# 绪论

本文属于作者原创，未经允许，不得传播。

联系方式：jeff-lv-1009080308@outlook.com

开始时间：2018-4-6

更新时间：2018-4-21

复习时间：

参考：

## 概述

本篇文章的主要目的是分析uboot的启动过程，研究其的内幕，并在此基础上可以根据用户定制，对uboot做一定修改。

说明1：源码采用友善之臂提供的tiny4412的uboot，版本2010.12

说明2：文中列出的代码会根据需要，舍去相关部分。

说明3：有些接口使用宏的方式定义，很难看出返回值和传入参数的类型，本篇文章会根据需要，用等价函数进行替换。

## 技巧

编译uboot到其他目录

make tiny4412\_config O=./obj

make O=./obj

make distclean O=./obj

# 配置和编译

参考：<https://blog.csdn.net/wanyong89/article/details/8503036>

<https://blog.csdn.net/groundhappy/article/details/52708792>

<https://blog.csdn.net/qq_38175979/article/details/73724754>

## 概述

对于uboot的分析，我们从Makefile入手。

说明1：uboot从2010.09版本后工程结构发生变化，所有关于板子配置信息都独立出来放在了boards.cfg文件中

## 整体

说明：本节分析的具体源码可以结合编译过程和配置过程来阅读，例如mkconfig可以在阅读完配置过程后再来阅读。

### 体系组成

uboot是一个庞大的工程，需要根据不同的配置，选取不同的源文件，采用不同的工具，最后生成一个可用的bootloader。这个过程依赖于Makefile以及相关配置文件的管理。

#### 原始文件

对于uboot中原始文件，总结如下：

1. 顶层Makefile：整体上控制着uboot的配置、编译。但这个过程是宏观的，相当于决定了具体的方向，具体的实现依赖于config.mk。例如Makefile中决定要编译u-boot.bin，但该目标编译的具体规则，用什么工具链，编译选项...内容都由config.mk来进一步实现。
2. 顶层config.mk：对Makefile实现的进一步补充。定义了使用的工具链，编译选项，具体编译规则…
3. 顶层mkconfig：在配置时由Makefile调用，生成编译过程中需要的文件。
4. 顶层boards.cfg：记录该uboot支持的单板。

#### 配置文件

配置过程中的文件主要由mkconfig产生：

1. include/config.mk，提供ARCH，CPU，BOARD，VENDOR ，SOC等信息

ARCH=arm, CPU=armv7, BOARD=tiny4412, VENDOR=samsung,

SOC=exynos

1. include/config.h，包含一些必要的头文件

#define CONFIG\_BOARDDIR board/samsung/tiny4412

#include <config\_defaults.h>

#include <configs/tiny4412.h>

#include <asm/config.h>

#### 编译文件

编译的文件主要由Makefile和config.mk产生：

1. include/autoconf.mk：记录CONFIG\_XX的信息，在阅读uboot源码时可以通过该文件判断uboot打开/关闭了哪些源码，类似于linux的.config
2. include/autoconf.mk.dep：包含了一大堆头文件的信息

### config.mk

config.mk在Makefile中第被包含（详见编译过程），其作用相当于对Makefile目标实现的具体化，主要的工作如下：

1. 定义编译工具链，在本篇文章中是arm-linux-
2. 包含一些和配置，ARCH，CPU，Board等相关的config.mk
3. 定义编译过程中的选项，例如CPPFLAGS，链接脚本…
4. 定义源文件的编译规则

#### 工具链

# Include the make variables (CC, etc...)

AS = $(CROSS\_COMPILE)as

LD = $(CROSS\_COMPILE)ld

CC = $(CROSS\_COMPILE)gcc

CPP = $(CC) -E

AR = $(CROSS\_COMPILE)ar

NM = $(CROSS\_COMPILE)nm

LDR = $(CROSS\_COMPILE)ldr

STRIP = $(CROSS\_COMPILE)strip

OBJCOPY = $(CROSS\_COMPILE)objcopy

OBJDUMP = $(CROSS\_COMPILE)objdump

RANLIB = $(CROSS\_COMPILE)RANLIB

设置编译工具链，在本篇文章中，CROSS\_COMPILE=arm-linux-

说明：CROSS\_COMPILE在arch/arm/config.mk中定义，为arm-linux-

#### 子config.mk

# Load generated board configuration

sinclude $(OBJTREE)/include/autoconf.mk

sinclude $(TOPDIR)/arch/$(ARCH)/config.mk #include architecture dependend rules

sinclude $(TOPDIR)/$(CPUDIR)/config.mk # include CPU specific rules

sinclude $(TOPDIR)/$(CPUDIR)/$(SOC)/config.mk #include SoC specific rules

BOARDDIR = $(VENDOR)/$(BOARD)

sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk # include board specific rules

包含一些和配置，ARCH，CPU，Board等相关的config.mk

#### 编译选项

……

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

OBJCFLAGS += --gap-fill=0xff

CPPFLAGS := $(DBGFLAGS) $(OPTFLAGS) $(RELFLAGS)

……

设置一些编译过程的选项和链接文件等

#### 导出变量

export HOSTCC HOSTCFLAGS HOSTLDFLAGS PEDCFLAGS HOSTSTRIP CROSS\_COMPILE \

AS LD CC CPP AR NM STRIP OBJCOPY OBJDUMP MAKE

export CONFIG\_SYS\_TEXT\_BASE PLATFORM\_CPPFLAGS PLATFORM\_RELFLAGS CPPFLAGS CFLAGS AFLAGS

导出编译过程需要用到的变量

#### 编译规则

# Allow boards to use custom optimize flags on a per dir/file basis

BCURDIR = $(subst $(SRCTREE)/,,$(CURDIR:$(obj)%=%))

$(obj)%.s: %.S

$(CPP) $(AFLAGS) $(AFLAGS\_$(BCURDIR)/$(@F)) $(AFLAGS\_$(BCURDIR)) \

-o $@ $<

$(obj)%.o: %.S

$(CC) $(AFLAGS) $(AFLAGS\_$(BCURDIR)/$(@F)) $(AFLAGS\_$(BCURDIR)) \

-o $@ $< -c

$(obj)%.o: %.c

$(CC) $(CFLAGS) $(CFLAGS\_$(BCURDIR)/$(@F)) $(CFLAGS\_$(BCURDIR)) \

-o $@ $< -c

$(obj)%.i: %.c

$(CPP) $(CFLAGS) $(CFLAGS\_$(BCURDIR)/$(@F)) $(CFLAGS\_$(BCURDIR)) \

-o $@ $< -c

$(obj)%.s: %.c

$(CC) $(CFLAGS) $(CFLAGS\_$(BCURDIR)/$(@F)) $(CFLAGS\_$(BCURDIR)) \

-o $@ $< -c –S

定义编译规则

### mkconfig

mkconfig是一个脚本文件，开头注解中对其总结是：创建一些头文件和链接文件，这些文件是特定板子的配置文件。

Makefile调用mkconfig方法是” mkconfig –A tiny4412”（详情见配置过程），整个mkconfig所做工作总结如下：

1. 重新录入传入给mkconfig的参数：$1=tiny4412, $2=arm, $3=armv7, $4=tiny4412, $5=samsung, $6=exynos
2. 创建相应的链接文件，呼应于源码的#include <asm/type.h>…书写形式

include/asm，指向arch/arm/include/asm

include/asm/arch，指向arch/arm/include/asm/arch-exynos

include/asm/proc，指向arch/arm/include/asm/proc-armv

1. 为编译创建相应的头文件：include/config.mk/include/config.h

include/config.mk：配合顶层Makefile，在编译时完成相应的配置筛选

ARCH=arm，CPU=armv7，BOARD=tiny4412，VENDOR=samsung，SOC=exynos

include/config.h：用于在编译时动态的配置代码

#define CONFIG\_BOARDDIR board/samsung/tiny4412

#include <config\_defaults.h>

#include <configs/tiny4412.h>

#include <asm/config.h>

#### 参数接析

下面是详细的代码分析过程（忽略为执行的代码）：

if [ \( $# -eq 2 \) -a \( "$1" = "-A" \) ] ; then

# Automatic mode

line=`egrep -i "^[[:space:]]\*${2}[[:space:]]" boards.cfg` || {

echo "make: \*\*\* No rule to make target \`$2\_config'. Stop." >&2

exit 1

}

set ${line} #重新录入输入参数

# add default board name if needed

[ $# = 3 ] && set ${line} ${1} //ignore

fi

上述代码的作用是在boards.cfg文件中搜索$2(tiny4412)开头的行，并把结果保存在变量line中。然后冲新录入mkconfig的输入参数（即重新对$1/2/3…赋值），最后得到的结果是：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| $1 | $2 | $3 | $4 | $5 | $6 |  |
| Target | ARCH | CPU | Board name | Vendor | SoC | Options |
| tiny4412 | arm | armv7 | tiny4412 | samsung | exynos |  |

#### 变量赋值

CONFIG\_NAME="${1%\_config}"

[ "${BOARD\_NAME}" ] || BOARD\_NAME="${1%\_config}"

arch="$2"

cpu="$3"

if [ "$4" = "-" ] ; then

board=${BOARD\_NAME}

else

board="$4"

fi

[ $# -gt 4 ] && [ "$5" != "-" ] && vendor="$5"

[ $# -gt 5 ] && [ "$6" != "-" ] && soc="$6"

上述代码给某些变量赋值：CONFIG\_NAME= tiny4412\_config; BOARD\_NAME= tiny4412\_config; arch=arm; cpu=armv7; board=tiny4412; vendor=samsung; soc=exynos。

#### 链接文件

if [ "$SRCTREE" != "$OBJTREE" ] ; then

……

else

cd ./include

rm -f asm

ln -s ../arch/${arch}/include/asm asm

fi

rm -f asm/arch

if [ -z "${soc}" ] ; then

ln -s ${LNPREFIX}arch-${cpu} asm/arch

else

ln -s ${LNPREFIX}arch-${soc} asm/arch

fi

if [ "${arch}" = "arm" ] ; then

rm -f asm/proc

ln -s ${LNPREFIX}proc-armv asm/proc

fi

从这里开始，mkconfig准备创建一些链接文件($SRCTREE和$OBJTREE由Makefile传入)：

1. 创建/include/asm，指向/arch/arm/include/asm
2. 创建/include/asm/arch，指向/arch/arm/include/asm/arch-exynos
3. 创建/include/asm/proc，指向/arch/arm/include/asm/proc-armv

说明：当${ SRCTREE } != ${OBJTREE}时，${LNPREFIX} = ${SRCTREE}/arch/${arch}/include/asm/。但在这里${LNPREFIX}为空，但被链接文件依然是${SRCTREE}/arch/${arch}/include/asm/

#### 头文件

echo "ARCH = ${arch}" > config.mk

echo "CPU = ${cpu}" >> config.mk

echo "BOARD = ${board}" >> config.mk

[ "${vendor}" ] && echo "VENDOR = ${vendor}" >> config.mk

[ "${soc}" ] && echo "SOC = ${soc}" >> config.mk

创建/include/config.mk文件，并向该文件写入：ARCH=arm，CPU=armv7，BOARD=tiny4412，VENDOR=samsung，SOC=exynos等信息

if [ -z "${vendor}" ] ; then

BOARDDIR=${board}

else

BOARDDIR=${vendor}/${board} #a

fi

# Create board specific header file

if [ "$APPEND" = "yes" ] # Append to existing config file

then

echo >> config.h

else

> config.h # a

fi

echo "/\* Automatically generated - do not edit \*/" >>config.h

cat << EOF >> config.h

#define CONFIG\_BOARDDIR board/$BOARDDIR

#include <config\_defaults.h>

#include <configs/${CONFIG\_NAME}.h>

#include <asm/config.h>

EOF

创建/include/config.h文件，并将上述代码红色部分(注解a的除外)写入到该文件中。

## 配置过程

在Makefile文件中和配置选项make tiny4412\_config匹配的规则是：

MKCONFIG := $(SRCTREE)/mkconfig

unconfig:

@rm -f $(obj)include/config.h $(obj)include/config.mk \

$(obj)board/\*/config.tmp $(obj)board/\*/\*/config.tmp \

$(obj)include/autoconf.mk $(obj)include/autoconf.mk.dep

%\_config:: unconfig

@$(MKCONFIG) -A $(@:\_config=)

上述代码的含义比较清晰，我们在使用make tiny4412\_config主要工作如下：

1. 执行unconfig目标下的command，主要是清理之前配置信息。
2. 执行%\_config目标下的command，”mkconfig –A tiny4412”

说明1：$(variable:search=replace)是替代引用规则：” :”号后为搜索字符串，” =”号后为用于替代的字符串。则在$(@:\_config=)中， $@代表目标”tiny4412\_config”， search是\_config，replace是” 空”；则$(@:\_config=)可以解释为在”tiny4412\_config”中，\_config用空来替换，则是tiny4412。

## 编译过程

uboot的Makefil中一共有两个all：第一个all没有依赖，也没有命令，仅需要构建一些头文件；第二个all有依赖，但没有命令。故最终执行make的结果是：

1. 构建第一个all需要的头文件
2. 实现第二个all的依赖

### all-1

从Makefile的目标的第一个all开始，逐一分析比较有看点的代码

all:

sinclude $(obj)include/autoconf.mk.dep

sinclude $(obj)include/autoconf.mk

执行make命令的核心操作之一就是构建include/autoconf.mk.dep和include/autoconf.mk文件，故在console上可以看到” Generating include/autoconf.mk…”等信息。

# load ARCH, BOARD, and CPU configuration

include $(obj)include/config.mk

export ARCH CPU BOARD VENDOR SOC

根据配置过程的include/config.mk，导出变量：ARCH，CPU，BOARD，VENDOR， SOC

# load other configuration

include $(TOPDIR)/config.mk

加载其他的配置，这些配置都在顶层Makefile的config.mk中

### all-2

# Always append ALL so that arch config.mk's can add custom ones

ALL += $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U\_BOOT\_NAND) $(U\_BOOT\_ONENAND)

all: $(ALL)

第二个all的终极目标就是实现依赖$(ALL)，即u-boot.srec，u-boot.bin，System.map，$(U\_BOOT\_NAND)，$(U\_BOOT\_ONENAND)。这里我们重点关注u-boot.bin

$(obj)u-boot.bin: $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

$(BOARD\_SIZE\_CHECK)

$(obj)u-boot: depend \

$(SUBDIRS) $(OBJS) $(LIBBOARD) $(LIBS) $(LDSCRIPT) $(obj)u-boot.lds

$(GEN\_UBOOT)

……

u-boot.bin由u-boot制作而成，而u-boot由一系列的依赖和$(GEN\_UBOOT)命令生成。接下来的动作就是去逐渐生成这些依赖，执行命令，完成对uboot.bin的构建。

我们的目标是了解uboot编译整体流程，分析也到这里为止。

### 杂记

本小节记录编译过程中比较重要的原料

# U-Boot objects....order is important (i.e. start must be first)

OBJS = $(CPUDIR)/start.o # arch/arm/cpu/armv7/start.o

…

LIBS += $(CPUDIR)/lib$(CPU).o # arch/arm/cpu/armv7/libarmv7.o

LIBS += $(CPUDIR)/$(SOC)/lib$(SOC).o # arch/arm/cpu/armv7/exynos/libexynos.o

LIBS += arch/$(ARCH)/lib/lib$(ARCH).o # arch/arm/lib/libarm.o

LIBS += $(CPUDIR)/s5p-common/libs5p-common.o

#arch/arm/cpu/armv7/ s5p-common/ libs5p-common.o

…

LIBBOARD = board/$(BOARDDIR)/lib$(BOARD).o

#board/samsung/tiny4412/ libtiny4412.o

定义编译所需要的obj和lib，上述代码仅列出和我们单板相关OBJS和LIBS

# 启动阶段一

参考：<https://blog.csdn.net/u011565090/article/details/41171559>

<http://www.cnblogs.com/CoderTian/p/5995409.html>

## 概述

本章分析tiny4412-uboot启动的第一阶段，源码始于arch/arm/cpu/armv7/start.S，

说明：start.S分析我们以流程为主，不去纠结于具体细节

## start.S

start.S位于arch/arm/cpu/armv7/，所做的工作总结如下：

1. 设置CPU为SVC32模式
2. cpu\_init\_crit ：CPU相关的初始化：关闭I/D cache，关闭MMU，调用底层板级初始化函数lowlevel\_init
3. lowlevel\_init：位于board\samsung\tiny4412\lowlevel\_init.S，最主要的工作是拷贝uboot的代码到RAM，然后跳转到board\_init\_f函数，执行uboot的第二阶段。

说明：在跳转到uboot第二阶段前，串口输出信息为”OK”

### svc32

.globl \_start

\_start: b reset

……

reset:

/\* set the cpu to SVC32 mode \*/

根据复位所在的异常向量表位置，系统上电会自动跳转到reset处。

进入reset的后的工作是：事cpu进入到svc32模式执行

### cpu\_init\_crit

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

bl cpu\_init\_crit

#endif

……

cpu\_init\_crit:

bl cache\_init

/\* Invalidate L1 I/D\*/

……

/\* disable MMU stuff and caches \*/

……

/ \* Jump to board specific initialization...

\* The Mask ROM will have already initialized

\* basic memory. Go here to bump up clock rate and handle

\* wake up conditions.\*/

……

bl lowlevel\_init @ go setup pll,mux,memory

mov lr, ip @ restore link

mov pc, lr @ back to my caller

在uboot源码中，我们没有定义CONFIG\_SKIP\_LOWLEVEL\_INIT，故源码会直跳转cpu\_init\_crit。cpu\_init\_crit所做工作如下：

1. 跳转到cache\_init，不过源码并没有定义cache\_init，故什么都不做
2. 使L1的I/D cache无效化
3. 关闭MMU和I/D cache
4. 跳转到板级相关的底层初始化函数lowlevel\_init

### lowlevel\_init

.globl lowlevel\_init

lowlevel\_init:

/\* use iROM stack in bl2 \*/

……

bl read\_om

/\* when we already run in ram, we don't need to relocate U-Boot.

\* and actually, memory controller must be configured before U-Boot

\* is running in ram.\*/

……

beq after\_copy /\* r0 == r1 then skip sdram init and u-boot.bin loading \*/

……

/\* init system clock \*/

bl system\_clock\_init

/\* Memory initialize \*/

bl mem\_ctrl\_asm\_init

/\* init uart for debug \*/

bl uart\_asm\_init @初始化完成之后打印O

b 1f

1:

bl tzpc\_init //trust zone相关

b load\_uboot

load\_uboot:

……

beq nand\_boot

……

beq onenand\_boot

……

beq mmcsd\_boot

……

beq emmc\_boot

……

beq emmc\_boot\_4\_4

……

beq nor\_boot

……

beq mmcsd\_boot

mmcsd\_boot:

……

bl movi\_uboot\_copy

b after\_copy

after\_copy:

#ifdef CONFIG\_ENABLE\_MMU

bl enable\_mmu

#endif

/\* store second boot information in u-boot C level variable \*/

/\* Print 'K' \*/

ldr r0, \_board\_init\_f

mov pc, r0

\_board\_init\_f:

.word board\_init\_f

lowlevel\_init位于和特定单板相关的目录下，故在移植是该函数是一个研究重点。tiny4412的该函数在board\samsung\tiny4412\lowlevel\_init.S，所做工作如下：

1. 设置BL2栈指针地址,进入read\_om，读取启动引脚，判断启动方式
2. 若当前程序是否已经在RAM中运行，直接跳转到after\_copy执行下一步动作。对于我们常用的SD卡启动来说，此时程序一般在在iRAM中运行。故这里不会跳转。
3. 既然程序在iRAM中运行， 这里会执行更底层的初始化：system\_clock\_init，mem\_ctrl\_asm\_init，uart\_asm\_init，然后跳转到标号1，在到load\_uboot
4. load\_uboot会根据外部启动的设备调用不同的方式将uboot拷贝到RAM，然后进入after\_copy。这里以SD启动为例，load\_uboot会调用mmcsd\_boot。mmcsd\_boot会调用movi\_uboot\_copy将uboot拷贝到RAM，然后跳转到after\_copy。
5. after\_copy所做工作如下：保存uboot第二阶段C变量的信息；启动MMU；打印出’K’；将PC指针设置为board\_init\_f函数，至此开始第二阶段的uboot执行。

说明：在函数uart\_asm\_init完成之后会打印出‘O’，而在artercopy跳转到board\_init\_f前会打印出’K’；故在跳转到uboot第二阶段前串口输出信息为”OK”

注意：在after\_copy中MMU处于打开状态，在做某些实验时请关闭。

# 启动阶段二

参考： <https://blog.csdn.net/pugu12/article/details/47011159>

<http://www.wowotech.net/u-boot/boot_flow_2.html> -2

## 概述

uboot的第二阶段需要关注点：board\_init\_f，relocate\_code，board\_init\_r，main\_loop

### 两次搬运

board\_init\_f(front)是uboot的前置板级初始化，board\_init\_r(rear)则是后置板级初始化。board\_init\_f过渡到board\_init\_r需要执行relocate\_code的动作，即将uboot搬移到DRAM中。但在第一阶段的lowlevel\_init中，我们已经看到了在调用board\_init\_f前，uboot已经被搬移到了DRAM，那这里为什么还需要再次调用relocate\_code进行搬运？而且uboot源码也不提供关闭relocate的开关？

主要原因猜测如下：

1. 某些情况下，在执行board\_init\_f时uboot可能还未在DRAM
2. 如果代码已经在u-boot中了，relocation的代价是极小的，仅仅是简单的内存复制，因此没有太大的必要做出两个分支，区别对待
3. 在relocate的过程中，也一并做了其它事情，特别是很多空间的预留，防止和kernel发生冲突（kernel一般在DRAM的低端地址解压执行，故uboot常放在DRAM的顶端地址）…

## 数据结构

### bd\_t

bd\_t在arch/arm/include/asm/u-boot.h中定义，记录的是和板子相关的信息。

typedef struct bd\_info {

int bi\_baudrate; /\* serial console baudrate \*/

unsigned long bi\_ip\_addr; /\* IP Address \*/

ulong bi\_arch\_number; /\* unique id for this board \*/

ulong bi\_boot\_params; /\* where this board expects params \*/

struct /\* RAM configuration \*/

{

ulong start;

ulong size;

} bi\_dram[CONFIG\_NR\_DRAM\_BANKS]; //4

} bd\_t;

来谈谈某些成员的含义：

### gd\_t

gd\_t在arch/arm/include/asm/global\_data.h中定义，用于记录uboot一些全局信息。

typedef struct global\_data {

bd\_t \*bd;

unsigned long flags;

unsigned long baudrate;

unsigned long have\_console; /\* serial\_init() was called \*/

unsigned long env\_addr; /\* Address of Environment struct \*/

unsigned long env\_valid; /\* Checksum of Environment valid? \*/

unsigned long fb\_base; /\* base address of frame buffer \*/

#ifdef CONFIG\_ARM

/\* "static data" needed by most of timer.c on ARM platforms \*/

unsigned long timer\_rate\_hz;

unsigned long tbl;

unsigned long tbu;

unsigned long long timer\_reset\_value;

unsigned long lastinc;

#endif

unsigned long relocaddr; /\* Start address of U-Boot in RAM \*/

phys\_size\_t ram\_size; /\* RAM size \*/

unsigned long mon\_len; /\* monitor len \*/

unsigned long irq\_sp; /\* irq stack pointer \*/

unsigned long start\_addr\_sp; /\* start\_addr\_stackpointer \*/

unsigned long reloc\_off;

void \*\*jt; /\* jump table \*/

char env\_buf[32]; /\* buffer for getenv() before reloc. \*/

……

} gd\_t;

来谈谈某些成员的含义：

1. flags：用于指示当前

## board\_init\_f()

board\_init\_f()位于arch/arm/lib/board.c，所做的主要工作如下：

1. 设置gd\_t结构在RAM中的位置，并清空该结构体，然后再后续逐一填充：env\_addr，env\_valid ，have\_console…
2. 调用init\_sequence数组里面的函数指针，初始化各个模块，主要工作有：
3. 设置gd\_t->env\_addr，填充默认环境变量：bootargs，bootcmd，bootdelay，baudrate，ethaddr，ipaddr，serverip，gatewayip，netmask
4. 打印相关信息：uboot版本，CPU信息，APLL和MPLL，板子名字
5. 调用dram\_init\_banksize()初始化DRAM，然后调用display\_dram\_config()在终端打印出DRAM的信息。其中dram\_init\_banksize()由uboot负责定义接口，开发板厂商提供该函数实现，位于board/samsung/tiny4412/tiny4412.c。display\_dram\_config()位于arch/arm/lib/board.c
6. 为再次relocate做准备，计算：gd\_t结构的 relocaddr,start\_addr\_sp,reloc\_off成员；然后跳转到start.S的relocate\_code

根据uboot的配置，提炼出board\_init\_f()核心的流程框架如下：

/\* Pointer is writable since we allocated a register for it \*/

gd = (gd\_t \*) ((CONFIG\_SYS\_INIT\_SP\_ADDR) & ~0x07);

/\* compiler optimization barrier needed for GCC >= 3.4 \*/

\_\_asm\_\_ \_\_volatile\_\_("": : :"memory");

memset((void\*)gd, 0, sizeof (gd\_t));

gd->mon\_len = \_bss\_end\_ofs;

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

if ((\*init\_fnc\_ptr)() != 0) {

hang();

}

}

......

addr = CONFIG\_SYS\_LOAD\_ADDR;

......

/\* reserve memory for malloc() arena \*/

addr\_sp = addr - TOTAL\_MALLOC\_LEN;

/\* (permanently) allocate a Board Info struct

\* and a permanent copy of the "global" data \*/

addr\_sp -= sizeof (bd\_t);

bd = (bd\_t \*) addr\_sp;

gd->bd = bd;

addr\_sp -= sizeof (gd\_t);

id = (gd\_t \*) addr\_sp;

gd->irq\_sp = addr\_sp; /\* setup stackpointer for exeptions \*/

addr\_sp -= 3; /\* leave 3 words for abort-stack \*/

addr\_sp &= ~0x07; /\* 8-byte alignment for ABI compliance \*/

......

gd->bd->bi\_baudrate = gd->baudrate;

dram\_init\_banksize(); \* Ram ist board specific, so move it to board code ... \*/

display\_dram\_config(); /\* and display it \*/

gd->relocaddr = addr;

gd->start\_addr\_sp = addr\_sp;

gd->reloc\_off = addr - \_TEXT\_BASE;

memcpy(id, (void \*)gd, sizeof (gd\_t));

relocate\_code(addr\_sp, id, addr);

/\* NOTREACHED - relocate\_code() does not return \*/

### init\_sequence

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

if ((\*init\_fnc\_ptr)() != 0) {

hang();

}

}

init\_sequence是一个函数指针数组，定义于arch/arm/lib/board.c中，该数组下保存了一些列板子各模块的初始化函数。上述代码的作用就是去逐一调用init\_sequence里的各个初始化函数执行，而且一旦失败整个uboot就陷入hang()而死掉。

对于tiny4412，init\_sequence里可用的初始化函数有：

1. timer\_init：在uboot的第一阶段已经初始化了时钟，故这里函数为空。
2. env\_init： 设置gd\_t->env\_addr指针，填充默认环境变量：bootargs，bootcmd，bootdelay，baudrate，ethaddr，ipaddr，serverip，gatewayip，netmask
3. console\_init\_f：在uboot的第一阶段已经初始化了uart，并且打印了”OK”，故这里仅设置gd\_t->have\_console=1，表示有console可以使用。
4. display\_banner：打印出uboot的版本，格式：时间(年月日时分秒)+版本号
5. print\_cpuinfo：打印出CPU的信息；APLL和MPLL的信息
6. checkboard：打印出板子信息，这里是"Board:TINY4412 "
7. dram\_init：DRAM在uboot第一阶段已经初始化，故这里函数为空

#### env\_init

不同的启动介质会各自的env\_init函数，在这里该函数由common/env\_auto.c提供。

uchar default\_environment[] = {

"bootargs=" CONFIG\_BOOTARGS "\0"

"bootcmd=" CONFIG\_BOOTCOMMAND" \0" //movi read kernel 0 40008000…

"bootdelay=" MK\_STR(CONFIG\_BOOTDELAY) "\0" //3

"baudrate=" MK\_STR(CONFIG\_BAUDRATE) "\0" //115200

"ethaddr=" MK\_STR(CONFIG\_ETHADDR) "\0" //00:40:5c:26:0a:5b

"ipaddr=" MK\_STR(CONFIG\_IPADDR)"\0" //192.168.0.20

"serverip=" MK\_STR(CONFIG\_SERVERIP)"\0" //"192.168.0.10

"gatewayip=" MK\_STR(CONFIG\_GATEWAYIP)"\0" //192.168.0.1

"netmask=" MK\_STR(CONFIG\_NETMASK)"\0" //255.255.255.0

......

"\0"

};

int env\_init(void)

{

#if defined(ENV\_IS\_EMBEDDED)

......

#else /\* ENV\_IS\_EMBEDDED \*/

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

gd->env\_valid = 1;

#endif /\* ENV\_IS\_EMBEDDED \*/

}

主要工作：设置gd\_t->env\_addr为默认default\_environment；并且设置gd\_t->env\_valid=1，表示打开环境变量。default\_environment在common/env\_common.c中提供，这里仅列出tiny4412中提供的env参数。

补充：ENV\_IS\_EMBEDDED主要目的是将环境变量嵌入到uboot的代码段，一般在使用Nor/Nand Flash才有效。故这里ENV\_IS\_EMBEDDED并未定义。详情请参考：https://blog.csdn.net/u013083059/article/details/70833278

## relocate\_code

relocate\_code位于arch/arm/cpu/armv7/start.S中，其作用是在此将uboot赋值的DRAM，同时简单的对DRAM进行布局。

1. 赋值uboot的代码到RAM
2. 清楚bss段，设置栈指针
3. 跳转到RAM，入口函数是board\_init\_r

.globl relocate\_code

relocate\_code:

mov r4, r0 /\* save addr\_sp \*/

mov r5, r1 /\* save addr of gd \*/

mov r6, r2 /\* save addr of destination \*/

/\* Set up the stack \*/

......

copy\_loop:

......

clear\_bss:

......

/\*\* We are done. Do not return, instead branch to second part of board

\* initialization, now running from RAM. \*/

jump\_2\_ram:

ldr r0, \_board\_init\_r\_ofs

adr r1, \_start

add lr, r0, r1

/\* setup parameters for board\_init\_r \*/

mov r0, r5 /\* gd\_t \*/

mov r1, r6 /\* dest\_addr \*/

/\* jump to it ... \*/

mov pc, lr

## board\_init\_r()

board\_init\_r()位于arch/arm/lib/board.c，主要工作如下：

1. 拿到gd\_t和bd\_t结构，并且设置gd\_t->flags= GD\_FLG\_RELOC，表示重定位操作已经完成
2. 调用board\_init初始化开发板，该函数由uboot定义接口，开发板移植人员负责实现。这里主要作用是设置机器码bd\_t->bi\_arch\_number，启动参数的位置bd\_t->bi\_boot\_params，打印一些tiny4412的信息：BL1 version，Boot Mode
3. 调用mem\_malloc\_init初始化堆空间，该操作完成之后，可以调用malloc
4. 调用mmc\_initialize初始化mmc设备。该函数位于drivers/mmc/mmc.c，根据MMC的通道数（设备数）逐一进行初始化。每初始化完成一个通道，就会打印出类似“MMC Device 0: 14804 MB”信息。
5. 调用env\_relocate重定位env。对于tiny4412来说，在board\_init\_f的init\_sequence->env\_init中已经设置了gd\_t->env\_addr和gd\_t->env\_valid，故这里实际上什么也没有做。
6. 其他方面的初始化：stdio\_init()；jumptable\_init()；这里说明一点，并不是执行stdio\_init()之后才能调用printf等函数。
7. 初始化和使能中断：interrupt\_init()，enable\_interrupts()；注意某些实验时关闭
8. 网络方面的初始化：bd\_t->bi\_ip\_addr，eth\_initialize()。其中eth\_initialize()用于初始化以太网卡，但在tiny4412并没有做对DM9621的任何初始化，故会看到终端打印“Net: No ethernet found.”
9. 进入main\_loop函数

根据uboot的配置，提炼出board\_init\_r()核心的流程框架如下：

......

gd = id;

bd = gd->bd;

gd->flags |= GD\_FLG\_RELOC; /\* tell others: relocation done \*/

monitor\_flash\_len = \_bss\_start\_ofs;

board\_init(); /\* Setup chipselects \*/

......

/\* The Malloc area is immediately below the monitor copy in DRAM \*/

malloc\_start = dest\_addr - TOTAL\_MALLOC\_LEN;

mem\_malloc\_init(malloc\_start, TOTAL\_MALLOC\_LEN);

......

mmc\_initialize(bd);

......

env\_relocate(); /\* initialize environment \*/

gd->bd->bi\_ip\_addr = getenv\_IPaddr("ipaddr"); /\* IP Address \*/

stdio\_init(); /\* get the devices list going. \*/

jumptable\_init();

......

interrupt\_init(); /\* set up exceptions \*/

enable\_interrupts(); /\* enable exceptions \*/

......

puts("Net:\t");

eth\_initialize(gd->bd);

......

/\* main\_loop() can return to retry autoboot, if so just run it again. \*/

for (;;) {

main\_loop();

}

/\* NOTREACHED - no way out of command loop except booting \*/

### board\_init

board\_init位于board/samsung/tiny4412/tiny4412.c，目的是为了初始化不同的板子，由提供开发版的原厂提供。对于tiny4412而言，主要的工作如下：

......

/\* display BL1 version \*/

#ifdef CONFIG\_TRUSTZONE

printf("\nBL1 version: N/A (TrustZone Enabled BSP)\n");

#else

......

#endif

#ifdef CONFIG\_TINY4412A

gd->bd->bi\_arch\_number = MACH\_TYPE\_TINY4412;

#else

......

#endif

gd->bd->bi\_boot\_params = (PHYS\_SDRAM\_1+0x100);

OmPin = INF\_REG3\_REG;

printf("\n\nChecking Boot Mode ...");

if (OmPin == BOOT\_ONENAND) {

….

} else if (OmPin == BOOT\_MMCSD) {

printf(" SDMMC\n");

......

## main\_loop

main\_loop()函数定义于common/main.c，主要负责两件事：1) 当用户在uboot delay时间里按下按键，则uboot进入命令行模式；2) 当用户在uboot delay时间未按下按键，则uboot进读取bootcmd，然后执行该命令。

main\_loop()实现上述功能的方式有两种：普通循环方式和hush方式，由宏CONFIG\_SYS\_HUSH\_PARSER(include/configs/tiny4412.h)控制。“hush方式”提供更为智能的解析，但实现相对复杂很多。故在tiny4412中，虽然定义了CONFIG\_SYS\_HUSH\_PARSER，但我们仍按普通循环的方式来分析，即认为CONFIG\_SYS\_HUSH\_PARSER未定义。

更多请参考：<https://blog.csdn.net/andy_wsj/article/details/8614905>

main\_loop()函数，所做的主要工作如下：

1. 获取bootdelay，bootcmd环境变量。
2. 若在uboot的bootdelay期间，未按下任何按键中断uboot启动kernel，则调用run\_command)准备执行bootcmd
3. 若在uboot的bootdelay期间，按下任何按键进入uboot命令行模式下，则进入循环：先调用readline()读取用户命令；然调用run\_command()执行

根据uboot的配置，提炼出main\_loop()核心的流程框架如下：

......

#ifdef CONFIG\_AUTO\_COMPLETE //目前测试打开此宏似乎没效果

install\_auto\_complete(); //实现uboot命令的自动补全

#endif

……

s = getenv ("bootdelay");

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

......

s = getenv ("bootcmd");

......

if (bootdelay >= 0 && s && !abortboot (bootdelay)) {

// abortboot()打印：Hit any key to stop autoboot

......

run\_command (s, 0);

......

}

/\* Main Loop for Monitor Command Processing \*/

for (;;) {

len = readline (CONFIG\_SYS\_PROMPT);

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

if (len > 0)

strcpy (lastcommand, console\_buffer);

else if (len == 0)

flag |= CMD\_FLAG\_REPEAT;

if (len == -1)

puts ("<INTERRUPT>\n");

else

rc = run\_command (lastcommand, flag);

if (rc <= 0) {

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

lastcommand[0] = 0;

}

}

### run\_command()

run\_command()定义于common/main.c，所做的主要工作如下：

1. 复制cmd到cmdbuf中，防止对env的破坏。
2. 循环检测输入的cmdbuf，提取具体的cmd，并对该cmd做一系列的处理(宏的替换)、校核工作。由于uboot允许一次输入多个cmd（”;”隔开），故需要提取每一条cmd执行。
3. 根据每条cmd开头的命令名从“command table”中寻找执行该cmd的具体函数，然后做相关检查后执行该函数

根据uboot的配置，提炼出run\_command()核心的流程框架如下：

......

strcpy (cmdbuf, cmd); //char \*str = cmdbuf;

/\* Process separators and check for invalid

\* repeatable commands \*/

while (\*str) {

/\* Find separator, or string end

\* Allow simple escape of ';' by writing "\;" \*/

......

/\* Limit the token to data between separators \*/

......

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

process\_macros (token, finaltoken);

/\* Extract arguments \*/

if ((argc = parse\_line (finaltoken, argv)) == 0) {…

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

if ((cmdtp = find\_cmd(argv[0])) == NULL) {…

/\* found - check max args \*/

if (argc > cmdtp->maxargs) {…

......//CONFIG\_CMD\_BOOTD，当cmd是do\_bootd时启动默认内核

/\* OK - call function to do the command \*/

if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {…

repeatable &= cmdtp->repeatable;

……

}

return rc ? rc : repeatable;

#### find\_cmd()

find\_cmd()定义在common/command.c中，主要作用是根据输入的cmd名，寻找对应的函数执行。

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

{

//int len = &\_\_u\_boot\_cmd\_end - &\_\_u\_boot\_cmd\_start;

int len = &\_\_u\_boot\_cmd\_showvar - &\_\_u\_boot\_cmd\_bdinfo + 1;

//return find\_cmd\_tbl(cmd, &\_\_u\_boot\_cmd\_start, len);

return find\_cmd\_tbl(cmd, &\_\_u\_boot\_cmd\_bdinfo, len);

}

\_\_u\_boot\_cmd\_showvar 和\_\_u\_boot\_cmd\_bdinfo 应该是用U\_BOOT\_CMD 宏定义的cmd\_tbl\_t，都在段Struct\_Section中，而且分别指示该段的结束和末位（详情参考5.2节）。故上述函数的核心就是去uboot支持的所有cmd\_tbl\_t中寻找cmd对应的cmd\_tbl\_t结构。

补充：\_\_u\_boot\_cmd\_bdinfo和\_\_u\_boot\_cmd\_showvar分别定义在common/下cmd\_bdinfo.c和hush.c中。

U\_BOOT\_CMD(

bdinfo, 1, 1, do\_bdinfo,

"print Board Info structure",

""

);

U\_BOOT\_CMD(

showvar, CONFIG\_SYS\_MAXARGS, 1, do\_showvar,

"print local hushshell variables",

"\n - print values of all hushshell variables\n"

"showvar name ...\n"

" - print value of hushshell variable 'name'"

);

# 启动内核

## 概述

uboot的终极目的是启动内核，使用的命令是bootm。本章的线索从bootm入手，研究uboot是如何启动内核。

bootm是uboot用来启动kernel的命令，对应函数是do\_bootm()，命令的用法：

bootm kernel\_addr [args…]

kernel\_addr：内核所在DRAM中的地址

args：附带参数，可ramdisk地址和device-tree blob地址

目前来说，uboot支持两种内核启动的方式，分别对应了bootm不同的使用方式：

1. IMAGE\_FORMAT\_LEGACY：tag传参的启动方式，bootm的使用： bootm kernel\_addr
2. IMAGE\_FORMAT\_FIT ：这就是大名鼎鼎的设备树传参的启动方式。bootm的使用： bootm { kernel\_addr } { initrd\_addr/ramdisk addr } { device-tree blob addr }

说明：原生uboot并不支持直接启动zImage。tiny4412提供的uboot中可以直接启动zImage，是因为samsung或友善之臂对其进行了移植（但这种移植有缺陷，破坏了uboot启动uImage的能力）。

### image

在学习uboot启动内核之前，有必要了解linux的常见的各种镜像：

1. vmlinux/vmlinuz：位于根目录，linux编译生成elf文件。
2. image：位于arch/arm/boot，使用objcopy将vmlinux/vmlinuz加工生成。
3. zImage：位于arch/arm/boot，对image进一步压缩，并在首部添加相应的自解压代码。
4. uImage：uImage并不是linux的产物。它使用mkimage对zImage进行加工得到(在zImage首部添加64Byte的描述信息)，便于uboot正确的启动内核。 说明：mkimage位于uboot源码目录下tools/，在uboot编译后得到。

### 补充

仅支持两种方式启动：LEGACY和FIT。LEGACY就是传统的uImage启动，FIT则是设备树传参启动。

## 数据结构

### bootm\_headers\_t

bootm\_headers定义在include/image.h中，用于描述uboot带启动image的信息。

typedef struct bootm\_headers {

image\_header\_t \*legacy\_hdr\_os; //描述

image\_header\_t legacy\_hdr\_os\_copy; //描述

ulong legacy\_hdr\_valid; //若为1，表示uboot启动的是zImgae

#ifndef USE\_HOSTCC

image\_info\_t os; /\* os image info \*/

ulong ep; /\* entry point of OS \*/

ulong rd\_start, rd\_end;/\* ramdisk start/end \*/

ulong ft\_len; /\* length of flat device tree \*/

ulong initrd\_start;

ulong initrd\_end;

ulong cmdline\_start;

ulong cmdline\_end;

bd\_t \*kbd;

#endif

int verify; /\* getenv("verify")[0] != 'n' \*/

#define BOOTM\_STATE\_START (0x00000001)

#define BOOTM\_STATE\_LOADOS (0x00000002)

#define BOOTM\_STATE\_RAMDISK (0x00000004)

#define BOOTM\_STATE\_FDT (0x00000008)

#define BOOTM\_STATE\_OS\_CMDLINE (0x00000010)

#define BOOTM\_STATE\_OS\_BD\_T (0x00000020)

#define BOOTM\_STATE\_OS\_PREP (0x00000040)

#define BOOTM\_STATE\_OS\_GO (0x00000080)

int state;

#ifdef CONFIG\_LMB

struct lmb lmb; /\* for memory mgmt \*/

#endif

......

} bootm\_headers\_t;

来谈谈某些成员的含义。

### image\_header\_t

image\_header\_t定义在include/image.h，用于描述zImage的头部信息

typedef struct image\_header {

uint32\_t ih\_magic; /\* Image Header Magic Number \*/

uint32\_t ih\_hcrc; /\* Image Header CRC Checksum \*/

uint32\_t ih\_time; /\* Image Creation Timestamp \*/

uint32\_t ih\_size; /\* Image Data Size \*/

uint32\_t ih\_load; /\* Data Load Address \*/

uint32\_t ih\_ep; /\* Entry Point Address \*/

uint32\_t ih\_dcrc; /\* Image Data CRC Checksum \*/

uint8\_t ih\_os; /\* Operating System \*/

uint8\_t ih\_arch; /\* CPU architecture \*/

uint8\_t ih\_type; /\* Image Type \*/

uint8\_t ih\_comp; /\* Compression Type \*/

uint8\_t ih\_name[IH\_NMLEN]; /\* Image Name \*/

} image\_header\_t;

来谈谈某些成员的含义

## do\_bootm()

do\_bootm()位于common/cmd\_bootm.c中，函数所做的主要工作如下：

1. 若宏CONFIG\_ZIMAGE\_BOOT打开，需要对判断是否是zImage：
2. 确定image在DRAM的位置：若用户直接传入，使用用户传入的地址；否则使用默认的地址，这里是0xc3e00000
3. 若image的0x35地址开始处4字节等于LINUX\_ZIMAGE\_MAGIC(注意字节序)，则该image是zImage
4. 若是zImage，需要填充相关头结构：所在DRAM地址，OS类型，启动入口地址…然后跳转到标号after\_header\_check处。此时consloe会打印“Boot with zImage”，接下来可以忽略步骤2~3，
5. 调用bootm\_start()进一步判断、校核image类型。bootm\_start()支持两种类型的image启动：遗留的uImage和设备树。同时也会根据image类型，提取相关的信息到bootm\_headers\_t结构的相关成员中。
6. 调用bootm\_load\_os()加载内核。bootm\_load\_os()会根据image的不同压缩类型进行不同的处理。一般会看到终端有这样的打印“Uncompressing ... OK”
7. after\_header\_check:
8. 若是zImage，用步骤1-c的信息填充image\_info\_t的相关成员。
9. 根据image\_info\_t->os成员在全局数组boot\_os[]寻找该OS对应具体的启动函数。对于linux来说，这里获取到的是do\_bootm\_linux()
10. 调用b中获取到的具体启动函数。linux是do\_bootm\_linux()
11. 根据前面的内容获取到启动的函数，在boot\_os数组(common/cmd\_bootm.c)中寻找，在这里是do\_bootm\_linux()。
12. 调用do\_bootm\_linux()继续启动kernel

根据uboot的配置，精简上述函数代码如下：

static bootm\_headers\_t images; //全局变量

......

#ifdef CONFIG\_ZIMAGE\_BOOT

#define LINUX\_ZIMAGE\_MAGIC 0x016f2818

image\_header\_t \*hdr;

ulong addr;

/\* find out kernel image address \*/

......

if (\*(ulong \*)(addr + 9\*4) == LINUX\_ZIMAGE\_MAGIC) {

u32 val;

printf("Boot with zImage\n");

//addr = virt\_to\_phys(addr);

hdr = (image\_header\_t \*)addr;

hdr->ih\_os = IH\_OS\_LINUX;

hdr->ih\_ep = ntohl(addr);

memmove (&images.legacy\_hdr\_os\_copy, hdr, sizeof(image\_header\_t));

images.legacy\_hdr\_os = hdr; /\* save pointer to image header \*/

images.legacy\_hdr\_valid = 1;

goto after\_header\_check;

}

#endif

......

if (bootm\_start(cmdtp, flag, argc, argv))...

iflag = disable\_interrupts(); //启动内核不允许有中断

......

ret = bootm\_load\_os(images.os, &load\_end, 1);

......

#if defined(CONFIG\_ZIMAGE\_BOOT)

after\_header\_check:

images.os.os = hdr->ih\_os;

images.ep = image\_get\_ep (&images.legacy\_hdr\_os\_copy);

#endif

boot\_fn = boot\_os[images.os.os];

......

arch\_preboot\_os();

boot\_fn(0, argc, argv, &images);

### bootm\_start()

bootm\_start()位于common/cmd\_bootm.c中，函数主要工作如下：

1. 调用boot\_get\_kernel()获得image的header信息。header中包含了不同image独有的描述，为下一步确认image类型和校核做准备。
2. 根据header继续调用genimg\_get\_format()获得imgae类型。根据不同的image类型调用不同函数填充bootm\_headers\_t相关成员，主要是获得如下的信息：image类型，压缩类型，操作系统类型，image加载位置和结束为止…
3. IMAGE\_FORMAT\_LEGACY：uImage启动的方式，该分支主要调用image\_get\_xxx()系列函数填充bootm\_headers\_t->image\_info\_t结构
4. CONFIG\_FIT：设备树启动的方式，该分支主要调用fit\_image\_get\_xxx()系列函数填充bootm\_headers\_t结构里fit\_xxx开头的系列成员和bootm\_headers\_t->image\_info\_t结构。
5. 寻找image的entry point。
6. 如果image\_info\_t->type是KERNEL/MULTI，并且此时image\_info\_t->os是LINUX，需要寻找ramdisk；若支持设备树启动，需要进一步寻找设备树的“flattened”。上述的内容若没有寻找到，则函数会失败。

根据uboot的配置，精简上述函数代码如下：

......

/\* get kernel image header, start address and length \*/

os\_hdr = boot\_get\_kernel (cmdtp, flag, argc, argv,

&images, &images.os.image\_start, &images.os.image\_len);

......

switch (genimg\_get\_format (os\_hdr)) { /\* get image parameters \*/

case IMAGE\_FORMAT\_LEGACY:

images.os.type = image\_get\_type (os\_hdr);

images.os.comp = image\_get\_comp (os\_hdr);

images.os.os = image\_get\_os (os\_hdr);

images.os.end = image\_get\_image\_end (os\_hdr);

images.os.load = image\_get\_load (os\_hdr);

break;

#if defined(CONFIG\_FIT)

case IMAGE\_FORMAT\_FIT:

if (fit\_image\_get\_type (images.fit\_hdr\_os,

images.fit\_noffset\_os, &images.os.type)) {...

if (fit\_image\_get\_comp (images.fit\_hdr\_os,

images.fit\_noffset\_os, &images.os.comp)) {...

if (fit\_image\_get\_os (images.fit\_hdr\_os,

images.fit\_noffset\_os, &images.os.os)) {...

images.os.end = fit\_get\_end (images.fit\_hdr\_os);

if (fit\_image\_get\_load (images.fit\_hdr\_os, images.fit\_noffset\_os,

&images.os.load)) {...

break;

#endif

......

}

if (images.legacy\_hdr\_valid) { /\* find kernel entry point \*/

images.ep = image\_get\_ep (&images.legacy\_hdr\_os\_copy);

#if defined(CONFIG\_FIT)

} else if (images.fit\_uname\_os) {

ret = fit\_image\_get\_entry (images.fit\_hdr\_os, images.fit\_noffset\_os, &images.ep);

......

#endif

}

......

if (((images.os.type == IH\_TYPE\_KERNEL) ||

(images.os.type == IH\_TYPE\_MULTI)) &&

(images.os.os == IH\_OS\_LINUX)) {

ret = boot\_get\_ramdisk (argc, argv, &images, IH\_INITRD\_ARCH,

&images.rd\_start, &images.rd\_end); /\* find ramdisk \*/

......

#if defined(CONFIG\_OF\_LIBFDT)

/\* find flattened device tree \*/

ret = boot\_get\_fdt (flag, argc, argv, &images, &images.ft\_addr, &images.ft\_len);

......

set\_working\_fdt\_addr(images.ft\_addr);

#endif

}

images.os.start = (ulong)os\_hdr;

images.state = BOOTM\_STATE\_START;

## do\_bootm\_linux()

do\_bootm\_linux()是do\_bootm()调用启动linux的函数，定义在arch/arm/lib/bootm.c。在分析此函数前，我们需要知道内核启动需要明白linux kernel启动需要的条件：r0=0，r1=机器码，r2=启动参数地址。

do\_bootm\_linux()所做的工作如下：

1. 调用getenv()获取环境变量bootargs和machid。但在uboot中，我们并没有设置machid，故该变量为空。
2. 调用boot\_get\_ramdisk()获取ramdisk
3. 如果支持设备树启动，且启动类型正好是设备树，则调用bootm\_linux\_fdt()进行设备树启动内核的方式
4. 设置函数kernel\_entry()的地址image的entry地址。
5. 调用setup\_start\_tag()，setup\_xxx\_tags()，setup\_end\_tag()安装一系列的环境变量。对于tiny4412，主要关注的是“bootargs”，“memory”
6. 调用announce\_and\_cleanup()通知内核准备启动了，然后做启动前的清理工作。该函数会打印“Starting kernel ...”，这也是uboot最后的打印信息。
7. 调用kernel\_entry()启动内核。传入参数r0=0，r1=机器码，r2=启动参数地址

根据uboot的配置，精简上述函数代码如下：

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

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

......

char \*commandline = getenv ("bootargs");

......

s = getenv ("machid"); //machid=NULL

......

ret = boot\_get\_ramdisk(argc, argv, ...//RAM文件系统

#ifdef CONFIG\_OF\_LIBFDT

if (images->ft\_len)

return bootm\_linux\_fdt(machid, images); //以设备树的方式启动内核

#endif

......

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

......

setup\_start\_tag (bd);

setup\_memory\_tags (bd); //CONFIG\_SETUP\_MEMORY\_TAGS

setup\_commandline\_tag (bd, commandline); //CONFIG\_CMDLINE\_TAG

if (images->rd\_start && images->rd\_end) //CONFIG\_INITRD\_TAG

setup\_initrd\_tag (bd, images->rd\_start, images->rd\_end);

setup\_end\_tag(bd);

......

announce\_and\_cleanup(); //打印Starting kernel ...,

kernel\_entry(0, machid, bd->bi\_boot\_params);

/\* does not return \*/

## tag传参

uboot在启动内核的时候，会告知内核某些参数。目前来说，参数传递的方式有两种：tag和设备树。本节介绍tag传参。

tag传参的主要思路是，uboot和内核共同维护一个struct tag结构。uboot在启动内核前，将某些参数填入struct tag中，并且将这些结构安按照一定顺序放到DRAM的某一区域（bd\_t->bi\_boot\_params）。在启动内核时，仅需要将首个struct tag地址告诉内核即可。

### 数据结构

#### tag/tag\_header

struct tag和struct tag\_header定义在arch/arm/include/asm/setup.h中。

struct tag仅仅是一个壳，里面描述的具体tag信息header指示：该tag大小，具体对应的tag结构(tag\_core、tag\_cmdline…)。

struct tag\_header {

u32 size; //tag的大小

u32 tag; //tag的类型

};

struct tag {

struct tag\_header hdr;

union {

struct tag\_core core; //setup\_start\_tag()

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;

struct tag\_acorn acorn; /\* Acorn specific \*/

struct tag\_memclk memclk; /\* DC21285 specific \*/

} u;

};

谈谈某些成员的含义：

#### tag\_cmdline

struct tag\_cmdline定义在arch/arm/include/asm/setup.h

struct tag\_cmdline {

char cmdline[1]; /\* this is the minimum size \*/

};

### setup\_start\_tag()

setup\_start\_tag()位于arch/arm/lib/bootm.c，用于指示tag传参的开始。函数所做的工作如下：

1. 给全局变量指针params(struct tag \*)赋值为bd\_t-> bi\_boot\_params，表明tag参数所在内存起始位置是bd\_t-> bi\_boot\_params
2. 填充第一个struct tag：类型为ATAG\_CORE，对应的数据结构是struct tag\_core
3. 跳转params，指向下一个struct tag开头的位置。

static struct tag \*params;

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);

}

### setup\_end\_tag()

setup\_start\_tag()位于arch/arm/lib/bootm.c，用于指示tag传参的结束。函数所做的工作如下：

1. 填充最后一个tag：类型为ATAG\_NONE，无具体的tag类型。

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

{

params->hdr.tag = ATAG\_NONE;

params->hdr.size = 0;

}

### setup\_commandline\_tag()

setup\_commandline\_tag()位于arch/arm/lib/bootm.c，用于传递commandline参数，函数所做的工作如下：

1. 对传入的commandline做一些预处理：去除开头的空格，检测是否合法…
2. 填充该struct tag：类型是ATAG\_CMDLINE，具体内容是tag\_cmdline。注意：struct tag\_cmdline结构的cmdline大小为1个char，但这里直接使用了strcpy。原因由于tag本来就是对一段内存区域按照一定格式填充，而uboot本来就是直接操作内存。
3. 跳转到下一个tag

static void setup\_commandline\_tag (bd\_t \*bd, char \*commandline)

{

char \*p;

if (!commandline) return;

for (p = commandline; \*p == ' '; p++); /\* eat leading white space \*/

/\* skip non-existent command lines so the kernel will still

\* use its default command line. \*/

if (\*p == '\0') return;

params->hdr.tag = ATAG\_CMDLINE;

params->hdr.size =

(sizeof (struct tag\_header) + strlen (p) + 1 + 4) >> 2;

strcpy (params->u.cmdline.cmdline, p);

params = tag\_next (params);

}

# 命令体系

## 概述

本章的来学习uboot命令体系。

## 数据结构

### cmd\_tbl\_t

cmd\_tbl\_t用来描述uboot下每个commad，定义于include/command.h中：

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 \* const []);

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

#ifdef CONFIG\_SYS\_LONGHELP

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

#endif

#ifdef CONFIG\_AUTO\_COMPLETE

int (\*complete)(int argc, … // do auto completion on the arguments

#endif

};

来谈谈某些成员的含义：

1. repeatable：该命令是否被重复执行；在uboot命令行下，若只是敲入回车键，会默认执行上一条uboot命令。

### 宏U\_BOOT\_CMD

include/command.h了相关宏，描述该uboot支持的宏定义：

#define U\_BOOT\_CMD(name,maxargs,rep,cmd,usage,help) \

cmd\_tbl\_t \_\_u\_boot\_cmd\_##name Struct\_Section = {#name, maxargs, rep, cmd, usage, help}

从实现基本可以看出，U\_BOOT\_CMD是在Struct\_Section段中，定义一个cmd\_tbl\_t结构体，然后为该结构的成员赋值。

### example

uboot支持cmd定义在common/cmd\_xxx.c类似的源码中，接下来举一些实例：

在common/cmd\_bootm.c中，定义了bootm的cmd\_tbl\_t结构：

U\_BOOT\_CMD(

bootm, CONFIG\_SYS\_MAXARGS, 1, do\_bootm,

……

);

在common/cmd\_fastboot.c中，定义了fastboot的cmd\_tbl\_t结构：

U\_BOOT\_CMD(

fastboot, 4, 1, do\_fastboot,

"fastboot- use USB Fastboot protocol\n",

……

);

# 设备树启动

参考：<http://www.cnblogs.com/pengdonglin137/tag/TINY4412/>

## 概述

本章主要目标是修改友善之臂提供的uboot，实现设备树启动linux的方式。

实验中将uImage加载到0x40008000，ramdisk加载到0x41000000, exynos4412-tiny4412.dtb（设备树镜像）加载到0x42000000，故bootm的启动命令是：

bootm 40008000 41000000 42000000

说明：在进行设备树的试验中，大部分情况都没有成功，实验停在了“Uncompressing Linux... done, booting the kernel.。但也有极少次数内核能成功启动。原因可能是由于设备树文件“exynos4412-tiny4412.dtb”在dnw下载时失败。

## uboot修改

由于设备树启动依赖于uImage，还需要打开支持设备树的宏CONFIG\_OF\_LIBFDT

### 支持uImage

直接启动uImage的修改文件是common/cmd\_bootm.c，最主要的操作是：当判断启动的image不是zImage时，images.os.os, images.ep不能被强制赋值。

修改完成之后可以编译生成uboot

--- src/uboot\_tiny4412\_0929/common/cmd\_bootm.c 2015-10-31 13:08:42.000000000 +0800

+++ uboot\_tiny4412\_0929/common/cmd\_bootm.c 2018-04-19 07:03:24.243707222 +0800

@@ -590,6 +590,7 @@

ulong load\_end = 0;

int ret;

boot\_os\_fn \*boot\_fn;

+ int is\_zImage = 0;

#ifdef CONFIG\_SECURE\_BOOT

#ifndef CONFIG\_SECURE\_BL1\_ONLY

@@ -613,6 +614,7 @@

if (\*(ulong \*)(addr + 9\*4) == LINUX\_ZIMAGE\_MAGIC) {

u32 val;

+ is\_zImage = 1;

printf("Boot with zImage\n");

//addr = virt\_to\_phys(addr);

@@ -723,8 +725,10 @@

#if defined(CONFIG\_ZIMAGE\_BOOT)

after\_header\_check:

- images.os.os = hdr->ih\_os;

- images.ep = image\_get\_ep (&images.legacy\_hdr\_os\_copy);

+ if (is\_zImage) {

+ images.os.os = hdr->ih\_os;

+ images.ep = image\_get\_ep (&images.legacy\_hdr\_os\_copy);

+ }

#endif

#ifdef CONFIG\_SILENT\_CONSOLE

#### uImage制作

在进行测试之前还需要制作可用的uImage。直接在linux源码执行“make uImage”生成的uImage是无法启动的，因为该镜像加载地址和entry point地址一致。

我们这里选择手动使用mkimage工具制作uImage：

mkimage -A arm -O linux -T kernel -C none -a 0x40008000 -e 0x40008040 -n Linux-3.5 -d zImage uImage

制作完成后的uImage可以加载到DRAM中0x40008000处启动。

### 宏开关

uboot需要修改的第二个点是宏开关的打开和关闭，主要是MMU的关闭以及支持设备树启动的宏打开。修改的文件位于include/configs/tiny4412.h

--- src/uboot\_tiny4412\_0929/include/configs/tiny4412.h 2015-10-31 13:08:42.000000000 +0800

+++ uboot\_tiny4412\_0929/include/configs/tiny4412.h 2018-04-21 20:29:16.590469927 +0800

@@ -308,7 +308,7 @@

#define CONFIG\_SYS\_MONITOR\_LEN (256 << 10) /\* 256 KiB \*/

#define CONFIG\_IDENT\_STRING " for TINY4412"

-#define CONFIG\_ENABLE\_MMU

+//#define CONFIG\_ENABLE\_MMU

#ifdef CONFIG\_ENABLE\_MMU

#define CONFIG\_SYS\_MAPPED\_RAM\_BASE 0xc0000000

@@ -441,4 +441,9 @@

#define MEMORY\_BASE\_ADDRESS 0x40000000

+

+//jf add

+#define CONFIG\_OF\_LIBFDT

+#define CONFIG\_SYS\_BOOTMAPSZ (20 << 26)

+

#endif /\* \_\_CONFIG\_H \*/

CONFIG\_SYS\_BOOTMAPSZ主要用于uboot对设备树镜像的重定位。在启动内核之前，为避免内核自解压时覆盖ramdisk和设备树镜像，需要对其进行重定位。

#### MMU

由于MMU的已经关闭，故需要修改uboot的链接地址，修改文件为：board/samsung/tiny4412/config.mk：

--- src/uboot\_tiny4412\_0929/board/samsung/tiny4412/config.mk 2015-10-31 13:08:42.000000000 +0800

+++ uboot\_tiny4412\_0929/board/samsung/tiny4412/config.mk 2018-03-24 02:31:15.016640259 +0800

@@ -10,4 +10,5 @@

#

-CONFIG\_SYS\_TEXT\_BASE = 0xc3e00000

+CONFIG\_SYS\_TEXT\_BASE = 0x43e00000

## ramdisk

使用设备树启动内核是需要ramdisk，对于ramdisk的制作，可以参考相关文档。这里我直接使用<http://www.cnblogs.com/pengdonglin137/p/5143516.html>提供的ramdisk.img镜像。

## 设备树镜像

友善之臂提供的linux-3.5内核目前不支持exynos4412相关的设备树（而且arch/arm/boot/dts/Makefile也不存在）。故我们使用linux-4.4.1版本的内核，该内核已经对exynos4412甚至tiny4412有了很好的支持。

故我们的步骤是：

1. 下载linux-4.4.1内核，解压，进入该内核目录。修改Makefile中ARCH和CROSS\_COMPILE变量。
2. 执行make exynos\_defconfig
3. arch/arm/boot/dts/exynos4412-tiny4412.dts文件，修改内容位于本节尾
4. 执行make dtbs(make dtbs clean用于清除)

上述步骤成功后，生成的arch/arm/boot/dts/ exynos4412-tiny4412.dtb就是我们需要的设备树镜像。

arch/arm/boot/dts/exynos4412-tiny4412.dts：

--- a/arch/arm/boot/dts/exynos4412-tiny4412.dts

+++ b/arch/arm/boot/dts/exynos4412-tiny4412.dts

@@ -21,6 +21,7 @@

chosen {

stdout-path = &;serial\_0;

+ bootargs = "root=/dev/ram0 rw rootfstype=ext4 console=ttySAC0,115200 init=/linuxrc earlyprintk";

};

memory {

@@ -78,7 +79,7 @@

bus-width = <;4>;

pinctrl-0 = <;&sd2\_clk &sd2\_cmd &sd2\_cd &sd2\_bus4>;

pinctrl-names = "default";

- status = "okay";

+ status = "disabled";

};

上述设备树描述中添加了bootargs（一般会被uboot本身的bootargs覆盖），并且禁止掉了SD卡控制器2（可能SD控制器的初始化还有些问题，会导致内核挂掉）。如果在u-boot中设置了bootargs的话，在bootm的时候u-boot会用自己的bootargs来覆盖设备树里的bootargs。