**综述**

建议读代码还是要使用sourceinsight这种软件，建议在研究u-boot代码前，先使用一下u-boot的功能。

**基本概念**

**固件**

有些程序在运行bootloader之前线运行一段固化的程序（固件，firmware），比如x86架构CPU的BIOS。在大多数嵌入式系统中没有固件，bootloader是上电后执行的第一个程序，一些是有的比如tiny6410。

**单阶段和多阶段**

多阶段：第一阶段使用汇编来实现，完成依赖于体系结构的初始化，如硬件设备初始化，RAM初始化，拷贝代码，设置好栈，跳转到C的入口点。第二阶段，使用C实现，完成更加复杂的功能，如初始化本阶段的硬件设备，检测系统内存映射（确定板子上使用了多少内存，一般可以直接设置，不必编写通用的程序），将内核映像和根文件系统映像从Flash上读到RAM空间中，为内核启动设置参数，调用内核（跳到入口点）。

**SPI Flash**

SPI FLASH对应的是nor flash，

在通信方式上Nor Flash 分为两种类型：CFI Flash（总线flash）和 SPI Flash。

SPI flash和 CFI Flash 的介质都是Norflash ，但是SPI 是通过通过SPI口对flash进行读写，而 CFI Flash 则以并行接口进行数据操作，需要你的MCU上有外部总线接口。

SPI容量都不是很大，市场上 SPI Flash 最大可以做到128Mbit，而且读写速度慢，但是价格便宜，操作简单。而parallel接口速度快，容量上市场上已经有1Gbit的容量，价格昂贵。

**u-boot结构**

u-boot的目录间可以大致分一个层次结构，可以为u-boot移植提供指导。lib\_generic common

post net fs disk dtt nand\_spl rtc drivers

board cpu lib\_xxx

最上层为通用的函数，中间层通用的设备驱动程序，最底层为相关于CPU和板子的程序。

**有意义的知识点**

1、bootloader的启动参数和启动程序分开存储（其实这是显然的，程序和数据就是分开存储的）

2、S3C2410被称为SoC（System on Chip），上面除了CPU还集成了UART，USB等外设的控制器。S3C2410的CPU为ARM920T。要注意区分SoC和CPU的差别。而板子是包括外部电路的，所以要注意区分板子和SoC。

3、编译u-boot前需要调用mkconfig做一下配置。指明架构，用于确定编译时使用的汇编语言（将asm连接到正确的架构上）；指明SOC，将arch指向对应的SOC上；创建顶层makefile包含的文件config.mk；创建config.h文件，用于包含开发板的头文件。

4、配置文件中以CONDIG\_开头的是选项，选项只是在几个选择中选择一个。以CFG开头的是参数，参数需要设置具体的值。

CONFIG\_除了设置一些参数外，主要用来设置u-boot的功能，选择使用文件中的哪一个部分（用于裁剪），而CFG\_用来设置更细节的参数。

5、bootloader和内核的交互

bootloader需要向板子传递RAM地址，指定根文件系统位置ramdisk，启动运行的第一个程序，控制台等信息。可以有三种方式

1、启动参数方式，已经很少使用。

2、约定好首地址，使用tag的数据结构，通过设置特定的tag标志起始和结束。

3、使用fdt

详细在下面的章节中叙述。

**imx6相关的u-boot**

**1、imx6的配置头文件**

imx6使用：u-boot-2015.04\include\configs\myimx6.h头文件，里面定义了板子相关的信息，比如一些宏定义，启动参数等。

#include<common.h>包含#include<config.h>,包含#include<myimx6.h>

**2、imx6所属架构：**

imx6属于cortex-a9架构，基于ARMv7-A架构，目前我们能见到的四核处理器大多都是属于Cortex-A9系列。所以在u-boot arch/arm/cpu目录底下的armv7目录。

**3、u-boot的lds文件和makefile文件分析**

**I、lds文件（arch/arm/cpu/u-boot.lds）：**

关于lds文件的解析参见“详细分析的文件”目录下的u-boot.lds文件。

**II、makefile文件：**

Makefile文件可以用来判断包含了那个文件。主目录的Makefile主要是将个目录下的文件综合起来，具体子目录中包含了哪些文件，需要看子目录下面的Makefile文件。

u-boot.elf: u-boot.bin

@$(OBJCOPY) -B aarch64 -I binary -O elf64-littleaarch64 \

$< u-boot-elf.o

@$(LD) u-boot-elf.o -o $@ \

--defsym=\_start=$(CONFIG\_SYS\_TEXT\_BASE) \

-Ttext=$(CONFIG\_SYS\_TEXT\_BASE)

指定程序的链接地址为CONFIG\_SYS\_TEXT\_BASE（运行地址），最终要将u-boot的代码拷贝到CONFIG\_SYS\_TEXT\_BASE代表的地址中，程序才能正常运行。

**注**：在\include\configs\myimx6.h文件中：

#define CONFIG\_SYS\_TEXT\_BASE 0x17800000

因此u-boot最终拷贝到SDRAM中的0x1780\_0000中。

**4、imx6的编译流程**

在编译前要修改顶层目录下的Makefile，设置CROSS\_COMPILE = yourCrossCompilePath/arm-none-linux-gnueabi-

注：yourCrossCompilePath表示你的交叉编译链路径，注意后面的需要是arm-none-linux-gnueabi-as/gcc/objcopy等等，否则可能不好用。对于我们这里的板子CROSS\_COMPILE = /home/hitxp/arm/opt/FriendlyARM/toolschain/4.4.3/bin/arm-none-linux-gnueabi-

同时也要将Makefile中的ARCH 设置为 ARCH = arm

u-boot-2015.04\include\configs

1) 执行sudo make myimx6ek200-6q\_defconfig

控制台将输出：

HOSTCC scripts/basic/fixdep

HOSTCC scripts/kconfig/conf.o

SHIPPED scripts/kconfig/zconf.tab.c

SHIPPED scripts/kconfig/zconf.lex.c

SHIPPED scripts/kconfig/zconf.hash.c

HOSTCC scripts/kconfig/zconf.tab.o

HOSTLD scripts/kconfig/conf

#

# configuration written to .config

#

这一步主要将相应的板子的信息myimx6ek200-6q\_defconfig写到顶层目录的.config文件中。

2) sudo make all:

先生成板子的config.h文件，然后根据config.h文件编译生成对应板子的u-boot.imx。

/\* Automatically generated - do not edit \*/

#define CONFIG\_IMX\_CONFIG board/myzr/myimx6/myimx6.cfg

#define CONFIG\_MX6Q 1

#define CONFIG\_DDR\_MB 1024

重新编译u-boot执行 make distclean 清除先前的影响。

**启动流程**

启动流程可以分成两个阶段：

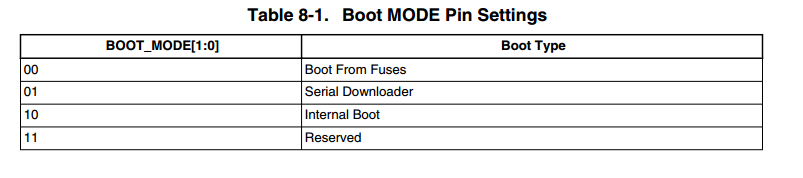
1、板子的启动：主要运行板子固化的ROM

2、u-boot的启动：u-boot启动又可以分成前期和后期。前期：完成硬件初始化，为后面运行功能程序构建环境。后期：运行u-boot指令，从而为内核启动创造条件。

**一、imx6q的启动流程：**

**1、与启动有关的设置和结构**

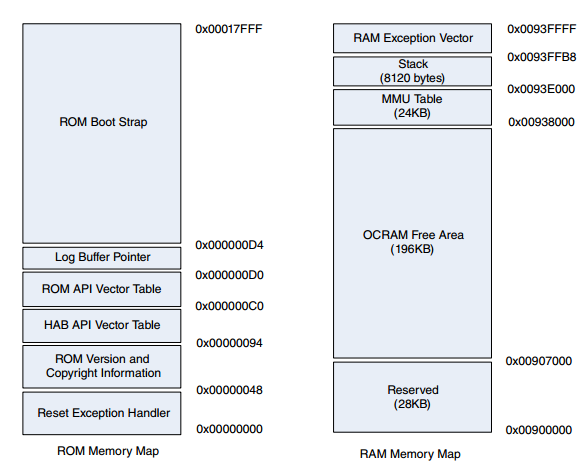
imx6q上电也是从0x00000000开始执行，有多种启动方式从mmc启动，从SD卡启动，从nand flash启动等等方式



设备有四种启动模式（其中一种为FSL的保留模式）。启动模式的选择基于保存在寄存器BOOT\_MODE中的二进制值。

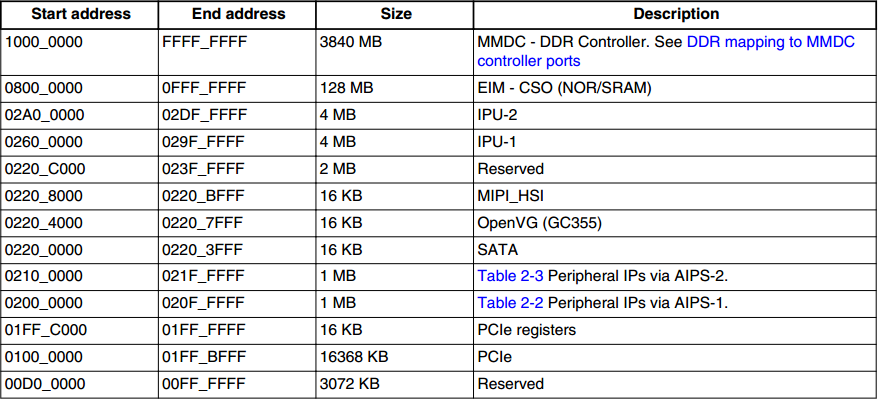
BOOT\_MODE的值 是在POR\_B的上升沿时 对BOOT\_MODE0和BOOT\_MODE1输入引脚的采样值，一旦此输入被采样，它们其后的状态就不再影响BOOT\_MODE内部寄存器的内容。

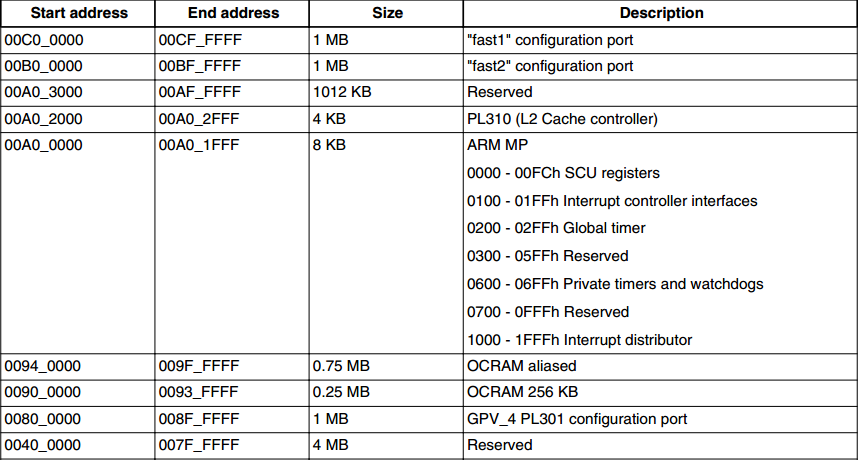
内部BOOT\_MODE寄存器的状态或许读自SOC的启动模式寄存器（SRC\_SBMR2）的BMOD[1:0] 域。可用的启动模式有：从Fuses启动，通过USB的串行启动，内部启动。

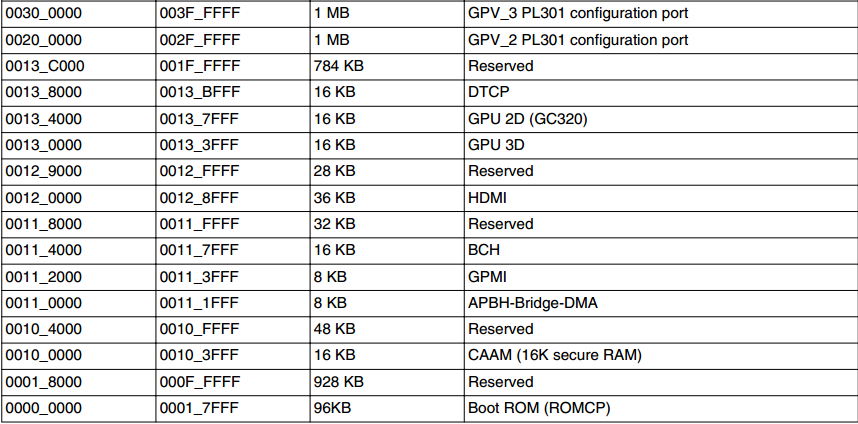


imx6q内部ROM以及RAM映射图

（boot strap：自举）







OCRAM为内部RAM,从SPI flash启动的时候，BOOT ROM会拷贝启动设备SPI FLASH中的前4K bytes数据到片内RAM(OCRAM)。这初始的4K数据中必须包含IVT(Image vector table)、DCD(Device configuration data)和Boot Data structures，但由于使用SPI FLASH启动时，IVT的偏移为1K (这1K通常内含分区表)，所以，剩余的3K数据中必须包含IVT、DCD和Boot Data structures。另外DCD最大长度限制为1768 bytes。

**IVT：**The Image Vector Table (IVT) is the data structure that the ROM reads from the boot device supplying the program image containing the required data omponents to perform a successful boot. **Google翻译：**图像矢量表（IVT）是ROM从引导设备读取的数据结构，提供包含所需数据组件的程序映像以执行成功的引导。

# IVT包含DCD的入口点和其他的入口点，给ROM使用。

# IVT在设备中的地址是固定的，这样ROM才能够找到。

**DCD：**开启启动的时候根据需要更改寄存器的默认值,方便一些外设一开机就配置

过程为：

**2、启动流程分析**

启动后，从0x0处开始执行，此时对应的程序就是固件中的BOOT ROM程序，BOOT ROM程序将完成以下功能：

1、BOOT ROM拷贝SPI FLASH 4K的数据到内部RAM(OCRAM)中去，这4K的空间包含1K的空余头(通常内含分区表)和内含IVT、DCD的数据块。

2、接着BOOT ROM检查OCRAM 4K数据中包含的IVT头标志0xD1，根据IVT中包含DCD地址定位的信息，找到DCD存储位置，然后执行DCD检查。

3、使用DCD中的配置参数，来初始化DDR，配置时钟，以及其他启动时必要的硬件设置。

4、从boot data structure 中解析出下一步拷贝的目的地址和长度（记为length），然后从SPI FLASH地址0，拷贝长度length的数据到上面解析出的目的地址处。

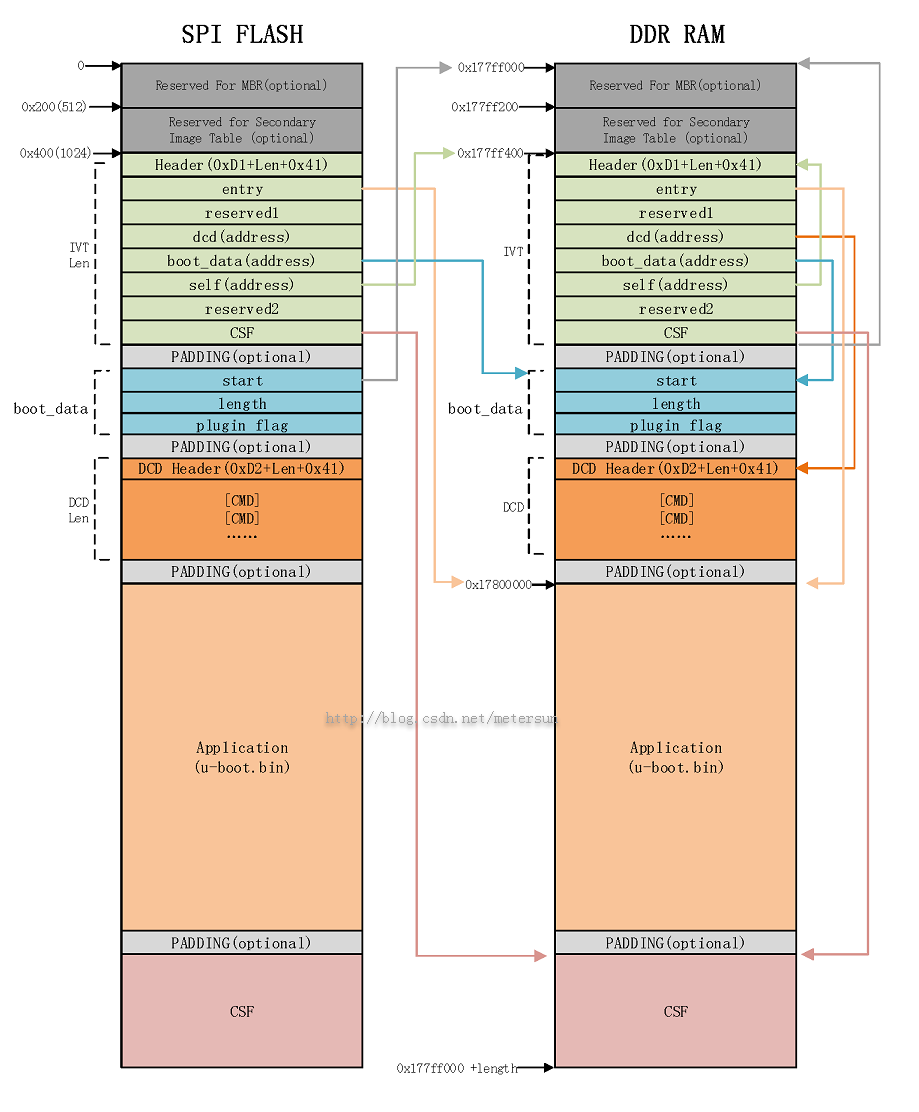
**注：**这里有一个如何在OCRAM中定位boot data structure地址的问题。我们编译生成的u-boot.imx，它的IVT中存储的boot data structure的地址，是一个绝对地址，实际为DDR地址，而OCRAM中此时的地址还是原始数据中的该地址。由于此时DDR还未准备好，还未执行u-boot.imx到DDR的拷贝，BOOT ROM不能使用IVT中的绝对(DDR地址)地址来定位boot data structure。而只能使用偏移地址在

OCRAM中定位boot data structure。   **计算方法：**IVT中还有一个self表项，它指向它自身，即IVT的首地址。比如hexdump出u-boot.imx的self的值为0x177ff400，IVT中boot data structure的地址值为0x177ff420，那么boot data structure的地址偏移为**boot data structure  offset = boot\_data - self(IVT首地址） = 0x20，**

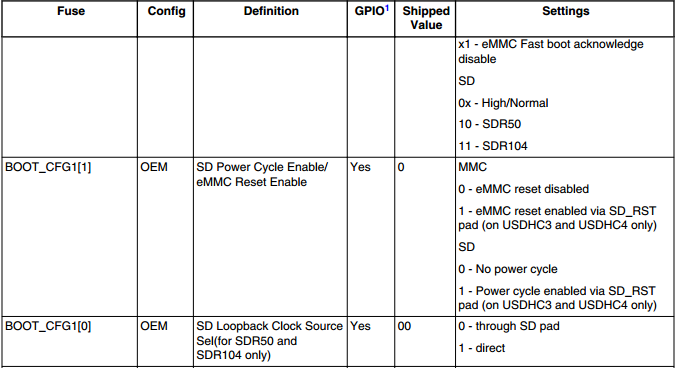
加载到OCRAM中后，boot data structure的有效地址为：OCRAM IVT 加载首地址 + boot data structure  offset，（关于如何生成前4k数据，以及如何对这部分数据进行修改参见“七、imx6的u-boot.imx和u-boot.bin”）

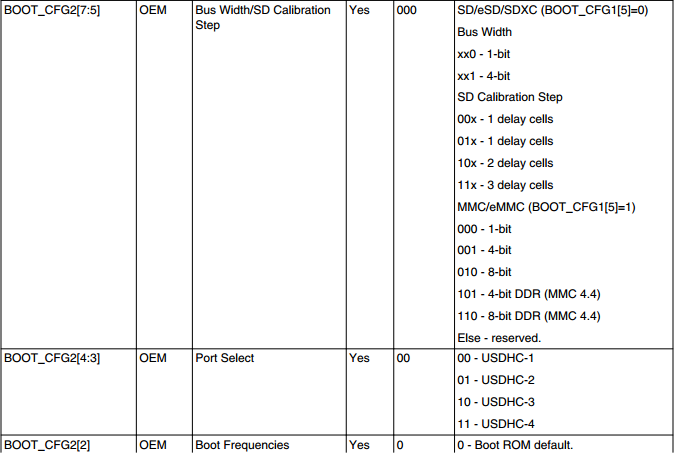
DCD数据的地址也是如此计算的。

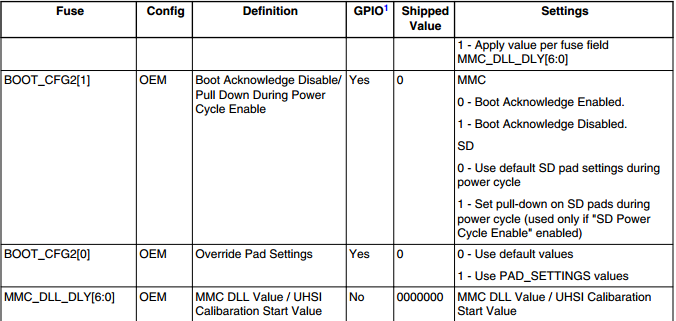
前4k代码执行完之后，u-boot.imx被全部拷贝到SDRAM中，，



从mmc/SD卡等启动方式都是从fuses启动，具体取决于寄存器BOOT\_CFG1[1]、BOOT\_CFG1[0]、BOOT\_CFG2[0]的设置。启动时上电执行内部ROM中的一段代码（起始地址0x00000000）,复制MMC/SD/eSD/SDXC/eMMC等设备中的前4k代码到RAM中。然后分析这4K代码，根据flash\_header 中的IVT(image vector table)结构体确定Uboot大小，初始化外设，再把Uboot Copy到DDR执行。







二、U-boot的前期启动流程分析

ARM U-boot的启动流程一般分为**两个部分**，

**第一部分**：初始化：初始化协处理器，看门狗，设置好栈，初始化外部设备，为u-boot执行命令做好准备。

**第二部分**：执行u-boot命令，将内核文件拷贝到RAM空间当中。为内核设置参数，然后调用内核。

可以认为前一部分和u-boot相关，后一部分和内核启动相关。本章介绍第一部分。

**imx6的ARM架构为cortex-A9,对应u-boot arch/arm/cpu/armv7目录**

**以下按照u-boot代码执行顺序依次分析文件**

1、start.S文件分析：在/u-boot-linux/u-boot-2015.04/arch/arm/cpu/armv7目录下面，这里将其拷贝到了“详细分析的文件”目录下，并将一些注释添加到里面。

总体上它实现的功能是，切换到管理模式并关闭irq,firq，设置中断向量表位置，关闭cache和mmu（通过设置cp15协处理器实现），设置临时栈。

下面详细分析：

(1)reset直接跳转到save\_boot\_params

(2)save\_boot\_params跳转到save\_boot\_params\_ret进行切换管理模式和设置异常向量表地址的操作。

(3) 跳转到cpu\_init\_cp15 初始化cp15寄存器，主要用来关闭cache、MMU、TLB等功能

(4) cpu\_init\_crit直接跳转到lowlevel\_init（该函数在arch/arm/cpu/armv7目录下的lowlevel\_init.s文件中）

(5) lowlevel\_init调用s\_init函数（./arch/arm/cpu/armv7/mx6/soc.c）用来设置某个外设的时钟使能。（感觉这个函数的实际意义不大）

注释:(3),(4),(5)由CONFIG\_SKIP\_LOWLEVEL\_INIT进行控制，如果定义了该宏，那么(3),(4),(5)均不发挥作用（这里应该没有定义这个宏）

(6)跳转到\_main函数执行（bl \_main）

2、crt0.S 文件分析(\_main函数在u-boot-2017.05\arch\arm\lib\crt0.S)

\_main在u-boot-2017.05\arch\arm\lib\crt0.S

**1**、设置堆栈，并在栈中预留出gd变量（gd变量是用来存储 **后面调用的函数** 的返回值的）的位置。这为调用board\_init\_f函数做了准备。

**注：**在栈中预留gd变量空间的原因是，此时还没有进行SDRAM的初始化，也就没有进行.data的拷贝和.bss的清零，所以程序中初始化和未初始化的全局变量都不可用，只有常量可用。所以要想存储数据，只能在当前可用的RAM中开辟空间（因为此时栈使用的是SRAM，不需要初始化）。

**注释：**有关gd的详细说明

gd全称是global data。由于u-boot在初始阶段可能在flash等不可写的位置处运行，所以不能使用.data数据段的全局变量。如果需要使用全局变量传递数据，可以使用global data方法。原理如下：

1）定义一个struct global\_data类型的数据结构，里面保存了各色各样需要传递的数据。数据结构定义在include/asm-generic/global\_data.h中，结构体一部分成员用于保存U-boot代码重定位后，在SDRAM中的地址。

relocaddr：保存代码重定位后U-boot在SDRAM中的地址

start\_addr\_sp：保存代码重定位之后栈的地址。

irq\_sp：保存代码重定位之后中断的栈的地址。

2）使用处理器不需要初始化的内存设置堆栈，堆栈配置好之后，在堆栈开始的位置，为struct global\_data预留空间，并将开始地址（就是一个struct global\_data指针）保存在一个寄存器中（ARM一般保存在r9寄存器中）。

3）在使用时通过 \_\_asm\_\_ volatile("mov %0, r9\n" : "=r" (gd\_ptr)); 获取

**2**、调用board\_init\_f函数。该函数进行硬件的初始化，由于SDRAM等可能并没有准备好，所以board\_init\_f函数将需要传递到下一阶段的数据存储在gd中。

下面的3,4,5对应的是non-spl模式下运行的（我们这里就是non-spl模式）

**3**、根据board\_init\_f的返回值（返回在gd中），重新设置堆栈，然后跳转到relocate\_code（relocate\_code这个函数将u-boot从当前位置拷贝到board\_init\_f函数计算的目的地址处）

**4**、最后完成所有的环境设置。一些CPU此时还需要设置memory那么将调用c\_runtime\_cpu\_setup，然后进行清.bss的操作，memset，coloured\_LED\_init, red\_led\_on这部分函数暂不深入研究。

**5**、跳转到board\_init\_r()执行

这里的初始化分成了两个阶段：board\_init\_f函数负责初始化一些后面急需用到的功能，如时钟，串口（用于打印信息），内存。board\_init\_r函数则用于初始化一些不是急需用到的，比如USB，MMC等外设的初始化。

2.2、函数board\_init\_f

board\_f.c文件分析（board\_init\_f(ulong boot\_flags)函数./common/board\_f.c）

u-boot-2017.05\common\board\_f.c 我们主要分析board\_init\_f(ulong boot\_flags)

**1**、初始化gd，该函数内设置了flags和have\_console

**2**、调用initcall\_run\_list(init\_sequence\_f)。init\_sequence\_f是一个函数数组，定义了很多和初始化相关的函数（一系列后半部分的板级初始化函数，这些函数都在./common/board\_f.c中定义，他们多数设置gd的参数，关于gd参数的解释参见include/asm-generic/global\_data.）。

2.2.2、initcall\_run\_list函数：路径./lib/initcall.c

initcall\_run\_list(init\_sequence\_r)依次执行init\_sequence\_r中的函数（一系列后半部分的板级初始化函数，这些函数多数都在./common/board\_f.c中定义，他们多数设置gd的参数，关于gd参数的解释参见include/asm-generic/global\_data.），h并最终调用run\_main\_loop函数 -> main\_loop函数后不再返回，从而完成u-boot第一阶段的启动过程。

1)、函数setup\_mon\_len

将u-boot的长度赋值给gd->mon\_len这和u-boot的拷贝有关（拷贝又称为重定位）。

**拷贝u-boot**还涉及的函数有dram\_init，setup\_dest\_addr，reserve\_x（x代表需要拷贝到RAM中的某一项，比如reserve\_video，reserve\_uboot，reserve\_global\_data等等）。有关的gd中的变量有ram\_top（指定u-boot使用的RAM的上限地址），relocaddr（指定u-boot在RAM中的开始地址），mon\_len（指明u-boot的长度）。

**过程：**先调用dram\_init，setup\_mon\_len，setup\_dest\_addr初始化RAM和相应的gd变量此时将relocaddr初始化为ram\_top的值，然后调用reserve\_x依次减去x需要的ram大小。最后调用拷贝函数进行拷贝操作。

2)、setup\_fdt函数

获取设备树的地址。

3)、initf\_malloc函数

设置gd关于malloc的值

4)、arch\_cpu\_init

该函数直接返回0

5)、board\_early\_init\_f，对于freescale imx6的板子，在./board/freescale/mx6sabresd/mx6sabresd.c文件

/\*不同的单板要进行不同的设置这里主要是初始化串口和显示 \*/

1、函数调用了setup\_iomux\_uart();

2、如果定义了CONFIG\_VIDEO\_IPUV3，函数还将调用setup\_display();

static void setup\_iomux\_uart(void)

{

imx\_iomux\_v3\_setup\_multiple\_pads(uart1\_pads, ARRAY\_SIZE(uart1\_pads));

}

static iomux\_v3\_cfg\_t const uart1\_pads[] = {

MX6\_PAD\_CSI0\_DAT10\_\_UART1\_TX\_DATA | MUX\_PAD\_CTRL(UART\_PAD\_CTRL),

MX6\_PAD\_CSI0\_DAT11\_\_UART1\_RX\_DATA | MUX\_PAD\_CTRL(UART\_PAD\_CTRL),

};

函数imx\_iomux\_v3\_setup\_multiple\_pads路径为：./arch/arm/imx-common/iomux-v3.c，使用UART1作为通用异步收发器，设置UART1的tx,rx所用引脚的寄存器。

6)、 get\_clocks函数./arch/arm/imx-common/speed.c

7)、timer\_init将定时器相关的gd值存储到gd中，特定于架构

8)、env\_init函数./common/env\_mmc.c,这里以mmc启动方式为例，也可以从网络文件系统启动。初始化gd与环境变量有关的内容。环境变量的说明，参见单独的章节。

9)、init\_baud\_rate初始化速率函数./common/board\_f.c

static int init\_baud\_rate(void)

{

/\*CONFIG\_BAUDRATE就是传输速率的值\*/

/\* getenv\_ulong 读取环境变量baudrate的值，如果没有设置该环境变量，那么使用CONFIG\_BAUDRATE 作为默认值，10表示CONFIG\_BAUDRATE 的基数是10，就是按照十进制解释CONFIG\_BAUDRATE \*/

gd->baudrate = getenv\_ulong("baudrate", 10, CONFIG\_BAUDRATE);

return 0;

}

ulong getenv\_ulong(const char \*name, int base, ulong default\_val)

{ /\* 路径./common/cmd\_nvedit.c\*/

/\*

\* We can use getenv() here, even before relocation, since the

\* environment variable value is an integer and thus short.

\*/

/\*调用getenv,获取baudrate这个变量的存储位置，位于gd结构体中，将地址赋值给str变量\*/

const char \*str = getenv(name);

return str ? simple\_strtoul(str, NULL, base) : default\_val;

}

char \*getenv(const char \*name)

{

if (gd->flags & GD\_FLG\_ENV\_READY) { /\* after import into hashtable \*/

ENTRY e, \*ep;

WATCHDOG\_RESET();

e.key = name;

e.data = NULL;

hsearch\_r(e, FIND, &ep, &env\_htab, 0);

return ep ? ep->data : NULL;

}

/\* restricted capabilities before import \*/

if (getenv\_f(name, (char \*)(gd->env\_buf), sizeof(gd->env\_buf)) > 0)

return (char \*)(gd->env\_buf);

return NULL;

}

getenv\_f函数：传入参数的名字name，使用gd->env\_buf来保存变量的存储地址。

10)、serial\_init串口初始化函数./drivers/serial/serial.c

11)、console\_init\_f console初始化函数./common/console.c

设置gd->have\_console=1

实际上11)和12)之间，还有一些信息打印函数，要想去除可以进行在此将其屏蔽掉

处理信息打印还有misc\_init\_f，init\_func\_i2c，init\_func\_spi模块的初始化，这些初始化时通过./include/configs/myimx.h中的宏定义进行配置的。

12)、dram\_init

DRAM初始化

/\* 配置可用的RAM大小，即设置 gd->ram\_size \*/

13)、setup\_dest\_addr函数

设置gd\_ram\_top。gd\_ram\_top的设置过程比较复杂，应该是最终定位到板子可用RAM的顶端，过程如下

/\* u-boot2015.04\include\configs\myimx.h \*/

#define CONFIG\_SYS\_SDRAM\_BASE PHYS\_SDRAM

#define PHYS\_SDRAM MMDC0\_ARB\_BASE\_ADDR

/\* u-boot2015.04\arch\arm\include\asm\arch-mx6\imx-regs.h \*/

#define MMDC0\_ARB\_BASE\_ADDR 0x10000000

gd->ram\_top= CONFIG\_SYS\_SDRAM\_BASE=0x10000000

**setup\_dest\_addr //上面是基址，这个函数在基址的基础上增加了RAM的大小，得到ram\_top**

gd->ram\_top=0x50000000

gd->relocaddr=0x50000000

**gd\_ram\_top的值在本函数中被赋给gd->relocaddr。**

14)、reserve\_x（x代表需要拷贝到RAM中的某一项，比如reserve\_video，reserve\_uboot，reserve\_global\_data等等）这些函数在gd->relocaddr的基础上减去x需要的大小，最终获得u-boot的实际重定位地址。（这一系列的函数都和u-boot的拷贝有关，参照1)setup\_mon\_len中的相关叙述。）

这些函数确定最终地址的过程如下：

**reserve\_mmu**

**start**

gd->arch.tlb\_size=0x10000

gd->relocaddr=4fff0000

**reserve\_trace**

**start**

gd->relocaddr=4fff0000

**reserve\_uboot**

**start**

gd->start\_addr\_sp=4ff7b000

gd->relocaddr=4ff7b000

**reserve\_malloc**

**start**

gd->start\_addr\_sp=4fb79000

TOTAL\_MALLOC\_LEN=402000

**reserve\_board**

**start**

gd->start\_addr\_sp=4fb78fb0

sizeof(bd\_t)=50

**setup\_machine**

**start**

gd->bd->bi\_arch\_number=7462

**reserve\_global\_data**

**start**

gd->start\_addr\_sp =4fb78eb8

gd->new\_gd =4fb78eb8

**reserve\_fdt**

**start**

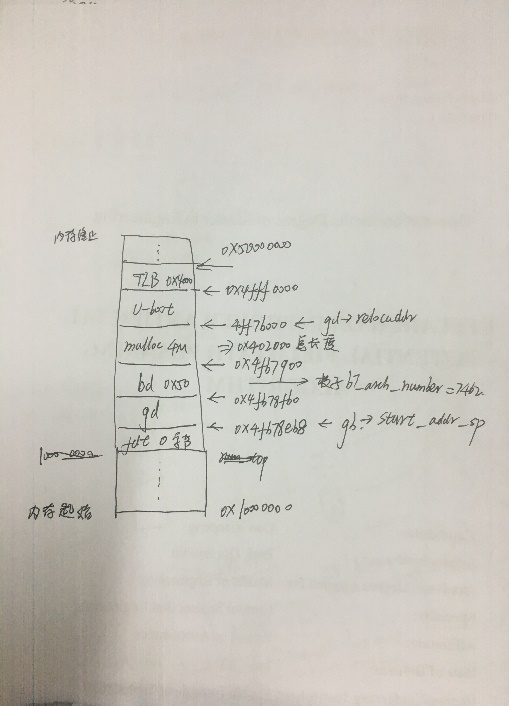
gd->start\_addr\_sp =4fb78eb8

gd->new\_fdt =0

**reserve\_stacks**

**start**

gd->start\_addr\_sp =4fb78ea0



最终的内存规划图

2.3、relocate\_code函数

在调用重定位函数前，将函数的传入参数存储在r0中，将函数的返回地址存储在lr寄存器中，然后调用函数。

relocate\_code函数代码位于./arch/arm/lib/relocate.S中，功能是1、将\_\_image\_copy\_start到\_\_image\_copy\_end之间的代码拷贝到RAM中。2、拷贝.rel.dyn。3、返回到u-boot-2017.05\arch\arm\lib\crt0.S的here处

2.4、board\_init\_r函数

board\_init\_r在u-boot2015/common/board\_r.c

该函数传入参数为gd和u-boot的重定位地址。功能为调用initcall\_run\_list函数运行init\_sequence\_r函数指针数组中的函数。

2.4.1、initcall\_run\_list函数：

功能为运行init\_sequence\_r函数指针数组中的函数

init\_sequence\_r函数指针数组中包含的函数：

1) initr\_trace 如果定义了CONFIG\_TRACE则初始化并使能u-boot的tracing system；如果没有定义则直接return 0，无作用。

2) initr\_reloc 设置relocation完成的标志，设置gd->flags，表示代码已经重定位并且malloc区域已经初始化了，所以malloc函数已经可以使用了。

3) initr\_reloc\_global\_data函数

4) initr\_barrier函数

5) initr\_malloc       malloc有关的初始化

6) bootstage\_relocate函数

7) initr\_dm           relocate之后，重新初始化DM

8) stdio\_init\_tables

9) initr\_serial         重新初始化串口

10) initr\_announce     宣布已经在RAM中执行

11) power\_init\_board

12) initr\_mmc

13) initr\_env              环境变量有关的初始化。

14) initr\_secondary\_cpu

15) initr\_pci

16) stdio\_add\_devices

各种输入输出设备的初始化，根据宏定义控制，这里只初始化了i2c总线，以及显示显示u-boot启动的logo。

17) initr\_jumptable。

18) console\_init\_r

19) interrupt\_init

20) initr\_ethaddr 在不使用网络的时候可以裁剪掉这个函数

21) board\_late\_init         由板级代码实现

22) initr\_net

初始化网卡，如果u-boot不带网启动，则可以裁掉这个函数

去掉这个函数，就只能从本地mmc启动，不能挂载网文系统。并且ping命令不好使。

23) run\_main\_loop         执行到main\_loop

2.4.1.1 run\_main\_loop函数,

该函数的作用就是循环调用main\_loop函数。run\_main\_loop是永远都不返回的函数。

**main\_loop函数**：位于u-boot2015.04/common/main.c

该函数主要进行u-boot的命令的解析和运行。

下面对main\_loop调用的函数进行解析

1)bootstage\_mark\_name保存u-boot启动的标记；

2)setenv设置环境变量ver为定义的一个值（并不重要）;

3) cli\_init：我们这里定义了CONFIG\_SYS\_HUSH\_PARSER，即配置u-boot使用hush shell来作为执行器。hush shell是一种轻量型的shell。cli\_init用来初始化hush shell使用的一些变量。hush shell的实现机制比较复杂，以下的hush shell相关实现代码都不做详尽跟踪分析

4) run\_preboot\_environment\_command函数从环境变量中获取"preboot"的定义，该变量包含了一些预启动命令，一般环境变量中不包含该项配置。这里并没有这项配置，所以run\_preboot\_environment\_command为空。

5) bootdelay\_process从环境变量中取出"bootdelay"和"bootcmd"的配置值，将取出的"bootdelay"配置值转换成整数，赋值给全局变量stored\_bootdelay，并"bootcmd"的配置值作为返回值返回，如下：

s =bootdelay\_process();

然后函数将执行

autoboot\_command(s);

6) cli\_process\_fdt：由于没有定义CONFIG\_OF\_CONTROL，函数cli\_process\_fdt返回false

7) autoboot\_command，该函数在common/autoboot.c中实现：

void autoboot\_command(const char \*s)

{

if (stored\_bootdelay != -1 && s && !abortboot(stored\_bootdelay)) {

run\_command\_list(s, -1, 0);

}

}

全局变量stored\_bootdelay在上面已做说明。静态函数abortboot中包含了CONFIG\_AUTOBOOT\_KEYED宏预处理分支，该宏定义用来使能用户名密码登录，这里它没有定义，而后调用了abortboot\_normal，在执行的时间stored\_bootdelay(秒)内，如无用户按键输入干预，那么abortboot\_normal函数将返回0，否则返回1。

当无用户按键干预时，接下来将调用run\_command\_list执行上述从环境变量中读取的"bootcmd"（关于bootcmd的具体值参见环境变量中的分析）配置值。注意该函数的参数s。run\_command\_list中调用了hush shell的命令解释器(parse\_stream\_outer函数)，解释bootcmd中的启动命令。环境变量bootcmd中的启动命令，用来设置linux必要的启动环境，然后加载和启动linux内核。u-boot启动linux内核后，将控制权交给linux内核，至此不再返回。

如用户在设定的bootdelay内有按键输入或者没有bootcmd，那么将退出autoboot\_command，然后运行cli\_loop执行hush shell命令解释器：

8) cli\_loop函数：

void cli\_loop(void)

{

parse\_file\_outer();

/\* This point is never reached \*/

for (;;);

}

parse\_file\_outer进行必要的初始化后，将调用hush shell的命令解释器。

run\_command\_list和parse\_file\_outer主要为命令解析过程，参照下面的“**u-boot命令章节**”。

函数前期执行流程总结：

（1）\_start（arch/arm/lib/vector.S）

              b     reset

（2）reset（arch/arm/cpu/armv7/start.S）

              bl    cpu\_init\_cp15

              bl    cpu\_init\_crit

              bl    \_main

（3）\_main（arch\arm\lib\crt0.S）

              board\_init\_f(common\Board\_f.c）

              b     relocate\_code

              ldr   lr, =board\_init\_r（common/Board\_r.c）

                     run\_main\_loop（）

                            main\_loop()

（4）main\_loop（）

（5）命令解析：  <https://blog.csdn.net/fengyuwuzu0519?t=1>

main\_loop->autoboot\_command->run\_command\_list->cli\_simple\_run\_command\_list->cli\_simple\_run\_command-> cmd\_process-> find\_cmd/ cmd\_call(result= (cmdtp->cmd)(cmdtp, flag, argc, argv))

cmd\_call会执行nand read和bootm函数，

注释：环境变量

u-boo会定义一个掉电不失的设备来存储环境变量，如Flash，MMC等，具体使用哪一个由u-boot-2017.05/include/configs/目录底下的.h文件中（不同的板子使用不同的.h文件，我们这里myimx6.h）的宏决定，

#define CONFIG\_ENV\_IS\_IN\_FLASH 1

#define CONFIG\_ENV\_OFFSET 0X40000

#define CONFIG\_ENV\_SIZE 0x20000

分别决定存储器，偏移地址和环境变量大小。上电后u-boot会先到对应存储器中寻找环境变量，如果有就使用存储器中的，如果没有就使用**默认的环境变量**。

u-boot的**默认环境变量**全部保存在u-boot-2017.05/include/configs/目录底下的.h文件中。这些环境变量都是一些字符串，如

#define CONFIG\_CONSOLE\_DEV "ttymxc0"

"console=" CONFIG\_CONSOLE\_DEV "\0"

这些字符串根据自己的类别，分别被定义成宏 CONFIG\_MFG\_ENV\_SETTINGS

CONFIG\_EXTRA\_ENV\_SETTINGS

CONFIG\_BOOTCOMMAND

在u-boot中和环境变量有关的操作有下面这些：

a、环境变量初始化env\_init：

u-boot-2017.05\common\board\_f.c中的env\_init函数，环境变量的首地址赋值给gd中的对应项。

int env\_init(void)

{

/\* use default \*/

gd->env\_addr = (ulong)&default\_environment[0];

gd->env\_valid = 1;

return 0;

}

default\_environment数组定义在./include/env\_default.h文件当中。它将u-boot-2017.05/include/configs/myimx.h文件的环境变量宏整合起来。代码如下

const uchar default\_environment[] = {

"bootcmd=" CONFIG\_BOOTCOMMAND "\0"

"bootdelay=" \_\_stringify(CONFIG\_BOOTDELAY) "\0"

"baudrate=" \_\_stringify(CONFIG\_BAUDRATE) "\0"

"ethaddr=" \_\_stringify(CONFIG\_ETHADDR) "\0"

"ethprime=" CONFIG\_ETHPRIME "\0"

"ipaddr=" \_\_stringify(CONFIG\_IPADDR) "\0"

"serverip=" \_\_stringify(CONFIG\_SERVERIP) "\0"

"loadaddr=" \_\_stringify(CONFIG\_LOADADDR) "\0"

CONFIG\_EXTRA\_ENV\_SETTINGS

} //可能有些出入，不具体分析了

其中各个宏，比如 CONFIG\_BOOTCOMMAND，CONFIG\_BOOTDELAY定义在u-boot-2017.05/include/configs/myimx.h文件中。

b、环境变量初始化initr\_env：

该函数位于u-boot2015/common/board\_r.c文件中。功能：1、进行环境变量重定位；2、获取load\_addr的值（loadaddr是内核的加载地址）。

1、重定位

imx6q的环境变量存储于SPI flash（myimx.h中通过定义宏，说明了使用SPI Flash）中，为了方便读取，可以重定位到RAM当中，然后使用结构体gd来保存这些环境变量。initr\_env 中重定位相关函数介绍如下：

**1、should\_load\_env**

是board\_r.c中的静态函数，经过编译预处理，直接返回1。接着执行下一个函数env\_relocate()。

**2、env\_relocate();**

位于u-boot-2015.04\common\ env\_common.c

就是环境变量重定位函数，它的作用就是将环境变量 **从存储设备比如spi flash 或者 在未定义存储设备或者指定存储设备中没有环境变量时，从默认的环境变量数组default\_environment中** 重定位到内存中的哈希表（哈希表是为了检索方便）当中。哈希表在内存中的存储位置为：env\_htab=4ffb4c04（在u-boot的代码范围当中4ff7b000-4fff0000，因此环境变量在u-boot的代码范围内）

对于imx6q，env\_relocate()直接调用env\_relocate\_spec();

**2.1、env\_relocate\_spec()**

函数实现位于common/Env\_sf.c中的（原因是：common/Makefile中有语句

obj-$(CONFIG\_ENV\_IS\_IN\_SPI\_FLASH) += env\_sf.o）。

env\_relocate\_spec用来对环境变量重定位，环境实际存储位置由myimx6.h中的宏确定，我们这里定义了

#define CONFIG\_ENV\_IS\_IN\_SPI\_FLASH //说明环境变量存储在SPI Flash中；#define CONFIG\_ENV\_OFFSET (768 \* 1024) //说明了env在SPI Flash 中的位置；

#define CONFIG\_ENV\_SIZE (8 \* 1024) //环境变量大小

函数功能：从SPI中拷贝环境变量，并进行CRC校验。如果其中任何一个步骤出错，则调用set\_default\_env(“reason”); set\_default\_env函数的传入参数为出错原因。

**2.1.1、set\_default\_env函数**

set\_default\_env函数作用是将default\_environment设为哈希表。它在common/env\_common.c中实现。其**核心语句为**

**(himport\_r(&env\_htab, (char \*)default\_environment,\ sizeof(default\_environment), '\0', flags, 0,\**

**0, NULL) == 0)**

default\_environment即为默认的环境变量，在include/Env\_default.h中定义的变量（在上面的env\_init的解析中详细分析过）。

**另外注意**，set\_default\_env函数中，gd->flags最后被赋值为

**gd->flags |= GD\_FLG\_ENV\_READY;**

而从SPI Flash中读取的环境变量，在使用后续函数env\_import时，该函数在返回前也会进行同样的赋值操作。**该标志代表环境变量已准备好，可以对其进行打印、编辑操作。**

环境变量的存储格式：

typedef struct environment\_s {

uint32\_t crc; /\* CRC32 over data bytes \*/

#ifdef CONFIG\_SYS\_REDUNDAND\_ENVIRONMENT

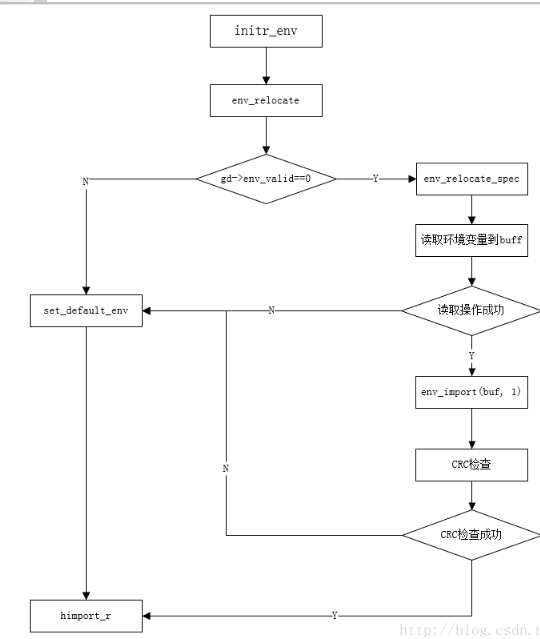
unsigned char flags; /\* active/obsolete flags \*/

#endif

unsigned char data[ENV\_SIZE]; /\* Environment data \*/

} env\_t

crc表示整个环境参数表的校验值，如果不往spi flash写数据，则crc=0xffffffff；往spi flash写入环境变量数据之后，crc被写到spi flash中，这是crc=实际校验值。则下次启动的时候会使用spi flash中的环境变量的值。



这里gd->env\_valid = 1.

总结

无论从默认的数组中读取环境变量的设置，还是从mmc中读取环境变量，最后都要将环境变量写到哈希表当中。第一次开机的时候，没有将环境变量写到mmc当中，因此默认是将默认数组中的环境变量写到哈希表当中。如果将环境变量写入到了mmc当中，下一次开机的时候就会从mmc中读取环境变量的值到哈希表当中。

之后使用printenv，setenv等操作都是对哈希表进行操作，saveenv可以将哈希表的内容存储到mmc当中。

三、u-boot启动内核bootz

这里有些板子选择使用bootz启动内核，有些板子使用bootm启动。由于前期没有找到imx6q使用哪个，所以分析bootm，后找到参考资料确定为bootz，因此将bootz命令的分析放在前面，之前分析bootm的留作参考。

注：**如何确定使用的是bootz**

在env\_default.h头文件中：

#ifdef CONFIG\_BOOTCOMMAND

"bootcmd=" CONFIG\_BOOTCOMMAND "\0"

参数bootcmd就等于我们的启动命令。

**CONFIG\_BOOTCOMMAND**在myimx6.h文件中使用宏定义进行说明：

#define CONFIG\_BOOTCOMMAND \

"mmc dev ${mmcdev}; " \

**/\***

**"mmcdev=" \_\_stringify(CONFIG\_SYS\_MMC\_ENV\_DEV) "\0" 宏为1**

**\*/**

"if run loadimage; then " \

**/\***

**"loadimage=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${image\_file}\0"**

**"mmcdev=" \_\_stringify(CONFIG\_SYS\_MMC\_ENV\_DEV) "\0" 宏为1**

**"mmcpart=" \_\_stringify(CONFIG\_SYS\_MMC\_IMG\_LOAD\_PART) "\0"宏为1**

**"image\_file=zImage-myimx6\0"**

**\*/**

"run **mmcboot**; " \

"else run netboot; " \

"fi; " \

"**mmcboot**=echo Booting from mmc ...; " \

"run mmcargs; " \ //run set\_disp **//setenv disp\_args ${display} //"display=\ \0"**

"if run loadfdt; then " \

**/\***

**"loadfdt=fatload mmc ${mmcdev}:${mmcpart} ${fdt\_addr} ${fdt\_file}\0"**

**"mmcdev=" \_\_stringify(CONFIG\_SYS\_MMC\_ENV\_DEV) "\0" 宏为1**

**"mmcpart=" \_\_stringify(CONFIG\_SYS\_MMC\_IMG\_LOAD\_PART) "\0"宏为1**

**#define CONFIG\_LOADADDR 0x12000000**

**#define CONFIG\_SYS\_LOAD\_ADDR CONFIG\_LOADADDR "fdt\_addr=0x18000000\0"**

**"fdt\_file="CONFIG\_BOARD\_NAME"-"EK\_SPEC".dtb\0" 宏为"myimx6ek200" "6qp"**

**\*/**

"bootz ${loadaddr} - ${fdt\_addr}; " \

**/\***

**bootz是命令，后面的${loadaddr} - ${fdt\_addr}是三个参数“-”占位也表示参数**

**\*/**

"else " \

"echo WARN: Cannot boot from mmc; " \

"fi;\0" \

**通过上述分析可以确定使用的是bootz指令启动内核，下面分析bootz的定义及其实现的功能。**

**定义：**

在u-boot-2015.04\include\configs\myimx6.h中我们定义了

#define CONFIG\_CMD\_BOOTZ

这个宏控制了bootz这个U\_BOOT\_CMD在u-boot2015.0/common/cmd\_bootm.c中的编译（关于u-boot命令是如何定义的参见“u-boot的命令”章节）。

在cmd\_bootm.c文件中有：

#ifdef CONFIG\_CMD\_BOOTZ

… … …

… … …

U\_BOOT\_CMD(

bootz, CONFIG\_SYS\_MAXARGS, 1, do\_bootz,

"boot Linux zImage image from memory", bootz\_help\_text

);

… … …

… … …

#endif

**功能分析：**

而根据myimx.h中关于mmcboot中的启动指令

bootz ${loadaddr} - ${fdt\_addr};

可知，传递给do\_bootz的参数就是loadaddr；“-”表示这个参数缺省，做占位用；fdt\_addr表示设备树的位置，根据关于fdt\_addr的定义可知，fdt\_addr=0x18000000。

1、do\_bootz

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

struct cmd\_tbl\_s {

char \*name; /\* Command Name \*/

int maxargs; /\* maximum number of arguments \*/

int repeatable; /\* autorepeat allowed? \*/

int (\*cmd)(struct cmd\_tbl\_s \*, int, int, char \*[]); /\* Implementation function \*/

char \*usage; /\* Usage message (short) \*/

#ifdef CFG\_LONGHELP

char \*help; /\* Help message (long) \*/

#endif

#ifdef CONFIG\_AUTO\_COMPLETE

/\* do auto completion on the arguments \*/

int (\*complete)(int argc, char \*argv[], char last\_char, int maxv, char \*cmdv[]);

#endif

};

**do\_bootz参数：**

cmdtp传递的是bootm的命令表项指针，保存着命令在定义时的性质，参见“u-boot的命令”章节、

argc=3

argv[0]="bootm",

argv[1]= loadaddr（#define CONFIG\_LOADADDR 0x12000000）,

arv2]= “-“ //这里表示没有需要单独加载的ramdisk

arv[3]= fdt\_addr（"fdt\_addr=0x18000000\0" \）,

do\_bootz的参数会在其函数调用的过程中，逐渐被其调用的函数使用，其中bootm\_find\_os使用loadaddr，bootm\_find\_other调用的boot\_find\_fdt使用fdt\_addr。

下面按照顺序依次分析do\_bootz函数中的重要代码：

**1、bootz\_start**

if (bootz\_start(cmdtp, flag, argc, argv, &images))

return 1;

**2、bootm\_disable\_interrupts**

\* We are doing the BOOTM\_STATE\_LOADOS state ourselves, so must

\* disable interrupts ourselves

\*/

bootm\_disable\_interrupts();

**3、**

// images.os.os=5，IH\_OS\_LINUX为linux操作系统的代码，指明操作系统为linux

images.os.os = IH\_OS\_LINUX;

**4、do\_bootm\_states**

ret = do\_bootm\_states(cmdtp, flag, argc, argv,

BOOTM\_STATE\_OS\_PREP | BOOTM\_STATE\_OS\_FAKE\_GO |

BOOTM\_STATE\_OS\_GO,

&images, 1);

下面详细介绍do\_bootz调用的函数

**1.1、bootz\_start**

do\_bootz函数先调用bootz\_start函数进行一些初始化操作

1、调用do\_bootm\_states函数，**状态为BOOTM\_STATE\_START**

2、 ret = bootz\_setup(images->ep, &zi\_start, &zi\_end);

3、if (bootm\_find\_ramdisk\_fdt(flag, argc, argv))

return 1;

**1.1.1、do\_bootm\_states函数：**

参见**u-boot启动内核bootm**目录下的解析。这里和1.2调用的函数以及**u-boot启动内核bootm**目录下函数是同一个，但是参数不同。所以，虽然函数相同，但是完成的功能并不相同。

这里的参数为 BOOTM\_STATE\_START ，因此只调用bootm\_start函数**bootm\_start**主要工作是清空images（一个全局变量），**标志当前状态**为bootm start。

**1.1.2、bootz\_setup函数：**

根据images->ep（这里表示内核的入口点，相关叙述参见 “设备树向内核传递信息”）bootz\_setup函数读取内核入口点的数据结构，获取开始和结束地址。

int bootz\_setup(ulong image, ulong \*start, ulong \*end)

{

struct zimage\_header \*zi;

zi = (struct zimage\_header \*)map\_sysmem(image, 0);

if (zi->zi\_magic != LINUX\_ARM\_ZIMAGE\_MAGIC) {

puts("Bad Linux ARM zImage magic!\n");

return 1;

}

\*start = zi->zi\_start;

\*end = zi->zi\_end;

printf("Kernel image @ %#08lx [ %#08lx - %#08lx ]\n", image, \*start,

\*end);

return 0;

}

**1.1.3、bootm\_find\_ramdisk\_fdt(函数：**

查找ramdisk和fdt设备文件，将他们的地址和长度信息填入images。

int bootm\_find\_ramdisk\_fdt(int flag, int argc, char \* const argv[])

{

if (bootm\_find\_ramdisk(flag, argc, argv))

return 1;

#if defined(CONFIG\_OF\_LIBFDT)

if (bootm\_find\_fdt(flag, argc, argv))

return 1;

#endif

return 0;

}

**1.2、do\_bootm\_states函数：**

调用指令为：

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

BOOTM\_STATE\_OS\_PREP | BOOTM\_STATE\_OS\_FAKE\_GO |

BOOTM\_STATE\_OS\_GO,

&images, 1);

这个函数最主要的功能就是获取内核启动函数，然后再跳转到内核启动函数，相当于调用**do\_bootm\_linux（分析参照“do\_bootm”下的函数分析）。**

**bootz启动内核总结：**

获取内核位置，ramdisk位置，fdt位值。然后调用do\_bootm\_linux，并在其中完成最后的跳转。

**三、u-boot启动内核bootm**

uboot下使用bootm命令启动内核镜像文件uImage，uImage是在zImage头添加了64字节的镜像信息供uboot解析使用

调用流程如下：

<https://blog.csdn.net/skyflying2012/article/details/35787971>

bootm会调用到do-bootm，这里直接从do\_bootm。开始分析

**1、do\_bootm**

do\_bootm在u-boot2015.04/common/cmd\_bootm.c文件中

do\_bootm->do\_bootm\_states（设置内核映像入口点）其指令为： do\_bootm\_states(cmdtp, flag, argc, argv, BOOTM\_STATE\_START | BOOTM\_STATE\_FINDOS | BOOTM\_STATE\_FINDOTHER | BOOTM\_STATE\_LOADOS |

BOOTM\_STATE\_OS\_PREP | BOOTM\_STATE\_OS\_FAKE\_GO |

BOOTM\_STATE\_OS\_GO, &images, 1); //这些宏控制着do\_bootm\_states的函数调用，它调用函数如下)

**1.1、do\_bootm\_states**

下面的函数在do\_bootm\_states函数中被依次调用

**1.1.1、bootm\_disable\_interrupts**

**1.1.2、bootm\_start**

**bootm\_start**主要工作是清空images（一个全局变量），**标志当前状态**为bootm start。

**1.1.3、bootm\_find\_os** //1、调用boot\_get\_kernel，函数较长，首先是获取image的load地址（如果bootm函数有参数，image的加载地址就是loadaddr）。2、**标志当前状态**3、调用genimg\_get\_image，该函数会检查当前的img\_addr是否在sdram中，如果是在flash中，则拷贝到sdram中CONFIG\_SYS\_LOAD\_ADDR处，修改img\_addr为该地址。4、genimg\_get\_format以buf为参数（buf = map\_sysmem(img\_addr, 0);）（实际上map\_sysmem为空函数，buf即为img\_addr）检查img头的数据结构img header的头4个字节，确定image的类型，有2种，legacy和FIT，这里使用的legacy，头4个字节为0x27051956。然后程序将根据image的类型进入到不同的分支。5、image\_get\_kernel则会来计算header的crc是否正确。6、然后image\_get\_type获取image的type，根据type来获取os的len和data起始地址。7、执行一些指令完成：最后将hdr的数据拷贝到images的legacy\_hdr\_os\_copy，防止kernel image在解压是覆盖掉hdr数据，保存hdr指针到legacy\_hdr\_os中，置位legacy\_hdr\_valid。

bootm\_find\_os总结：到这里bootm\_find\_os就结束了，主要工作是根据image的hdr来做crc，获取一些基本的os信息到images结构体中

**1.1.4、bootm\_find\_other**

该函数大体看一下，主要函数为bootm\_find\_ramdisk\_fdt，该函数又调用bootm\_find\_ramdisk以及bootm\_find\_fdt。bootm\_find\_ramdisk对于legacy类型的image，获取查询是否有ramdisk，此处我们没有用单独的ramdisk，ramdisk是直接编译到kernel image中的。boot\_get\_fdt根据fdt\_addr参数得到DTB镜像后，进行DTB镜像校验，在确认DTB镜像无误之后，会将该地址保存在环境变量“fdtaddr”中。接着，uboot会把DTB镜像reload一次，使得DTB镜像所在的物理内存归lmb所管理。**注**：**boot\_get\_fdt是boot\_find\_fdt调用的函数，而boot\_find\_fdt用bootm\_find\_other调用。**

**1.1.5、bootm\_load\_os**

(加载内核到指定位置上)：load\_buf是之前find\_os是根据hdr获取的load addr，image\_buf是find\_os获取的image的开始地址（去掉64字节头）。之后则是根据hdr的comp类型来解压拷贝image到load addr上。

**1.1.6、bootm\_os\_get\_boot\_func**（获取到对应操作系统的启动函数，被存储到boot\_fn 中）

对应指令boot\_fn = bootm\_os\_get\_boot\_func(images->os.os);

则boot\_fn中存储着启动函数，实际上bootm\_os\_get\_boot\_func的返回值为do\_bootm\_linux）

（bootm\_os\_get\_boot\_func通过查询u-boot2015.04/common/bootm\_os.c中的boot\_os函数数组得到do\_bootm\_linux）

**1.1.7、do\_bootm\_linux/ boot\_fn**

boot\_fn(BOOTM\_STATE\_OS\_PREP, argc, argv, images); 等价于do\_bootm\_linux(BOOTM\_STATE\_OS\_PREP, argc, argv, images);//这个函数应该是一个不返回的函数

**下面分析do\_bootm\_linux在arch/arm/lib/bootm.c中**

do\_bootm\_linux->（下面的两个函数被do\_bootm\_linux依次调用）

boot\_prep\_linux

boot\_jump\_linux

boot\_jump\_linux->kernel\_entry（跳转到kernel中，离开uboot）

**标签：uboot传参给内核**

**由于这部分启动和U-boot传参数给内核是重复的，为了避免重复以及检索方便添加了这个标签“标签：uboot传参给内核”。**

u-boot在启动内核时，会向内核传递一些参数。u-boot目前可以通过三种方法传递参数给内核。**方法一**：是旧的参数结构方式（parameter\_struct)，用于2.4之前的内核使用的方式，现在已经很少使用。**方法二**：是标记列表 (tagged list) 方式。**方法三**：使用设备树。

u-boot和内核之间传递参数不仅仅取决于u-boot，还要看linux内核是否支持对应的形式。

由于方法一已经很少使用，这里不再介绍。

1.1.7.1、boot\_prep\_linux函数

1、boot\_prep\_linux首先执行的是char \*commandline = getenv("bootargs");

该语句从环境变量中获取启动参数。之后，函数将根据宏定义进行分支。不同的分支对应不同的向内核传递参数的方法。

2、分支的判断语句如下：

if (**IMAGE\_ENABLE\_OF\_LIBFDT** && images->ft\_len) {

#ifdef **CONFIG\_OF\_LIBFDT**

if (image\_setup\_linux(images)) {

printf("FDT creation failed! hanging...");

hang();

}

#endif

} else if (**BOOTM\_ENABLE\_TAGS**) {

setup\_start\_tag(gd->bd);

setup\_serial\_tag(&params);

setup\_commandline\_tag(gd->bd, commandline);

setup\_revision\_tag(&params);

setup\_memory\_tags(gd->bd);

setup\_initrd\_tag(gd->bd, images->rd\_start,

images->rd\_end);

}

setup\_board\_tags(&params);

setup\_end\_tag(gd->bd);

} else {

printf("FDT and ATAGS support not compiled in - hanging\n");

hang();

}

}

**上面的判断语句，根据宏定义决定使用fdt还是tag向内核传递参数。**

**对于imx6q**，在myimx.h中有#define CONFIG\_OF\_LIBFDT同时在u-boot-2015.04\include有：

#ifdef CONFIG\_OF\_LIBFDT

# define IMAGE\_ENABLE\_OF\_LIBFDT 1

#else

# define IMAGE\_ENABLE\_OF\_LIBFDT 0

#endif

可知这个判断语句将执行第一个分支，即imx6q使用fdt的方式传递参数。但是之前分析了tag方式，这里留作参考。

**1、设备树向内核传递信息：**

对应上面判断语句的第一个分支。

现在的u-boot向内核传递参数都使用设备树文件，设备树文件可以在**编译内核**的时候一起生成，也可以单独生成。u-boot可以读取设备树文件（.dtb文件）（imx6q读取的是**myimx6ek200-6qp.dtb**），修改设备树文件的节点或者直接加入新的节点，向内核传递启动参数。

**设备树原理参见设备树一章，这里介绍u-boot对于设备树的操作。**

u-boot向内核传参时将会调用image\_setup\_linux(images)函数（位于boot2015.04/common/image.c），下面详细分析该函数。

**1）传入参数**

首先其传入参数images的定义为

bootm\_headers\_t images; //images是一个在u-boot2015.04/common/cmd\_bootm.c //中的一个全局变量

该结构体在bootm\_start中清零。在初始化过程中，逐渐为其结构内的变量赋值，其间调用的函数会将image结构体中fdt相关的变量填充好。

其中bootm\_headers\_t的定义为：

uboot用于记录os、initrd、fdt信息的数据结构bootm\_headers，其定义在/include/image.h中，下面截取其与设备树相关的部分。

typedef struct bootm\_headers {

… … …

#if defined(CONFIG\_FIT)

const char \*fit\_uname\_cfg; /\* configuration node unit name \*/

void \*fit\_hdr\_os; /\* os FIT image header \*/

const char \*fit\_uname\_os; /\* os subimage node unit name \*/

int fit\_noffset\_os; /\* os subimage node offset \*/

void \*fit\_hdr\_rd; /\* init ramdisk FIT image header \*/

const char \*fit\_uname\_rd; /\* init ramdisk subimage node unit name \*/

int fit\_noffset\_rd; /\* init ramdisk subimage node offset \*/

void \*fit\_hdr\_fdt; /\* FDT blob FIT image header 指向DTB设备树镜像的头\*/

const char \*fit\_uname\_fdt; /\* FDT blob subimage node unit name \*/

int fit\_noffset\_fdt;/\* FDT blob subimage node offset \*/

void \*fit\_hdr\_setup; /\* x86 setup FIT image header \*/

const char \*fit\_uname\_setup; /\* x86 setup subimage node name \*/

int fit\_noffset\_setup;/\* x86 setup subimage node offset \*/

#endif

… … …

#ifdef CONFIG\_LMB

struct lmb lmb; /\* for memory mgmt。lmb为uboot下的一种内存管理机制，全称为logical memory blocks。用于管理镜像的内存。lmb所记录的内存信息最终会传递给kernel。这里对lmb不做展开描述。在/include/lmb.h和/lib/lmb.c中有对lmb的接口和定义的具体描述。\*/

#endif

} bootm\_headers\_t;

**2）函数内部详解**

int image\_setup\_linux(bootm\_headers\_t \*images)

{

ulong of\_size = images->ft\_len;

char \*\*of\_flat\_tree = &images->ft\_addr;

ulong \*initrd\_start = &images->initrd\_start;

ulong \*initrd\_end = &images->initrd\_end;

struct lmb \*lmb = &images->lmb;

ulong rd\_len;

int ret;

if (IMAGE\_ENABLE\_OF\_LIBFDT)

**boot\_fdt\_add\_mem\_rsv\_regions(lmb, \*of\_flat\_tree);**

if (IMAGE\_BOOT\_GET\_CMDLINE) {

ret = **boot\_get\_cmdline(lmb, &images->cmdline\_start,**

**&images->cmdline\_end);**

if (ret) {

puts("ERROR with allocation of cmdline\n");

return ret;

}

}

if (IMAGE\_ENABLE\_RAMDISK\_HIGH) {

rd\_len = images->rd\_end - images->rd\_start;

ret = **boot\_ramdisk\_high(lmb, images->rd\_start, rd\_len,**

**initrd\_start, initrd\_end);**

if (ret)

return ret;

}

if (IMAGE\_ENABLE\_OF\_LIBFDT) {

ret = **boot\_relocate\_fdt(lmb, of\_flat\_tree, &of\_size);**

if (ret)

return ret;

}

if (IMAGE\_ENABLE\_OF\_LIBFDT && of\_size) {

ret = **image\_setup\_libfdt(images, \*of\_flat\_tree, of\_size, lmb);**

if (ret)

return ret;

}

return 0;

}

①boot\_fdt\_add\_mem\_rsv\_regions会将原先的内存DTB镜像所在的内存置为reserve，保证该段内存不会被其他函数非法使用，保证接下来的reload数据是正确的；

②boot\_get\_cmdline从环境变量中获取命令行参数存储到lmb中，即images结构体中。

③boot\_ramdisk\_high从环境变量中获取initrd相关的内容存储到lmb中即images结构体中。

④boot\_relocate\_fdt会在bootmap区域中申请一块未被使用的内存，接着将DTB镜像内容复制到这块区域（即归lmb所管理的区域）

注：查看环境变量，可以发现存在“fdt\_high=0xffffffff”。这是因为在函数boot\_relocate\_fdt中，

如果fdt\_high不等于0xffffffff，将会调用lmb\_alloc\_base函数来分配DTB镜像reload的地址空间，此时若分配失败，则会停止bootm操作。

因而，建议设置“fdt\_high=0xffffffff”参数此时将根据设备树地址和设备树长度进行分配。

⑤image\_setup\_libfdt，调用函数fdt\_chosen()创建一个chosen节点用来存储bootargs（chosen节点参见设备树章节）；调用函数arch\_fixup\_fdt创建pci相关的节点。其他函数也是和创建节点相关的函数，这里不详细分析，如果需要对节点进行操作，或者修改dtb文件，应该仔细分析这个函数。

**2、tag结构传参**

**1、setup\_start\_tag(gd->bd)函数：**

static void setup\_start\_tag (bd\_t \*bd)

{

params = (struct tag \*)bd->bi\_boot\_params;

params->hdr.tag = ATAG\_CORE;

params->hdr.size = tag\_size (tag\_core);

params->u.core.flags = 0;

params->u.core.pagesize = 0;

params->u.core.rootdev = 0;

params = tag\_next (params); //指针params向下移动

}

**params**：为一个全局的静态变量（static struct **tag** \*params;注意params是一个tag类型的结构体的指针），最开始时，将bd->bi\_boot\_params的值赋给params。

setup\_start\_tag功能：1、初始化了第一个tag，是tag\_core类型的tag。2、调用tag\_next跳到第一个tag末尾，为下一个tag做准备。

注：gd->bd->bi\_boot\_params指向的内存地址为0x80000000+0x100=0x80000100(CONFIG\_MX6SL, CONFIG\_MX6SX, CONFIG\_MX6UL),或者0x10000100(其他)

**2、setup\_serial\_tag函数，设置串口有关的启动信息**

static void setup\_serial\_tag(struct tag \*\*tmp)

{

struct tag \*params = \*tmp; //使用传入的tmp保存串口tag的地址的地址

struct tag\_serialnr serialnr;

get\_board\_serial(&serialnr); //获取串口信息结构体

params->hdr.tag = ATAG\_SERIAL; //设置标识符

params->hdr.size = tag\_size (tag\_serialnr); //设置tag的大小

params->u.serialnr.low = serialnr.low;

params->u.serialnr.high= serialnr.high;

/\* #define tag\_next(t) ((struct tag \*)((u32 \*)(t) + (t)->hdr.size))\*/

params = tag\_next (params); // params往下移动一个tag结构体大小，用于存放下一个tag结构

\*tmp = params;

}

**3、setup\_commandline\_tag**

该函数设置第二个tag的hdr.tag为ATAG\_CMDLINE，然后拷贝cmdline到tags的cmdline结构体中，跳到下一个tag。

**4、setup\_memory\_tag**

将第三个tag设为ATAG\_MEM，将mem的start，size保存在此处，如果有多片ram（CONFIG\_NR\_DRAM\_BANKS > 1），则将下一个tag保存下一片ram的信息，依次类推。

**5、 setup\_board\_tags**

这个函数是\_\_weak属性，我们可以在自己的板级文件中去实现来保存跟板子相关的参数，如果没有实现，则是空函数

**6、 setup\_initrd\_tag**

tag性质为ATAG\_INITRD，用来告知内核压缩的ramdisk的位置，

**7、setup\_end\_tags**

将最末尾的tag设置为ATAG\_NONE,标志tag结束。

static void setup\_end\_tag(bd\_t \*bd)

{

params->hdr.tag = ATAG\_NONE;

params->hdr.size = 0;

}

注释：tag结构体：

struct tag {

struct tag\_header hdr;

union {

struct tag\_core core;

struct tag\_mem32 mem;

struct tag\_videotext videotext;

struct tag\_ramdisk ramdisk;

struct tag\_initrd initrd;

struct tag\_serialnr serialnr;

struct tag\_revision revision;

struct tag\_videolfb videolfb;

struct tag\_cmdline cmdline;

/\*

\* Acorn specific

\*/

struct tag\_acorn acorn;

/\*

\* DC21285 specific

\*/

struct tag\_memclk memclk;

} u;

};

struct tag\_header {

u32 size; //每个tag的大小

u32 tag; //每个tag的标识符

};

tag包括hdr和各种类型的tag\_\*,hdr来标志当前的tag是哪种类型的tag。

**标签：uboot传参给内核部分结束**

1.1.7.2、boot\_jump\_linux函数

主要作用就是跳转到os中运行，跳转过程如下：

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

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

kernel\_entry(0, machid, r2);

static void boot\_jump\_linux(bootm\_headers\_t \*images, int flag)

{

#ifdef CONFIG\_ARM64

/\*定义一个函数指针kernel\_entry，指向内核程序的首地址，四个参数，\*/

………..

#else

/\* machid板子的机器类型id\*/

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 = getenv("machid");

if (s) {

strict\_strtoul(s, 16, &machid);

printf("Using machid 0x%lx from environment\n", machid);

}

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;

**/\*r2寄存器用来存储u-boot向内核传递参数的首地址。如果定义了IMAGE\_ENABLE\_OF\_LIBFDT，即使用了fdt，则r2中存放fdt的首地址；如果不使用fdt，则使用tag，则r2中存储tag的首地址。**

**内核后续会根据r2中的内容使用u-boot传入的参数\*/**

if (!fake) {

#if defined(CONFIG\_ARMV7\_NONSEC) || defined(CONFIG\_ARMV7\_VIRT)

if (armv7\_boot\_nonsec()) {

armv7\_init\_nonsec();

secure\_ram\_addr(\_do\_nonsec\_entry)(kernel\_entry,

0, machid, r2);

} else

#endif

kernel\_entry(0, machid, r2);

}

#endif

}

整体栈的位置的分析sp:

u-boot在两个位置设置了栈，第一个位置，程序在SRAM中运行时，在汇编语言调用C语言之前设置了栈。第二个位置，在程序已经拷贝到SDRAM中后，重新设置栈。

I、./arch/arm/lib/crt0.S文件中，调用borad\_init\_f函数之前对sp进行了设置：

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

**/\***

**此时sp=0x00930000**

**sp=0x00900000+0x00040000-0x00010000=0x00930000**

**0x00900000=CONFIG\_SYS\_INIT\_RAM\_ADDR：RAM的起始地址**

**0x00040000= IRAM\_SIZE：RAM的大小**

**0x00010000= GENERATED\_GBL\_DATA\_SIZE：gd结构体的数据大小的大小**

**#define GENERATED\_GBL\_DATA\_SIZE 256 /\* (sizeof(struct global\_data) + 15) & ~15@ \*/+15 &（~15）的目的是为了让后四位为0，即16位字节对齐**

**\*/**

bic sp, sp, #7

**/\* 8-byte alignment for ABI compliance**

**将sp与7的反码与，后三位清零，sp=0x00930000,8字节对齐\*/**

mov r2, sp

sub sp, sp, #GD\_SIZE

**/\* allocate one GD above SP\*/**

**/\* #define GD\_SIZE 248 注：sizeof(struct global\_data) \*/**

bic sp, sp, #7 **/\* 8-byte alignment for ABI compliance \*/**

mov r9, sp

**/\* GD is above SP,r9中存放GD段的开始地址r9=0x00930000- GD\_SIZE \*/**

mov r1, sp

mov r0, #0

clr\_gd:  **/\*清除gd段的内容\*/**

cmp r1, r2 /\* while not at end of GD \*/

strlo r0, [r1] /\* clear 32-bit GD word \*/

addlo r1, r1, #4 /\* move to next \*/

blo clr\_gd

#if defined(CONFIG\_SYS\_MALLOC\_F\_LEN)

sub sp, sp, #CONFIG\_SYS\_MALLOC\_F\_LEN

/\*CONFIG\_SYS\_MALLOC\_F\_LEN=0x400=1kb\*/

str sp, [r9, #GD\_MALLOC\_BASE]

/\*GD\_MALLOC\_BASE = 192 (0xC0)，把sp的值存到r9+ 0xc0地址处。\*/

#endif

II、board\_init\_f函数执行完之后，对SDRAM进行了内存区域的划分，

之后执行代码的重定位后，sp就跳转到SDRAM的存储区域，主存中。

ldr sp, [r9, #GD\_START\_ADDR\_SP] /\* sp = gd->start\_addr\_sp \*/

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

ldr r9, [r9, #GD\_BD] /\* r9 = gd->bd \*/

sub r9, r9, #GD\_SIZE /\* new GD is below bd \*/

adr lr, here

ldr r0, [r9, #GD\_RELOC\_OFF] /\* r0 = gd->reloc\_off \*/

add lr, lr, r0

ldr r0, [r9, #GD\_RELOCADDR] /\* r0 = gd->relocaddr \*/

b relocate\_code

此时sp指针应该指向SDRAM栈分配内存的起始地址

四、实际调试

1、初始的时候执行各项初始化的时间

查看uboot各个启动项占用时间情况，可以采用如下方法：

1) 通过串口调试助手显示的时间。但是通过串口调试助手显示的时间很不精确，尤其是uboot启动过程重新设置了串口和控制台。

2) 在uboot/lib目录下，找到initcall.c文件，修改其中的initcall\_run\_list函数，用get\_timer来取得精确时间，单位ms。可以用printf打印信息到串口控制台，一般，串口控制台输出主要用于确定函数的执行顺序和内容等信息。

**下面介绍方法二：**

**关于initcall\_run\_list的参数：**init\_sequence[]对应init\_sequence\_f和init\_sequence\_r，分别定义在uboot/common/目录下的board\_f.c和board\_r.c文件中，代表了重定位之前和之后执行的启动项。

**关于对initcall\_run\_list的修改**：下面标注黄色的是添加的有关时间计算的内容，加粗是添加的注释。

int initcall\_run\_list(const init\_fnc\_t init\_sequence[])

{

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

ulong start; //start ticker var,

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

unsigned long reloc\_ofs = 0;

int ret;

start = get\_timer(0); //get time begin

if (gd->flags & GD\_FLG\_RELOC)

reloc\_ofs = gd->reloc\_off;

debug("initcall: %p", (char \*)\*init\_fnc\_ptr - reloc\_ofs);

if (gd->flags & GD\_FLG\_RELOC)

debug(" (relocated to %p)\n", (char \*)\*init\_fnc\_ptr);

else

debug("\n");

ret = (\*init\_fnc\_ptr)();

printf("initcall\_run\_list:%d->%lu\n", init\_fnc\_ptr - init\_sequence， get\_timer(start));

**/\***

**//get time end, print the info**

**init\_fnc\_ptr - init\_sequence当前函数指针减去函数列表的头指针，用来获得当前调用的函数在init\_sequence[]中的序号，具体对应哪个函数只能大致估算，与头文件定义了哪些宏有关。**

**\*/**

if (ret) {

printf("initcall sequence %p failed at call %p (err=%d)\n",

init\_sequence,

(char \*)\*init\_fnc\_ptr - reloc\_ofs, ret);

return -1;

}

}

return 0;

}

**上述函数对应的串口控制台输出：**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:14->7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:15->3

CPU: Freescale i.MX6Q rev1.5 at 792 MHz

CPU: Temperature 43 C

Reset cause: POR

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:16->21

Board: MYZR i.MX6 Evaluation Kit

Model: MY-IMX6-EK200-6Q-1G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:17->8

I2C: ready

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:18->4

DRAM: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:19->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:20->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:21->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:22->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:23->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:24->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:25->2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:26->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:27->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:28->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:29->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:30->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:31->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:32->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:33->3

1 GiB

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:34->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:35->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:36->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:37->3

**//上面是board\_init\_f的启动，下面开始board\_init\_r的启动**

\*\*\*\*\*\*\*\*\*board init r start

\*\*\*\*\*\*\*\*\*initcall\_run\_list start

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:0->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:1->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:2->5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:3->2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:4->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:5->37

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:6->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:7->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:8->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:9->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:10->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:11->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:12->3

MMC: FSL\_SDHC: 0, FSL\_SDHC: 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:13->8

SF: Detected SST25VF016B with page size 256 Bytes, erase size 4 KiB, total 2 MiB

\*\*\* Warning - bad CRC, using default environment

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:14->20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:15->3

No panel detected: default to Hannstar-XGA

Display: Hannstar-XGA (1024x600)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:16->46

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:17->3

In: serial

Out: serial

Err: serial

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:18->7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:19->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:20->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:21->3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:22->2

Net: FEC [PRIME]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*initcall\_run\_list:23->24

2、加入打印信息之后的输出：

console\_init\_f start

\*\*\*\*\*\*\*\*\*initcall\_run\_list:13->2

[22:19:02.343] U-Boot 2015.04 (Jun 04 2018 - 07:15:10)

[22:19:02.343]

[22:19:02.343] \*\*\*\*\*\*\*\*\*initcall\_run\_list:14->4 (display\_options)

[22:19:02.343] \*\*\*\*\*\*\*\*\*initcall\_run\_list:15->0 (display\_text\_info start)

[22:19:02.343] CPU: Freescale i.MX6Q rev1.5 at 792 MHz

[22:19:02.343] CPU: Temperature 41 C

[22:19:02.343] Reset cause: POR

[22:19:02.343] \*\*\*\*\*\*\*\*\*initcall\_run\_list:16->17 (print\_cpuinfo)

[22:19:02.432] show\_board\_info

[22:19:02.432] Board: MYZR i.MX6 Evaluation Kit

[22:19:02.432] Model: MY-IMX6-EK200-6Q-1G

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:17->7 (show\_board\_info)

[22:19:02.432] I2C: ready

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:18->1

[22:19:02.432] DRAM: \*\*\*\*\*\*\*\*\*initcall\_run\_list:19->1

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:20->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:21->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:22->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:23->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:24->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:25->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:26->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:27->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:28->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:29->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:30->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:31->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:32->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:33->0

[22:19:02.432] 1 GiB

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:34->1

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:35->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:36->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*initcall\_run\_list:37->0

[22:19:02.432] \*\*\*\*\*\*\*\*\*board init r start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:0->0 //trace

[22:19:02.561] initr\_reloc start\*\*\*\*\*\*\*\*\*initcall\_run\_list:1->1

[22:19:02.561] initr\_caches start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:2->3

[22:19:02.561] initr\_reloc\_global\_data start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:3->3

[22:19:02.561] initr\_barrier start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:4->1

[22:19:02.561] initr\_malloc start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:5->10

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:6->0

[22:19:02.561] initr\_dm start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:7->1

[22:19:02.561] board\_init start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:8->1

[22:19:02.561] stdio\_init\_tables start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:9->2

[22:19:02.561] initr\_serial start\*\*\*\*\*\*\*\*\*initcall\_run\_list:10->2

[22:19:02.561] initr\_announce start

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:11->2

[22:19:02.561] power\_init\_board

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:12->1

[22:19:02.561] initr\_mmc

[22:19:02.561] MMC: FSL\_SDHC: 0, FSL\_SDHC: 1

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:13->6

[22:19:02.561] initr\_env

[22:19:02.561] SF: Detected SST25VF016B with page size 256 Bytes, erase size 4 KiB, total 2 MiB

[22:19:02.561] \*\*\* Warning - bad CRC, using default environment

[22:19:02.561]

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:14->18

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:15->0

[22:19:02.561] stdio\_add\_devices(I2C)

[22:19:02.561] No panel detected: default to Hannstar-XGA

[22:19:02.561] Display: Hannstar-XGA (1024x600)

[22:19:02.561] \*\*\*\*\*\*\*\*\*initcall\_run\_list:16->45

[22:19:02.634] initr\_jumptable

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:17->2

[22:19:02.634] console\_init\_r

[22:19:02.634] In: serial

[22:19:02.634] Out: serial

[22:19:02.634] Err: serial

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:18->5

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:19->0

[22:19:02.634] initr\_enable\_interrupts

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:20->2

[22:19:02.634] initr\_ethaddr

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:21->1

[22:19:02.634] board\_late\_init

[22:19:02.634] \*\*\*\*\*\*\*\*\*initcall\_run\_list:22->2

[22:19:02.634] initr\_net

[22:19:02.634] Net: FEC [PRIME]

[22:19:02.661] \*\*\*\*\*\*\*\*\*initcall\_run\_list:23->21

[22:19:02.661] Normal Boot

3、修改：

(1)CONFIG\_SYS\_MALLOC\_LEN默认16M，可以缩减为4M，减少uboot启动时间(从30ms降到4ms)

CONFIG\_SYS\_MALLOC\_LEN为手动设置的堆区的大小，可以修改小一点：

(2) 修改延时

u-boot-2015.04\include\configs\myimx6.h

#define CONFIG\_BOOTDELAY 0

默认延时为1秒，可以将其修改为0

文件./include/env\_default.h中

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

"bootdelay=" \_\_stringify(CONFIG\_BOOTDELAY) "\0"

#endif

./common/autoboot.c文件中：

函数const char \*bootdelay\_process(void)：该函数在main\_loop函数中被调用

s = getenv("bootdelay");

bootdelay = s ? (int)simple\_strtol(s, NULL, 10) : CONFIG\_BOOTDELAY;

如果没有从env\_default.h文件中获取到bootdelay变量的值，就使用宏定义，否则使用env\_default.h文件中获取的延时值，

stored\_bootdelay = bootdelay;

真正的延时在./common/autoboot.c文件中的static int abortboot\_normal(int bootdelay)函数实现。abortboot调用abortboot\_normal函数->autoboot\_command

static int abortboot(int bootdelay)

{

#ifdef CONFIG\_AUTOBOOT\_KEYED

return abortboot\_keyed(bootdelay);

#else

return abortboot\_normal(bootdelay);

#endif

}

最后main\_loop函数调用autoboot\_command

void autoboot\_command(const char \*s)

{

debug("### main\_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

if (stored\_bootdelay != -1 && s && !abortboot(stored\_bootdelay)) {

#if defined(CONFIG\_AUTOBOOT\_KEYED) && !defined(CONFIG\_AUTOBOOT\_KEYED\_CTRLC)

int prev = disable\_ctrlc(1); /\* disable Control C checking \*/

#endif

run\_command\_list(s, -1, 0);

#if defined(CONFIG\_AUTOBOOT\_KEYED) && !defined(CONFIG\_AUTOBOOT\_KEYED\_CTRLC)

disable\_ctrlc(prev); /\* restore Control C checking \*/

#endif

}

#ifdef CONFIG\_MENUKEY

if (menukey == CONFIG\_MENUKEY) {

s = getenv("menucmd");

if (s)

run\_command\_list(s, -1, 0);

}

#endif /\* CONFIG\_MENUKEY \*/

}

(3)去掉board\_init\_f调用函数中的print\_cpuinfo函数

print\_cpuinfo：打印CPU信息

/\*#define CONFIG\_DISPLAY\_CPUINFO\*/

(4)去掉board\_init\_f调用函数数组中的打印板子信息的函数：

/\*#define CONFIG\_DISPLAY\_BOARDINFO\*/

3/4完成启动时间：794->691,u-boot大小：375->371

(4)裁剪大小，新版u-boot大小之所很大，是因为支持了很多新的命令。

I、去掉USB相关的命令以及宏定义

/\*

#define CONFIG\_CMD\_USB 支持USB命令集，这样就可以使用U-Boot命令访问USB设备

#if defined(CONFIG\_CMD\_USB)

#define CONFIG\_USB\_EHCI

#define CONFIG\_USB\_EHCI\_MX6

#define CONFIG\_USB\_STORAGE

#define CONFIG\_EHCI\_HCD\_INIT\_AFTER\_RESET

#define CONFIG\_USB\_HOST\_ETHER

#define CONFIG\_USB\_ETHER\_ASIX

#define CONFIG\_MXC\_USB\_PORTSC (PORT\_PTS\_UTMI | PORT\_PTS\_PTW)

#define CONFIG\_MXC\_USB\_FLAGS 0

\*/

/\*#define CONFIG\_USB\_MAX\_CONTROLLER\_COUNT 1\*/ /\* Enabled USB controller number \*/

/\*#endif\*/

u-boot支持USB，主要是为了让内核文件以及文件系统能够存放在U盘上面，这样就可以通过U盘来启动。

II、去掉SATA有关的命令

#if defined(CONFIG\_CMD\_SATA)

#define CONFIG\_DWC\_AHSATA

#define CONFIG\_SYS\_SATA\_MAX\_DEVICE 1

#define CONFIG\_DWC\_AHSATA\_PORT\_ID 0

#define CONFIG\_DWC\_AHSATA\_BASE\_ADDR SATA\_ARB\_BASE\_ADDR

#define CONFIG\_LBA48

#define CONFIG\_LIBATA

#endif

SATA有关的命令，可以支持从sata 硬盘启动

III、去掉pci总线的命令：

#if defined(CONFIG\_CMD\_PCI)

#define CONFIG\_PCI

#define CONFIG\_PCI\_PNP

#define CONFIG\_PCI\_SCAN\_SHOW

#define CONFIG\_PCIE\_IMX

#define CONFIG\_PCIE\_IMX\_POWER\_GPIO IMX\_GPIO\_NR(6, 8)

#if defined(CONFIG\_TARGET\_MYIMX6EK200)

#define CONFIG\_PCIE\_IMX\_PERST\_GPIO IMX\_GPIO\_NR(6, 7)

#elif defined(CONFIG\_TARGET\_MYIMX6EK314)

#define CONFIG\_PCIE\_IMX\_PERST\_GPIO IMX\_GPIO\_NR(6, 14)

#endif

IV、去掉I2C、memory的相关命令

/\*#define CONFIG\_CMD\_I2C\*/

/\*#define CONFIG\_CMD\_MEMTEST\*/

与i2c有关的地方：

\board\myzr\myimx6\myimx6.c：int board\_init(void)->

setup\_i2c(1, CONFIG\_SYS\_I2C\_SPEED, 0x7f, &i2c\_pad\_info1);

myimx6.h中的定义：

/\*#define CONFIG\_CMD\_I2C\*/

/\*

#define CONFIG\_SYS\_I2C

#define CONFIG\_SYS\_I2C\_MXC

#define CONFIG\_SYS\_I2C\_SPEED 100000

\*/

board\_f.c文件中：

#if defined(CONFIG\_HARD\_I2C) || defined(CONFIG\_SYS\_I2C)

/\*init\_func\_i2c,\*/

#endif

board\_r.c文件中：

stdio\_add\_devices函数中//i2c\_init\_all();

4、老师的打印信息：

[10:19:37.431] U-Boot 2015.04 (Apr 29 2018 - 14:52:17)

[10:19:37.431]

[10:19:37.431] CPU: Freescale i.MX6Q rev1.5 at 792 MHz

[10:19:37.439] CPU: Temperature 33 C

[10:19:37.452] Reset cause: POR

[10:19:37.452] Board: MYZR i.MX6 Evaluation Kit

[10:19:37.452] Model: MY-IMX6-EK200-6Q-1G

[10:19:37.462] DRAM: 1 GiB

[10:19:37.462] MMC: FSL\_SDHC: 0, FSL\_SDHC: 1

[10:19:37.480] Using default environment

[10:19:37.480]

[10:19:37.480] In: serial

[10:19:37.488] Out: serial

[10:19:37.488] Err: serial

[10:19:37.488] Normal Boot

[10:19:37.488] Hit any key to stop autoboot: 0

[10:19:37.488] switch to partitions #0, OK

[10:19:37.687] mmc1(part 0) is current device

[10:19:37.687] reading zImage-myimx6

[10:19:37.699] 5841888 bytes read in 163 ms (34.2 MiB/s)

[10:19:37.860] Booting from mmc ...

[10:19:37.860] reading myimx6ek200-6q.dtb

[10:19:37.870] 40956 bytes read in 18 ms (2.2 MiB/s)

[10:19:37.891] Kernel image @ 0x12000000 [ 0x000000 - 0x5923e0 ]

[10:19:37.901] ## Flattened Device Tree blob at 18000000

[10:19:37.901] Booting using the fdt blob at 0x18000000

[10:19:37.901] Using Device Tree in place at 18000000, end 1800cffb

[10:19:37.909]

[10:19:37.909] Starting kernel ...

5、u-boot程序被搬运了两次

在固化的ROM程序中，u-boot被从spi flash 搬运到SDRAM中的0x17800000地址处，然后程序跳转到该地址执行u-boot的代码，之后在分配SDRAM的空间的时候，又重新把u-boot搬运到了0x4ff7b000地址处。可以从board\_init\_f和board\_init\_r两个函数的地址可以看出来。

board\_init\_f=17804eac

board\_init\_r=4ff80504

五、问题分析：

1、u-boot的下载位置，不同的板子不一样，通过熔丝位决定，对于imx6q的板子(明远智睿)，下载到spi-falsh;对于imx6d的板子，厂家不同，下载到mmc flash。需要使用厂家提供的标准的烧写工具来烧写。

2、不要使用//注释#define

//#define不能这样使用

六、u-boot的命令：

1、在u-boot底下添加一个命令：

所有命令程序都在common目录底下，以cmd\_开头

I、在./common目录下面建立一个cmd\_zjp.c文件：

#include <common.h>

#include <command.h>

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

{

printf("do\_zjp command is ready now!");

return 0;

}

U\_BOOT\_CMD(

zjp, 2, 0, do\_zjp,

"zjp - this is a command, do nothing",

"- this is a command, do nothing"

);

II、在myimx6.h 中建立宏定义：

/\* ZJP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define CONFIG\_CMD\_ZJP

III、修改makefile文件：

加入：

ifdef CONFIG\_CMD\_ZJP

obj-$(CONFIG\_CMD\_ZJP) += cmd\_zjp.o

endif

IV、重新编译u-boot即可。

2、u-boot下命令的实现原理：

实战见效果了，我们继续分析。不难发现，只要能搞清上面代码中的“U\_BOOT\_CMD”，就能弄明白uboot命令实现原理。

U\_BOOT\_CMD格式如下：

U\_BOOT\_CMD(\_name, \_maxargs, \_rep, \_cmd, \_usage, \_help)

其中，各个参数解释如下：

| 参数名 | 说明 |
| --- | --- |
| \_name | 命令名，非字符串，但在U\_BOOT\_CMD中用“#”符号转化为字符串 |
| \_maxargs | 命令的最大参数个数 |
| \_rep | 是否自动重复（按Enter键是否会重复执行） |
| \_cmd | 该命令对应的响应函数 |
| \_usage | 简短的使用说明（字符串） |
| \_help | 较详细的使用说明（字符串） |

在u-boot.lds文件中：

. = ALIGN(4);

.u\_boot\_list : {

KEEP(\*(SORT(.u\_boot\_list\*)));

}

3、u-boot下命令的执行

如果在u-boot的延迟时间内有按键按下将执行run\_command\_list，如果没有按键按下将执行cli\_loop()函数：

1、cli\_loop()分支

void cli\_loop(void)

{

#ifdef CONFIG\_SYS\_HUSH\_PARSER 进入hush命令模式

parse\_file\_outer();

/\* This point is never reached \*/

for (;;);

#else

cli\_simple\_loop(); //进入通用模式

#endif /\*CONFIG\_SYS\_HUSH\_PARSER\*/

}

void cli\_simple\_loop(void)

{

static char lastcommand[CONFIG\_SYS\_CBSIZE + 1] = { 0, };

int len;

int flag;

int rc = 1;

for (;;) {

len = cli\_readline(CONFIG\_SYS\_PROMPT);

// 打印CONFIG\_SYS\_PROMPT，然后从串口读取一行作为命令，存储在console\_buffer中char console\_buffer[CONFIG\_SYS\_CBSIZE + 1]; CONFIG\_SYS\_CBSIZE为控制台buffer的大小

// 在tiny210中定义如下：./configs/tiny210\_defconfig:203:CONFIG\_SYS\_PROMPT="TINY210 => "

flag = 0; /\* assume no special flags for now \*/

if (len > 0)

strlcpy(lastcommand, console\_buffer,

CONFIG\_SYS\_CBSIZE + 1);

// 如果获得了一个新行时，命令会存储在console\_buffer，将命令复制到lastcommand中

else if (len == 0)

flag |= CMD\_FLAG\_REPEAT;

// 只是得到一个单纯的换行符时，设置重复标识，后续重复执行上一次命令

if (len == -1)

puts("<INTERRUPT>\n");

// 获得非数据值时，直接打印中断

else

rc = run\_command\_repeatable(lastcommand, flag);

// 否则，执行lastcommand中的命令

if (rc <= 0) {

/\* invalid command or not repeatable, forget it \*/

lastcommand[0] = 0;

// 命令执行出错时，清空lastcommand，防止下一次重复执行这个命令

}

}

}

/\* run\_command\_repeatable实现如下 \*/

int run\_command\_repeatable(const char \*cmd, int flag)

{

return cli\_simple\_run\_command(cmd, flag);

}

2、run\_command\_list分支

该分支是在u-boot的delay期间没有按键按下且存在bootcmd时，运行的分支，用来执行bootcmd中的指令。

1) run\_command\_list

根据是否定义了CONFIG\_SYS\_HUSH\_PARSER判断使用hush解析器分析命令还是调用cli\_simple\_run\_command\_list运行命令。

int run\_command\_list(const char \*cmd, int len, int flag)

{

int need\_buff = 1;

char \*buff = (char \*)cmd; /\* cast away const \*/

int rcode = 0;

if (len == -1) {

len = strlen(cmd);

#ifdef CONFIG\_SYS\_HUSH\_PARSER

/\* hush will never change our string \*/

need\_buff = 0;

#else

/\* the built-in parser will change our string if it sees \n \*/

need\_buff = strchr(cmd, '\n') != NULL;

#endif

}

if (need\_buff) {

buff = malloc(len + 1);

if (!buff)

return 1;

memcpy(buff, cmd, len);

buff[len] = '\0';

}

#ifdef CONFIG\_SYS\_HUSH\_PARSER

rcode = parse\_string\_outer(buff, FLAG\_PARSE\_SEMICOLON);

#else

/\*

\* This function will overwrite any \n it sees with a \0, which

\* is why it can't work with a const char \*. Here we are making

\* using of internal knowledge of this function, to avoid always

\* doing a malloc() which is actually required only in a case that

\* is pretty rare.

\*/

rcode = cli\_simple\_run\_command\_list(buff, flag);

if (need\_buff)

free(buff);

#endif

return rcode;

}

2、cli\_simple\_run\_command\_list

逐行获取命令，调用cli\_simple\_run\_command进行每行命令的运行。

int cli\_simple\_run\_command\_list(char \*cmd, int flag)

{

char \*line, \*next;

int rcode = 0;

/\*

\* Break into individual lines, and execute each line; terminate on

\* error.

\*/

next = cmd;

line = cmd;

while (\*next) {

if (\*next == '\n') {

\*next = '\0';

/\* run only non-empty commands \*/

if (\*line) {

debug("\*\* exec: \"%s\"\n", line);

if (cli\_simple\_run\_command(line, 0) < 0) {

rcode = 1;

break;

}

}

line = next + 1;

}

++next;

}

if (rcode == 0 && \*line)

rcode = (cli\_simple\_run\_command(line, 0) < 0);

return rcode;

}

汇合点

从这点开始，上述两个分支开始调用同一个函数，进行命令的运行。

/\* cli\_simple\_run\_command实现如下 \*/

功能：对每行命令进行规范化整合，调用cmd\_process对命令进行处理

int cli\_simple\_run\_command(const char \*cmd, int flag)

{

char cmdbuf[CONFIG\_SYS\_CBSIZE]; /\* working copy of cmd \*/

char \*token; /\* start of token in cmdbuf \*/

char \*sep; /\* end of token (separator) in cmdbuf \*/

char finaltoken[CONFIG\_SYS\_CBSIZE];

char \*str = cmdbuf;

char \*argv[CONFIG\_SYS\_MAXARGS + 1]; /\* NULL terminated \*/

int argc, inquotes;

int repeatable = 1;

int rc = 0;

debug\_parser("[RUN\_COMMAND] cmd[%p]=\"", cmd);

if (DEBUG\_PARSER) {

/\* use puts - string may be loooong \*/

puts(cmd ? cmd : "NULL");

puts("\"\n");

}

clear\_ctrlc(); /\* forget any previous Control C \*/

if (!cmd || !\*cmd)

return -1; /\* empty command \*/

if (strlen(cmd) >= CONFIG\_SYS\_CBSIZE) {

puts("## Command too long!\n");

return -1;

}

strcpy(cmdbuf, cmd);

/\* Process separators and check for invalid

\* repeatable commands

\*/

debug\_parser("[PROCESS\_SEPARATORS] %s\n", cmd);

while (\*str) {

// 这里过滤掉一些对命令进行处理的部分代码

/\* find macros in this token and replace them \*/

cli\_simple\_process\_macros(token, finaltoken);

/\* Extract arguments \*/

argc = cli\_simple\_parse\_line(finaltoken, argv);

// 对命令进行加工处理，转化成argv和argc格式。

if (argc == 0) {

rc = -1; /\* no command at all \*/

continue;

}

if (cmd\_process(flag, argc, argv, &repeatable, NULL))

rc = -1;

// 调用cmd\_process对命令进行处理

/\* Did the user stop this? \*/

if (had\_ctrlc())

return -1; /\* if stopped then not repeatable \*/

}

cmd\_process函数：先调用find\_cmd,再调用cmd\_call执行命令

enum command\_ret\_t cmd\_process(int flag, int argc, char \* const argv[],

int \*repeatable, ulong \*ticks)

{

enum command\_ret\_t rc = CMD\_RET\_SUCCESS;

cmd\_tbl\_t \*cmdtp;

/\* Look up command in command table \*/

cmdtp = find\_cmd(argv[0]);

if (cmdtp == NULL) {

printf("Unknown command '%s' - try 'help'\n", argv[0]);

return 1;

}

/\* found - check max args \*/

if (argc > cmdtp->maxargs)

rc = CMD\_RET\_USAGE;

#if defined(CONFIG\_CMD\_BOOTD)

/\* avoid "bootd" recursion \*/

else if (cmdtp->cmd == do\_bootd) {

if (flag & CMD\_FLAG\_BOOTD) {

puts("'bootd' recursion detected\n");

rc = CMD\_RET\_FAILURE;

} else {

flag |= CMD\_FLAG\_BOOTD;

}

}

#endif

/\* If OK so far, then do the command \*/

if (!rc) {

if (ticks)

\*ticks = get\_timer(0);

rc = cmd\_call(cmdtp, flag, argc, argv);

if (ticks)

\*ticks = get\_timer(\*ticks);

\*repeatable &= cmdtp->repeatable;

}

if (rc == CMD\_RET\_USAGE)

rc = cmd\_usage(cmdtp);

return rc;

}

查找命令：

cmd\_tbl\_t \*find\_cmd(const char \*cmd)

{

cmd\_tbl\_t \*start = ll\_entry\_start(cmd\_tbl\_t, cmd);

// 获取命令表的地址，start表示指向命令表的指针，具体实现看后面

const int len = ll\_entry\_count(cmd\_tbl\_t, cmd);

// 获取命令表的长度，具体实现看后面

return find\_cmd\_tbl(cmd, start, len);

// 以命令表的指针和命令表的长度为参数，查找和cmd匹配的表项，也就是cmd\_tbl\_t结构，并返回给调用者。

}

include/linker\_lists.h

#define ll\_entry\_start(\_type, \_list) \

({ \

static char start[0] \_\_aligned(4) \_\_attribute\_\_((unused, \

section(".u\_boot\_list\_2\_"#\_list"\_1"))); \

(\_type \*)&start; \

})

// 因为传进来的\_list是cmd，所以这里就是获取命令表的起始地址，也就是.u\_boot\_list\_2\_cmd\_1的地址，

#define ll\_entry\_end(\_type, \_list) \

({ \

static char end[0] \_\_aligned(4) \_\_attribute\_\_((unused, \

section(".u\_boot\_list\_2\_"#\_list"\_3"))); \

(\_type \*)&end; \

})

// 因为传进来的\_list是cmd，所以这里就是获取命令表的结束地址，也就是.u\_boot\_list\_2\_cmd\_3的地址，

#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; \

})

// 因为传进来的\_list是cmd，所以这里就是计算命令表的长度

执行命令：调用cmd\_call函数

static int cmd\_call(cmd\_tbl\_t \*cmdtp, int flag, int argc, char \* const argv[])

{

int result;

result = (cmdtp->cmd)(cmdtp, flag, argc, argv);

// 直接执行命令表项cmd\_tbl\_t 中的cmd命令处理函数

if (result)

debug("Command failed, result=%d\n", result);

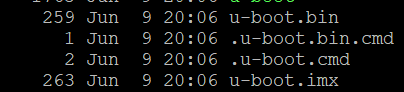
// 命令返回非0值时，报错

return result;

}

七、imx6的u-boot.imx和u-boot.bin

u-boot.bin和u-boot.imx的区别就是u-boot.imx在u-boot.bin的头部加入了4KB的头部。



u-boot.bin大小：259k; u-boot.imx大小：263k

4KB头部包括：IVT头部，启动数据结构Boot Data structures，DCD(device configure data)。作用： This IVT header is for our boot ROM to identify the u-boot's location & function etc...为ROM启动程序确定u-boot在SDRAM中的位置。

<https://blog.csdn.net/metersun/article/details/52334040>

(1)如何生成前4k的头部：

u-boot-2015.07及以后的版本中，更改了u-boot.imx的产生方式，不再使用上述的2009版本中的flash\_header.S，而是使用mkimage工具直接生成。此处使用的mkimage包含的目标文件类型-T参数中新增了imximage选项，mkimage会调用格式类型imximage的解释器来生成针对i.mx(5系6系)系列芯片的 IVT、boot Data、DCD的格式头。解释器代码实现在tools/imximage.c中，这里不做深入研究。

在u-boot2015.04/.u-boot.imx.cmd文件中包含着生成u-boot.imx的指令，对于该文件中指令的说明如下：

|  |  |
| --- | --- |
| -T imximage | 指定生成image的类型 |
| -e 0x17800000 | 指定entry address，和V2009中不同，这里的entry address是真正的U-Boot的入口，而不是将image拷贝到的目标地址 |
| -d u-boot.bin | 对应真正u-boot的binary，这个u-boot.bin将不再像V2009中那样包含为开始的blank和DCD/Plugin |
| u-boot.imx | 生成目标文件 |

0x17800000:u-boot.bin程序的入口地址entry

打开编译生成的System.map文件，可以看到程序入口\_start标号的地址为：0x17800000:

17800000 T \_\_image\_copy\_start

17800000 T \_start

**以我们这里的u-boot为例进行说明**

在./.u-boot.imx.cmd文件中存在下面语句：

./.u-boot.imx.cmd:1:cmd\_u-boot.imx := ./tools/mkimage -n board/myzr/myimx6/myimx6.cfg.cfgtmp -T imximage -e 0x17800000 -d u-boot.bin u-boot.imx >/dev/null

 其中-n指明输入文件为board/myzr/myimx6/myimx6.cfg.cfgtmp，-T指明解释器为imximage，-e为镜像程序入口点。

mkimage调用选项-T的imximage 解释器解析输入文件board/myzr/myimx6/myimx6.cfg.cfgtmp，而文件board/myzr/myimx6/myimx6.cfg.cfgtmp为myimx6.cfg预处理(使用gcc -E选项)后产生的临时文件。

关于myimx6.cfg

1、myimx6.cfg 其中包含的IMAGE\_VERSION 2和BOOT\_FROM spi，它们将在下面会被使用。

在u-boot2015.04/tools/imximage.c中的函数parse\_cfg\_cmd使用前导函数从输入文件myimx6.cfg.cfgtmp取出的参数，当参数是CMD\_IMAGE\_VERSION，它对应myimx6.cfg.cfgtmp文件中的MAGE\_VERSION 2，然后调用get\_cfg\_value得到版本号为2。imximage的头格式分版本号1和2。同理CMD\_BOOT\_FROM，对应于myimx6.cfg.cfgtmp文件中的BOOT\_FROM   spi，调用get\_table\_entry\_id在表imximage\_boot\_offset找到spi对应的偏移量FLASH\_OFFSET\_SPI(它在文件imximage.h中定义为0x400，这里反映了不同的启动设备其IVT偏移量的不同)。后续的操作以此为参数，然后联合其他的解析结果，生成总的结构体imx\_header\_v2\_t，最后mkimage将这个结构体插入文件u-boot.bin的头部，生成u-boot.imx文件。

2、myimx6.cfg 该文件预处理后会将ddr-setup.cfg、1066mhz\_4x128mx16.cfg、clocks.cfg的内容复制到myimx6.cfg.cfgtmp中，而前三个文件即为DCD配置文件，如ddr-setup.cfg的内容为：

1. DATA 4, MX6\_IOM\_DRAM\_SDQS0, 0x00000030
2. DATA 4, MX6\_IOM\_DRAM\_SDQS1, 0x00000030
3. DATA 4, MX6\_IOM\_DRAM\_SDQS2, 0x00000030
4. DATA 4, MX6\_IOM\_DRAM\_SDQS3, 0x00000030

如MX6\_IOM\_DRAM\_SDQS0 又在文件arch/arm/include/asm/arch/mx6q-ddr.h中定义：

#define MX6\_IOM\_DRAM\_SDQS0 0x020e05a8

预处理后为(删除了空行和注解)：

1. IMAGE\_VERSION 2
2. BOOT\_FROM spi
3. DATA4,0x020e05a8,0x00000030
4. DATA4,0x020e05b0,0x00000030
5. DATA4,0x020e0524,0x00000030
6. DATA4,0x020e051c,0x00000030

八、[设备树](https://blog.csdn.net/x356982611/article/details/77833347)

**1、定义：**

“A data structure by which bootloaders pass hardware layout to Linux in a device-independent manner, simplifying hardware probing.” 翻译：设备树是一种**数据结构**，通过使用它，bootloader能够**不依赖具体设备**地将**硬件布局传给内核**，简化了硬件探测。

Device Tree可以描述的信息包括**CPU**的**数量**和**类别**、**内存**基地址和大小、**总线**和**桥**、外设连接、**中断**控制器和中断使用情况、**GPIO**控制器和GPIO使用情况、**Clock控制器**和Clock**使用情况**。

设备树的**主要优势**：对于同一SOC的不同主板，只需更换设备树文件.dtb即可实现不同主板的无差异支持，而无需更换内核文件。

**2、设备树的组成：**

设备树包含DTC（device tree compiler），DTS（device tree source和DTB（device tree blob）关系为：

DTC将DTS转换为DTB

(1)dts文件：

.dts文件是一种以ASCII文本对Device Tree的描述，放置在**内核**/arch/arm/boot/dts目录。一般而言，一个.dts文件对应一个ARM的machine。 由于一个SOC可能有多个不同的电路板，而每个电路板拥有一个 .dts。这些dts势必会存在许多共同部分，为了减少代码的冗余，设备树将这些共同部分提炼保存在.dtsi文件中，供不同的dts共同使用。**.dtsi的使用方法，类似于C语言的头文件**，在dts文件中需要进行include .dtsi文件。当然，dtsi本身也支持include 另一个dtsi文件。

(2) dtc文件

DTC为编译工具，它可以将.dts文件编译成.dtb文件。DTC的源码位于内核的scripts/dtc目录，内核选中CONFIG\_OF，编译内核的时候，主机可执行程序DTC就会被编译出来。 即scripts/dtc/Makefile中

hostprogs-y := dtc

always := $(hostprogs-y) ﻿​

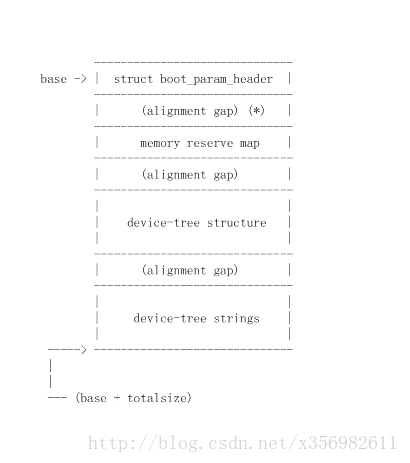
(3)dtb文件：

DTC编译.dts生成的二进制文件（.dtb），bootloader在引导内核时，会预先读取.dtb到内存，进而由内核解析。

在内核的arch/arm/boot/dts/Makefile中，若选中某种SOC，则与其对应相关的所有dtb文件都将编译出来。在linux下，make dtbs可单独编译dtb。

下面介绍dtb的结构

DTB由三部分组成：头（Header）、结构块（device-tree structure）、字符串块（string block）。下面将详细介绍这三部分的内容。



**头**：在\kernel\include\linux\of\_fdt.h文件中有相关定义struct boot\_param\_header

**memory reserve map**：这个区域包括了若干的reserve memory描述符。每个reserve memory描述符是由address和size组成。其中address和size都是用U64来描述。

有些系统，我们也许会保留一些memory有特殊用途（例如DTB或者initrd image），或者在有些DSP+ARM的SOC platform上，有些memory被保留用于ARM和DSP进行信息交互。这些保留内存不会进入内存管理系统。

**device-tree structure**：设备树结构块是一个线性化的结构体，是设备树的主体，以节点的形式保存了主板上的设备信息。

**一个节点的结构如下：**

(1)节点开始标志：一般为OF\_DT\_BEGIN\_NODE（0x00000001）。

(2)节点路径或者节点的单元名(version<3以节点路径表示，version>=0x10以节点单元名表示)

(3)填充字段（对齐到四字节）

(4)节点属性。每个属性以宏OF\_DT\_PROP(0x00000003)开始，后面依次为属性值的字节长度(4字节)、属性名称在字符串块中的偏移量(4字节)、属性值和填充（对齐到四字节）。

(5)如果存在子节点，则定义子节点。

(6)节点结束标志OF\_DT\_END\_NODE(0x00000002)。

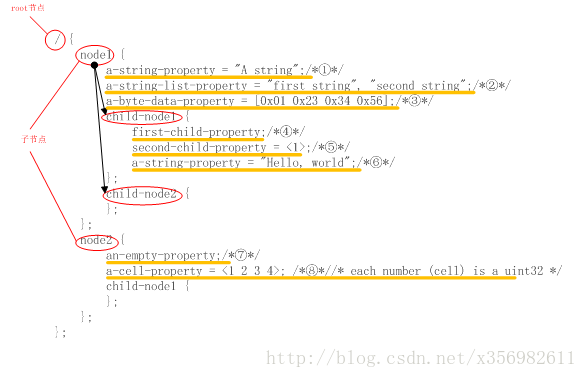
**字符串块**：通过节点的定义知道节点都有若干属性，而不同的节点的属性又有大量相同的属性名称，因此将这些属性名称提取出一张表，当节点需要应用某个属性名称时，直接在属性名字段保存该属性名称在字符串块中的偏移量。

**3、设备树与bootloader的关系**

Bootloader需要将设备树在内存中的地址传给内核。在ARM中通过bootm或bootz命令来进行传递。   
 bootm [kernel\_addr] [initrd\_address] [dtb\_address]，其中kernel\_addr为内核镜像的地址，initrd为initrd的地址，dtb\_address为dtb所在的地址。若initrd\_address为空，则用“-”来代替。

**4、设备树的语法：**

Device Tree由节点和属性构成。属性为key-value对，节点包括了各种属性，也可以包含子节点。



1 唯一的根节点 “/”。

**注：**如果看过内核/arch/arm/boot/dts目录的读者看到这可能有一个疑问。在每个.dsti和.dts中都会存在一个“/”根节点，那么如果在一个设备树文件中include一个.dtsi文件，那么岂不是存在多个“/”根节点了么。其实不然，编译器DTC在对.dts进行编译生成dtb时，会对node进行合并操作，最终生成的dtb只有一个root node。Dtc会进行合并操作这一点从属性上也可以得到验证。

2 一些节点：node1 node2

3 子节点 node1的子节点child-node1和child-node2

4 属性

**属性**都是简单的key-value对，其中value也可以是空的或包含任意的byte流。以下是一些属性的基本数据结构：

1 双引号包含的字符信息

string-property = "a string";

2 cells单位信息是32位无符号整型数据

cell-property = <0xFF01 412 0x12341283>;

3 二进制数据流

binary-property = [0x01 0x02 0x03 0x04];

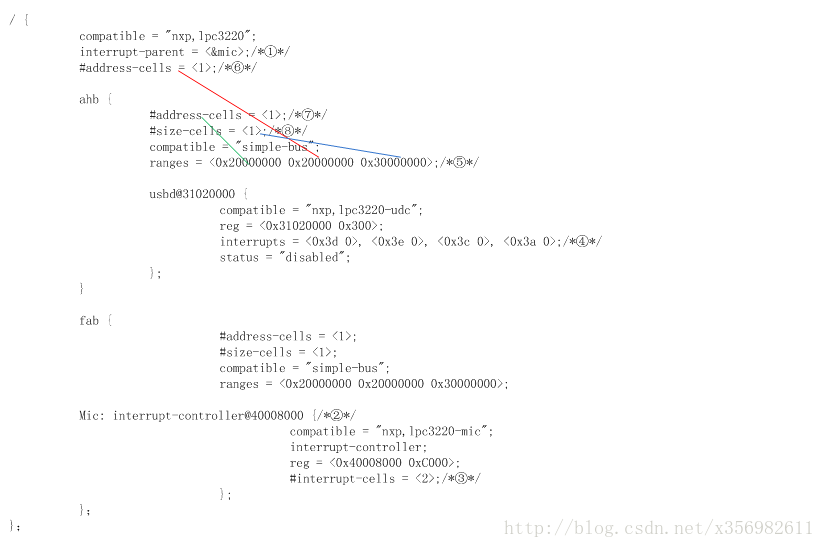
4 混合数据用逗号隔开

mixed-property = "a string", [0x01 0x02 0x03 0x04], <0xFF01 412 0x12341283>;

5 字符列表

string-list = "string test1", "string test2";

**属性的例子**



1、reg

reg是描述父节点的memory-mapped IO register的offset和length，对应cells属性。子节点的reg属性address和length长度取决于父节点对应的#address-cells和#size-cells的值

例子：

reg = <0x31020000,0x300>

其中0x31020000为address，0x300为size

2、compatible

compatible属性为string list，用来将设备匹配对应的driver驱动，优先级为从左向右。本例中usbd的驱动优先考虑“nxp，lpc3220-udc”驱动；若没有“fsl，aips-bus”驱动，则用字符串“simple-bus”来继续寻找合适的驱动。

3、interrupt属性

设备节点通过interrupt-parent来指定它所依附的中断控制器，当节点没有指定interrupt-parent时，则从parent节点中继承。上面例子中，root节点的interrupt-parent = <&mic>。这里使用了引用，即mic引用了②中的inrerrupt-controller @40008000；root节点的子节点并没有指定interrupt-controller，如ahb、fab，它们均使用从根节点继承过来的mic，即位于0x40008000的中断控制器。

若子节点使用到中断(中断号、触发方法等等)，则需用interrupt属性来指定，该属性的数值长度受中断控制器中#inrerrupt-controller值③控制，即interrupt属性<>中数值的个数为#inrerrupt-controller的值；本例中#inrerrupt-controller=<2>，因而④中interrupts的值为<0x3d 0>形式，具体每个数值的含义由驱动实现决定。

4、ranges属性

ranges属性为地址转换表，这在pcie中使用较为常见，它表明了该设备在到parent节点中所对应的地址映射关系。ranges格式长度受当前节点#address-cell、parent节点#address-cells、当前节点#size-cell所控制。顺序为ranges=<前节点#address-cell, parent节点#address-cells , 当前节点#size-cell。在本例中，当前节点#address-cell=<1>,对应于⑤中的第一个0x20000000；parent节点#address-cells=<1>，对应于⑤中的第二个0x20000000；当前节点#size-cell=<1>，对应于⑤中的0x30000000。即ahb0节点所占空间从0x20000000地址开始，对应于父节点的0x20000000地址开始的0x30000000地址空间大小。

注：对于相同名称的节点，dtc会根据定义的先后顺序进行合并，其相同属性，取后定义的那个。

**节点例子**

**3.1 chosen node**

chosen {

bootargs = "tegraid=40.0.0.00.00 vmalloc=256M video=tegrafb console=ttyS0,115200n8 earlyprintk";

};﻿​

**解释**

chosen node 主要用来描述由系统指定的runtime parameter，它并没有描述任何硬件设备节点信息。原先通过tag list传递的一些linux kernel运行的参数，可以通过chosen节点来传递。如command line可以通过bootargs这个property来传递。如果存在chosen node，它的parent节点必须为“/”根节点。

**3.2 aliases node**

aliases {

i2c6 = &pca9546\_i2c0;

i2c7 = &pca9546\_i2c1;

i2c8 = &pca9546\_i2c2;

i2c9 = &pca9546\_i2c3;

};﻿​

**解释**

aliases node用来定义别名，类似C++中引用。上面是一个在.dtsi中的典型应用，当使用i2c6时，也即使用pca9546\_i2c0，使得引用节点变得简单方便。例：当.dts include 该.dtsi时，将i2c6的status属性赋值为okay，则表明该主板上的pca9546\_i2c0处于enable状态；反之，status赋值为disabled，则表明该主板上的pca9546\_i2c0处于disenable状态。如下是引用的具体例子：

&i2c6 {

status = "okay";

};﻿​

**3.3 memory node**

memory {

device\_type = "memory";

reg = <0x00000000 0x20000000> /\* 512 MB \*/

};﻿​

**解释**

对于memory node，device\_type必须为memory，由之前的描述可以知道该memory node是以0x00000000为起始地址，以0x20000000为结束地址的512MB的空间。

一般而言，在.dts中不对memory进行描述，而是通过bootargs中类似521M@0x00000000的方式传递给内核。

**5、u-boot操作设备树文件dtb：**

参照前面的启动顺序分析。

九、[ramdisk(initrd)](https://blog.csdn.net/ruixj/article/details/3772752)

在Linux中可以将一部分内存mount为分区来使用，通常称之为RamDisk，分为：Ramdisk, ramfs, tmpfs。因此initrd是ramdisk的一种，用于linux的开机启动。下面解释initrd，其与u-boot启动内核有关。

linux在启动之初，由于硬盘驱动存储在硬盘中的根文件系统中，而没有硬盘驱动式无法挂载硬盘的，这就构成矛盾。为了解决这个矛盾，于是出现了initrd(boot loader initialized RAM disk)。

initrd是一个被压缩过的小型根目录，这个目录中包含了启动阶段中必须的驱动模块，可执行文件和启动脚本。

当系统启动的时候，booloader会把initrd文件读到内存中，然后把initrd的起始地址告诉内核。内核在运行过程中会解压initrd，然后把 initrd挂载为根目录，然后执行根目录中的/initrc脚本，您可以在这个脚本中运行initrd中的udevd，让它来自动加载设备驱动程序以及 在/dev目录下建立必要的设备节点。在udevd自动加载磁盘驱动程序之后，就可以mount真正的根目录，并切换到这个根目录中。

您可以通过下面的方法来制作一个initrd文件。

# dd if=/dev/zero of=initrd.img bs=4k count=1024

# mkfs.ext2 -F initrd.img

# mount -o loop initrd.img /mnt

# cp -r miniroot/\* /mnt

# umount /mnt

# gzip -9 initrd.img

通过上面的命令，我们制作了一个4M的initrd，其中miniroot就是一个根目录。最后我们得到一个名为initrd.img.gz的压缩文件。

利用initrd内核在启动阶段可以顺利的加载设备驱动程序，然而initrd存在以下缺点：

1、initrd大小是固定的，例如上面的压缩之前的initrd大小是4M(4k\*1024)，假设您的根目录(上例中的miniroot/)总大小仅仅是 1M，它仍然要占用4M的空间。如果您在dd阶段指定大小为1M，后来发现不够用的时候，必须按照上面的步骤重新来一次。

2、initrd是一个虚拟的块设备，在上面的例子中，您可是使用fdisk对这个虚拟块设备进行分区。在内核中，对块设备的读写还要经过缓冲区管理模块，也 就是说，当内核读取initrd中的文件内容时，缓冲区管理层会认为下层的块设备速度比较慢，因此会启用预读和缓存功能。这样initrd本身就在内存 中，同时块设备缓冲区管理层还会保存一部分内容。

为了避免上述缺点，于是出现了initramfs，它的作用和initrd类似，您可以使用下面的方法来制作一个initramfs：

# find miniroot/ | cpio -c -o > initrd.img

# gzip initrd.img

这样得到的initrd.img大小是可变的，它取决于您的小型根目录miniroot/的总大小，由于首选使用cpio把根目录进行打包， 因此这个initramfs又被称为cpio initrd. 在系统启动阶段，**bootloader除了从磁盘上机制内核镜像bzImage之外，还要加载initrd.img.gz，然后把initrd.img.gz 的起始地址传递给内核。**能不能把这两个文件合二为一呢？答案是肯定的（**注意：即使二者合二为一，initrd.img.gz也是由bootloader加载的**），在Linux 2.6的内核中，可以把initrd.img.gz链接到内核文件(ELF格式)的一个特殊的数据段中，这个段的名字为.init.ramfs。其中全局 变量\_\_initramfs\_start和\_\_initramfs\_end分别指向这个数据段的起始地址和结束地址。内核启动时会对.init.ramfs段中的数据进行解压，然后使用它作为临时的根文件系统。别看这个过程复杂，您只需要在make menuconfig中配置以下选项就可以了：

General setup --->

[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support

(../miniroot/) Initramfs source file(s)

其中../miniroot/就是我们的小型根目录。这样就只需要一个内核镜像文件就可以了。 内核在启动过程中，必须对以下几种情况进行处理：

1、如果.init.ramfs数据段大小不为0(initramfs\_end - initramfs\_start != 0)，就说明这是initrd集成在内核数据段中。并且是cpio的initrd.

2、initrd是由bootloader加载到内存中的，这时bootloader会把起始地址和结束地址传递给内核，内核中的全局 initrd\_start和initrd\_end分别指向initrd的起始地址和结束地址。现在内核还需要判断这个initrd是新式的cpio格式的 initrd还是旧的initrd.

十、总结：

对于u-boot主要理解：

1、imx6的上电启动流程，u-boot.imx和u-boot.bin的区别，前4k代码的信息，ROM启动程序如何通过前4k代码提供的信息来确定u-boot应该搬运到SDRAM中的哪个位置。

2、u-boot代码的执行过程,如何裁剪board\_init\_f和board\_init\_r,打印出两个函数调用的函数指针数组，使用get\_timer获取每个函数执行的时间。针对时间较大的函数进行裁剪。

3、环境变量的重定位到内存中的哈希表

4、内存的总体分配方式，

5、u-boot如何修改设备树文件，使用设备树向内核传递一些变量，比如：bootarg

6、u-boot中命令的实现方式，bootz命令的启动过程

board\_init\_f:

print\_cpuinfo

show\_board\_info

init\_func\_i2c

board\_init\_r:

stdio\_add\_devices,去掉打印u-boot启动的log

u-boot的驱动：<https://www.chromium.org/chromium-os/firmware-porting-guide/u-boot-drivers>

参考博客：

<http://unicornx.github.io/2015/12/26/20151226-transfer-params-btw-uboot-and-kernel/>

<https://blog.csdn.net/fengyuwuzu0519/article/details/79560249>

<https://blog.csdn.net/metersun/article/details/52584242>

<https://blog.csdn.net/kris_fei/article/details/50474002>

<https://blog.csdn.net/u013904227/article/details/51648179#chatqa>

<https://blog.csdn.net/davidsky11/article/details/25162575>

<https://blog.csdn.net/qq_38144425/article/details/73460095>

<https://blog.csdn.net/qq_29350001/article/details/51481227>