mxc_serial_setbrg,
        .putc   = mxc_serial_putc,
        .puts   = default_serial_puts,
        .getc   = mxc_serial_getc,
        .tstc   = mxc_serial_tstc,
};
```

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

最终会调用到mxc_serial_init函数，往里面继续探究看看：

```
static int mxc_serial_init(void)
{
        _mxc_serial_init(mxc_base, false);

        serial_setbrg();

        return 0;
}
```

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

继续往里探究，其中mxc_base的定义如下：

```
#define mxc_base        ((struct mxc_uart *)CONFIG_MXC_UART_BASE)
```

- 1
- 2

还没能看清楚使用的是哪个串口，那就继续往下追：

```
#define CONFIG_MXC_UART_BASE            UART1_BASE
```

- 1
- 2

看完参数看函数，看看_mxc_serial_init如何实现：

```
static void _mxc_serial_init(struct mxc_uart *base, int use_dte)
{
        writel(0, &base->cr1);
        writel(0, &base->cr2);

        while (!(readl(&base->cr2) & UCR2_SRST));

        if (use_dte)
                writel(0x404 | UCR3_ADNIMP, &base->cr3);
        else
                writel(0x704 | UCR3_ADNIMP, &base->cr3);

        writel(0x704 | UCR3_ADNIMP, &base->cr3);
        writel(0x8000, &base->cr4);
        writel(0x2b, &base->esc);
        writel(0, &base->tim);

        writel(0, &base->ts);
}
```

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

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

最终就是直接操作寄存器了。到此为止，串口初始化函数`serial_init`基本追踪完毕了。

3.4.2.6 display_options：打印u-boot版本、编译时间

接着查看一下u-boot启动时一系列打印辅助信息函数，我们可以通过这些打印信息确定程序是否能执行到这里或者串口等问题。

```
int display_options(void)
{
        char buf[DISPLAY_OPTIONS_BANNER_LENGTH];

        display_options_get_banner(true, buf, sizeof(buf));
        printf("%s", buf);

        return 0;
}


char *display_options_get_banner(bool newlines, char *buf, int size)
{
        return display_options_get_banner_priv(newlines, BUILD_TAG, buf, size);
}


char *display_options_get_banner_priv(bool newlines, const char *build_tag,
                                      char *buf, int size)
{
        int len;

        len = snprintf(buf, size, "%s%s", newlines ? "\n\n" : "",
                       version_string);
        ...
        return buf;
}


const char __weak version_string[] = U_BOOT_VERSION_STRING;


#define U_BOOT_VERSION_STRING U_BOOT_VERSION " (" U_BOOT_DATE " - " \
        U_BOOT_TIME " " U_BOOT_TZ ")" CONFIG_IDENT_STRING


#define U_BOOT_DATE "Feb 05 2021"
#define U_BOOT_TIME "17:14:08"
#define U_BOOT_TZ "+0800"
#define U_BOOT_DMI_DATE "02/05/2021"
#define U_BOOT_BUILD_DATE 0x20210205
#define U_BOOT_EPOCH 1612516448
```

- 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

最终就是串口打印出u-boot的版本以及编译时间等等，继续查看下一个打印函数。

3.4.2.7 display_text_info：打印u-boot的地址

```
static int display_text_info(void)
{
#if !defined(CONFIG_SANDBOX) && !defined(CONFIG_EFI_APP)
        ulong bss_start, bss_end, text_base;

        bss_start = (ulong)&__bss_start;
        bss_end = (ulong)&__bss_end;

#ifdef CONFIG_SYS_TEXT_BASE
        text_base = CONFIG_SYS_TEXT_BASE;
#else
        text_base = CONFIG_SYS_MONITOR_BASE;
#endif

        debug("U-Boot code: %08lX -> %08lX  BSS: -> %08lX\n",
              text_base, bss_start, bss_end);
#endif

        return 0;
}
```

这个函数就只是debug一下u-boot的代码在内存中的位置而已，相对上面的简单一点。

3.4.2.8 print_cpuinfo：打印CPU信息

```
int print_cpuinfo(void)
{
        u32 cpurev;
        __maybe_unused u32 max_freq;

        cpurev = get_cpu_rev();

#if defined(CONFIG_IMX_THERMAL) || defined(CONFIG_IMX_TMU)
        struct udevice *thermal_dev;
        int cpu_tmp, minc, maxc, ret;

        printf("CPU:   Freescale i.MX%s rev%d.%d",
               get_imx_type((cpurev & 0x1FF000) >> 12),
               (cpurev & 0x000F0) >> 4,
               (cpurev & 0x0000F) >> 0);
        max_freq = get_cpu_speed_grade_hz();
        if (!max_freq || max_freq == mxc_get_clock(MXC_ARM_CLK)) {
                printf(" at %dMHz\n", mxc_get_clock(MXC_ARM_CLK) / 1000000);
        } else {
                printf(" %d MHz (running at %d MHz)\n", max_freq / 1000000,
                       mxc_get_clock(MXC_ARM_CLK) / 1000000);
        }
#else
        ...
#endif

#if defined(CONFIG_IMX_THERMAL) || defined(CONFIG_IMX_TMU)
        puts("CPU:   ");
        switch (get_cpu_temp_grade(&minc, &maxc)) {
        case TEMP_AUTOMOTIVE:
                puts("Automotive temperature grade ");
                break;
        case TEMP_INDUSTRIAL:
                puts("Industrial temperature grade ");
                break;
        case TEMP_EXTCOMMERCIAL:
                puts("Extended Commercial temperature grade ");
                break;
        default:
                puts("Commercial temperature grade ");
                break;
        }
        printf("(%dC to %dC)", minc, maxc);
        ret = uclass_get_device(UCLASS_THERMAL, 0, &thermal_dev);
        if (!ret) {
                ret = thermal_get_temp(thermal_dev, &cpu_tmp);

                if (!ret)
                        printf(" at %dC", cpu_tmp);
                else
                        debug(" - invalid sensor data\n");
```

```
        } else {
                debug(" - invalid sensor device\n");
        }
        puts("\n");
#endif

        printf("Reset cause: %s\n", get_reset_cause());
        return 0;
}
```

3.4.2.9 show_board_info：打印单板信息

```
int __weak show_board_info(void)
{
#ifdef CONFIG_OF_CONTROL
        DECLARE_GLOBAL_DATA_PTR;
        const char *model;

        model = fdt_getprop(gd->fdt_blob, 0, "model", NULL);

        if (model)
                printf("Model: %s\n", model);
#endif

        return checkboard();
}
```

由于imx6ull默认配置中定义了`CONFIG_OF_CONTROL`，也就是在u-boot中也使用了设备树文件，所以从设备树中获取“model”属性并打印出来，然后继续调用了`checkboard`函数：

```
int checkboard(void)
{
        if (is_cpu_type(MXC_CPU_MX6ULZ))
                puts("Board: MX6ULZ 14x14 EVK\n");
        else
                puts("Board: MX6ULL 14x14 EVK\n");

        return 0;
}
```

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

3.4.2.10 init_func_i2c：初始化**i2c**并打印

```
static int init_func_i2c(void)
{
        puts("I2C:   ");
        i2c_init_all();
        puts("ready\n");
        return 0;
}
```

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

3.4.2.11 announce_dram_init：打印**“DRAM: ”**字符串

```
static int announce_dram_init(void)
{
        puts("DRAM:  ");
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6

3.4.2.12 dram_init：设置**gd**结构体的**ram_size**成员

```
int dram_init(void)
{
        gd->ram_size = imx_ddr_size();

        return 0;
}
```

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

函数并非初始化，其实就只是根据MMDC的配置来设置一下前面所说的gd全局变量。

3.4.2.13 reserve_fdt：预留设备树文件的内存，并设置设备树新的内存地址到**gd->new_fdt**

```
static int reserve_fdt(void)
{
#ifndef CONFIG_OF_EMBED

        if (gd->fdt_blob) {
                gd->fdt_size = ALIGN(fdt_totalsize(gd->fdt_blob), 32);

                gd->start_addr_sp = reserve_stack_aligned(gd->fdt_size);
                gd->new_fdt = map_sysmem(gd->start_addr_sp, gd->fdt_size);
                debug("Reserving %lu Bytes for FDT at: %08lx\n",
                      gd->fdt_size, gd->start_addr_sp);
        }
#endif

        return 0;
}
```

- 1
- 2

```
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15
    • 16
    • 17
    • 18
    • 19
    • 20
    • 21
```

3.4.2.14 dram_init_banksize：设置**gd**结构体的**dram**信息

```
__weak int dram_init_banksize(void)
{
        gd->bd->bi_dram[0].start = gd->ram_base;
        gd->bd->bi_dram[0].size = get_effective_memsize();

        return 0;
}
```

```
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
```

3.4.2.15 show_dram_config：打印内存的大小

```
static int show_dram_config(void)
{
        unsigned long long size;
        int i;

        debug("\nRAM Configuration:\n");
        for (i = size = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
                size += gd->bd->bi_dram[i].size;
                debug("Bank #%d: %llx ", i,
                      (unsigned long long)(gd->bd->bi_dram[i].start));
#ifdef DEBUG
                print_size(gd->bd->bi_dram[i].size, "\n");
#endif
        }
        debug("\nDRAM:  ");

        print_size(size, "");
        board_add_ram_info(0);
        putc('\n');

        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
```

这个函数才真正的打印出DRAM的大小，如果还想知道单位等等就继续往`print_size`里查看，这里不继续往里探讨。

3.4.2.16 display_new_sp：打印新的**sp**栈内存地址

```
static int display_new_sp(void)
{
        debug("New Stack Pointer is: %08lx\n", gd->start_addr_sp);

        return 0;
}
```

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

3.4.2.17 reloc_fdt：拷贝设备树内容到新的地址去（重定位），并重新设置**gd->fdt_blob**

```
static int reloc_fdt(void)
{
#ifndef CONFIG_OF_EMBED
        if (gd->flags & GD_FLG_SKIP_RELOC)
                return 0;
        if (gd->new_fdt) {
                memcpy(gd->new_fdt, gd->fdt_blob, fdt_totalsize(gd->fdt_blob));
                gd->fdt_blob = gd->new_fdt;
        }
#endif

        return 0;
}
```

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

3.4.2.18 setup_reloc：设置重定位相关的信息，后面会用到

```
static int setup_reloc(void)
{
        if (gd->flags & GD_FLG_SKIP_RELOC) {
                debug("Skipping relocation due to flag\n");
                return 0;
        }

#ifdef CONFIG_SYS_TEXT_BASE
#ifdef ARM
        gd->reloc_off = gd->relocaddr - (unsigned long)__image_copy_start;
#elif defined(CONFIG_M68K)

        gd->reloc_off = gd->relocaddr - (CONFIG_SYS_TEXT_BASE + 0x400);
#elif !defined(CONFIG_SANDBOX)
        gd->reloc_off = gd->relocaddr - CONFIG_SYS_TEXT_BASE;
#endif
#endif
        memcpy(gd->new_gd, (char *)gd, sizeof(gd_t));

        debug("Relocation Offset is: %08lx\n", gd->reloc_off);
        debug("Relocating to %08lx, new gd at %08lx, sp at %08lx\n",
              gd->relocaddr, (ulong)map_to_sysmem(gd->new_gd),
              gd->start_addr_sp);

        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
- 30

`setup_reloc`函数计算了重定位的偏移地址值赋给`gd->reloc_off`，并且将gd结构体拷贝到前面`reserve_xxx`系列函数计算得到的新地址`gd->new_gd`，这里所涉及的2个gd结构体成员很重要，`board_init_f`函数返回到`_main`函数里马上就用上了：

```
/* file: arch/arm/lib/crt0.S */
        bl      board_init_f

        ldr     r0, [r9, #GD_START_ADDR_SP]     /* sp = gd->start_addr_sp */
        bic     r0, r0, #7      /* 8-byte alignment for ABI compliance */
        mov     sp, r0
        ldr     r9, [r9, #GD_NEW_GD]            /* r9 <- gd->new_gd */

        adr     lr, here
        ldr     r0, [r9, #GD_RELOC_OFF]         /* r0 = gd->reloc_off */
        add     lr, lr, r0
#if defined(CONFIG_CPU_V7M)
        orr     lr, #1                          /* As required by Thumb-only */
#endif
        ldr     r0, [r9, #GD_RELOCADDR]         /* r0 = gd->relocaddr */
        b       relocate_code
```

所以接下来的任务就是分析`relocate_code`代码重定位了。

未完待续...

(1条消息**) u-boot-2021.01**（**imx6ull**）启动流程分析之四：**relocate_code**和**relocate_vectors**重定位分析**___ASDFGH**的博客**-CSDN**博客

3.4.3 relocate_code

顾名思义，代码重定位。

```
/* file: arch/arm/lib/relocate.S */
ENTRY(relocate_code)
        ldr     r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
        subs    r4, r0, r1              /* r4 <- relocation offset */
        beq     relocate_done           /* skip relocation */
        ldr     r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */

copy_loop:
        ldmia   r1!, {r10-r11}          /* copy from source address [r1]    */
        stmia   r0!, {r10-r11}          /* copy to   target address [r0]    */
        cmp     r1, r2                  /* until source end address [r2]    */
        blo     copy_loop

        /*
         * fix .rel.dyn relocations
         */
        ldr     r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
        ldr     r3, =__rel_dyn_end      /* r3 <- SRC &__rel_dyn_end */
fixloop:
        ldmia   r2!, {r0-r1}            /* (r0,r1) <- (SRC location,fixup) */
        and     r1, r1, #0xff
        cmp     r1, #R_ARM_RELATIVE
        bne     fixnext

        /* relative fix: increase location by offset */
        add     r0, r0, r4
        ldr     r1, [r0]
        add     r1, r1, r4
        str     r1, [r0]
fixnext:
        cmp     r2, r3
        blo     fixloop

relocate_done:
...
        /* ARMv4- don't know bx lr but the assembler fails to see that */

#ifdef __ARM_ARCH_4__
        mov     pc, lr
#else
        bx      lr
#endif

ENDPROC(relocate_code)
```

从上面的代码可以看到，先将__image_copy_start~__image_copy_end部分重定位，根据u-boot.lds文件可以知道，这个区间就已经包含了vendor向量、text代码段和data段等等。

3.4.3.1 “image”部分重定位过程

在进入relocate_code部分之前，r0寄存器就保存了重定位的地址值（这点可以回到_main函数中验证）。

1. 把代码链接地址赋给r1寄存器；
2. 计算r4=r0-r1=“重定位目标地址”-“重定位源地址”，得到的是地址偏移量，**r4**这个偏移值后面在**.rel.dyn**段的配置会用上；
3. 如果r1和r0相等，那就根本不用重定位了，直接跳到重定位完毕的relocate_done标志去了。但是，如果刚才一路追踪着内存地址的设置，就会发现它们不会相等，所以需要重定位；
4. 通过ldmia命令（**ld**r/**m**any/**i**ncrease/**a**fter）读取”重定位源地址r1“开始的2个32位地址的数据保存到r10和r11寄存器中（完成后r1的值会更新）；
5. 然后又将r10和r11两个寄存器的值存放到“重定位目标地址r0”开始的2个地址处（完成后r0的值也会更新），这样就完成了2个32位的代码重定位；
6. 完成后判断“更新后的源地址r1”和“重定位结束的地址r2”是否相等，相等则代表重定位完成，不相等则需要返回第4步继续拷贝。

3.4.3.2 .rel.dyn部分配置过程

在分析.rel.dynd段fix（配置）之前，需要先知道的一点就是：程序运行时应当处于链接地址，要想访问某个变量，就得先得到变量的地址。 但由于重定位代码之后地址会发生改变，当需要访问这些数据时，就不应该使用绝对地址。在u-boot程序中是使用了pc寄存器来偏移获得这个数据的Label值，Label就保存着这个变量的地址，一旦知道Label所保存的地址那就可以访问这个数据，所以重定位之后修改Label所保存的值就能正确地访问了。至于它的原理，举个例子看一下就清楚了，这里就添加一个函数来查看一下变量的存储方式（仅供测试，不考虑程序运行情况）：

```
@@ -951,8 +951,18 @@ static const init_fnc_t init_sequence_f[] = {
        NULL,
 };

+int rel_test_val = 0x100;
+
+void rel_test_fun(void)
+{
+       rel_test_val = 0x200;
+       printf("rel_test!\n");
+}
+
 void board_init_f(ulong boot_flags)
 {
+       rel_test_fun();
+
        gd->flags = boot_flags;
        gd->have_console = 0;
```

再执行`arm-linux-gnueabihf-objdump -D -m arm u-boot > u-boot.dis`进行反汇编一下看看关于这部分的实现：

```
/* file: u-boot.dis */
// Disassembly of section .text_rest:
8780b978 <rel_test_fun>:
8780b978: f44f 7200   mov.w r2, #512  ; 0x200
8780b97c: 4b02        ldr r3, [pc, #8]  ; (8780b988 <rel_test_fun+0x10>)
8780b97e: 4803        ldr r0, [pc, #12] ; (8780b98c <rel_test_fun+0x14>)
8780b980: 601a        str r2, [r3, #0]
8780b982: f025 be68   b.w 87831656 <printf>
8780b986: bf00        nop
8780b988: 878469d0      ; <UNDEFINED> instruction: 0x878469d0
8780b98c: 8783fa93      ; <UNDEFINED> instruction: 0x8783fa93

8780b990 <board_init_f>:
8780b990: b570        push  {r4, r5, r6, lr}
8780b992: 4604        mov r4, r0
8780b994: f7ff fff0   bl  8780b978 <rel_test_fun>
8780b998: 2200        movs  r2, #0
8780b99a: 4d0e        ldr r5, [pc, #56] ; (8780b9d4 <board_init_f+0x44>)
...

// Disassembly of section .data:
878469d0 <rel_test_val>:
878469d0: 00000100    andeq r0, r0, r0, lsl #2
...

// Disassembly of section .rel.dyn:
8784b6dc:   8780b988    strhi   fp, [r0, r8, lsl #19]
8784b6e0:   00000017    andeq   r0, r0, r7, lsl r0
```

看到以上反汇编内容，一条一条指令分析（冒号左侧为内存地址，右侧为机器码、指令）：

- 8780b994：`board_init_f`函数通过bl命令跳转到了`8780b978`地址处的`rel_test_fun`函数；
- 8780b978：在`rel_test_fun`函数中，将0x200写入到r2寄存器；
- 8780b97c：将pc寄存器的值加上8赋给r3寄存器。由于ARM三级流水线结构（在执行第一条指令时，第二条指令正在译码阶段，第三条指令正在取指阶段，pc寄存器就是指向取指的地址。参考文章），而随后的2条指令都是16位的thumb指令集，所以pc=8780b97c+2+2=8780b980，而不是8780b97c+4+4，故r3=0x8780b980+8=0x8780b988；

- 8780b988：它随着程序地址变化而变化，俗称位置无关码，也就是**“Label”**，它保存的就是一个地址。在这个例子中保存rel_test_val的地址值是878469d0；
- 878469d0：rel_test_val的地址，冒号右侧就是保存着它的值；
- 8780b980：将r2的0x200保存到r3保存的地址中，这样就完成了一次rel_test_val的赋值。

这个函数在重定位之前是正常运行的，但是重定位之后，代码和数据的地址都发生了变化，而Label处保存的值还是原样的，所以访问该数据就还是重定位之前的数据。

然而，在编译u-boot的时候已经加上了**“-pie”**选项，它的作用就是生成位置无关码，编译时生成一个.rel.dyn段，通过这个段可以对重定位后的代码进行“补充纠正”。那就是`relocate_code`函数后半部分的**“fix配置”**了，它的过程如下：

1. 首先使用ldmia命令读取r2保存的地址开始的2个32位地址的值存到r0和r1（r1保存的是高地址的值）；
2. 将r1寄存器和0xff进行**“&运算”**后保存到r1寄存器，目的是想获得低8位的值；
3. 将r1和R_ARM_RELATIVE比较，这个值在`include/elf.h`文件中定义，它的值为23，也就是16进制的0x17；
4. 如果r1的低8位不是0x17，则跳到后面比较地址判断.rel.dyn段是否已经配置完成，如果地址不相等则返回去继续第1步；
5. 如果r1的低8位是0x17，则代表r0保存的地址值是一个**“Label”**，例如上面反汇编文件中的8780b988地址；
6. 更新**“Label”**保存的值：

   ① 将Label保存的地址加上重定位偏移的值r4得到新Label，假设r4=0x10002000，那么就可以知道重定位后新Label是在8780b988+10002000=9780d988处，把它保存到r0寄存器中；

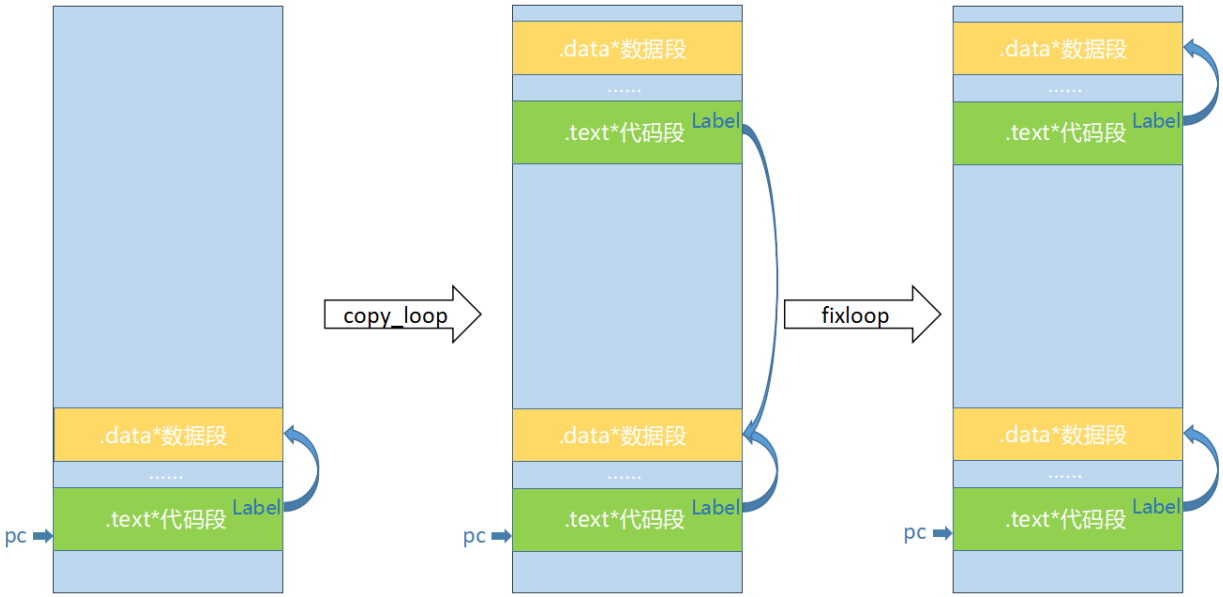
   ② 读取r0（新Label）的值存到r1寄存器中，这时新Label保存的还是变量旧的地址：878469d0；

   ③ 将r1（变量旧的地址）也加上重定位偏移的值r4，就变成了878469d0+10002000=978489d0；

   ④ 再将r1的值（变量新的地址978489d0）写回到r0（新Label）处，这样就完成了一次Label的更新。

其实，以上6个步骤的操作可以总结为一句话：将重定位后代码段中的**Label**所保存的地址值更新为对应的新地址。 以上面反汇编为例，在重定位后将Label保存的地址值878469d0更新为978489d0，这样才能正确地访问到更新后的rel_test_val变量。

整个`relocate_code`代码重定位可以简化为一张图：



relocate_code执行过程

需要注意的是，在`relocate_code`函数末尾部分可以看到以下部分：

```
#ifdef __ARM_ARCH_4__
        mov     pc, lr
#else
        bx      lr
#endif
```

由于没有定义`__ARM_ARCH_4__`，所以执行的是`bx lr`语句。需要知道的一点是，当使用bl或blx跳转去执行的时候，lr（r14）寄存器保存的是pc（r15）寄存器减4的地址，也就是返回的地址（参考文章），所以这时还没有跳转到重定位之后的内存地址中去。那它是什么时候跳转的呢？它又是如何跳转的呢？那就是在`_main`函数里调用完`relocate_code`和其他一些函数之后，才使用绝对跳转跳到重定位之后的地址去的：

```
#if CONFIG_IS_ENABLED(SYS_THUMB_BUILD)
        ldr     lr, =board_init_r       /* this is auto-relocated! */
        bx      lr
#else
        ldr     pc, =board_init_r       /* this is auto-relocated! */
#endif
```

这里知道跳转的时机和方式即可，board_init_r函数后面会研究。

3.4.4 relocate_vectors

从名字也可以知道它的功能，它就是重定位vectors向量表:

```
/* file: arch/arm/lib/relocate.S */
ENTRY(relocate_vectors)

#ifdef CONFIG_CPU_V7M
        /*
         * On ARMv7-M we only have to write the new vector address
         * to VTOR register.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        ldr     r1, =V7M_SCB_BASE
        str     r0, [r1, V7M_SCB_VTOR]
#else
#ifdef CONFIG_HAS_VBAR
        /*
         * If the ARM processor has the security extensions,
         * use VBAR to relocate the exception vectors.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        mcr     p15, 0, r0, c12, c0, 0  /* Set VBAR */
#else
        /*
         * Copy the relocated exception vectors to the
         * correct address
         * CP15 c1 V bit gives us the location of the vectors:
         * 0x00000000 or 0xFFFF0000.
         */
        ldr     r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
        mrc     p15, 0, r2, c1, c0, 0   /* V bit (bit[13]) in CP15 c1 */
        ands    r2, r2, #(1 << 13)
        ldreq   r1, =0x00000000         /* If V=0 */
        ldrne   r1, =0xFFFF0000         /* If V=1 */
        ldmia   r0!, {r2-r8,r10}
        stmia   r1!, {r2-r8,r10}
        ldmia   r0!, {r2-r8,r10}
        stmia   r1!, {r2-r8,r10}
#endif
#endif
        bx      lr

ENDPROC(relocate_vectors)
```

根据宏定义的条件判断，最终由于判断CONFIG_HAS_VBAR部分条件为真，所以整段代码也就只有简化成以下三句内容:

```
ldr     r0, [r9, #GD_RELOCADDR]
mcr     p15, 0, r0, c12, c0, 0
bx      lr
```

根据注释也可以很清楚知道，它就只是将cp15协处理器对应的VBAR设置为重定位之后的uboot起始地址，也即vectors的地址。因为vectors区是在uboot的最前面，而前面的relocate_code已经也将它一起重定位了，所以只需要简单设置新的地址即可。（还记得前面_start部分已经设置过cp15寄存器支持重定向vectors区并且设置当时的地址，现在重定位之后当然需要将地址修改过来。）

重定位完成之后，就继续调用board_init_r函数初始化。

未完待续...

(1条消息) u-boot-2021.01（imx6ull）启动流程分析之五：board_init_r函数分析___ASDFGH的博客-CSDN博客

3.4.5 board_init_r

从函数名称也可以知道，它也是负责一些初始化，但它还有一个目的就是通过层层调用之后启动内核。将相关的宏定义简化一下如下：

```
void board_init_r(gd_t *new_gd, ulong dest_addr)
{

        gd->flags &= ~GD_FLG_LOG_READY;

        if (initcall_run_list(init_sequence_r))
                hang();


        hang();
}
```

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

类似地，同样可以找到函数的重点——`init_sequence_r`函数指针数组，将宏定义简化一下如下：

```
static init_fnc_t init_sequence_r[] = {
        initr_trace,
        initr_reloc,
    initr_caches,
    initr_reloc_global_data,
    initr_barrier,
        initr_malloc,
        log_init,
        initr_bootstage,
        initr_console_record,
    initr_of_live,
    initr_dm,
    board_init,
        efi_memory_init,
    initr_binman,
        initr_dm_devices,
```

```
        stdio_init_tables,
        serial_initialize,
        initr_announce,
    INIT_FUNC_WATCHDOG_RESET
    initr_mmc,
    initr_env,
    stdio_add_devices,
        initr_jumptable,
        console_init_r,
    interrupt_init,
    initr_ethaddr,
    board_late_init,
    initr_net,
    run_main_loop,
};
```

函数比较多，所以和`board_init_f`函数一样，也只挑一部分比较重要的函数研究，其余的函数也类似地自行研究即可。

3.4.5.1 serial_initialize：注册各个厂家的串口驱动

```
int serial_initialize(void)
{
        atmel_serial_initialize();
        mcf_serial_initialize();
        mpc85xx_serial_initialize();
        mxc_serial_initialize();
```

```
        ns16550_serial_initialize();
        pl01x_serial_initialize();
        pxa_serial_initialize();
        sh_serial_initialize();
        mtk_serial_initialize();

        serial_assign(default_serial_console()->name);

        return 0;
}
```

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

看到前面那一大坨的`xxx_serial_initialize`，这里包括不同芯片厂家的串口“初始化”，但其实它并不是真正的“初始化”，查看它的内部实现就会发现，里面都是使用`serial_register`来将各个厂商的串口加入到链表。加入到链表也得用才行，那就是`serial_assign`函数：

```
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;
}
```

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

而函数的实参default_serial_console()->name似曾相识，在分析board_init_f函数时就已经接触过了。看了前面这些调用，其实serial_initialize函数的主要功能是将串口注册进链表，然后从链表里“挑”出一个串口来作为stdin/stdout/stderr，imx6ull芯片就很明显会用到mxc_serial_drv驱动，而不像函数名一样误认为是初始化所有厂家的SoC串口。

3.4.5.2 initr_announce：debug一下u-boot重定位后的地址

```
static int initr_announce(void)
{
        debug("Now running in RAM - U-Boot at: %08lx\n", gd->relocaddr);
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6

3.4.5.3 initr_mmc：初始化mmc设备

```
static int initr_mmc(void)
{
        puts("MMC:   ");
        mmc_initialize(gd->bd);
        return 0;
}
```

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

继续往里面探索：

```
int mmc_initialize(struct bd_info *bis)
{
        static int initialized = 0;
        int ret;
        if (initialized)
                return 0;
        initialized = 1;

#if !CONFIG_IS_ENABLED(BLK)
#if !CONFIG_IS_ENABLED(MMC_TINY)
        mmc_list_init();
#endif
#endif
        ret = mmc_probe(bis);
```

```
        if (ret)
                return ret;

#ifndef CONFIG_SPL_BUILD
        print_mmc_devices(',');
#endif

        mmc_do_preinit();
        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

里面主要调用了3个函数，其中mmc_list_init函数初始化链表，后面初始化mmc设备会用到，但不用深入了解，知道作用即可；主要还是看mmc_probe和mmc_do_preinit，先看mmc_probe：

```
static int mmc_probe(struct bd_info *bis)
{
        if (board_mmc_init(bis) < 0)
                cpu_mmc_init(bis);

        return 0;
}
```

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

只有两个函数：board_mmc_init和cpu_mmc_init；

```
__weak int board_mmc_init(struct bd_info *bis)
{
        return -1;
}
```

- 1
- 2
- 3
- 4
- 5

所以判断返回之后还会调用到cpu_mmc_init:

```
int cpu_mmc_init(struct bd_info *bis)
{
        return fsl_esdhc_mmc_init(bis);
}
```

- 1
- 2
- 3
- 4
- 5

继续看调用:

```
int fsl_esdhc_mmc_init(struct bd_info *bis)
{
        struct fsl_esdhc_cfg *cfg;

        cfg = calloc(sizeof(struct fsl_esdhc_cfg), 1);
        cfg->esdhc_base = CONFIG_SYS_FSL_ESDHC_ADDR;
        cfg->sdhc_clk = gd->arch.sdhc_clk;
        return fsl_esdhc_initialize(bis, cfg);
}
```

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

函数里面使用“基地址”和“时钟频率”传递到fsl_esdhc_initialize函数中初始化，函数里面调用的内容分析起来比较多，就只列出个别重要的操作:

```
fsl_esdhc_initialize(bis, cfg);
    fsl_esdhc_cfg_to_priv(cfg, priv);
        priv->esdhc_regs = (struct fsl_esdhc *)(unsigned long)(cfg->esdhc_base);
        priv->bus_width = cfg->max_bus_width;
        priv->sdhc_clk = cfg->sdhc_clk;
        ...
    fsl_esdhc_init(priv, plat);
                regs = priv->esdhc_regs;
                esdhc_reset(regs);
                esdhc_setbits32(&regs->sysctl, ...
```

```
                esdhc_write32(&regs->mixctrl, 0);
                cfg = &plat->cfg;
                cfg->name = "FSL_SDHC";
                cfg->ops = &esdhc_ops;
                        .getcd          = esdhc_getcd,
                        .init           = esdhc_init,
                        .send_cmd       = esdhc_send_cmd,
                        .set_ios        = esdhc_set_ios,
                ...
        mmc_create(&plat->cfg, priv);
                mmc->cfg = cfg;
                mmc->priv = priv;
                bdesc = mmc_get_blk_desc(mmc);
                bdesc->if_type = IF_TYPE_MMC;
                bdesc->removable = 1;
                bdesc->devnum = mmc_get_next_devnum();
                bdesc->block_read = mmc_bread;
                bdesc->block_write = mmc_bwrite;
                bdesc->block_erase = mmc_berase;
                mmc_list_add(mmc);
```

需要注意的是，上面这些内容主要是配置Soc里面的eSDHC控制器和一些mmc驱动的属性和操作函数。也就是说mmc设备还没初始化，前面说的`mmc_do_preinit`函数还没研究呢，回去看看：

```
void mmc_do_preinit(void)
{
        struct udevice *dev;
        struct uclass *uc;
        int ret;
```

```
        ret = uclass_get(UCLASS_MMC, &uc);
        if (ret)
                return;
        uclass_foreach_dev(dev, uc) {
                struct mmc *m = mmc_get_mmc_dev(dev);

                if (!m)
                        continue;
                if (m->preinit)
                        mmc_start_init(m);
        }
}
```

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

前面的遍历是为了得到找到mmc设备，一旦找到该设备，就进行以下函数调用：

```
mmc_start_init(m);
        mmc_get_op_cond(mmc);
                mmc_power_init(mmc);
                mmc_power_cycle(mmc);
                mmc->cfg->ops->init(mmc);
                mmc_set_initial_state(mmc);
                mmc_go_idle(mmc);
                ...
```

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

可以看到，前面配置的那些属性和函数在这里就用上了。

3.4.5.4 initr_ethaddr：设置网卡mac地址

```
static int initr_ethaddr(void)
{
        struct bd_info *bd = gd->bd;


        eth_env_get_enetaddr("ethaddr", bd->bi_enetaddr);

        return 0;
}
```

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

从函数定义可以知道函数从环境变量`ethaddr`来获取值来设置`bd->bi_enetaddr`。

3.4.5.5 initr_env：初始化环境变量

```
static int initr_env(void)
{

        if (should_load_env())
                env_relocate();
        else
                env_set_default(NULL, 0);

        if (IS_ENABLED(CONFIG_OF_CONTROL))
                env_set_hex("fdtcontroladdr",
                            (unsigned long)map_to_sysmem(gd->fdt_blob));


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

        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

- 16
- 17
- 18

函数一开始先是调用should_load_env函数判断是否需要加载环境变量:

```
static int should_load_env(void)
{
        if (IS_ENABLED(CONFIG_OF_CONTROL))
                return fdtdec_get_config_int(gd->fdt_blob,
                                             "load-environment", 1);

        if (IS_ENABLED(CONFIG_DELAY_ENVIRONMENT))
                return 0;

        return 1;
}
```

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

由于imx6ull默认配置中定义了CONFIG_OF_CONTROL宏，所以继续调用fdtdec_get_config_int函数读取设备树信息:

```
int fdtdec_get_config_int(const void *blob, const char *prop_name,
                          int default_val)
{
        int config_node;

        debug("%s: %s\n", __func__, prop_name);
        config_node = fdt_path_offset(blob, "/config");
        if (config_node < 0)
                return default_val;
        return fdtdec_get_int(blob, config_node, prop_name, default_val);
}
```

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

整个函数的调用流程可以大概看出程序是在根节点下的config节点找到"load-environment"属

性，如果找不到则使用默认值1。

回到initr_env函数之后，继续调用env_relocate函数，从名字就可以猜测它是重定位环境变量的。然后接着就是设置环境变量“fdtcontroladdr”，最后获取环境变量“loadaddr”的值来设置全局变量“image_load_addr”，这个变量保存的就是kernel存放的地址。

3.4.5.6 board_late_init：根据cpu型号来设置环境变量

```
int board_late_init(void)
{
#ifdef CONFIG_CMD_BMODE
        add_board_boot_modes(board_boot_modes);
#endif

#ifdef CONFIG_ENV_VARS_UBOOT_RUNTIME_CONFIG
        if (is_cpu_type(MXC_CPU_MX6ULZ))
                env_set("board_name", "ULZ-EVK");
        else
                env_set("board_name", "EVK");
        env_set("board_rev", "14X14");
#endif

        return 0;
}
```

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

3.4.5.7 initr_net：初始化网卡

它是网卡的初始化函数，在使用不同厂家的网卡芯片需要进行修改，函数定义如下：

```
static int initr_net(void)
{
        puts("Net:   ");
        eth_initialize();
#if defined(CONFIG_RESET_PHY_R)
        debug("Reset Ethernet PHY\n");
        reset_phy();
```

```
#endif
        return 0;
}
```

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

往函数eth_initialize里面继续探究：

```
int eth_initialize(void)
{
        int num_devices = 0;
        struct udevice *dev;

        eth_common_init();
    ...
}
```

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

继续往eth_common_init里走：

```
void eth_common_init(void)
{
        bootstage_mark(BOOTSTAGE_ID_NET_ETH_START);
#if defined(CONFIG_MII) || defined(CONFIG_CMD_MII) || defined(CONFIG_PHYLIB)
        miiphy_init();
#endif

#ifdef CONFIG_PHYLIB
        phy_init();
#endif
}
```

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

- 10
- 11
- 12

继续往phy_init里走:

```
int phy_init(void)
{
#ifdef CONFIG_NEEDS_MANUAL_RELOC

        struct list_head *head = &phy_drivers;

        head->next = (void *)head->next + gd->reloc_off;
        head->prev = (void *)head->prev + gd->reloc_off;
#endif

#ifdef CONFIG_B53_SWITCH
        phy_b53_init();
#endif
#ifdef CONFIG_MV88E61XX_SWITCH
        phy_mv88e61xx_init();
#endif
...
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23

由于函数内部是初始化各个网卡厂家的芯片，所以这里仅列出几个。以mv88e61xx系列为例，查看phy_mv88e61xx_init函数内部实现:

```
int phy_mv88e61xx_init(void)
{
        phy_register(&mv88e61xx_driver);
        phy_register(&mv88e609x_driver);
        phy_register(&mv88e6071_driver);

        return 0;
}
```

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

可以看到，在phy_init函数里根据宏定义来决定使用哪个厂商的网卡，而在对应的厂商函数内部就是不同型号的网卡芯片进行注册。这部分对于移植u-boot来说相当重要，尤其是开发板使用与公版不同厂商的网卡芯片。

3.4.5.8 run_main_loop：启动内核/解析命令行输入

一般来说，如果程序执行到这里，后续几乎不会出现什么奇怪的现象了，因为不管哪个SoC，这部分都是属于相同的。对于移植u-boot，后续一般不需要理会，但如果需要添加新功能，还是可以继续往下探究。函数定义如下：

```
static int run_main_loop(void)
{
#ifdef CONFIG_SANDBOX
        sandbox_main_loop_init();
#endif

        for (;;)
                main_loop();
        return 0;
}
```

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

比较显眼的就是main_loop函数，函数调用比较复杂，所以只列出函数调用过程：

```
main_loop
        env_set("ver", ...
        cli_init
        s = bootdelay_process();
                s = env_get("bootdelay");
                bootdelay = s ? (int)...
                bootdelay = fdtdec_get_config_int(gd->fdt_blob, "bootdelay", ...
                s = env_get("bootcmd");
                stored_bootdelay=bootdelay;
                return s;
        cli_process_fdt(&s)
                cli_secure_boot_cmd(s);
```

```
autoboot_command(s);
        if(s && (stored_bootdelay == -2 ||
        (stored_bootdelay != -1 && !abortboot(stored_bootdelay))))
                run_command_list(s, -1, 0);
        parse_string_outer(buff, FLAG_PARSE_SEMICOLON);
                setup_string_in_str(&input, p);
                parse_stream_outer(&input, flag);
                parse_stream(&temp, ...
                    run_list(ctx.list_head);
                        run_list_real(pi);
                            run_pipe_real(pi);
                                cmd_process(flag, child->argc, ...
                                    find_cmd
                                    cmd_call
                                    cmd_usage
cli_loop();
        bootstage_mark(BOOTSTAGE_ID_ENTER_CLI_LOOP);
        parse_file_outer();
                setup_file_in_str(&input);
    parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);
```

- 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

看到上面的函数调用关系可以知道，`main_loop`函数先获取环境变量`bootdelay`和`bootcmd`，如果u-boot配置了`CONFIG_OF_CONTROL`，则在`cli_process_fdt`函数里获取设备树的`bootcmd`和`bootsecure`属性，如果返回的`bootsecure`属性不为0，则进入`cli_secure_boot_cmd`函数解析并执行设备树中的`bootcmd`属性内容；否则进入到`autoboot_command`函数里倒计时，倒计时结束后就会解析并执行环境变量`bootcmd`的内容；如果倒计时过程中检测到串口有输入，就会退出该函数，进入`cli_loop`函数里面解析命令行的输入。

至于find_cmd、cmd_call和cmd_usage三个函数的实现，必须先看u-boot命令在源码中的组织形式：

```
#define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help)          \
        U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, NULL)


#define U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, _comp) \
        ll_entry_declare(struct cmd_tbl, _name, cmd) =                  \
                U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,  \
                                          _usage, _help, _comp);


#define ll_entry_declare(_type, _name, _list)                           \
        _type _u_boot_list_2_##_list##_2_##_name __aligned(4)           \
                        __attribute__((unused,                          \
                        section(".u_boot_list_2_"#_list"_2_"#_name)))

#define U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,          \
                                  _usage, _help, _comp)                 \
                { #_name, _maxargs,                                     \
                  _rep ? cmd_always_repeatable : cmd_never_repeatable,  \
                  _cmd, _usage, _CMD_HELP(_help) _CMD_COMPLETE(_comp) }


# define _CMD_COMPLETE(x) x,
# define _CMD_HELP(x) x,
```

就以“version”命令为例，解析简化之后它的定义：

```
U_BOOT_CMD(
        version,        1,              1,      do_version,
        "print monitor, compiler and linker version",
        ""
```

```
);
```

- 1
- 2
- 3
- 4
- 5
- 6

经过转化之后的定义就是：

```
struct cmd_tbl _u_boot_list_2_cmd_2_version __aligned(4) \
    __attribute__((unused, section(".u_boot_list_2_cmd_2_version"))) = {
    version, 1, cmd_always_repeatable,
        do_version, "print monitor, compiler and linker version", "", NULL
};
```

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

其中，结构体`cmd_tbl`原型如下：

```
struct cmd_tbl {
        char            *name;
        int             maxargs;
        int             (*cmd_rep)(struct cmd_tbl *cmd, int flags, int argc,
                                   char *const argv[], int *repeatable);

        int             (*cmd)(struct cmd_tbl *cmd, int flags, int argc,
                               char *const argv[]);
        char            *usage;
#ifdef  CONFIG_SYS_LONGHELP
        char            *help;
#endif
#ifdef CONFIG_AUTO_COMPLETE

        int             (*complete)(int argc, char *const argv[],
                                    char last_char, int maxv, char *cmdv[]);
#endif
};
```

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

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

所以，回到“查找命令”的`find_cmd`函数：

```
struct cmd_tbl *find_cmd(const char *cmd)
{
        struct cmd_tbl *start = ll_entry_start(struct cmd_tbl, cmd);
        const int len = ll_entry_count(struct cmd_tbl, cmd);
        return find_cmd_tbl(cmd, start, len);
}
```

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

里面分别调用了`ll_entry_start`宏和`ll_entry_count`宏来获得命令所在的区间范围：

```
#define ll_entry_start(_type, _list)                                    \
({                                                                      \
        static char start[0] __aligned(4) __attribute__((unused,        \
                section(".u_boot_list_2_"#_list"_1")));                 \
        (_type *)&start;                                                \
})
```

```
#define ll_entry_count(_type, _list)                                    \
({                                                                  \
        _type *start = ll_entry_start(_type, _list);            \
        _type *end = ll_entry_end(_type, _list);                \
        unsigned int _ll_result = end - start;                  \
        _ll_result;                                             \
})

#define ll_entry_end(_type, _list)                                      \
({                                                                      \
        static char end[0] __aligned(4) __attribute__((unused,          \
                section(".u_boot_list_2_"#_list"_3")));                 \
        (_type *)&end;                                                  \
})
```

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

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23

根据上面宏定义也可以看出回去程序所在的section段的区间里计算_type类型的个数。获得个数之后，`find_cmd`函数里就可以调用`find_cmd_tbl(cmd, start, len);`来找出匹配的命令并且作为返回值:

```
struct cmd_tbl *find_cmd_tbl(const char *cmd, struct cmd_tbl *table,
                             int table_len)
{
    ...
    for (cmdtp = table; cmdtp != table + table_len; cmdtp++) {
                if (strncmp(cmd, cmdtp->name, len) == 0) {
                        if (len == strlen(cmdtp->name))
                                return cmdtp;
                        ...
                }
        }
}
```

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

既然能找出命令所在的结构体的位置了，那调用它的成员函数就比较简单了。首先就是调用该命令的执行函数:

```
static int cmd_call(struct cmd_tbl *cmdtp, int flag, int argc,
                    char *const argv[], int *repeatable)
{
        int result;

        result = cmdtp->cmd_rep(cmdtp, flag, argc, argv, repeatable);
        if (result)
                debug("Command failed, result=%d\n", result);
        return result;
```

```
}
```

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

如果命令输入不正确等情况，那就返回调用者调用其他函数打印的用法:

```
int cmd_usage(const struct cmd_tbl *cmdtp)
{
        printf("%s - %s\n\n", cmdtp->name, cmdtp->usage);

#ifdef  CONFIG_SYS_LONGHELP
        printf("Usage:\n%s ", cmdtp->name);

        if (!cmdtp->help) {
                puts ("- No additional help available.\n");
                return 1;
        }

        puts(cmdtp->help);
        putc('\n');
#endif
        return 1;
}
```

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

至此，启动流程讲得差不多了。但是别忘了u-boot的最终目的是启动内核，所以还没有结束，接着往下看！

未完待续...

(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

- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

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;
}
```

- 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

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

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;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

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！