# 一、连接脚本u-boot.lds详解

1、从u-boot.lds可知，uboot入口地址为\_start

\_\_image\_copy\_start -》 0x87800000

.vetcors -> 0x87800000 存放中断向量表

arch/arm/cpu/armv7/start.o start.c

\_\_image\_copy\_end -》 0x8785dc6c

\_\_rel\_dyn\_start -> 0x8785dc6c rel段

\_\_rel\_dyn\_end -> 0x878668a4

\_\_end -》 0x878668a4

\_image\_binary\_end -》 0x878668a4

\_\_bss\_start -》 0x8785dc6c bss段。

\_\_bss\_end -》 0x878a8d74

# 二、uboot启动流程

1、reset函数

bicne=bic + ne

①、reset函数目的是将处理器设置为SVC模式，并且关闭FIQ和IRQ.

②、设置中断向量。

③、初始化CP15

④、

2、lowlevel\_init函数

CONFIG\_SYS\_INIT\_SP\_ADDR

#define CONFIG\_SYS\_INIT\_SP\_ADDR \

(CONFIG\_SYS\_INIT\_RAM\_ADDR + CONFIG\_SYS\_INIT\_SP\_OFFSET)

#define CONFIG\_SYS\_INIT\_SP\_OFFSET \

(CONFIG\_SYS\_INIT\_RAM\_SIZE - GENERATED\_GBL\_DATA\_SIZE)

#define CONFIG\_SYS\_INIT\_RAM\_SIZE IRAM\_SIZE

#define IRAM\_SIZE 0x00020000

#define CONFIG\_SYS\_INIT\_RAM\_ADDR IRAM\_BASE\_ADDR

#define IRAM\_BASE\_ADDR 0x00900000 6ULL内部OCRAM

#define GENERATED\_GBL\_DATA\_SIZE 256

0x00900000 + CONFIG\_SYS\_INIT\_SP\_OFFSE =>

0x00900000 + CONFIG\_SYS\_INIT\_RAM\_SIZE - GENERATED\_GBL\_DATA\_SIZE =>

0x00900000 **+** 0x00020000 – 256 = 0x0091ff00

设置SP指针、R9寄存器

3、s\_init函数

空函数。

4、\_main函数

5、board\_init\_f函数

initcall\_run\_lis此函数会调用一系列的函数，这些函数保存在init\_sequence\_f数组里面，

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

U\_BOOT\_VERSION = U-Boot 2016.03

TOTAL\_MALLOC\_LEN (CONFIG\_SYS\_MALLOC\_LEN + CONFIG\_ENV\_SIZE)

CONFIG\_SYS\_MALLOC\_LEN (16 \* SZ\_1M)

CONFIG\_ENV\_SIZE SZ\_8K

mx6ullevk.c mx6ullevk.h这两个文件长打交道

6、relocate\_code函数

relocate\_code函数有一个参数，r0=gd->relocaddr=0X9FF47000，uboot重定位后的首地址。

r1=0X87800000 源地址起始地址。

r4=0X9FF47000-0X87800000=0X18747000 偏移。

r2=0x8785dc6c

当简单粗暴的将uboot从0X87800000拷贝到其他地方以后，关于函数调用、全局变量引用就会出问题。Uboot对于这个的处理方法就是采用位置无关码，这个就需要借助于.rel.dyn段。

8785dcf8 <rel\_a>:

8785dcf8: 00000000 andeq r0, r0, r0

878042b4 <rel\_test>:

878042b4: e59f300c ldr r3, [pc, #12] ; 878042c8 <rel\_test+0x14>

878042b8: e3a02064 mov r2, #100 ; 0x64

878042bc: e59f0008 ldr r0, [pc, #8] ; 878042cc <rel\_test+0x18>

878042c0: e5832000 str r2, [r3]

878042c4: ea00d64c b 87839bfc <printf>

878042c8: 8785dcf8 ; <UNDEFINED> instruction: 0x8785dcf8

878042cc: 87842aaf strhi r2, [r4, pc, lsr #21]

设置r3为878042b4+8+12=878042c8的值，r3=8785dcf8。这里并没有直接去读取rel\_a的地址，而是借助了878042c。

878042c8叫做Label。

重定位以后

9ffa4cf8 <rel\_a>:

9ffa4cf8: 00000000 andeq r0, r0, r0

9ff4b2b4<rel\_test>:

9ff4b2b4: e59f300c ldr r3, [pc, #12] ; 878042c8 <rel\_test+0x14>

9ff4b2b8: e3a02064 mov r2, #100 ; 0x64

9ff4b2bc: e59f0008 ldr r0, [pc, #8] ; 878042cc <rel\_test+0x18>

9ff4b2c0: e5832000 str r2, [r3]

9ff4b2c4: ea00d64c b 87839bfc <printf>

9ff4b2c8: 8785dcf8 ; <UNDEFINED> instruction: 0x8785dcf8

9ff4b2cc: 87842aaf strhi r2, [r4, pc, lsr #21]

Label中的值还是原来的！必须要将8785dcf8换为重定位后的rel\_a地址。读取9ff4b2c8里面的数据，也就是老的rel\_a的地址=8785dcf8+0x18747000=0x9ffa4cf8

重定位以后，需要对所有的Label保存的数据加上偏移！！

8785dcec: 87800020 strhi r0, [r0, r0, lsr #32]

8785dcf0: 00000017 andeq r0, r0, r7, lsl r0

……

8785e2fc: 878042c8 strhi r4, [r0, r8, asr #5]

8785e300: 00000017 andeq r0, r0, r7, lsl r0

878042c8+offset = 读取新的Label处的数据+offset

完成这个功能在连接的时候需要加上”-pie”

7、relocate\_vectors函数

设置VBAR寄存器为重定位后的中断向量表起始地址。

8、board\_init\_r函数

Board\_init\_r函数和board\_init\_f函数很类似。board\_init\_r也是执行init\_sequence\_r初始化序列。

9、run\_main\_loop函数

run\_main\_loop

-> main\_loop

-> bootdelay\_process 获取bootdelay的值，然后保存到stored\_bootdelay

全局变量里面，获取bootcmd环境变量值，并且将其

返回

-> autoboot\_command 参数是bootcmd的值。

-> abortboot 参数为boot delay，此函数会处理倒计时

-> abortboot\_normal 参数为boot delay，此函数会处理倒计时

-> cli\_loop uboot命令模式处理函数。

-> parse\_file\_outer

-> parse\_stream\_outer

-> parse\_stream 解析输入的字符，得到命令

-> run\_list 运行命令

-> run\_list\_real

-> run\_pipe\_real

-> cmd\_process 处理命令，也就是执行命令

10、cli\_loop函数

11、cmd\_process函数

Uboot使用U\_BOOT\_CMD来定义一个命令。CONFIG\_CMD\_XXX来使能uboot中的某个命令。

U\_BOOT\_CMD最终是定义了一个cmd\_tbl\_t类型的变量，所有的命令最终都是存放在.u\_boot\_list段里面。cmd\_tbl\_t的cmd成员变量就是具体的命令执行函数，命令执行函数都是do\_xxx。

cmd\_process

->find\_cmd 从.u\_boot\_list段里面查找命令，当找到对应的命令以后以返回值的

形式给出，为cmd\_tbl\_t类型

->cmd\_call

->cmdtp->cmd 直接引用cmd成员变量

# 三、bootz启动Linux内核过程

Uboot启动Linux内核使用bootz命令，bootm。。bootz是如何启动Linux内核？uboot的生命是怎么终止的呢？linux又是怎么启动的呢？

1、image全局变量

bootm\_headers\_t images;

2、do\_bootz函数

tftp 80800000 zImage

tftp 83000000 imx6ull-14x14-emmc-7-1024x600-c.dtb

bootz 80800000 - 83000000

bootz命令的执行函数，do\_xxxx，do\_bootz是bootz的执行函数。

do\_bootz

-> bootz\_start

-> do\_bootm\_states 阶段为BOOTM\_STATE\_START

-> bootm\_start 对images全局变量清零，

-> images->ep = 0X80800000

->bootz\_setup 判断zImage是否正确

-> bootm\_find\_images

· -> boot\_get\_fdt 找到设备树，然后将设备树起始地址和长度，写入到images

的ft\_addr和ft\_len成员变量中。

-> bootm\_disable\_interrupts 关闭中断相关

-> images.os.os = IH\_OS\_LINUX; 表示要启动Linux系统

-> do\_bootm\_states 状态BOOTM\_STATE\_OS\_PREP 、BOOTM\_STATE\_OS\_FAKE\_GO 、

BOOTM\_STATE\_OS\_GO,

-> bootm\_os\_get\_boot\_func 查找Linux内核启动函数。找到Linux内核启动函数

do\_bootm\_linux，赋值给boot\_fn。

-> boot\_fn(BOOTM\_STATE\_OS\_PREP, argc, argv, images); 就是do\_bootm\_linux。

-> boot\_prep\_linux 启动之前的一些工作，对于使用设备树来说，他会将

Bootargs传递给Linux内核，通过设备树完成。也就是向

Linux内核传参。

-> boot\_selected\_os BOOTM\_STATE\_OS\_GO, do\_bootm\_linux

-> do\_bootm\_linux，BOOTM\_STATE\_OS\_GO

-> boot\_jump\_linux

-> machid= gd->bd->bi\_arch\_number;

-> void (\*kernel\_entry)(int zero, int arch, uint params);

-> kernel\_entry = (void (\*)(int, int, uint))images->ep; 0X80800000。

-> announce\_and\_cleanup 输出Starting kernel……

-> kernel\_entry(0, machid, r2); 启动Linux内核。Uboot的最终使命，启动Linux内核。

zimage\_header 的zi\_magic为zimage的幻数，魔术数。应该为0x016f2818。前面有9个32位的数据，那么9\*4=36，0~35，第36个字节的数据开始就是zimage的幻数。

3、do\_bootm\_states函数

4、bootm\_os\_get\_boot\_func函数

5、do\_bootm\_linux函数