## **一、s5pv210 上电启动的几个阶段**

**1、BL0**

运行在IROM上！！！

加载说明

上电后启动的第一个阶段。其代码固化在s5pv210的IROM中，也就是出厂后已经完成，无法进行修改。上电后CPU会直接从IROM上的BL0的代码上开始执行。

**主要工作有**：

初始化系统时钟、特殊设备的控制器、启动设备、看门狗、堆栈、SRAM等等

验证BL1镜像

从存储介质上（比如SD\eMMC\nand flash）或者通过USB加载BL1镜像到对应内部SRAM上

跳转到BL1镜像所在的地址上

**2、BL1**

运行在内部SRAM上！！！

加载说明

上电后启动的第二个阶段。BL0阶段会将其镜像或者代码从存储介质上（比如SD\eMMC\nand flash）上加载到内部SRAM上.

**主要工作有：**

初始化部分时钟（和SDRAM相关）

初始化DDR（外部SDRAM）

从存储介质上（比如SD\eMMC\nand flash）将BL2镜像加载到SDRAM上

验证BL2镜像的合法性

跳转到BL2镜像所在的地址上

**3、BL2**

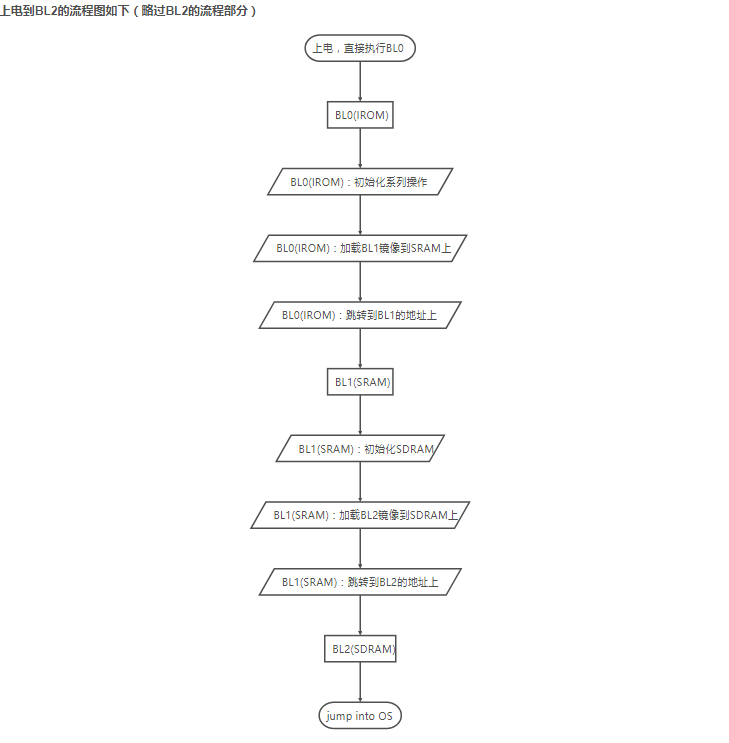
运行在外部SDRAM上！！！

加载说明

上电后启动的第三个阶段，BL1阶段会将其镜像或者代码从存储介质上（比如SD\eMMC\nand flash）上加载到外部SDRAM上.

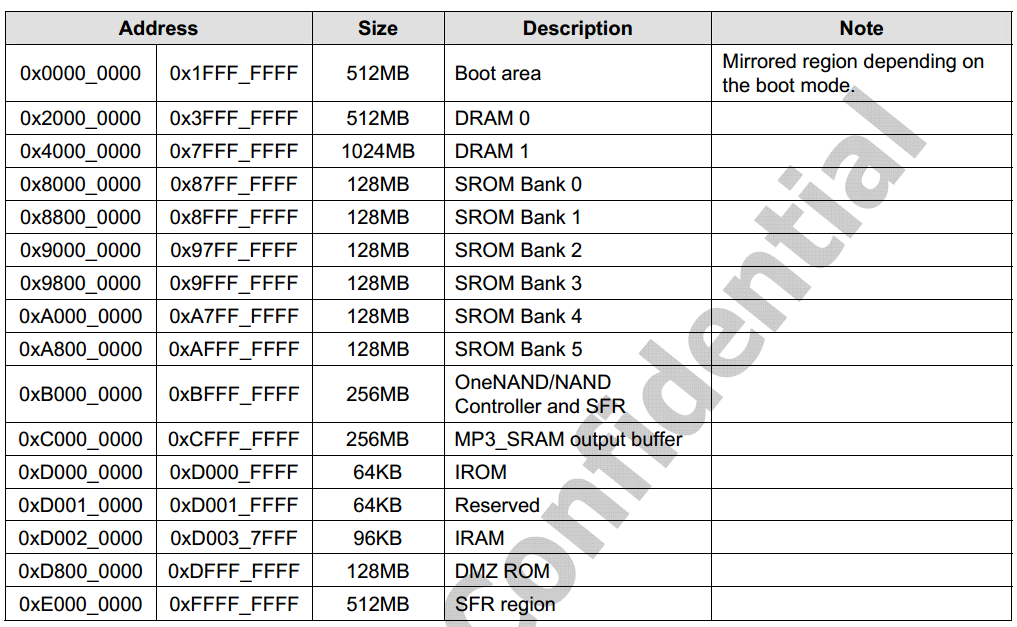
主要功能

BL2就是指传统意义上的bootloader，也就是我们这里的uboot的主体，负责flash操作、uboot命令操作等等，并且最终目标是加载OS和启动OS，如linux。关于uboot的实现，会在后面具体分析，这里不详细说明。

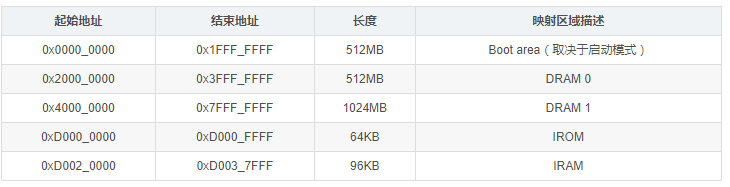


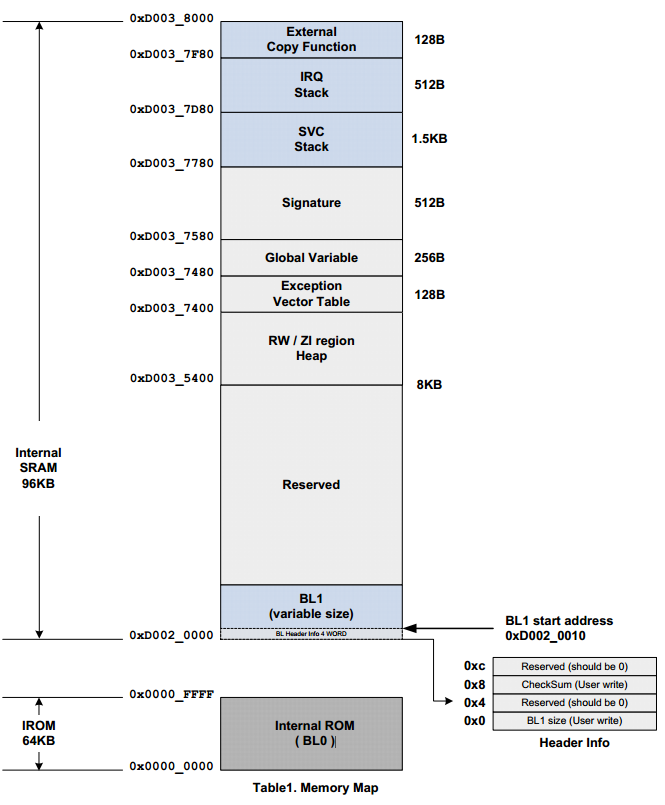
## **二、各个阶段在地址空间上的运行位置**

### 1，**s5pv210地址空间**



重点关注如下几个地址空间：





### **2、BL0的运行地址**



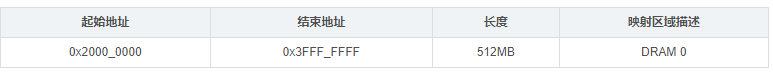
### **3、BL1的运行地址**



### **4、BL2的运行地址**



TQ210的ddr是512M并且使用的是DRAM0的地址空间，故BL2的运行空间是



但是在实际使用中，我把uboot加载到了ddr上0x2000\_0000（具体参考CONFIG\_SYS\_TEXT\_BASE的定义）的地址上。

**三、启动模式（如何将对应BL1加载到对应RAM上面）**

在上述中，我们知道BL0负责将BL1的镜像加载到对应的地址上。

那么，有一个问题，BL1的镜像可能存放在SD卡，eMMC，Nand flash等存储介质上，也有可能是直接通过USB上写入到对应RAM的地址上。那么，BL0如何判断要从哪里去加载BL1的镜像呢，这就取决于启动模式了。

**1、s5pv210支持的启动模式**

OneNand Boot模式

在这种模式下，BL0会从OneNAND的page0开始加载，先加载16B的header到0xD002\_0000，通过header中的size判断需要读取多少page，然后再加载剩余的page到0xD002\_0010上。

最后计算除header外的数据的校验和，和header中的校验和比较，一旦匹配，则跳转到0xD002\_0010。

因此，当我们选择这种方式的时候，需要给BL1的镜像加上16B的header，然后以OneNand page0位置开始存放整个镜像。

Nand Boot (with H/W 8/16-Bit ECC)

在这种模式下，BL0会从NAND的page0开始加载，先加载16B的header到0xD002\_0000，通过header中的size判断需要读取多少page，然后再加载剩余的page到0xD002\_0010上。

最后计算除header外的数据的校验和，和header中的校验和比较，一旦匹配，则跳转到0xD002\_0010。

因此，当我们选择这种方式的时候，需要给BL1的镜像加上16B的header，然后以Nand page0位置开始存放整个镜像。

SD / MMC Boot

在这种模式下，BL0会跳过SD的block0，从SD的block1加载，先加载block1到0xD002\_0000，通过header中的size判断需要读取多少block，然后再加载剩余的block。

最后计算除header外的数据的校验和，和header中的校验和比较，一旦匹配，则跳转到0xD002\_0010。

因此，当我们选择这种方式的时候，需要给BL1的镜像加上16B的header，然后以block1位置开始存放整个镜像。

eSSD Boot

在这种模式下，BL0会跳过eSSD的block0，从eSSD的block1加载，先加载block1到0xD002\_0000，通过header中的size判断需要读取多少block，然后再加载剩余的block。

最后计算除header外的数据的校验和，和header中的校验和比较，一旦匹配，则跳转到0xD002\_0010。

因此，当我们选择这种方式的时候，需要给BL1的镜像加上16B的header，然后以block1位置开始存放整个镜像。

eMMC Boot

在这种模式下，BL0会从eMMC的block0加载，先加载block0到0xD002\_0000，通过header中的size判断需要读取多少block，然后再加载剩余的block。

最后计算除header外的数据的校验和，和header中的校验和比较，一旦匹配，则跳转到0xD002\_0010。

因此，当我们选择这种方式的时候，需要给BL1的镜像加上16B的header，然后以block0位置开始存放整个镜像。

UART Boot

在这种模式下，BL0会检测串口数据，并且将其复制到0xD002\_0000开始的位置。在PC侧通过DWN软件将镜像通过串口进行写入。

因为这种模式下不需要验证header，一旦下载完成之后，会直接跳转到0xD002\_0000进入BL1。

USB Boot

在这种模式下，BL0会检测USB数据，并且将其复制到0xD002\_0000开始的位置。在PC侧通过DWN软件将镜像通过USB进行写入。

因为这种模式下不需要验证header，一旦下载完成之后，会直接跳转到0xD002\_0000进入BL1。



### **2、s5pv210选择启动模式的方法**



# uboot流程——概述

**一、bootloader & uboot**

**1、bootloader的概念**

　Bootloader是在操作系统运行之前执行的一段小程序。而这段小程序的最终目的，正确地设置好软硬件环境，使之能够成功地引导操作系统。

**2、bootloader的核心功能**

bootloader的核心功能就是引导操作系统，部分工作如下

初始化部分硬件，包括时钟、内存等等

加载内核到内存上

加载文件系统、atags或者dtb到内存上

根据操作系统启动要求正确配置好一些硬件

启动操作系统

**3、bootloader的monitor功能**

上述2是bootloader的核心功能，也就是引导操作系统的功能。

但是部分bootloader还支持monitor功能，提供了更多的命令行接口，具体部分功能如下：

进行调试

读写内存

烧写Flash

配置环境变量

命令引导操作系统

**4、嵌入式几种常见的bootloader**

uboot

这也是最常见的bootloader，开源，常用于ARM,MIPS等平台。

支持monitor功能，也是在项目project X中，使用的bootloader

所以后续两节会针对这个bootloader进行说明

superboot

不开源，友善之臂的tiny210代码中默认使用这个bootloader

LK(Little Kernel)

常用于高通平台，支持monitor功能。

**二、uboot-spl & uboot**

**1、uboot-spl**

由uboot编译生成，对应于BL1阶段，也就是BL1的镜像，uboot-spl.bin。

根据《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》，其代码运行于IRAM中

主要工作有：

初始化部分时钟（和SDRAM相关）

初始化DDR（外部SDRAM）

从存储介质上（比如SD\eMMC\nand flash）将BL2镜像加载到SDRAM上

验证BL2镜像的合法性

跳转到BL2镜像所在的地址上

后续会从编译和代码流程两方面来介绍uboot-spl。

对应文章：

《[uboot] （第二章）uboot流程——uboot-spl编译流程》

《[uboot] （第三章）uboot流程——uboot-spl代码流程》

**2、uboot**

由uboot编译生成，对应于BL2阶段，也就是BL2的镜像,uboot.bin。

根据《[project X] tiny210(s5pv210)上电启动流程（BL0-BL2）》，其代码运行于SDRAM中.

主要工作有：

初始化部分硬件，包括时钟、内存等等

加载内核到内存上

加载文件系统、atags或者dtb到内存上

根据操作系统启动要求正确配置好一些硬件

启动操作系统

monitor工作，主要是处理命令行的命令，以下是部分操作：

flash操作

环境变量操作

启动操作

后续会从编译、代码整体流程以及部分功能的具体流程来介绍uboot-spl。

# uboot流程——uboot编译流程（u-boot201701）

## **一、uboot编译和生成文件**

### **1、编译方法**

make smdkv210\_defconfig

make

### **生成文件**

最终编译完成之后，会在u-boot-2017.01下生成如下文件：

api config.mk dts Kconfig net System.map u-boot.cfg u-boot.map

arch configs examples lib post test u-boot.cfg.configs u-boot-nodtb.bin

board disk fs Licenses README tools u-boot.dtb u-boot.srec

cmd doc include MAINTAINERS scripts u-boot u-boot-dtb.bin u-boot.sym

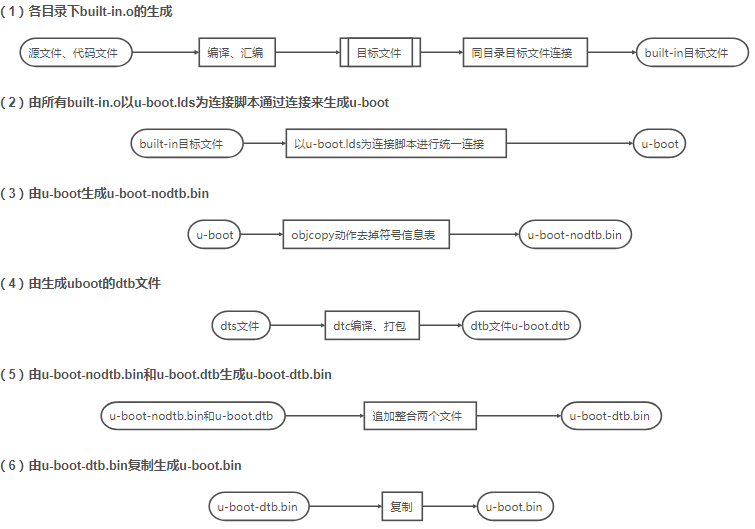
common drivers Kbuild Makefile snapshot.commit u-boot.bin u-boot.lds

其中，arch、common、dts、include、board、drivers、fs等等目录是对应代码的编译目录，****各个目录下都会生成相应的built.o，是由同目录下的目标文件连接而成。****   
重点说一下以下几个文件：



## **二、uboot编译流程**

### **1、编译整体流程**



### **2、具体编译流程分析**

我们直接从make命令分析，也就是从uboot下的Makefile的依赖关系来分析整个编译流程。   
注意，这个分析顺序和上述的整体编译流程的顺序是反着的。

1. ****入口分析****

all: $(ALL-y) @ALL-y=checkarmreloc u-boot.srec u-boot.bin u-boot.sym System.map binary\_size\_check u-boot.dtb

u-boot.bin就是我们的目标，所以后需要主要研究u-boot.bin的依赖关系。

****（2）u-boot.bin的依赖关系****

ifeq ($(CONFIG\_OF\_SEPARATE),y) @*CONFIG\_OF\_SEPARATE用于定义是否有DTB并且是否是和uboot分开编译的。CONFIG\_OF\_SEPARATE=y,所以编译此段。*

u-boot-dtb.bin: u-boot-nodtb.bin dts/dt.dtb FORCE @*由u-boot-nodtb.bin和dts/dt.dtb连接在一起，先生成u-boot-dtb.bin*

$(call if\_changed,cat)*@$(call if\_changed,cat)会调用到cmd\_cat函数，具体实现我们不分析了*

u-boot.bin: u-boot-dtb.bin FORCE *@直接将u-boot-dtb.bin复制为u-boot.bin*

$(call if\_changed,copy) *@$(call if\_changed,copy)会调用到cmd\_copy函数，具体实现我们不分析了*

else

u-boot.bin: u-boot-nodtb.bin FORCE

$(call if\_changed,copy)

endif

****（3）u-boot-nodtb.bin的依赖关系****

****u****-boot-nodtb.bin: u-boot FORCE

$(call if\_changed,objcopy)

$(call DO\_STATIC\_RELA,$<,$@,$(CONFIG\_SYS\_TEXT\_BASE))

$(BOARD\_SIZE\_CHECK)

*@$(call if\_changed,objcopy)表示当依赖文件发生变化时，将依赖文件经过objcopy处理之后得到目标文件。也就是通过objcopy把u-boot的符号信息以及一些无用信息去掉之后，得到了u-boot-nodtb.bin。*

如上述Makefile代码u-boot-nodtb.bin依赖于u-boot，并且由u-boot经过objcopy操作之后得到。

****（4）u-boot的依赖关系****

u-boot: $(u-boot-init) $(u-boot-main) u-boot.lds FORCE

$(call if\_changed,u-boot\_\_) *@$(call if\_changed,u-boot\_\_)来生成目标，$(call if\_changed,u-boot\_\_)对应cmd\_u-boot\_\_命令*

ifeq ($(CONFIG\_KALLSYMS),y)

$(call cmd,smap)

$(call cmd,u-boot\_\_) common/system\_map.o

endif

如上，u-boot依赖于$(u-boot-init) 、$(u-boot-main)和u-boot.lds，并且最终会调用cmd\_u-boot\_\_来生成u-boot。

u-boot-init=$(u-boot-init)=arch/arm/cpu/armv7/start.o

u-boot-main=$(u-boot-main)=arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/lib/built-in.o arch/arm/mach-s5pc1xx/built-in.o board/samsung/common/built-in.o board/samsung/smdkv210/built-in.o cmd/built-in.o common/built-in.o disk/built-in.o drivers/built-in.o drivers/dma/built-in.o drivers/gpio/built-in.o drivers/i2c/built-in.o drivers/mmc/built-in.o drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o drivers/mtd/spi/built-in.o drivers/net/built-in.o drivers/net/phy/built-in.o drivers/pci/built-in.o drivers/power/built-in.o drivers/power/battery/built-in.o drivers/power/domain/built-in.o drivers/power/fuel\_gauge/built-in.o drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o drivers/power/regulator/built-in.o drivers/serial/built-in.o drivers/spi/built-in.o drivers/usb/common/built-in.o drivers/usb/dwc3/built-in.o drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o drivers/usb/ulpi/built-in.o fs/built-in.o lib/built-in.o net/built-in.o test/built-in.o test/dm/built-in.o

cmd\_u-boot\_\_实现如下：

cmd\_u-boot\_\_ ?= $(LD) $(LDFLAGS) $(LDFLAGS\_u-boot) -o $@ \

-T u-boot.lds $(u-boot-init) \

--start-group $(u-boot-main) --end-group \

$(PLATFORM\_LIBS) -Map u-boot.map

将cmd\_u-boot\_\_通过echo命令打印出来之后得到如下（拆分出来看的）：

**cmd\_u-boot\_\_**=arm-linux-gnueabihf-ld.bfd

-pie --gc-sections -Bstatic -Ttext 0x34800000

-o u-boot

-T u-boot.lds

arch/arm/cpu/armv7/start.o --start-group

arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/lib/built-in.o arch/arm/mach-s5pc1xx/built-in.o board/samsung/common/built-in.o board/samsung/smdkv210/built-in.o cmd/built-in.o common/built-in.o disk/built-in.o drivers/built-in.o drivers/dma/built-in.o drivers/gpio/built-in.o drivers/i2c/built-in.o drivers/mmc/built-in.o drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o drivers/mtd/spi/built-in.o drivers/net/built-in.o drivers/net/phy/built-in.o drivers/pci/built-in.o drivers/power/built-in.o drivers/power/battery/built-in.o drivers/power/domain/built-in.o drivers/power/fuel\_gauge/built-in.o drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o drivers/power/regulator/built-in.o drivers/serial/built-in.o drivers/spi/built-in.o drivers/usb/common/built-in.o drivers/usb/dwc3/built-in.o drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o drivers/usb/ulpi/built-in.o fs/built-in.o lib/built-in.o net/built-in.o test/built-in.o test/dm/built-in.o --end-group

arch/arm/lib/eabi\_compat.o arch/arm/lib/lib.a -Map u-boot.map

可以看出上述是一条连接命令，以u-boot.lds为链接脚本，把$(u-boot-init) 、$(u-boot-main)的指定的目标文件连接到u-boot中。 ****并且已经指定输出文件为u-boot，连接脚本为u-boot.lds。****

连接很重要的东西就是连接标识，也就是 $(LD) $(LDFLAGS) $(LDFLAGS\_u-boot)的定义。   
尝试把$(LD) 、$(LDFLAGS) 、$(LDFLAGS\_u-boot)) 打印出来，结果如下：

LD=arm-linux-gnueabihf-ld.bfd

LDFLAGS=

LDFLAGS\_u-boot=-pie --gc-sections -Bstatic -Ttext 0x34800000

LDFLAGS\_u-boot定义如下：

LDFLAGS\_u-boot += $(LDFLAGS\_FINAL) *@LDFLAGS\_FINAL= --gc-sections -Bstatic*

ifneq ($(CONFIG\_SYS\_TEXT\_BASE),) *@CONFIG\_SYS\_TEXT\_BASE=0x34800000*

LDFLAGS\_u-boot += -Ttext $(CONFIG\_SYS\_TEXT\_BASE)

endif

## 当指定CONFIG\_SYS\_TEXT\_BASE时，会配置连接地址。在tiny210项目中，定义如下：

## include/configs/tiny210.h:52:#define CONFIG\_SYS\_TEXT\_BASE 0x23E00000

## $(LDFLAGS\_FINAL)在如下几个地方定义了

## ./config.mk:19:LDFLAGS\_FINAL :=

## ./config.mk:80:LDFLAGS\_FINAL += -Bstatic

## ./arch/arm/config.mk:16:LDFLAGS\_FINAL += --gc-sections

## 通过上述LDFLAGS\_u-boot=-pie --gc-sections -Bstatic -Ttext 0x23E00000也就可以理解了

## 对应于上述二、1（2）流程。

1. ****u-boot-init & u-boot-main依赖关系（代码是如何被编译的）****

先看一下这两个值打印出来的

u-boot-init=$(u-boot-init)=arch/arm/cpu/armv7/start.o

u-boot-main=$(u-boot-main)=arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o arch/arm/lib/built-in.o arch/arm/mach-s5pc1xx/built-in.o board/samsung/common/built-in.o board/samsung/smdkv210/built-in.o cmd/built-in.o common/built-in.o disk/built-in.o drivers/built-in.o drivers/dma/built-in.o drivers/gpio/built-in.o drivers/i2c/built-in.o drivers/mmc/built-in.o drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o drivers/mtd/spi/built-in.o drivers/net/built-in.o drivers/net/phy/built-in.o drivers/pci/built-in.o drivers/power/built-in.o drivers/power/battery/built-in.o drivers/power/domain/built-in.o drivers/power/fuel\_gauge/built-in.o drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o drivers/power/regulator/built-in.o drivers/serial/built-in.o drivers/spi/built-in.o drivers/usb/common/built-in.o drivers/usb/dwc3/built-in.o drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o drivers/usb/ulpi/built-in.o fs/built-in.o lib/built-in.o net/built-in.o test/built-in.o test/dm/built-in.o

可以观察到是一堆目标文件的路径。这些目标文件最终都要被连接到u-boot中。

 u-boot-init & u-boot-main的定义如下代码

u-boot-init := $(head-y)

u-boot-main := $(libs-y)

*## head-y定义在如下位置*

arch/arm/Makefile:head-y := arch/arm/cpu/$(CPU)/start.o *@CPU=armv7*

那么u-boot-init & u-boot-main是如何生成的呢？   
需要看一下对应的依赖如下：

$(sort $(u-boot-init) $(u-boot-main)): $(u-boot-dirs) ;

*## 也就是说$(u-boot-init) $(u-boot--main)依赖于$(u-boot-dirs)*

*## sort函数根据首字母进行排序并去除掉重复的。*

*##u-boot-dirs := $(patsubst %/,%,$(filter %/, $(libs-y))) tools examples*

*## $(filter %/, $(libs-y)过滤出'/'结尾的字符串，注意，此时$(libs-y)的内容还没有加上built-in.o文件后缀*

*## patsubst去掉字符串中最后的'/'的字符。*

*## 最后u-boot-dirs打印出来如下：*

*##u-boot-dirs=arch/arm/cpu arch/arm/cpu/armv7 arch/arm/lib arch/arm/mach-s5pc1xx board/samsung/common board/samsung/smdkv210 cmd common disk drivers drivers/dma drivers/gpio drivers/i2c drivers/mmc drivers/mtd drivers/mtd/onenand drivers/mtd/spi drivers/net drivers/net/phy drivers/pci drivers/power drivers/power/battery drivers/power/domain drivers/power/fuel\_gauge drivers/power/mfd drivers/power/pmic drivers/power/regulator drivers/serial drivers/spi drivers/usb/common drivers/usb/dwc3 drivers/usb/emul drivers/usb/eth drivers/usb/gadget drivers/usb/gadget/udc drivers/usb/host drivers/usb/musb-new drivers/usb/musb drivers/usb/phy drivers/usb/ulpi fs lib net test test/dm tools examples*

u-boot-dirs依赖规则如下：

PHONY += $(u-boot-dirs)

$(u-boot-dirs): prepare scripts

$(Q)$(MAKE) $(build)=$@ *@Q=@ MAKE=make -f ./scripts/Makefile.build obj即@make -f ./scripts/Makefile.build obj=$@*

*## 依赖于prepare scripts*

*## prepare会导致prepare0、prepare1、prepare2、prepare3目标被执行，最终编译了tools目录下的东西，生成了一些工具*

*## 然后执行$(Q)$(MAKE) $(build)=$@*

*## 也就是会对每一个目标文件依次执行make $(build)=目标文件*

对每一个目标文件依次执行make $(build)=目标文件

$(build)定义如下：u-boot-2017.01/scripts/Kbuild.include

build := -f $(srctree)/scripts/Makefile.build obj

以arch/arm/mach-s5pc1xx为例   
“$(Q)$(MAKE) $(build)=$@”展开后格式如下   
@make -f ./scripts/Makefile.build obj=arch/arm/mach-s5pc1xx 如果make执行的命令前面加了@字符，则不显示命令本身而只显示它的结果

Makefile.build定义built-in.o、.lib以及目标文件.o的生成规则。这个Makefile文件生成了子目录的.lib、built-in.o以及目标文件.o。 

Makefile.build第一个编译目标是\_\_build，如下:

PHONY := \_\_build

\_\_build:

*## 所以会直接编译执行\_\_build这个目标，其依赖如下*

\_\_build: $(if $(KBUILD\_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \

$(if $(KBUILD\_MODULES),$(obj-m) $(modorder-target)) \

$(subdir-ym) $(always)

@:

*## 和built-in.o相关的是依赖builtin-target。下面来看这个依赖。*

*KBUILD\_BUILTIN=1 KBUILD\_MODULES=*

builtin-target := $(obj)/built-in.o

*## 以obj=arch/arm/mach-s5pc1xx为例，那么builtin-target就是arch/arm/mach-s5pc1xx/built-in.o*

*## 依赖关系如下：*

$(builtin-target): $(obj-y) FORCE

$(call if\_changed,link\_o\_target)

*## $(call if\_changed,link\_o\_target)将所有依赖连接到$(builtin-target)，也就是相应的built-in.o中了。*

*## 具体实现可以查看cmd\_link\_o\_target的实现，这里不详细说明了。*

*## 那么$(obj-y)是从哪里来的呢？是从相应目录下的Makefile中include得到的。*

*## The filename Kbuild has precedence(*优先*) over Makefile*

kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))

kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)

include $(kbuild-file)

*## 当obj=arch/arm/mach-s5pc1xx时,得到对应的kbuild-file=u-boot-2017.01/arch/arm/mach-s5pc1xx/Makefile*

*## 而在u-boot-2017.01/arch/arm/mach-s5pc1xx/Makefile中定义了obj-y如下：*

*## obj-y = cache.o*

*## obj-y += reset.o*

*## obj-y += clock.o*

*## 对应obj-y对应一些目标文件，由C文件编译而来，这里就不说明了。*

后面来看目标文件的编译流程

u-boot-2017.01/scripts/Makefile.build

*# Built-in and composite(合成，复合) module parts*

$(obj)/%.o: $(src)/%.c $(recordmcount\_source) FORCE

$(call cmd,force\_checksrc)

$(call if\_changed\_rule,cc\_o\_c) *@## 调用cmd\_cc\_o\_c对.c文件进行编译*

*##* *cmd\_cc\_o\_c格式如下：*

cmd\_cc\_o\_c = $(CC) $(c\_flags) -c -o $@ $<

*## $(CC) $(c\_flags)打印出来如下：*

*CC=arm-linux-gnueabihf-gcc*

*c\_flags=-Wp,-MD,..d -nostdinc -isystem /usr/local/arm/7.3.1/bin/../lib/gcc/arm-linux-gnueabihf/7.3.1/include -Iinclude -I./arch/arm/include -include ./include/linux/kconfig.h -D\_\_KERNEL\_\_ -D\_\_UBOOT\_\_ -Wall -Wstrict-prototypes -Wno-format-security -fno-builtin -ffreestanding -Os -fno-stack-protector -fno-delete-null-pointer-checks -g -fstack-usage -Wno-format-nonliteral -Werror=date-time -D\_\_ARM\_\_ -marm -mno-thumb-interwork -mabi=aapcs-linux -mword-relocations -fno-pic -mno-unaligned-access -ffunction-sections -fdata-sections -fno-common -ffixed-r9 -msoft-float -pipe -march=armv7-a -D\_\_LINUX\_ARM\_ARCH\_\_=7 -I./arch/arm/mach-s5pc1xx/include -D"KBUILD\_STR(s)=#s" -D"KBUILD\_BASENAME=KBUILD\_STR()"*

对应于上述二、1（1）流程。

1. ****u-boot.lds依赖关系****

这里主要是为了找到一个匹配的连接文件。

*## u-boot-2017.01/Makefile*

u-boot.lds: $(LDSCRIPT) prepare FORCE

$(call if\_changed\_dep,cpp\_lds)

*# If there is no specified link script, we look in a number of places for it*

ifndef LDSCRIPT

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/$(CPUDIR)/u-boot.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/arch/$(ARCH)/cpu/u-boot.lds

endif

endif

*## 也就是说依次从board/板级目录、cpudir目录、arch/架构/cpu/目录下去搜索u-boot.lds文件。*

*## 例如，tq210(s5vp210 armv7)最终会在./arch/arm/cpu/下搜索到u-boot.lds LDSCRIPT=./arch/arm/cpu/u-boot.lds*

综上，最终指定了u-boot-2017.01/arch/arm/cpu/u-boot.lds作为连接脚本。

1. ****dts/dt.dtb依赖关系****

该依赖关系的主要目的是生成dtb文件。   
首先了解dts文件被放在了arch/arm/dts里面，并通过dts下的Makefile进行选择。   
Makefile如下（剪切出一部分）

u-boot-2017.01/arch/arm/dts/Makefile

dtb-$(CONFIG\_S5PC110) += s5pc1xx-goni.dtb

dtb-$(CONFIG\_EXYNOS4) += exynos4210-origen.dtb \

exynos4210-smdkv310.dtb \

exynos4210-universal\_c210.dtb \

exynos4210-trats.dtb \

exynos4412-trats2.dtb \

exynos4412-odroid.dtb

*## 填充选择dtb-y*

targets += $(dtb-y)

*# Add any required device tree compiler flags here*

DTC\_FLAGS +=

*## 用于添加DTC编译选项*

PHONY += dtbs

dtbs: $(addprefix $(obj)/, $(dtb-y)) *@$(addprefix $(obj)/, $(dtb-y))=arch/arm/dts/s5pc1xx-goni.dtb*

@:

*## 伪目标，其依赖为$(dtb-y)加上了源路径，如下*

*## arch/arm/dts/**s5pc1xx-goni.dtb*

*## 后续会使用到这个伪目标*

*u-boot-2017.01/Makefile*

PHONY += dtbs

dtbs: dts/dt.dtb

@:

dts/dt.dtb: checkdtc u-boot

$(Q)$(MAKE) $(build)=dts dtbs

*## checkdtc依赖用于检查dtc的版本*

*## u-boot一旦发生变化那么就重新编译一遍dtb*

*## 重点关注命令 $(Q)$(MAKE) $(build)=dts dtbs*

*## 展开来就是make -f ./scripts/Makefile.build obj=dts dtbs*

*## 我们相当于值在/scripts/Makefile.build下执行了目标dtbs*

在scripts/Makefile.build中dtbs的目标定义在哪里呢

kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)

include $(kbuild-file)

*##kbuild-file=./arch/arm/cpu/armv7/Makefile,包含所有目录下的Makefile进来*

*## 把对应的Makefile路径包含了进去，也就是arch/arm/dts/Makefile*

*## 如前面所说，arch/arm/dts/Makefile中定义了dtbs的目标*

*## dtbs: $(addprefix $(obj)/, $(dtb-y))*

*## @:*

*## 这里我们就找到对应的依赖关系了，依赖就是$(obj)/, $(dtb-y)，举个例子就是arch/arm/dts/s5pc1xx-goni.dtb*

include scripts/Makefile.lib

*## 包含了scripts/Makefile.lib，在编译dts的时候会用到*

*## scripts/Makefile.lib*

$(obj)/%.dtb: $(src)/%.dts FORCE

$(call if\_changed\_dep,dtc)

*## 使用了通配符的方式* *## 这样就通过dtc对dts编译生成了dtb文件*

对应于上述二、1（4）流程。

## **三、一些重点定义**

1、连接标志

在二、2（4）中说明。

连接命令在cmd\_u-boot\_\_中，如下

cmd\_u-boot\_\_ ?= $(LD) $(LDFLAGS) $(LDFLAGS\_u-boot) -o $@ \

-T u-boot.lds $(u-boot-init) \

--start-group $(u-boot-main) --end-group \

$(PLATFORM\_LIBS) -Map u-boot.map

连接标识如下：

LD=arm-linux-gnueabihf-ld.bfd

LDFLAGS=

LDFLAGS\_u-boot=-pie --gc-sections -Bstatic -Ttext 0x34800000

LDFLAGS\_u-boot定义如下：

LDFLAGS\_u-boot += -pie

LDFLAGS\_u-boot += $(LDFLAGS\_FINAL)

ifneq ($(CONFIG\_SYS\_TEXT\_BASE),)

LDFLAGS\_u-boot += -Ttext $(CONFIG\_SYS\_TEXT\_BASE)

endif

****‘-o’指定了输出文件是u-boot，’-T’是指定了连接脚本是当前目录下的u-boot.lds, -Ttext指定了连接地址是CONFIG\_SYS\_TEXT\_BASE。****

****2、连接地址****

在二、2（4）中说明。

CONFIG\_SYS\_TEXT\_BASE指定了u-boot.bin的连接地址。这个地址也就是uboot的起始运行地址。

对于tiny210，其定义如下（可以进行修改）

/include/configs/tiny210.h

#define CONFIG\_SYS\_TEXT\_BASE 0x23E00000

1. ****连接脚本****

在二、2（6）中说明。

u-boot/arch/arm/cpu/u-boot.lds

u-boot.lds: $(LDSCRIPT) prepare FORCE

$(call if\_changed\_dep,cpp\_lds)

ifndef LDSCRIPT

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/$(CPUDIR)/u-boot.lds

endif

ifeq ($(wildcard $(LDSCRIPT)),)

LDSCRIPT := $(srctree)/arch/$(ARCH)/cpu/u-boot.lds

endif

endif

综上，最终指定了arch/arm/cpu/u-boot.lds作为连接脚本。

## **四、uboot链接脚本说明**

### **1、连接脚本整体分析**

相对比较简单，直接看连接脚本的内容arch/arm/cpu/u-boot.lds

#include <config.h>

#include <asm/psci.h>

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")

OUTPUT\_ARCH(arm)

ENTRY(\_start) *@//定义了地址为\_start的地址，所以我们分析代码就是从这个函数开始分析的！！！*

SECTIONS

{

#ifndef CONFIG\_CMDLINE

/DISCARD/ : { \*(.u\_boot\_list\_2\_cmd\_\*) }

#endif

#if defined(CONFIG\_ARMV7\_SECURE\_BASE) && defined(CONFIG\_ARMV7\_NONSEC)

/\*

\* If CONFIG\_ARMV7\_SECURE\_BASE is true, secure code will not

\* bundle with u-boot, and code offsets are fixed. Secure zone

\* only needs to be copied from the loading address to

\* CONFIG\_ARMV7\_SECURE\_BASE, which is the linking and running

\* address for secure code.

\*

\* If CONFIG\_ARMV7\_SECURE\_BASE is undefined, the secure zone will

\* be included in u-boot address space, and some absolute address

\* were used in secure code. The absolute addresses of the secure

\* code also needs to be relocated along with the accompanying u-boot

\* code.

\*

\* So DISCARD is only for CONFIG\_ARMV7\_SECURE\_BASE.

\*/

/DISCARD/ : { \*(.rel.\_secure\*) }

#endif

. = 0x00000000;

*@以下定义文本段*

. = ALIGN(4);

.text :

{

\*(.\_\_image\_copy\_start) *@定义\_\_image\_copy\_start这个标号地址为当前地址*

\*(.vectors) *@所有目标文件的vectors段，也就是中断向量表连接到这里来*

CPUDIR/start.o (.text\*)

\*(.text\*)

}

#ifdef CONFIG\_ARMV7\_NONSEC

/\* Align the secure section only if we're going to use it in situ \*/

.\_\_secure\_start :

#ifndef CONFIG\_ARMV7\_SECURE\_BASE

ALIGN(CONSTANT(COMMONPAGESIZE))

#endif

{

KEEP(\*(.\_\_secure\_start))

}

#ifndef CONFIG\_ARMV7\_SECURE\_BASE

#define CONFIG\_ARMV7\_SECURE\_BASE

#define \_\_ARMV7\_PSCI\_STACK\_IN\_RAM

#endif

.secure\_text CONFIG\_ARMV7\_SECURE\_BASE :

AT(ADDR(.\_\_secure\_start) + SIZEOF(.\_\_secure\_start))

{

\*(.\_secure.text)

}

.secure\_data : AT(LOADADDR(.secure\_text) + SIZEOF(.secure\_text))

{

\*(.\_secure.data)

}

#ifdef CONFIG\_ARMV7\_PSCI

.secure\_stack ALIGN(ADDR(.secure\_data) + SIZEOF(.secure\_data),

CONSTANT(COMMONPAGESIZE)) (NOLOAD) :

#ifdef \_\_ARMV7\_PSCI\_STACK\_IN\_RAM

AT(ADDR(.secure\_stack))

#else

AT(LOADADDR(.secure\_data) + SIZEOF(.secure\_data))

#endif

{

KEEP(\*(.\_\_secure\_stack\_start))

/\* Skip addreses for stack \*/

. = . + CONFIG\_ARMV7\_PSCI\_NR\_CPUS \* ARM\_PSCI\_STACK\_SIZE;

/\* Align end of stack section to page boundary \*/

. = ALIGN(CONSTANT(COMMONPAGESIZE));

KEEP(\*(.\_\_secure\_stack\_end))

#ifdef CONFIG\_ARMV7\_SECURE\_MAX\_SIZE

/\*

\* We are not checking (\_\_secure\_end - \_\_secure\_start) here,

\* as these are the load addresses, and do not include the

\* stack section. Instead, use the end of the stack section

\* and the start of the text section.

\*/

ASSERT((. - ADDR(.secure\_text)) <= CONFIG\_ARMV7\_SECURE\_MAX\_SIZE,

"Error: secure section exceeds secure memory size");

#endif

}

#ifndef \_\_ARMV7\_PSCI\_STACK\_IN\_RAM

/\* Reset VMA but don't allocate space if we have secure SRAM \*/

. = LOADADDR(.secure\_stack);

#endif

#endif

.\_\_secure\_end : AT(ADDR(.\_\_secure\_end)) {

\*(.\_\_secure\_end)

LONG(0x1d1071c); /\* Must output something to reset LMA \*/

}

#endif

. = ALIGN(4);

.rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) }

. = ALIGN(4);

.data : {

\*(.data\*)

}

. = ALIGN(4);

. = .;

. = ALIGN(4);

.u\_boot\_list : {

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

}

. = ALIGN(4);

.\_\_efi\_runtime\_start : {

\*(.\_\_efi\_runtime\_start)

}

.efi\_runtime : {

\*(efi\_runtime\_text)

\*(efi\_runtime\_data)

}

.\_\_efi\_runtime\_stop : {

\*(.\_\_efi\_runtime\_stop)

}

.efi\_runtime\_rel\_start :

{

\*(.\_\_efi\_runtime\_rel\_start)

}

.efi\_runtime\_rel : {

\*(.relefi\_runtime\_text)

\*(.relefi\_runtime\_data)

}

.efi\_runtime\_rel\_stop :

{

\*(.\_\_efi\_runtime\_rel\_stop)

}

. = ALIGN(4);

.image\_copy\_end :

{

\*(.\_\_image\_copy\_end)

}

.rel\_dyn\_start :

{

\*(.\_\_rel\_dyn\_start)

}

.rel.dyn : {

\*(.rel\*)

}

.rel\_dyn\_end :

{

\*(.\_\_rel\_dyn\_end)

}

.end :

{

\*(.\_\_end)

}

\_image\_binary\_end = .;

/\*

\* Deprecated: this MMU section is used by pxa at present but

\* should not be used by new boards/CPUs.

\*/

. = ALIGN(4096);

.mmutable : {

\*(.mmutable)

}

/\*

\* Compiler-generated \_\_bss\_start and \_\_bss\_end, see arch/arm/lib/bss.c

\* \_\_bss\_base and \_\_bss\_limit are for linker only (overlay ordering)

\*/

.bss\_start \_\_rel\_dyn\_start (OVERLAY) : {

KEEP(\*(.\_\_bss\_start));

\_\_bss\_base = .;

}

.bss \_\_bss\_base (OVERLAY) : {

\*(.bss\*)

. = ALIGN(4);

\_\_bss\_limit = .;

}

.bss\_end \_\_bss\_limit (OVERLAY) : {

KEEP(\*(.\_\_bss\_end));

}

.dynsym \_image\_binary\_end : { \*(.dynsym) }

.dynbss : { \*(.dynbss) }

.dynstr : { \*(.dynstr\*) }

.dynamic : { \*(.dynamic\*) }

.plt : { \*(.plt\*) }

.interp : { \*(.interp\*) }

.gnu.hash : { \*(.gnu.hash) }

.gnu : { \*(.gnu\*) }

.ARM.exidx : { \*(.ARM.exidx\*) }

.gnu.linkonce.armexidx : { \*(.gnu.linkonce.armexidx.\*) }

}

重点关注   
\* \_\_image\_copy\_start & \_\_image\_copy\_end   
界定了代码空间的位置，用于重定向代码的时候使用，在uboot relocate的过程中，需要把这部分拷贝到uboot的新的地址空间中，后续在新地址空间中运行。