EK 之 UBOOT 学习

Creat at 2018-1-19 By ErKun

目录

前言

最近在学习 UBOOT，看 Makefile、看链接脚本。看 UBOOT 源码、看网上博客的分析讲解。。。。

总之看了很多，也就着开发板做了一些测试验证。明白了很多，同时还有很多还不明白，后来一想这么多大神这么久写出来的代码，岂能是我等晚辈三两天、三两周就学会的。后面有时间再继续多研究研究，那个时候的主要精力应该是放在代码的具体实现上，学习 C 语言编程的高级技巧了。

所以这回我主要是想记录下来 UBOOT 的主要执行流程和一些实现上的方式，对于那个过程的具体实现、具体的代码分析留在后续。

俗话说好记性不如烂笔头，但是看明白了，要是不赶紧记下来，那多半怕是就白看了。所以我也想把最近看到的学到的先记下来，同时也是再复习一遍吧！

在看的过程中，也发现 UBOOT 的不同版本还是有些差别的，总的来说就是不用版本的 UBOOT 虽然做的事情都是一样的，但是可能走的路不一样，实现起来的方式不一样，源码文件的结构有些变动。但这些在强大的 Source Insight 代码编辑器下通通都不叫个事儿。

因为我用的是创龙的 AM43XX-EVM 的开发板，所以我看的 UBOOT 也是配套的，是 U-BOOT-2014-07 的版本。

http://processors.wiki.ti.com/index.php/Linux_Core_U-Boot_User%27s_Guide

ti 官方的 uboot 用户指导。

一 、环境搭建

构建开发环境，安装 Linux 虚拟机，安装 Ubuntu 桌面版的长期支持版本（LTS）不要装国产的那个麒麟版本。建议安装 32 位的，因为你的 ARM 处理器就是 32 位的，而且 64 位的话会需要自行安装一些 32 位的支持库，如果你不知道的话，就是强行给自己增加难度。我有切身体会的...

对于 Ubuntu 的版本我觉得应该安装和开发板厂家使用的一样的版本，因为不同版本也有些差别，同样切身体会，就是在做 SD 系统卡时的脚本，在不同版本运行结果不同，就是因为他们的命令都变了。。。等后面全面了解 Linux 后这些就不是问题了。我装了 Ubuntu 的 1204 的 32 位版本和 Ubuntu1604 的 32 位和 64 位版本。说多了都是泪。主要是有个追逐新版的毛病。

安装完成后，做一些后续工作。比如安装 VMware Tools、建立和 windows 之间的共享文件夹、安装 VIM 的完整版然后可以在按自己习惯配置一下，这些事百度上讲的很多很详细。

安装编译工具链，因为 UBOOT 在编译的过程中会创建很多符号链接，而符号链接是 Linux 独有的。Windows 中没有，所以必须把 UBOOT 的源码拷贝到 Linux 的原生目录下去编译，不能在共享文件夹里。

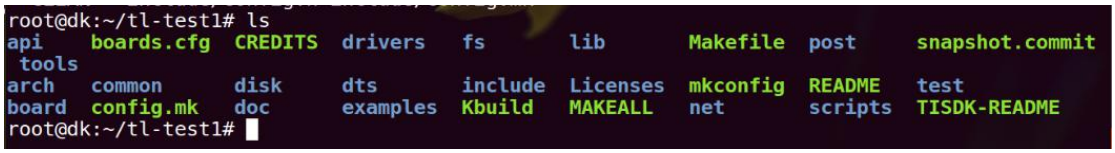
所谓的编译工具链的安装也就是直接解压就行。Linux 的目录一般都有一些分类的所以这些工具也最好安装在/usr/local 目录下。在/usr/local 下新建一个 arm 的文件夹，然后把交叉编译工具链解压在这个目录下就算安装完成了。编译工具链的版本就安装厂家提供的相应版本就行，版本一样就行，可以自行去下载，我就是自己下的，因为提供的在我那解压出错。

解压完后还需要设置一下环境变量，这样才能在其他目录下也直接使用编译工具链的名字而不用加路径。export PATH=安装路径:$PATH 这个命令只能在这一次生效。还可以在~/.bashrc 文件中添加上面那句话，然后关闭命令终端，在重进去一次就可以了。

在命令行中输入 root@dk:~# arm-linux-gnueabihf-gcc -v 查看编译工具链的版本，能打印出来东西并且最后一行显示 gcc version 4.7.3 20130328 (prerelease) (crosstool-NG linaro-1.13.1-4.7-2013.04-20130415 - Linaro GCC 2013.04)就表示安装完成了。可以开始肆意交叉“感染”了。

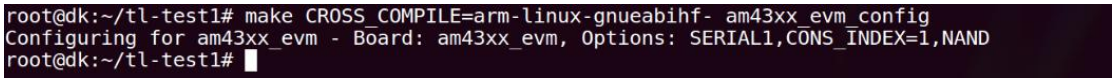
二、UBOOT 初体验

1.将 UBOOT 源码拷贝到 linux 下的某个目录后解压出源码，然后进入源码目录下。



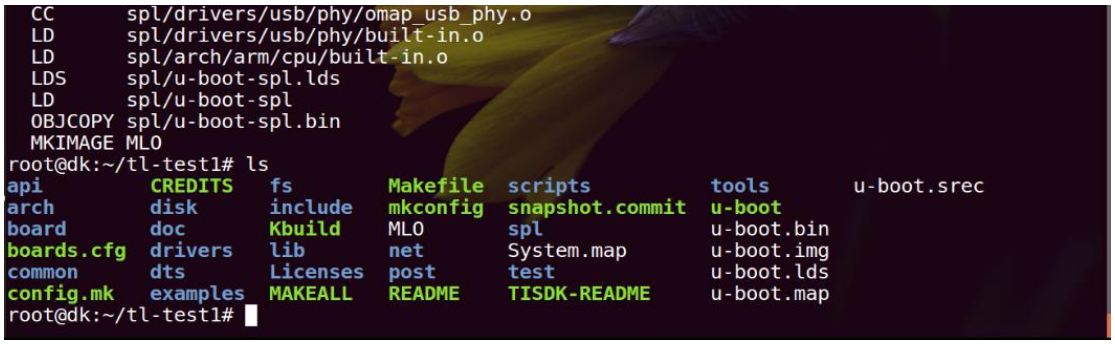
2.清理 UBOOT 用 make CROSS_COMPILE=arm-linux-gnueabihf- distclean 主要是清除上一回编译后生成的中间文件和结果

3.配置 UBOOT。UBOOT 在编译之前是需要配置的，因为 UBOOT 是通用的引导程序支持很多 CPU 和开发板，所以在编译之前，需要配置一下来告诉 Makefile 要编译那些文件。

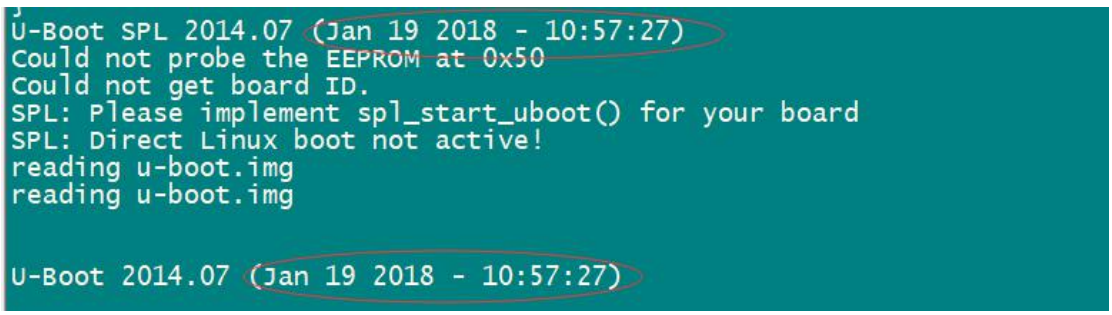


4.编译，配置完成后即可编译 UBOOT

完成后可以再当前目录下看到生成的烧写文件。



其中 MLO 和 u-boot.img 就是可以烧写的程序文件。可以拷贝到 SD 卡中做测试，如何判断执行的就是你刚编译好的呢？



那个时间就是编译时系统的时间，也就是你编译的时间吗，前面是版本信息。

5.一些解释 make CROSS_COMPILE=arm-linux-gnueabihf- distclean

命令 make 就是直接按照文件中的主 Makefile 文件来编译生成指定的目标 CROSS_COMPILE=arm-linux-gnueabihf- 是用来给 Makefile 传参的，就是告诉 Makefile 当前使用的交叉编译工具链是 arm-linux-gnueabihf-，当然这个只是前缀，-后面的选项例如 gcc、ld 等都一样所以已经在 Makefile 中指定了。前缀每个人可能使用的不一样所以自己指定。

如果你确定使用某种编译工具链后，也可以直接在 Makefile 文件中直接指定，也就是给 CROSS_COMPILE=arm-linux-gnueabihf-。这样每次使用 make 是就不用在添加 CROSS_COMPILE=arm-linux-gnueabihf-来传参了。

distclean 是 Makefile 中指定的一个伪目标。主要用来清除编译和配置时生成的中间文件。

am43xx_evm_config 是 Makefile 中的一个用来配置的目标。会根据配置生成一些包含信息的头文件和一些符号链接。

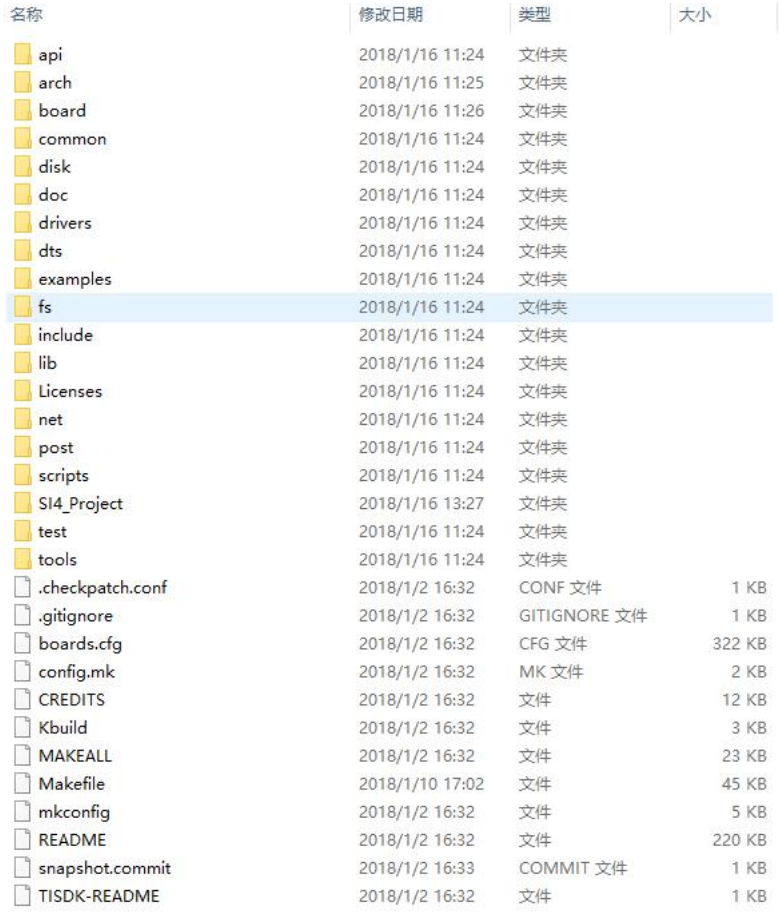
make 命令不带目标时，就是生成 Makefile 中默认的目标 all，也就是最终的 MLO 和 u-boot.img 和其他一些文件。

三 、UBOOT 源码目录

1.UBOOT 的源码目录如下图，必须要说的是 UBOOT 的不同版本之间在目录架构上是有些差别的，但不大，内容上基本一致的。

2008 年 8 月及以前按版本号命名：u-boot-1.3.4.tar.bz2(2008 年 8 月更新)

2008 年 8 月以后均按日期命名。如 u-boot-2017.09.tar.bz2(2017 年 9 月更新)



名称	修改日期	类型	大小
api	2018/1/16 11:24	文件夹	
arch	2018/1/16 11:25	文件夹	
board	2018/1/16 11:26	文件夹	
common	2018/1/16 11:24	文件夹	
disk	2018/1/16 11:24	文件夹	
doc	2018/1/16 11:24	文件夹	
drivers	2018/1/16 11:24	文件夹	
dts	2018/1/16 11:24	文件夹	
examples	2018/1/16 11:24	文件夹	
fs	2018/1/16 11:24	文件夹	
include	2018/1/16 11:24	文件夹	
lib	2018/1/16 11:24	文件夹	
Licenses	2018/1/16 11:24	文件夹	
net	2018/1/16 11:24	文件夹	
post	2018/1/16 11:24	文件夹	
scripts	2018/1/16 11:24	文件夹	
SI4_Project	2018/1/16 13:27	文件夹	
test	2018/1/16 11:24	文件夹	
tools	2018/1/16 11:24	文件夹	
.checkpatch.conf	2018/1/2 16:32	CONF 文件	1 KB
.gitignore	2018/1/2 16:32	GITIGNORE 文件	1 KB
boards.cfg	2018/1/2 16:32	CFG 文件	322 KB
config.mk	2018/1/2 16:32	MK 文件	2 KB
CREDITS	2018/1/2 16:32	文件	12 KB
Kbuild	2018/1/2 16:32	文件	3 KB
MAKEALL	2018/1/2 16:32	文件	23 KB
Makefile	2018/1/10 17:02	文件	45 KB
mkconfig	2018/1/2 16:32	文件	5 KB
README	2018/1/2 16:32	文件	220 KB
snapshot.commit	2018/1/2 16:33	COMMIT 文件	1 KB
TISDK-README	2018/1/2 16:32	文件	1 KB

2.文件夹列表

api：存放 uboot 源码提供的接口函数

arch：主要存放和 CPU 架构体系相关的代码，是 UBOOT 的重点

board：开发板，是和开发板相关的代码

common：通用代码。包括一些命令和校验之类。还包含了和 SPL 相关的代码。

disk：磁盘相关的代码，目前我还没有涉及到

doc：说明文件

drivers：驱动，所有类型的驱动，每个驱动是一个文件夹

dts：里面就 Makefile 和一个 git 相关文件，不明觉厉。

examples：示例程序

fs：文件系统，支持嵌入式开发板常见的文件系统

include：通用的头文件均在这里

lib：调用的一些库文件

Lisences：许可文件一堆.txt 文件

net：网络相关的代码一些小型的协议栈

post：加电自检程序

scripts：一些脚本程序

SI4_Project：真不好意思。这个是我用 Source Insight 看代码时建的工程，跟 UBOOT 没关系

test：测试程序

tools 辅助工具程序，用于编译生成的 uboot 文件

3.文件目录列表

讲几个用到的吧，有些我目前也不知道啥用。

boards.cfg：开发板的配置信息，前面配置 UBOOT 时就是最后会在这个文件里面查找信息，包括状态、架构、CPU、SoC、厂商、开发板名和选项等一众信息。

config.mk：Makefile 调用的一个配置文件里面又包含了其他目录下的 config.mk

Makefile：这个 UBOOT 的主 Makefile 文件，可以重点看看

mkconfig：配置时就是调用这个文件，这个会读取 boards.cfg 中的信息的

剩下的大致一看，能力不行看不懂，先略过。。。。

4.一些话

因为我是刚开始看 UBOOT，所以我觉得不要过分的去想要知道源码中的某个文件底具体的都是干什么的，大致了解下是存放哪一类代码的即可，有一个整体的框架。到后面看源码的时候你绝对会翻烂这些文件的。一开始接触就钻入到某个细节的话那就真的出不来了，而且很容易就没兴趣了。我看 Makefile 时就是这样，一脸的看不懂，两脸的还是看不懂，感觉写这些的那些人得是有多厉害啊，后来我觉得具体的细节语法什么的用到的时候在查吧。

四 、UBOOT 的配置解析

有关 Makefile 的一些规则可以看《跟我一起写 Makefile》和《GNU-make 的中文手册》，不亚于学习一门新的语言。

1.UBOOT 的配置

之前提过，uboot 的配置就是执行 Makefile 中定义的配置目标。可以在主 Makefile 中查找一下关于 am43xx_evm_config 的这个目标

```
%_config:: outputmakefile
    @$(MKCONFIG) -A $(@:_config=)
```

%是 Makefile 中的通配符，am43xx_evm_config 就是带 _config 后缀的目标，所以会执行这个目标。目标依赖于 outputmakefile

```
outputmakefile:
ifneq ($(KBUILD_SRC),)
    $(Q)ln -fsn $(srctree) source
    $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
        $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
endif
```

这个目标没有依赖了，会判断 KBUILD _SRC 这个变量的值是不是为空来执行下面的命令。因为没有使用 KBUILD 所以关于 KBUILD 的基本上都可以忽略。所以配置的时候就执行了@$(MKCONFIG) -A $(@:_config=)这条语句。MKCONFIG 这个变量在前面有赋值为源码目录下的 mkconfig 文件。$(@:_config=)这句话是把目标中的_config 替换为了空。

```
168
169  MKCONFIG    := $(srctree)/mkconfig
170  export MKCONFIG
```

综合一下就是调用了 uboot 根目录下的 mkconfig 脚本并且传参 am43xx_evm。

mkconfig -A am43xx_evm。

进而进入 mkconfig 文件中进行分析。

```
23
24  if [ \( $# -eq 2 \) -a \( "$1" = "-A" \) ] ; then
25      # Automatic mode
26      line=`awk '($0 !~ /^#/ && $7 ~ /^'"$2"'$/) { print $1, $2, $3, $4, $5, $6, $7, $8 }' $srctree/boards.cfg`
27      if [ -z "$line" ] ; then
28          echo "make: *** No rule to make target \`$2_config'.  Stop." >&2
29          exit 1
30      fi
31
32      set ${line}
33      # add default board name if needed
34      [ $# = 3 ] && set ${line} ${1}
35  fi
```

我们执行的是 mkconfig -A am43xx_evm，$#表示参数的个数，$1 表示第一个参数，

line 就是在 boards.cfg 中 am43xx_evm 的那行。

在 boards.cfg 中有

Status：Active $1

Arch：arm $2

CPU：armv7 $3

SoC：am33xx $4

Vendor：ti $5

Board name：am43xx $6

Target：am43xx_evm $7

Options：am43xx_evm:SERIAL1,CONS_INDEX=1,NAND $8

//Maintainers： Lokesh Vutla <lokeshvutla@ti.com> 这个忽略吧，因为只要 8 个

命令中 set ${line}就是在脚本运行中给出了其他的参数， 这个时候 参数就是上面的 8 个了

然后主要是做些判断就把这些参数赋值给内部定义的相应的变量。比如下面的 options 变量。

```
if [ "$options" ] ; then
    echo "Configuring for ${BOARD_NAME} - Board: ${CONFIG_NAME}, Options: ${options}"
else
    echo "Configuring for ${BOARD_NAME} board..."
fi
```

配置时打印出来的话就是在这打印出来的，就是判断 options 选项然后就打印出来了。

1）接下来就是创建一些符号链接包括 asm 的链接进入 arch/arm/include 文件下删除该文件里面的 asm/arch 然后重新建立 asm/arch 链接指向通文件夹下的 arch-am33

```
if [ -n "$KBUILD_SRC" ] ; then
    mkdir -p ${objtree}/include
    LNPREFIX=${srctree}/arch/${arch}/include/asm/
    cd ${objtree}/include
    mkdir -p asm
else
    cd arch/${arch}/include
fi

rm -f asm/arch

if [ "${soc}" ] ; then
    ln -s ${LNPREFIX}arch-${soc} asm/arch
elif [ "${cpu}" ] ; then
    ln -s ${LNPREFIX}arch-${cpu} asm/arch
fi
```

脚本中的-n var 用来判断 var 变量是否有值。这里没有值，直接进入 arch/arm/include 目录下先删除然后在建立符号链接。

2）然后创建 make 配置文件，注意此时目录已经切换到了的源码的/include 目录下，在这个目录下新建了一个 config.mk 的文件，然后把一些信息写了进去，其实就是写了 CPU 架构和 BOARD 之类的一些信息。如下所示

```
ARCH   = arm
CPU    = armv7
BOARD  = am43xx
VENDOR = ti
SOC    = am33xx
```

```
#
# Create include file for Make
#
( echo "ARCH   = ${arch}"
    if [ ! -z "$spl_cpu" ] ; then
    echo 'ifeq ($(CONFIG_SPL_BUILD),y)'
    echo "CPU    = ${spl_cpu}"
    echo "else"
    echo "CPU    = ${cpu}"
    echo "endif"
    else
    echo "CPU    = ${cpu}"
    fi
    echo "BOARD  = ${board}"

    [ "${vendor}" ] && echo "VENDOR = ${vendor}"
    [ "${soc}"    ] && echo "SOC    = ${soc}"
    exit 0 ) > config.mk
```

3）然后创建了开发板的特殊的头文件 config.h，并且往里面写了一些东西。同样是在 include 文件夹里面的。这个文件是自动生成的所以不要改动，内容如下面所示。

```
#
# Create board specific header file
#
if [ "$APPEND" = "yes" ]    # Append to existing config file
then
    echo >> config.h
else
    > config.h      # Create new config file
fi
echo "/* Automatically generated - do not edit */" >>config.h

for i in ${TARGETS} ; do
    i="`echo ${i} | sed '/=/ {s/=/  /;q; } ; { s/$/ 1/; }'`"
    echo "#define CONFIG_${i}" >>config.h ;
done

echo "#define CONFIG_SYS_ARCH  \"${arch}\""  >> config.h
echo "#define CONFIG_SYS_CPU   \"${cpu}\""   >> config.h
echo "#define CONFIG_SYS_BOARD \"${board}\"" >> config.h

[ "${vendor}" ] && echo "#define CONFIG_SYS_VENDOR \"${vendor}\"" >> config.h

[ "${soc}"    ] && echo "#define CONFIG_SYS_SOC    \"${soc}\""    >> config.h

[ "${board}"  ] && echo "#define CONFIG_BOARDDIR board/$BOARDDIR" >> config.h
cat << EOF >> config.h
```

```
 1 /* Automatically generated - do not edit */
 2 #define CONFIG_SERIAL1  1
 3 #define CONFIG_CONS_INDEX       1
 4 #define CONFIG_NAND     1
 5 #define CONFIG_SYS_ARCH  "arm"
 6 #define CONFIG_SYS_CPU   "armv7"
 7 #define CONFIG_SYS_BOARD "am43xx"
 8 #define CONFIG_SYS_VENDOR "ti"
 9 #define CONFIG_SYS_SOC    "am33xx"
10 #define CONFIG_BOARDDIR board/ti/am43xx
11 #include <config_cmd_defaults.h>
12 #include <config_defaults.h>
13 #include <configs/am43xx_evm.h>
14 #include <asm/config.h>
15 #include <config_fallbacks.h>
16 #include <config_uncmd_spl.h>
```

五 、主 Makefile 大致解析

有关 Makefile 的一些规则可以看《跟我一起写 Makefile》和《GNU-make 的中文手册》，不亚于学习一门新的语言。

http://blog.csdn.net/dndxhej/article/details/8134148

http://blog.csdn.net/guyongqiangx/article/details/52565493

https://www.cnblogs.com/asulove/p/6010322.html

上面这几个是其他人讲解的 Makefile 的一些分析，可以看一下的。

1）上来就是先定义了 UBOOT 的版本，执行时打印出来的版本就是在这定义的，后面会把这些版本信息弄成一个叫 UBOOTVERSION 的宏定义写入到一个头文件中。

2）这个主要是指定编译的结果的输出目录的，origin 这个函数主要用来判断 O 这个参数是来自于哪里，是自己定义的还是命令行输入的。也就是谁 make 的时候可以带 o=一个目录 来把一些编译生成的中间文件输出到指定目录下。

```
106 # OK, Make called in directory where kernel src resides
107 # Do we want to locate output files in a separate directory?
108 ifeq ("$(origin O)", "command line")
109   KBUILD_OUTPUT := $(O)
110 endif
111
112 ifeq ("$(origin W)", "command line")
113   export KBUILD_ENABLE_EXTRA_GCC_CHECKS := $(W)
114 endif
```

3）这里有指定交叉编译的工具链，我们可以不管他，直接在 endif 后面添加一句 CROSS_COMPILE :=arm-linux-gnueabihf- 这样以后 make 是就不用在传参了。省点事儿。

```
197 # set default to nothing for native builds
198 ifeq ($(HOSTARCH),$(ARCH))
199 CROSS_COMPILE ?=
200 endif
201
```

哎呀，剩下的内容好多啊，还有好多看不明白啊，不想写了啊，反正这个 Makefile 能用。

不能再一行一行看了，会忍不住放弃的。。

4）顶层目标依赖

A.首先定义了伪目标_all。注释都说了这是我们默认的目标当在命令行光 make 不指定目标的是时候。想想，我们配置完后就是直接 make 的没有任何参数（交叉编译工具链不算目标）

```
116 # That's our default target when none is given on the command line
117 PHONY := _all
118 _all:
```

然后添加了_all 目标的依赖项视另一个伪目标 all

```
153 PHONY += all
154 ifeq ($(KBUILD_EXTMOD),)
155 _all: all
156 else
157 _all: modules
158 endif
```

all 目标依赖于$(ALL-y) ALL-y 是一个变量

```
762 all:          $(ALL-y)
```

B. ALL-y 变量包含了所有最终需要生成的文件包括一众 u-boot 开头的各种文件

```
709 # Always append ALL so that arch config.mk's can add custom ones
710 ALL-y += u-boot.srec u-boot.bin System.map binary_size_check
711
712 ALL-$(CONFIG_ONENAND_U_BOOT) += u-boot-onenand.bin
713 ifeq ($(CONFIG_SPL_FSL_PBL),y)
714 ALL-$(CONFIG_RAMBOOT_PBL) += u-boot-with-spl-pbl.bin
715 else
716 ALL-$(CONFIG_RAMBOOT_PBL) += u-boot.pbl
717 endif
718 ALL-$(CONFIG_SPL) += spl/u-boot-spl.bin
719 ALL-$(CONFIG_SPL_FRAMEWORK) += u-boot.img
720 ALL-$(CONFIG_TPL) += tpl/u-boot-tpl.bin
721 ALL-$(CONFIG_OF_SEPARATE) += u-boot.dtb u-boot-dtb.bin
722 ifeq ($(CONFIG_SPL_FRAMEWORK),y)
723 ALL-$(CONFIG_OF_SEPARATE) += u-boot-dtb.img
724 endif
725 ALL-$(CONFIG_OF_HOSTFILE) += u-boot.dtb
726 ifneq ($(CONFIG_SPL_TARGET),)
727 ALL-$(CONFIG_SPL) += $(CONFIG_SPL_TARGET:"%"=%)
728 endif
729 ALL-$(CONFIG_REMAKE_ELF) += u-boot.elf
730
731 # enable combined SPL/u-boot/dtb rules for tegra
732 ifneq ($(CONFIG_TEGRA),)
733 ifeq ($(CONFIG_SPL),y)
734 ifeq ($(CONFIG_OF_SEPARATE),y)
735 ALL-y += u-boot-dtb-tegra.bin
736 else
737 ALL-y += u-boot-nodtb-tegra.bin
738 endif
739 endif
740 endif
```

带只有 ALL-后面直接带 y 的是通用文件，其他的都是需要想用的变量也为 y 是才包含进内。

u-boot.serc 依赖于 u-boot 和 FORCE

```
784 u-boot.hex u-boot.srec: u-boot FORCE
785     $(call if_changed,objcopy)
```

u-boot.bin 依赖于 u-boot 和 FORCE

```
801
802 u-boot.bin: u-boot FORCE
803     $(call if_changed,objcopy)
804     $(call DO_STATIC_RELA,$<,$@,$(CONFIG_SYS_TEXT_BASE))
805     $(BOARD_SIZE_CHECK)
```

System.map 这个文件就是神器，后面分析连接脚本和源代码段是贼好用。

依赖于 u-boot

```
System.map: u-boot
	@$(call SYSTEM_MAP,$<) > $@
```

binary_size_check　依赖于 u-boot.bin System.map FORCE

```
binary_size_check: u-boot.bin System.map FORCE
    @file_size=`stat -c %s u-boot.bin` ; \
    map_size=$(shell cat System.map | \
        awk '/_image_copy_start/ {start = $$1} /_image_binary_end/ {end = $$1} END {if (sta
        | bc); \
    if [ "" != "$$map_size" ]; then \
        if test $$map_size -ne $$file_size; then \
            echo "System.map shows a binary size of $$map_size" >&2 ; \
            echo "  but u-boot.bin shows $$file_size" >&2 ; \
            exit 1; \
        fi \
    fi
```

u-boot 很多文件依赖于 u-boot 但 u-boot 依赖于 u-boot-init 和 u-boot-main 变量和连接脚本 u-boot.lds

```
u-boot: $(u-boot-init) $(u-boot-main) u-boot.lds
    $(call if_changed,u-boot__)
ifeq ($(CONFIG_KALLSYMS),y)
    smap=`$(call SYSTEM_MAP,u-boot) | \
        awk '$$2 ~ /[tTwW]/ {printf $$1 $$3 "\\\\000"}'` ; \
    $(CC) $(c_flags) -DSYSTEM_MAP="\"$${smap}\"" \
        -c $(srctree)/common/system_map.c -o common/system_map.o
    $(call cmd,u-boot__) common/system_map.o
endif
```

u-boot-init 和 u-boot-main 变量有分别是 head-y 和 libs-y 变量定义的

```
u-boot-init := $(head-y)
u-boot-main := $(libs-y)
```

u-boot.lds 依赖于$LDSCRIPT 变量 prepare 和 FORCE

```
u-boot.lds: $(LDSCRIPT) prepare FORCE
	$(call if_changed_dep,cpp_lds)
```

没完没了了啊怎么。。。。

head-y 和 libs-y 这里终于看到了对源文件的包含了啊 head 应该就是头，整个文件的头就是 start.S 和一众需要配置的才能添加的，就是变量配置为 y

```
head-y := $(CPUDIR)/start.o
head-$(CONFIG_4xx) += arch/powerpc/cpu/ppc4xx/resetvec.o
head-$(CONFIG_MPC85xx) += arch/powerpc/cpu/mpc85xx/resetvec.o
```

libs-y 包含了一众的需要编译的文件路径，具体要编译哪个文件还是需要在该目录下的的的

Makefile 文件中指定的。

```
libs-y += lib/
libs-$(HAVE_VENDOR_COMMON_LIB) += board/$(VENDOR)/common/
libs-y += $(CPUDIR)/
ifdef SOC
libs-y += $(CPUDIR)/$(SOC)/
endif
libs-$(CONFIG_OF_EMBED) += dts/
libs-y += arch/$(ARCH)/lib/
libs-y += fs/
libs-y += net/
libs-y += disk/
```

u-boot.lds 话说这文件不应该是源码里的一个连接脚本么，怎么变成一个目标了，难道自动生成，还是说在多种情况下来选择用那个。。。。

$(LDSCRIPT)这个变量其实就是指向了我们要用到的链接脚本，下面这段代码就是来判断和寻找 lds 文件的，果然是存在多个，然后根据配置来选择。先在开发板目录下找，找不到就去 CPU 目录下找，再找不到就去 arch/arm/cpu 目录下找。

我去翻了一下源码目录，确实只有在 arch/arm/cpu 下面有一个 u-boot.lds 文件。

没错那就是它了。

```
ifndef LDSCRIPT
    #LDSCRIPT := $(srctree)/board/$(BOARDDIR)/u-boot.lds.debug
    ifdef CONFIG_SYS_LDSCRIPT
        # need to strip off double quotes
        LDSCRIPT := $(srctree)/$(CONFIG_SYS_LDSCRIPT:"%"=%)
    endif
endif

# 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
```

这里是一些列的 prepare 相关的目标和依赖，啊这的是不明觉厉。。。

```
# prepare2 creates a makefile if using a separate output directory
prepare2: prepare3 outputmakefile

prepare1: prepare2 $(version_h) $(timestamp_h)
ifeq ($(__HAVE_ARCH_GENERIC_BOARD),)
ifeq ($(CONFIG_SYS_GENERIC_BOARD),y)
    @echo >&2 "  Your architecture does not support generic board."
    @echo >&2 "  Please undefine CONFIG_SYS_GENERIC_BOARD in your board config file."
    @/bin/false
endif
endif
ifeq ($(wildcard $(LDSCRIPT)),)
    @echo >&2 "  Could not find linker script."
    @/bin/false
endif

archprepare: prepare1 scripts_basic

prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.

# All the preparing..
prepare: prepare0
```

这里有涉及到两个文件，看名字就是描述版本和时间戳的。

```
$(version_h): include/config/uboot.release FORCE
    $(call filechk,version.h)

$(timestamp_h): $(srctree)/Makefile FORCE
    $(call filechk,timestamp.h)
```

include/config/uboot.release 文件中如下，就是 uboot 的版本号

```
1 2015.07
```

```
415 version_h := include/generated/version_autogenerated.h
416 timestamp_h := include/generated/timestamp_autogenerated.h
```

这都是和这相关的。待我去编译好的文件夹了搂一眼去。

这个是 include/generated/timestap_autogenerated.h 文件中的内容，怎么感觉时间不准呢，今天明明是 1 月 19 号。

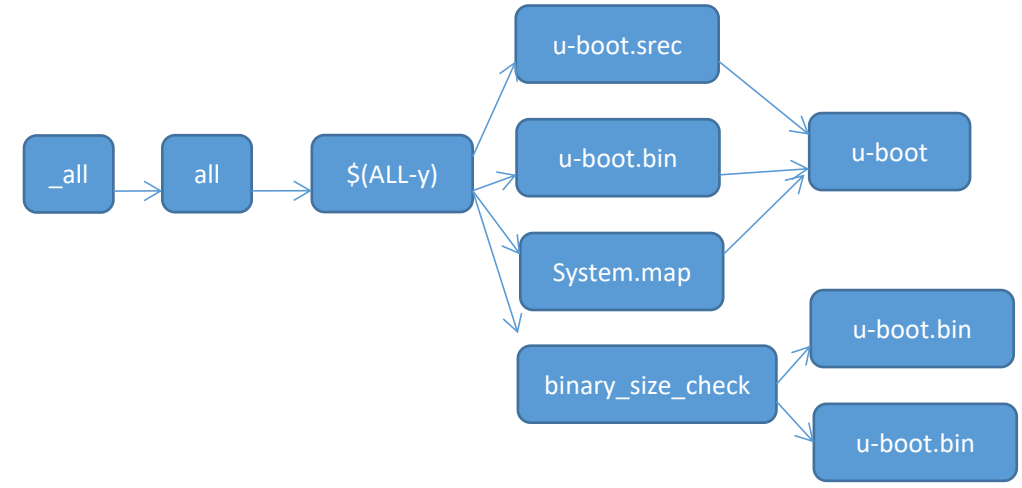
```
1 #define U_BOOT_DATE "Jan 20 2018"
2 #define U_BOOT_TIME "17:48:44"
```

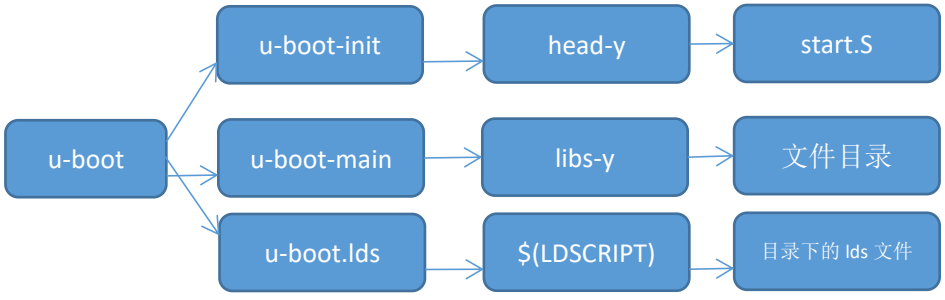
这个是 include/generated/version_autogenerated.h 文件中的内容，你看是个宏定义吧，最后源码里都会调用这些头文件的

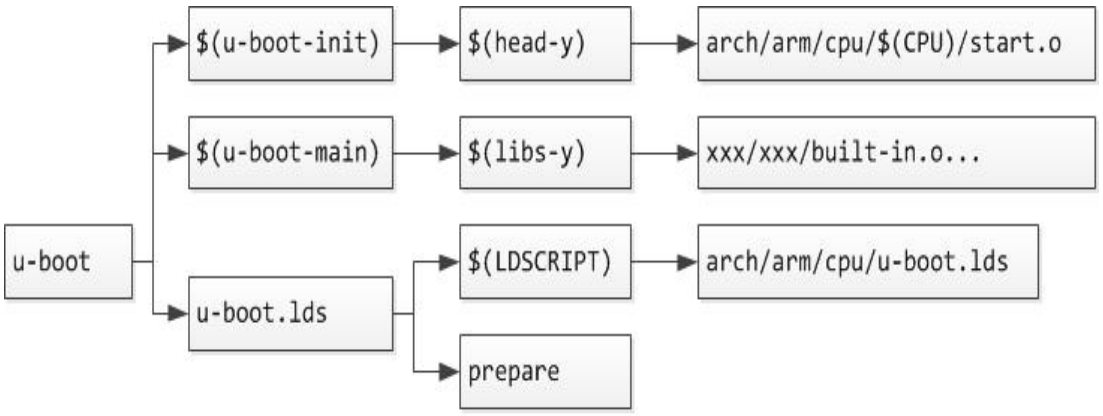
```
1 #define PLAIN_VERSION "2014.07"
2 #define U_BOOT_VERSION "U-Boot " PLAIN_VERSION
3 #define CC_VERSION_STRING "arm-linux-gnueabihf-gcc (crosstool-NG linaro-1.13.1-4.7-2013.04-20130
  CC 2013.04) 4.7.3 20130328 (prerelease)"
4 #define LD_VERSION_STRING "GNU ld (crosstool-NG linaro-1.13.1-4.7-2013.04-20130415 - Linaro GCC
  1"
```

在 include/generated/下面还生成里其他的头文件，都是自动生成的一些信息，描述一些偏移什么的。等用到的时候在看，mark 一下。

最后照着大神来个图，直观体现一下。







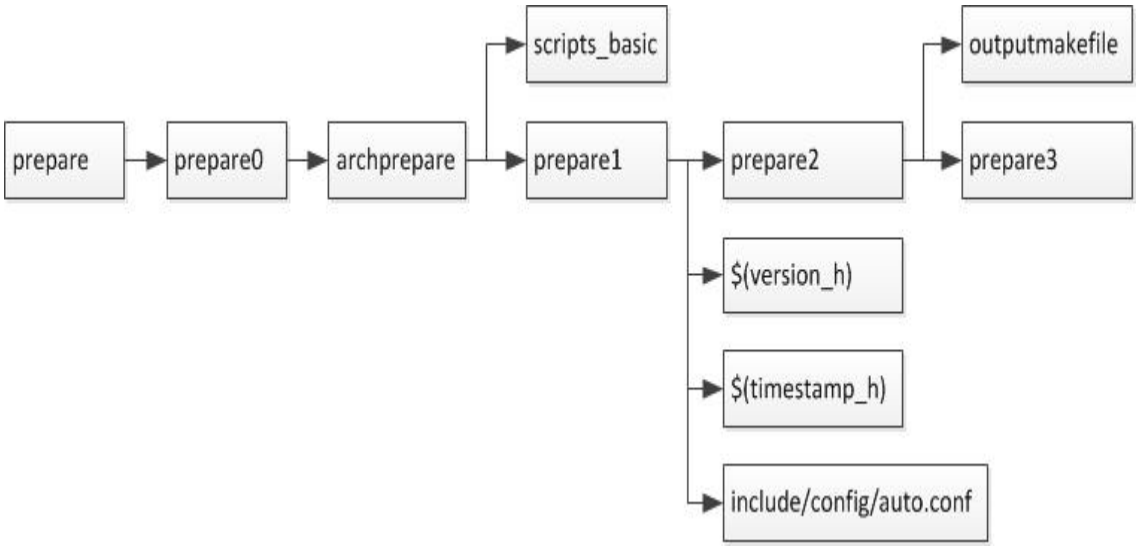
u-boot文件目标的依赖关系

好图必须得收藏

FORCE 目标本身是一个空目标，所以先忽略。

libs 为各层目录下的 build-in.o 的集合，各目录下的.o 文件合并变成一个 build-in.o 供外部链接使用。

prepare 是一系列的伪目标和动作组合，完成编译前的准备工作。



2018-01-19 周五

六 、MLO/SPL 的生成

在上个阶段编译生成 u-boot.bin 和 u-boot.img 镜像后，这些镜像都有 300KB-500KB 这么大。这些都大于了 SoC 内部自带的 SRAM 的大小，ROM code 根本无法直接执行他们，所以就有了用来引导 uboot 的代码，叫做 SPL，在 ti 的叫法中也可以用 MLO，因为 ROM code 就认 MLO 这个名字。所以下文 SPL 和 MLO 就说的是一个东西了。

SPL 的代码是和 UBOOT 的代码在一起的，是 UBOOT 代码中的一部分吧，是在编译的时候通过一些条件编译和一些宏配置来划分出来的。不过我一直觉得为什么不把 UBOOT 的前一段代码直接截取出来作为 SPL，截出来这段只要小于内部 SRAM 的大小并且包含了初始化 DDR 和重定位 UBOOT 的代码就可以啊。而且之前确实有其他的 UBOOT 就是这么干的啊。估计还是和代码有关系，这样的话就必须保证前一段代码是位置无关码了，可能这样就不是很通用了。

接着说，make 编译生成后会在源码目录下生成 MLO 文件，同时会生成一个叫 spl 的文件，这个文件内部是一些用来生成 u-boot-spl、u-boot-spl.bin 等的一些.o 文件。感觉就是低配版的 uboot。

```
root@dk:~/tl-test1# ls
api         config.mk  dts       lib       MLO         snapshot.commit  tools       u-boot.map
arch        CREDITS    examples  Licenses  net         spl              u-boot      u-boot.srec
board       disk       fs        MAKEALL   post        System.map       u-boot.bin
boards.cfg  doc        include   Makefile  README      test             u-boot.img
common      drivers    Kbuild    mkconfig  scripts     TISDK-README     u-boot.lds
root@dk:~/tl-test1# cd spl
root@dk:~/tl-test1/spl# ls
arch  board  common  disk  drivers  fs  lib  net  u-boot-spl  u-boot-spl.bin  u-boot-spl.lds  u-boot
```

MLO/spl 是和 u-boot.bin 等一起一次性生成的，所以还是分析 make 命令，分析主 Makefile 文件。

在文件中有根据变量来判断在默认目标中添加 spl/u-boot-spl.bin 文件的。我想这些应该是包含在内了把。

```
718 ALL-$(CONFIG_SPL) += spl/u-boot-spl.bin
719 ALL-$(CONFIG_SPL_FRAMEWORK) += u-boot.img
```

特意找了一下这些 CONFIG 开头的一堆变量的定义。

```
1104 include/autoconf.mk.dep: include/config.h include/common.h
1105     $(call cmd,autoconf_dep)
1106
1107 quiet_cmd_autoconf = GEN     $@
1108       cmd_autoconf = \
1109     $(CPP) $(c_flags) -DDO_DEPS_ONLY -dM $(srctree)/include/common.h > $@.tmp && \
1110     sed -n -f $(srctree)/tools/scripts/define2mk.sed $@.tmp > $@; \
1111     rm $@.tmp
1112
1113 include/autoconf.mk: include/config.h
1114     $(call cmd,autoconf)
```

```
479 # Read in config
480 -include include/autoconf.mk
481 -include include/autoconf.mk.dep
482
483 # load other configuration
484 include $(srctree)/config.mk
```

这些都是包含在这几个自动生成的文件中尤其 include/autoconf.mk 文件。这个文件是依赖于 include/config.h 文件，这个文件是在配置时生成的里面包含了几个 CPU 之类的宏后还包含了一些头文件，其中就包括源码中的 include/configs 文件夹下面的板子配置的头文件，里面才是真正的这些变量的来源。也是就是说用工具将这里面的宏变成了 include/autoconf.mk 里面的一众配置变量供 Makefile 调用。

源码目录下的 config.mk 好像没干什么事，就是用 Sinclude 又包含了一些其他的目录下的 config.mk。这些.mk 不是自动生成的。用 sinclude 意思是如果目录下没有也不会报错的。所以有的目录下就没有 config.mk 这个文件。

回归 SPL。spl/u-boot-spl.bin 依赖于 spl/u-boot-spl

```
1129
1130 spl/u-boot-spl.bin: spl/u-boot-spl
1131     @:
```

spl/u-boot-spl 依赖于 tools 和 prepare

```
1132 spl/u-boot-spl: tools prepare
1133     $(Q)$(MAKE) obj=spl -f $(srctree)/scripts/Makefile.spl all
```

tools 又依赖于 prepare

```
1010 tools: prepare
1011 # The "tools" are needed early
```

看似找不到了，但是目标 spl/u-boot-spl 的命令里面是转移到了 scripts 下面的 Makefile.spl 里面的 all 目标。

下面分析是在 scripts/Makefile.spl 文件里

一样的老套路啊，all 依赖于 ALL-y 而 ALL-y 其实就是 u-boot-spl.bin 文件

```
209
210 all:    $(ALL-y)
211
```

```
198 ALL-y   += $(obj)/$(SPL_BIN).bin
```

```
36 SPL_BIN := u-boot-spl
```

剩下的简直和 UBOOT 是一样的套路。

```
228
229 $(obj)/$(SPL_BIN).bin: $(obj)/$(SPL_BIN) FORCE
230     $(call if_changed,objcopy)
```

```
249
250 $(obj)/$(SPL_BIN): $(u-boot-spl-init) $(u-boot-spl-main) $(obj)/u-boot-spl.lds
251     $(call cmd,u-boot-spl)
```

```
144 u-boot-spl-init := $(head-y)
145 u-boot-spl-main := $(libs-y)
```

```
262
263 $(obj)/u-boot-spl.lds: $(LDSCRIPT) FORCE
264     $(call if_changed_dep,cpp_lds)
```

MLO 也是有 u-boot-spl.bin 用 mkimage 工具生成的。

```
183
184 MLO MLO.byteswap: $(obj)/u-boot-spl.bin
185     $(call if_changed,mkimage)
```

七 、镜像文件的差异

一开始我也是郁闷的，都是个 uboot 镜像和 spl 的镜像怎么还这么多版本。后来找了个二进制阅读文件看了一些。二进制文件阅读和比较有比较好的工具，比如 Beyond Compare 和 UltraEdit 等，但都是需要钱的啊，在试用了一个月后只好放弃。公司内也不敢用破解的。。。

http://blog.csdn.net/mathsoftware/article/details/51423664

后来网上找了一个 HexCmp 的小软件，也不用安装，解压后直接就能用的挺不错的。

我拿来对比了一下两个文件。

File size	
- HEX	0x16BB0
- DECIMAL	93104
Offset	
- HEX	0x0
- DECIMAL	0
Value	
- Character	'@'
- Bit-Set	01000000
- Byte (HEX)	0x40
- Byte (DEC)	64
- Word (HEX)	0x40
- Word (DEC)	64
- DWord (HEX)	0x40
- DWord (DEC)	64

First File - F:\winshare\tl\tl-test2\MLO

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	40	00	00	00	0C	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	43	48	53	45	54	54	49	4E	47	53	00	00
00000020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
00000030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
00000040	C1	C0	C0	C0	00	01	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

File size	
- HEX	0x169A8
- DECIMAL	92584
Offset	
- HEX	0x0
- DECIMAL	0
Value	
- Character	'u'
- Bit-Set	01110101
- Byte (HEX)	0x75
- Byte (DEC)	117
- Word (HEX)	0x75
- Word (DEC)	117
- DWord (HEX)	0xEB000075
- DWord (DEC)	-352321419

Second File - F:\winshare\tl\tl-test2\spl\u-boot-spl.bin

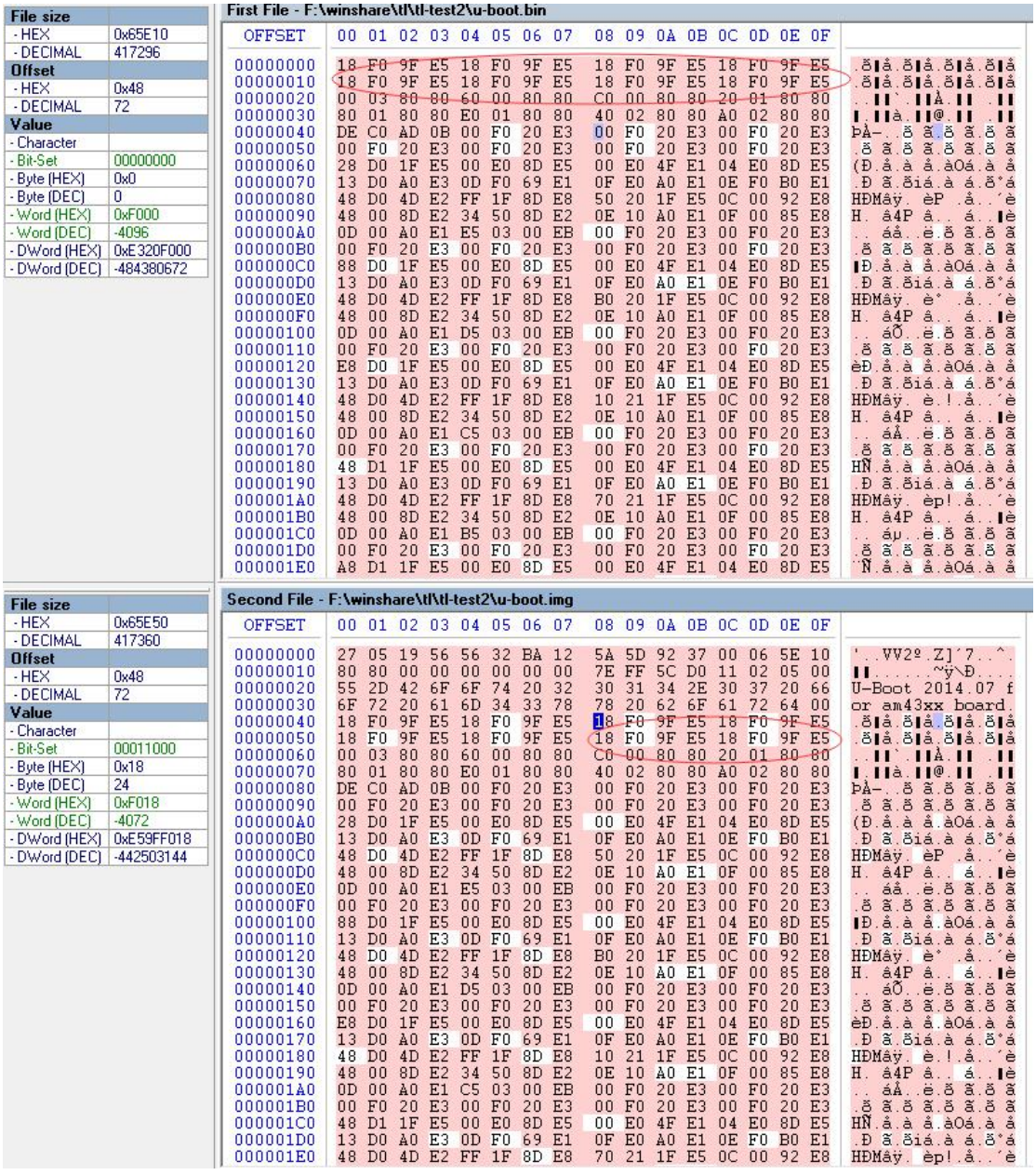
OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	75	00	00	EB	00	00	0F	E1	1F	10	00	E2	1A	00	31	E3
00000010	1F	00	C0	13	13	00	80	13	C0	00	80	E3	00	F0	29	E1
00000020	10	0F	11	EE	02	0A	C0	E3	10	0F	01	EE	64	00	9F	E5
00000030	10	0F	0C	EE	08	00	00	EB	15	00	00	EB	16	00	00	EB
00000040	15	0F	07	EE	9A	0F	07	EE	95	0F	07	EE	44	00	9F	E5
00000050	10	0F	0C	EE	1E	FF	2F	E1	1E	FF	2F	E1	00	00	A0	E3
00000060	17	0F	08	EE	15	0F	07	EE	D5	0F	07	EE	9A	0F	07	EE
00000070	95	0F	07	EE	10	0F	11	EE	02	0A	C0	E3	07	00	C0	E3
00000080	02	00	80	E3	02	0B	80	E3	01	0A	80	E3	10	0F	01	EE
00000090	0E	F0	A0	E1	48	00	00	EA	00	00	00	00	82	D4	A0	E3
000000A0	07	D0	CD	E3	B8	D0	4D	E2	07	D0	CD	E3	0D	90	A0	E1
000000B0	00	00	A0	E3	FF	FF	FF	EB	20	00	9F	E5	00	10	A0	E3
000000C0	1C	20	9F	E5	08	40	2D	E9	02	20	60	E0	81	30	00	EB
000000D0	00	00	A0	E3	0C	90	9F	E5	00	10	A0	E1	F7	05	00	EB
000000E0	00	00	A0	80	10	49	A4	80	48	A1	30	40	08	40	2D	E9
000000F0	00	50	A0	E1	14	30	9F	E5	0C	40	93	E5	C7	00	00	EB
00000100	00	00	A0	E3	00	10	E0	E3	05	20	A0	E1	34	FF	2F	E1
00000110	04	00	A0	80	00	00	A0	E3	1E	FF	2F	E1	08	40	2D	E9
00000120	1C	00	9F	E5	0E	17	00	EB	50	03	0C	E3	C4	30	00	EB
00000130	F7	FF	FF	EB	00	00	A0	E3	D1	00	00	EB	00	00	A0	E3
00000140	08	80	BD	E8	30	63	30	40	08	D0	4D	E2	00	30	A0	E3
00000150	02	00	00	EA	00	00	A0	E1	04	30	9D	E5	01	30	83	E2
00000160	04	30	8D	E5	04	30	9D	E5	63	00	53	E3	F8	FF	FF	DA
00000170	08	D0	8D	E2	1E	FF	2F	E1	10	40	2D	E9	F1	FF	FF	EB
00000180	10	4F	11	EE	EF	FF	FF	EB	01	3A	C4	E3	10	3F	01	EE
00000190	10	80	BD	E8	10	40	2D	E9	EA	FF	FF	EB	10	4F	11	EE
000001A0	04	00	14	E3	10	80	BD	08	E6	FF	FF	EB	6F	00	00	EB
000001B0	05	40	C4	E3	10	4F	01	EE	10	80	BD	E8	10	D0	9F	E5
000001C0	07	D0	CD	E3	0C	90	9F	E5	00	50	2D	E9	DF	03	00	EB
000001D0	00	90	BD	E8	40	FF	33	40	48	A1	30	40	14	10	9F	E5
000001E0	00	00	81	E5	1E	FF	2F	E1	F0	4F	2D	E9	08	C0	9F	E5

猛一瞅吧，我擦都不一样啊。首先对比两个文件的大小。MLO 比 u-boot-spl 多了 520 个字节。然后剩下的就都是一模一样了。

还挺暧昧的，MLO 对 u-boot-spl.bin 说：我只比你多了一个 520！

MLO 是在 u-boot-spl.bin 的基础上添加了一个 header 用来启动时校验等的吧。

然后 u-boot.bin 和 u-boot.img 也是如此，只不过这回不是 520 了



File size	
- HEX	0x65E10
- DECIMAL	417296
Offset	
- HEX	0x48
- DECIMAL	72
Value	
- Character	
- Bit-Set	00000000
- Byte (HEX)	0x0
- Byte (DEC)	0
- Word (HEX)	0xF000
- Word (DEC)	-4096
- DWord (HEX)	0xE320F000
- DWord (DEC)	-484380672

First File - F:\winshare\tl\tl-test2\u-boot.bin

OFFSET	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F
00000000	18 F0 9F E5 18 F0 9F E5	18 F0 9F E5 18 F0 9F E5
00000010	18 F0 9F E5 18 F0 9F E5	18 F0 9F E5 18 F0 9F E5
00000020	00 03 80 80 60 00 80 80	C0 00 80 80 20 01 80 80
00000030	80 01 80 80 E0 01 80 80	40 02 80 80 A0 02 80 80
00000040	DE C0 AD 0B 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000050	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000060	28 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
00000070	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
00000080	48 D0 4D E2 FF 1F 8D E8	50 20 1F E5 0C 00 92 E8
00000090	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
000000A0	0D 00 A0 E1 E5 03 00 EB	00 F0 20 E3 00 F0 20 E3
000000B0	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
000000C0	88 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
000000D0	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
000000E0	48 D0 4D E2 FF 1F 8D E8	B0 20 1F E5 0C 00 92 E8
000000F0	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
00000100	0D 00 A0 E1 D5 03 00 EB	00 F0 20 E3 00 F0 20 E3
00000110	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000120	E8 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
00000130	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
00000140	48 D0 4D E2 FF 1F 8D E8	10 21 1F E5 0C 00 92 E8
00000150	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
00000160	0D 00 A0 E1 C5 03 00 EB	00 F0 20 E3 00 F0 20 E3
00000170	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000180	48 D1 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
00000190	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
000001A0	48 D0 4D E2 FF 1F 8D E8	70 21 1F E5 0C 00 92 E8
000001B0	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
000001C0	0D 00 A0 E1 B5 03 00 EB	00 F0 20 E3 00 F0 20 E3
000001D0	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
000001E0	A8 D1 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5

File size	
- HEX	0x65E50
- DECIMAL	417360
Offset	
- HEX	0x48
- DECIMAL	72
Value	
- Character	
- Bit-Set	00011000
- Byte (HEX)	0x18
- Byte (DEC)	24
- Word (HEX)	0xF018
- Word (DEC)	-4072
- DWord (HEX)	0xE59FF018
- DWord (DEC)	-442503144

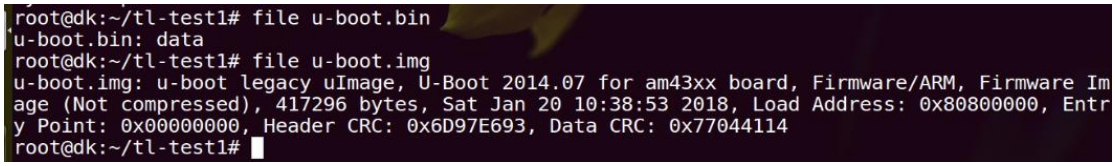
Second File - F:\winshare\tl\tl-test2\u-boot.img

OFFSET	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F
00000000	27 05 19 56 56 32 BA 12	5A 5D 92 37 00 06 5E 10
00000010	80 80 00 00 00 00 00 00	7E FF 5C D0 11 02 05 00
00000020	55 2D 42 6F 6F 74 20 32	30 31 34 2E 30 37 20 66
00000030	6F 72 20 61 6D 34 33 78	78 20 62 6F 61 72 64 00
00000040	18 F0 9F E5 18 F0 9F E5	18 F0 9F E5 18 F0 9F E5
00000050	18 F0 9F E5 18 F0 9F E5	18 F0 9F E5 18 F0 9F E5
00000060	00 03 80 80 60 00 80 80	C0 00 80 80 20 01 80 80
00000070	80 01 80 80 E0 01 80 80	40 02 80 80 A0 02 80 80
00000080	DE C0 AD 0B 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000090	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
000000A0	28 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
000000B0	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
000000C0	48 D0 4D E2 FF 1F 8D E8	50 20 1F E5 0C 00 92 E8
000000D0	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
000000E0	0D 00 A0 E1 E5 03 00 EB	00 F0 20 E3 00 F0 20 E3
000000F0	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000100	88 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
00000110	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
00000120	48 D0 4D E2 FF 1F 8D E8	B0 20 1F E5 0C 00 92 E8
00000130	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
00000140	0D 00 A0 E1 D5 03 00 EB	00 F0 20 E3 00 F0 20 E3
00000150	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
00000160	E8 D0 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
00000170	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
00000180	48 D0 4D E2 FF 1F 8D E8	10 21 1F E5 0C 00 92 E8
00000190	48 00 8D E2 34 50 8D E2	0E 10 A0 E1 0F 00 85 E8
000001A0	0D 00 A0 E1 C5 03 00 EB	00 F0 20 E3 00 F0 20 E3
000001B0	00 F0 20 E3 00 F0 20 E3	00 F0 20 E3 00 F0 20 E3
000001C0	48 D1 1F E5 00 E0 8D E5	00 E0 4F E1 04 E0 8D E5
000001D0	13 D0 A0 E3 0D F0 69 E1	0F E0 A0 E1 0E F0 B0 E1
000001E0	48 D0 4D E2 FF 1F 8D E8	70 21 1F E5 0C 00 92 E8

只是多了一个 64 字节的头部信息。其他的还是一样的。u-boot.img 是由 u-boot.bin 在 tools/mkimage 工具生成的。这是传参也就是头部信息的。

```
827 MKIMAGEFLAGS_u-boot.img = -A $(ARCH) -T firmware -C none -O u-boot \
828     -a $(CONFIG_SYS_TEXT_BASE) -e $(CONFIG_SYS_UBOOT_START) \
829     -n "U-Boot $(UBOOTRELEASE) for $(BOARD) board"
```

在 linux 中用 file 命令可以来查看文件的属性。u-boot.bin 就是二进制 data。而 u-boot.img 就是 u-boot 的 legacy uImage。后面那些应该就是头部信息。包含了版本和板件还有地址之类的信息。



八 、链接脚本的一看

源码在编译生成.o 的文件后需要使用 lds 链接脚本来指定这些.o 文件的排列方式最后生成二进制文件。

http://blog.csdn.net/itxiebo/article/details/50938753

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :
    {
        *(.__image_copy_start)
        *(.vectors)
        CPUDIR/start.o (.text*)
        *(.text*)
    }

    . = 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);

    .image_copy_end :
    {
```

输出格式 elf32 的 arm 小端模式。输出架构是 arm。整个程序的入口地址是 _start

然后就开始排列了。.0x00000000;就是说当前地址是 0，这个在调用 lds 脚本是会在外部指定链接地址的。外部指定的比这个 0 的优先级要高，所以并不会真正的连接到 0 地址的。

```
ifneq ($(CONFIG_SYS_TEXT_BASE),)
LDFLAGS_u-boot += -Ttext $(CONFIG_SYS_TEXT_BASE)
```

```
CONFIG_SYS_TEXT_BASE=0x80800000
```

.__image_copy_start;一点就是指当前地址，也就是__image_copy_start 这个标号就是指向当前地址，这个主要是便于程序中重定位是拷贝程序是用的，后面还有一个 end。

然后开始排列_start 入口和其他的所有的 text 代码段。完了是 rodata 段、data 段 .ALIGN(4)是当前地址开始 4 字节对齐后在接着放。

接下来是 u_boot_list 段，这个是后面分析命令时有用的。后面还有一些其他的段，包括 bss 段。

下图则是 System.map 中的关于代码的排列顺序。

```
80800000 T __image_copy_start
80800000 T _start
80800020 t _reset
80800024 T _undefined_instruction
80800028 T _software_interrupt
8080002c T _prefetch_abort
80800030 T _data_abort
80800034 T _not_used
80800038 T _irq
8080003c T _fiq
80800040 T IRQ_STACK_START_IN
80800060 t undefined_instruction
808000c0 t software_interrupt
80800120 t prefetch_abort
80800180 t data_abort
808001e0 t not_used
80800240 t irq
808002a0 t fiq
80800300 T reset
80800338 T c_runtime_cpu_setup
80800354 T cpu_init_cp15
80800390 T save_boot_params
8080039c T set_pl310_ctrl_reg
808003b4 t v7_maint_dcache_all
```

九 、MLO 程序启动流程

MLO 程序在编译时在 Makefile.spl 中会生成 CONFIG_SPL_BUILD 的宏定义，这个就区分了 MLO 和 UBOOT 的不同的执行流程。

那么开始吧。我以为_start 会在 start.S 文件中，而且很多版本的 uboot 都是在这个文件中，但是我看的这个就是在 vectors.S 文件中。

```
50: _start:
51:     ldr pc, _reset
52:     ldr pc, _undefined_instruction
53:     ldr pc, _software_interrupt
54:     ldr pc, _prefetch_abort
55:     ldr pc, _data_abort
56:     ldr pc, _not_used
57:     ldr pc, _irq
58:     ldr pc, _fiq
```

然后跳转_reset

```
78: _reset:         .word reset
```

reset 在 start.S 文件中后面尽量不截图代码了，感觉太多，截不过来，还是需要对着代码来看的。

```
35: reset:
36:     bl  save_boot_params
37:     /*
38:      * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
39:      * except if in HYP mode already
40:      */
41:     mrs r0, cpsr
42:     and r1, r0, #0x1f       @ mask mode bits
43:     teq r1, #0x1a       @ test for HYP mode
44:     bicne   r0, r0, #0x1f       @ clear all mode bits
45:     orrne   r0, r0, #0x13       @ set SVC mode
46:     orr r0, r0, #0xc0       @ disable FIQ and IRQ
47:     msr cpsr,r0
48:
```

保存一些参数，一般不用管它。我觉得现在的水平真不能每个函数都去扣，先了解主要流程再说吧。下面是在关闭 FIQ 和 IRQ，就是关闭了中断，然后设置进入 SVC32 模式。Supervisor 超级管理模式。

然后设置向量表，然后调用这两个都是在 start.S 文件中

```
67:     bl  cpu_init_cp15
68:     bl  cpu_init_crit
```

cpu_init_cp15　关闭 L1 的 I/Dcache 就是关闭了数据和指令的 cache 缓冲。

cpu_init_crit　直接跳转 lowlevel_init

```
191:     b   lowlevel_init       @ go setup pll,mux,memory
```

lowlevel_init 在 lowlevel_init.S 文件中，设置了暂时的栈，为了调用 C 函数。sp 指向 CONFIG_SYS_INIT_SP_ADDR。这个宏是在 include/configs/ti_armv7_common.h

中定义的。然后减去了全局变量结构体的大小。

sp 8 字节对齐 设置全局变量 保存 ip 指针 跳转到 s_init。

s_init 在 arch/arm/cpu/armv7/am33xx/board.c 文件中。有些宏定义在 Source Insight 显示是没有定义的，但是也不敢保证就真的没有定义。哎。。。。

```
#ifdef CONFIG_SPL_BUILD
    save_omap_boot_params();
#endif
    watchdog_disable();
    timer_init();
    set_uart_mux_conf();
    setup_clocks_for_console();
    uart_soft_reset();
```

```
#elif defined(CONFIG_SPL_BUILD)
    gd = &gdata;
    preloader_console_init();
#endif
```

```
void preloader_console_init(void)
{
    gd->bd = &bdata;
    gd->baudrate = CONFIG_BAUDRATE;

    serial_init();      /* serial communications setup */

    gd->have_console = 1;

    puts("\nU-Boot SPL " PLAIN_VERSION " (" U_BOOT_DATE " - " \
            U_BOOT_TIME ")\n");
#ifdef CONFIG_SPL_DISPLAY_PRINT
    spl_display_print();
#endif
}
```

UBOOT 的串口初始化在这。SPL 启动的时候打印的这句话就是在这打印出来的。

```
#ifdef CONFIG_SPL_BUILD
    board_early_init_f();
    sdram_init();
#endif
```

在 board_early_init_f();中初始化时钟 PLL 和使能了板子上的引脚复用。

board_early_init_f();函数中有 prcm_init();

在 prcm_init();函数中有 scale_vcores();还有 PLL 时钟的各种设置。

在 scale_vcores();函数函数中处理读取 eeprom 来获取板件信息。这个是 ti 的 evm 系列的开发板有的，根据这个来获取开发板具体型号的。

sdram_init();就是完成了 DDR 的初始化。

回到 start.S 后又跳转到_main，这个位于 arch/arm/lib/crt0.S 文件中

再一次设置了栈指针指向了 CONFIG_SPL_STACK 这个地址是 DDR 开始地址 0x800000000+32KB 的地方 8 字节对齐后调用了 board_init_f(0);多看一下注释讲的很清楚的。r0 就是传给 board_init_f 参数。当参数小于等于 4 个时是保存在 r0123

寄存器里的再多的话是在栈中保存的。可以看出来全局变量也在栈里参合着呢目前看来。

```
58: ENTRY(_main)
59:
60: /*
61:  * Set up initial C runtime environment and call board_init_f(0).
62:  */
63:
64: #if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
65:     ldr sp, =(CONFIG_SPL_STACK)
66: #else
67:     ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
68: #endif
69:     bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
70:     sub sp, sp, #GD_SIZE    /* allocate one GD above SP */
71:     bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
72:     mov r9, sp      /* GD is above SP */
73:     mov r0, #0
74:     bl  board_init_f
75:
76: #if ! defined(CONFIG_SPL_BUILD)
77:
```

说一句的是_main 中的 board_init_f();函数之后的代码通通没有执行。要记着在 MLO/SPL 执行的时候 CONFIG_SPL_BUILD 是有效的啊。

board_init_f();函数在 arch/arm/lib/spl.c 文件中

```
26: void __weak board_init_f(ulong dummy)
27: {
28:     /* Clear the BSS. */
29:     memset(__bss_start, 0, __bss_end - __bss_start);
30:
31:     /* Set global data pointer. */
32:     gd = &gdata;
33:
34:     board_init_r(NULL, 0);
35: }
36:
```

清除了 bss 段获取了全局变量的地址，然后调用了位于 common/spl/spl.c 文件中的 board_init_r();函数。

在这个函数中执行 spl_board_init();初始化了一些开发板上的用到的一些初始化。比如用到了 gpmc 接口的初始化。如果有定义的话还有一些 USB 之类的初始化。

然后调用了 spl_boot_device();来获取启动介质，接着就是 switch 语句来加载和启动 uboot 了。这个函数其实就是去读取了配置启动引脚的那些配置来确定的。

```
153:
154:        boot_device = spl_boot_device();
155:        debug("boot device - %d\n", boot_device);
156:        switch (boot_device) {
157:    #ifdef CONFIG_SPL_RAM_DEVICE
158:        case BOOT_DEVICE_RAM:
159:            spl_ram_load_image();
160:            break;
161:    #endif
162:    #ifdef CONFIG_SPL_MMC_SUPPORT
163:        case BOOT_DEVICE_MMC1:
164:        case BOOT_DEVICE_MMC2:
165:        case BOOT_DEVICE_MMC2_2:
166:            spl_mmc_load_image();
167:            break;
168:    #endif
```

MMC 是就用的 spl_mmc_load_image();函数，在函数中会调用 spl_start_boot();这两句是在 spl_start_boot();打印出来的。

```
SPL: Please implement spl_start_uboot() for your board
SPL: Direct Linux boot not active!
```

在 spl_mmc_load_image(); 函数中初始化了 MMC 相关的东西。spl_boot_mode();函数获取了 boot 方式，有 RAW 的也有 FAT 方式的。FAT 方式是在 windows 和 linux 下直接能看的见得所以把程序直接拷贝带 SD 卡就能执行而不用费劲去写 sd 卡内部的扇区了。spl_load_image_fat_os();函数会读取环境变量中的文件名。但在 SPL 中好像没有设置环境变量呢。（实测没有环境变量的）失败后会退回执行 spl_load_image_fat();函数。并且掺入下面这两个作为参数。

```
/* FAT sd card locations. */
#define CONFIG_SYS_MMC_SD_FAT_BOOT_PARTITION    1
#define CONFIG_SPL_FAT_LOAD_PAYLOAD_NAME    "u-boot.img"
```

这里还出现了 u-boot.img 的头部信息。正好就是比 u-boot.bin 多的 64 个字节啊。

```
254:  */
255: typedef struct image_header {
256:     __be32      ih_magic;   /* Image Header Magic Number    */
257:     __be32      ih_hcrc;    /* Image Header CRC Checksum    */
258:     __be32      ih_time;    /* Image Creation Timestamp */
259:     __be32      ih_size;    /* Image Data Size      */
260:     __be32      ih_load;    /* Data  Load  Address      */
261:     __be32      ih_ep;      /* Entry Point Address      */
262:     __be32      ih_dcrc;    /* Image Data CRC Checksum  */
263:     uint8_t     ih_os;      /* Operating System     */
264:     uint8_t     ih_arch;    /* CPU architecture     */
265:     uint8_t     ih_type;    /* Image Type           */
266:     uint8_t     ih_comp;    /* Compression Type     */
267:     uint8_t     ih_name[IH_NMLEN];  /* Image Name       */
268: } image_header_t;
269:
```

```
    err = file_fat_read(filename, header, sizeof(struct image_header));
    if (err <= 0)
        goto end;

    spl_parse_image_header(header);

    err = file_fat_read(filename, (u8 *)spl_image.load_addr, 0);
```

file_fat_read();函数就是读取并且填充了头部信息。然后解析了头部信息。因为里面包含了 u-boot 的加载地址。然后再一次将 uboot 读取进来。这里是 0 不知道为啥啊？

```
long file_fat_read_at(const char *filename, unsigned long pos, void *buffer,
            unsigned long maxsize)
{
    printf("reading %s\n", filename);
    return do_fat_read_at(filename, pos, buffer, maxsize, LS_NO, 0);
}
```

回归 board_init_r();函数。最后执行这个函数后跳转执行 u-boot 去了。

```
239:     jump_to_image_no_args(&spl_image);
240: } « end board_init_r »
```

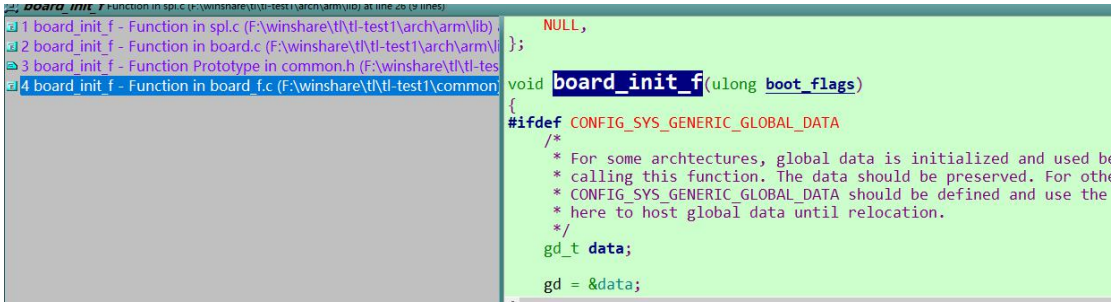
这个里面就是函数指针方式的跳转。

2018-01-20 周六

十 、UBOOT 程序启动流程

uboot 的执行流程其实和 MLO 基本一致的，只是因为 CONFIG_SPL_BUILD 这个宏定义在 uboot 执行时是没有定义的所以会不在执行已经在 MLO 执行过的函数。比如 DDR 和 PLL 以及 PINMUX 之类的代码。

uboot 的运行在进入 ctr0.S 文件下的_main 后开始和 MLO 就分了。同样是 board_init_f(); 这个函数名，但是不是在 spl.c 文件里面那个了。是在 common/board_f.c 文件中的同名函数。



这个函数显示处理了一下 global data，然后就是 initcall_run_list(init_sequence_f);

自己真的是孤陋寡闻、薄才惰艺。这种操作还真是头回见。

```
982:
983:     gd->flags = boot_flags;
984:     gd->have_console = 0;
985:
986:     if (initcall_run_list(init_sequence_f))
987:         hang();
988:
```

init_sequence_f 是一个函数指针数组名。里面包含了一众参数是 void 返回值是一个 int 的函数。然后一个 initcall_run_list 就把所有的函数执行一遍。

```
typedef int (*init_fnc_t)(void);
```

```
779: static init_fnc_t init_sequence_f[] = {
780: #ifdef CONFIG_SANDBOX
781:     setup_ram_buf,
782: #endif
783:     setup_mon_len,
784:     setup_fdt,
785:     trace_early_init,
786: #if defined(CONFIG_MPC85xx) || defined(CONFIG_MPC86xx)
787:     /* TODO: can this go into arch_cpu_init()? */
788:     probecpu,
789: #endif
790:     arch_cpu_init,      /* basic arch cpu dependent setup */
791: #ifdef CONFIG_X86
792:     cpu_init_f,     /* TODO(sjg@chromium.org): remove */
```

让我们荡起双桨，小船儿。。额，让我们来大致看一下这一堆函数吧。

```
static init_fnc_t init_sequence_f[] = {
    setup_mon_len,
    setup_fdt,
    trace_early_init,
    arch_cpu_init,
    mark_bootstage,
    board_early_init_f, //跟MLO一样DDR PLL PINMUX
    env_init,           //环境变量初始化
    init_baud_rate,     //波特率赋值给gd
    serial_init,        //串口
    console_init_f,
    display_options,    //打印版本信息
    display_text_info,
    print_cpuinfo,
    INIT_FUNC_WATCHDOG_INIT
    INIT_FUNC_WATCHDOG_RESET
    init_func_i2c,      //打印I2C
    announce_dram_init, //打印DRAM
    INIT_FUNC_WATCHDOG_RESET
    INIT_FUNC_WATCHDOG_RESET
    INIT_FUNC_WATCHDOG_RESET
    setup_dest_addr,    //设置了relocaddr地址从DDR顶端向下预留内存
    reserve_round_4k,   //更改reloaddr地址预留出4K
    reserve_trace,
    reserve_uboot,
```

```
    reserve_trace,
    reserve_uboot,
    reserve_malloc,
    reserve_board,      //各种预留内存后更改relocaddr地址
    setup_machine,
    reserve_global_data, //预留了全局变量
    reserve_fdt,
    reserve_stacks,
    setup_dram_config,
    show_dram_config,   //打印DDR大小 512MiB
    display_new_sp,
    INIT_FUNC_WATCHDOG_RESET
    reloc_fdt,
    setup_reloc,        //在这reloaddr=0x9F731000了
    jump_to_copy,       //跳转去复制了
    NULL,
};
```

环境变量的初始化，是直接使用了默认值。后面还回有从启动介质中读取环境变量的过程，如果成功的话会覆盖这些默认值的。环境变量后面还回单独拿出来研究的。测了一下，jump_to_copy 不是在这执行的。

```
3 env_init - Function in env_remote.c (F:\winshare\tl\tl-test1\common)
4 env_init - Function in env_nowhere.c (F:\winshare\tl\tl-test1\commo
5 env_init - Function in env_mmc.c (F:\winshare\tl\tl-test1\common) a
6 env_init - Function in env_flash.c (F:\winshare\tl\tl-test1\common) a
7 env_init - Function in env_nand.c (F:\winshare\tl\tl-test1\common) a
8 env_init - Function in env_dataflash.c (F:\winshare\tl\tl-test1\commc
9 env_init - Function in env_nvram.c (F:\winshare\tl\tl-test1\common)
10 env_init - Function in env_onenand.c (F:\winshare\tl\tl-test1\comm
11 env_init - Function in env_eeprom.c (F:\winshare\tl\tl-test1\commo
```

```
int env_init(void)
{
    /* use default */
    gd->env_addr    = (ulong)&default_environment[0];
    gd->env_valid   = 1;

    return 0;
}
```

定时器初始化串口波特率串口初始化。。。

```
836:    init_baud_rate,     /* initialze baudrate settings */
837:    serial_init,        /* serial communications setup */
838:    console_init_f,     /* stage 1 init of console */
```

因为我是拿的开发板配套的 uboot 在看，所以我特别喜欢看打印出来的东西，然后去源码里查找打印的地方来追踪执行流程。

```
U-Boot 2014.07 (Jan 19 2018 - 10:57:27)

SI2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
reading uboot.env

** Unable to read "uboot.env" from mmc0:1 **
Using default environment

Could not probe the EEPROM at 0x50
Could not get board ID.
Net:   <ethaddr> not set. Validating first E-fuse MAC
cpsw, usb_ether
```

第一行版本信息。是在这打印出来的。

```
int display_options (void)
{
#if defined(BUILD_TAG)
    printf ("\n\n%s, Build: %s\n\n", version_string, BUILD_TAG);
#else
    printf ("\n\n%s\n\n", version_string);
#endif
    return 0;
}
```

第二行 SI2C 就比较尴尬了，其实这 I2C 那个 S 是我自己添加的，看看是不是我找的那个地方打印出来的。后来忘了改回去了。

```
static int init_func_i2c(void)
{
    puts("I2C:   ");
#ifdef CONFIG_SYS_I2C
    i2c_init_all();
#else
    i2c_init(CONFIG_SYS_I2C_SPEED, CONFIG_SYS_I2C_SLAVE);
#endif
    puts("ready\n");
    return (0);
}
```

第三行的 DRAM：是一个来自于 announce_dram_init 函数，

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

在另一个函数中 show_dram_config 中显示了容量的大小。512MiB

```
943:    show_dram_config,
```

回到 crt0.S 文件的_main 中运行 relocate_code 在 arch/arm/lib 下的 relocate.S 中

```
23: ENTRY(relocate_code)
24:     ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
25:     subs    r4, r0, r1      /* r4 <- relocation offset */
26:     beq relocate_done       /* skip relocation */
27:     ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
28:
```

链接脚本中的__image_cpoy_start 排上用场了。借用 ti 的话：实现 uboot 的重定位，也即是将 uboot 的程序搬移到 DDR 内存中合适的位置去执行。作用有二：一是为 kernel 腾出低端空间，防止接下来的引导进来的 kernel 解压覆盖 uboot，而是静态存储器（spiflash nandflash）启动，这个 relocation 是必须的。

不是很明白。uboot 在连接时就已经指定了运行地址了，而且 uboot 有很多全局变量所以是位置相关代码，怎么能在运行时在挪到其他地方呢。当 kernel 解压的时候 uboot 就已经完成使命了，覆盖了也没啥事啊。spiflash nandflash 不只是一个代码存储介质么，当 MLO 把 uboot 读到 DDR 中不就没它什么事了么。

http://www.360doc.com/content/17/0525/17/37911645_657208881.shtml

这个链接是讲述为什么进行这种重定位，准确的说应该是讲解如何实现的重定位。

接下来设置了 C 的运行环境，是最后的运行环境了。并且清除了 bss 段，这里的代码已经是 relocation 后的了。

```
93:     b   relocate_code
94: here:
95:
96: /* Set up final (full) environment */
97:
98:     bl  c_runtime_cpu_setup /* we still call old routine here */
99:
100:    ldr r0, =__bss_start    /* this is auto-relocated! */
101:    ldr r1, =__bss_end      /* this is auto-relocated! */
102:
103:    mov r2, #0x00000000     /* prepare zero to clear BSS */
104:
```

接着调用 board_init_r();并且传参到 r0 和 r1。

```
112:
113:    /* call board_init_r(gd_t *id, ulong dest_addr) */
114:    mov     r0, r9                  /* gd_t */
115:    ldr r1, [r9, #GD_RELOCADDR] /* dest_addr */
116:    /* call board_init_r */
117:    ldr pc, =board_init_r   /* this is auto-relocated! */
118:
119:    /* we should not return here. */
```

在 common/board_init_r.c 文件中。

```
935:
936:    if (initcall_run_list(init_sequence_r))
937:        hang();
938:
```

一样的套路。

在这里输出了 NAND：nand_init();里面输出了 512MiB

```
int initr_nand(void)
{
    puts("NAND:  ");
    nand_init();
    return 0;
}
```

在这里输出了 MMC 相关的那行信息。

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

这个函数打印出了和环境变量相关的东西。环境变量是以一个叫 uboot.env 的文件存在 SD 启动卡的 boot 分区的。如果没有这个文件就会使用默认的环境变量。

common/board_r.c 的 initr_env();

common/env_common.c 中 env_relocate();

common/env_fat.c 中 env_relocate_spec();

fs/fat/fat.c 中 file_fat_read()；涉及到读取文件了啊

```
444: static int initr_env(void)
445: {
446:     /* initialize environment */
447:     if (should_load_env())
448:         env_relocate();
449:     else
450:         set_default_env(NULL);
451:
452:     /* Initialize from environment */
453:     load_addr = getenv_ulong("loadaddr", 16, load_addr);
```

网络初始化在这，但是我在 uboot 中用网口也是不太能行啊，目前没有细看这块的初始化。

```
static int initr_net(void)
{
    puts("Net:   ");
    eth_initialize(gd->bd);
#if defined(CONFIG_RESET_PHY_R)
    debug("Reset Ethernet PHY\n");
    reset_phy();
#endif
    return 0;
}
```

最后直接进入了 run_main_loop 函数，然后又进入 common/main.c 里面的 main_loop 函数

```
704: static int run_main_loop(void)
705: {
706: #ifdef CONFIG_SANDBOX
707:     sandbox_main_loop_init();
708: #endif
709:     /* main_loop() can return to retry autoboot, if so just run it again */
710:     for (;;)
711:         main_loop();
712:     return 0;
713: }
```

在 main_loop 中先进行了 bootdelay 的功能，然后根据 bootdelay 的不同来选择进入命令的命令体系还是直接启动内核。感觉又是比较重要的，或者说这才是 uboot 真正干实事的地方。

十一 、main_loop 简单分析

modem_init();内部有个条件编译，所以这是个空函数。没干什么事情。

setenv();是把新生成的版本信息写进环境变量中。

```
57: void main_loop(void)
58: {
59:     const char *s;
60:
61:     bootstage_mark_name(BOOTSTAGE_ID_MAIN_LOOP, "main_loop");
62:
63: #ifndef CONFIG_SYS_GENERIC_BOARD
64:     puts("Warning: Your board does not use generic board. Please read\n");
65:     puts("doc/README.generic-board and take action. Boards not\n");
66:     puts("upgraded by the late 2014 may break or be removed.\n");
67: #endif
68:
69:     modem_init();
70: #ifdef CONFIG_VERSION_VARIABLE
71:     setenv("ver", version_string);  /* set version variable */
72: #endif /* CONFIG_VERSION_VARIABLE */
73:
74:     cli_init();
75:
76:     run_preboot_environment_command();
77:
78: #if defined(CONFIG_UPDATE_TFTP)
79:     update_tftp(0UL);
80: #endif /* CONFIG_UPDATE_TFTP */
81:
82:     s = bootdelay_process();
83:     if (cli_process_fdt(&s))
84:         cli_secure_boot_cmd(s);
85:
86:     autoboot_command(s);
87:
88:     cli_loop();
89: } « end main_loop »
```

cli_init();开始为后面循环做一些初始化，比如解析器的初始化。

bootdelay_process();读取环境变量和默认的比较。

```
45:     s = getenv("bootdelay");
46:     bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
47:
```

在 bootdelay_process();函数中同时也读取了 bootcmd 环境变量。这个主要就是自动 boot 时执行的命令。

```
272:         s = getenv("bootcmd");
273:
274:     process_fdt_options(gd->fdt_blob);
275:     stored_bootdelay = bootdelay;
276:
277:     return s;
278: } « end bootdelay_process »
```

autoboot_command(s);执行了 bootdelay 延时，然后要么投币，要么刷卡，要么滚蛋，戴个墨镜，当自己黑客帝国呢！额。。要么执行 S 也就是 bootcmd 自动启

动 kernel，要么退出来进入 cli_loop();进入命令体系中。

```
280: void autoboot_command(const char *s)
281: {
282:     debug("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");
283:
284:     if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay)) {
285: #if defined(CONFIG_AUTOBOOT_KEYED) && !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
286:         int prev = disable_ctrlc(1);    /* disable Control C checking */
287: #endif
288:
289:         run_command_list(s, -1, 0);
290:
291: #if defined(CONFIG_AUTOBOOT_KEYED) && !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
292:         disable_ctrlc(prev);    /* restore Control C checking */
293: #endif
294:     }
```

abortboot();函数执行调用 abortboot_normal(bootdelay);

```
152:     if (bootdelay >= 0)
153:         printf("Hit any key to stop autoboot: %2d ", bootdelay);
```

那句话就是在这里打印出来的。然后检测是否有按键输入，有就直接返回 1.退出自动启动。否则返回 0，则在 autoboot_command();中 if 语句成立，进入里面执行 run_command_list(s,-1,0);开始自动启动，其实就是执行了 bootcmd 里面的命令。对于命令的执行在后面会再来讲解 uboot 的命令体系。

看后缀带个_list 就是一个命令列表，而且 bootcmd 也确实是多个命令的集合。

common/cli.c 文件中 cli_loop();这里面根据 CONFIG_SYS_HUSH_PARSER 这个宏定义了两种循环方式。cli_simple_loop();是比较好看懂的读取一行输入，然后解析然后执行然后在读取....这样的方式。另一种时采用了 HUSH 的解析器方式。看着挺吓人的。

```
198: void cli_loop(void)
199: {
200: #ifdef CONFIG_SYS_HUSH_PARSER
201:     parse_file_outer();
202:     /* This point is never reached */
203:     for (;;);
204: #else
205:     cli_simple_loop();
206: #endif /*CONFIG_SYS_HUSH_PARSER*/
207: }
```

common/cli_hush.c 文件中 parse_file_outer();

```
3260: int parse_file_outer(void)
3261: #endif
3262: {
3263:     int rcode;
3264:     struct in_str input;
3265: #ifndef __U_BOOT__
3266:     setup_file_in_str(&input, f);
3267: #else
3268:     setup_file_in_str(&input);
3269: #endif
3270:     rcode = parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);
3271:     return rcode;
3272: }
3273:
```

setup_file_in_str();函数就是输入的初始化。

http://blog.csdn.net/andy_wsj/article/details/8614905

这个链接主要讲 main_loop 循环方式的。

```
static void setup_file_in_str(struct in_str *i)
#endif
{
    i->peek = file_peek;
    i->get = file_get;
    i->__promptme=1;
    i->promptmode=1;
#ifndef __U_BOOT__
    i->file = f;
#endif
    i->p = NULL;
}
```

file_peek 调用的是 fgetc();

```
13: static int file_peek(struct in_str *i)
14: {
15: #ifndef __U_BOOT__
16:     if (i->p && *i->p) {
17: #endif
18:         return *i->p;
19: #ifndef __U_BOOT__
20:     } else {
21:         i->peek_buf[0] = fgetc(i->file);
22:         i->peek_buf[1] = '\0';
23:         i->p = i->peek_buf;
24:         debug_printf("b_peek: got a %d\n", *i->p);
25:         return *i->p;
26:     }
27: #endif
28: }
```

file_get 调用了 get_user_input(i);它又调用了 cli_readline();函数，感觉又回到了那个 simple 的套路了啊。

```
1006:         n = cli_readline(CONFIG_SYS_PROMPT);
1007:     } else {
```

```
#define CONFIG_SYS_PROMPT       "U-Boot# "
#define CONFIG_SYS_CONSOLE_INFO_QUIET
```

那个宏就是在 uboot 的命令时前端的引导标识啊。

虽然初始化时将 i->get = file_get ，但是在哪使用呢？？？

初始化之后，parse_file_outer 函数内调用主循环函数：

rcode = parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);

将初始化之后的 input 传给了主循环，主循环继续调用

rcode = parse_stream(&temp, &ctx, inp, '\n');

将 inp(即 input)传递给 parse_stream

在 parse_stream 函数内循环读取输入：

while ((ch=b_getch(input))!=EOF) {

b_getch 是一个宏： #define b_getch(input) ((input)->get(input))

实际就是 while ((ch = input->get(input)) != EOF) {

如此以来就调用到了 readline ，然后调用关系：readline-->readline_into_buffer-->getc

getc 再调用 serial_getc，实现与串口输入关联，与 simple 方式一相同。

parse_stream_outer 函数中以 do{}while();方式实现循环。

```
3162:     o_string temp=NULL_O_STRING;
3163:     int rcode;
3164: #ifdef __U_BOOT__
3165:     int code = 0;
3166: #endif
3167:     do {
3168:         ctx.type = flag;
3169:         initialize_context(&ctx);
3170:         update_ifs_map();
3171:         if (!(flag & FLAG_PARSE_SEMICOLON) || (flag & FLAG_REPARSING)) mapset
3172:         inp->promptmode=1;
3173:         rcode = parse_stream(&temp, &ctx, inp, '\n');
```

rcode = parse_stream(&temp, &ctx, inp, '\n');这个函数是读取一条命令并解析之。

code = run_list(ctx.list_head);执行命令　　common/cli_hush.c

run_list_real(pi);　　common/cli_hush.c

rcode = run_pipe_real(pi);　　common/cli_hush.c

```
1655:         /* Process the command */
1656:         return cmd_process(flag, child->argc, child->argv,
1657:                            &flag_repeat, NULL);
```

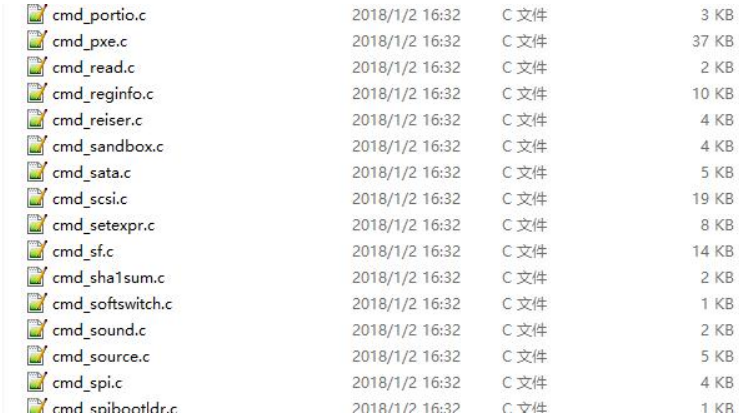
大致意思就是将收到的指令通过一系列处理加入一个执行列表，然后执行这个列表。还有一大堆的控制、出错处理.....

最后，hush 方式也调用到 cmd_process 函数，这又与 simple 相同。

2018-01-22 周一

十二 、UBOOT 的命令体系

在启动 UBOOT 后，如果在 bootdelay 执行完之前按下了任何按键，则就进入了 uboot 的命令行中。开始进入上一节讲述的循环中接受和处理命令。在 UBOOT 中试定义了一些命令的。代码主要存放在 common/目录下以 cmd_xx.c 格式命名的一堆文件中和 command.c 文件。还有 include 目录下的 command.h 文件。



cmd_portio.c	2018/1/2 16:32	C 文件	3 KB
cmd_pxe.c	2018/1/2 16:32	C 文件	37 KB
cmd_read.c	2018/1/2 16:32	C 文件	2 KB
cmd_reginfo.c	2018/1/2 16:32	C 文件	10 KB
cmd_reiser.c	2018/1/2 16:32	C 文件	4 KB
cmd_sandbox.c	2018/1/2 16:32	C 文件	4 KB
cmd_sata.c	2018/1/2 16:32	C 文件	5 KB
cmd_scsi.c	2018/1/2 16:32	C 文件	19 KB
cmd_setexpr.c	2018/1/2 16:32	C 文件	8 KB
cmd_sf.c	2018/1/2 16:32	C 文件	14 KB
cmd_sha1sum.c	2018/1/2 16:32	C 文件	2 KB
cmd_softswitch.c	2018/1/2 16:32	C 文件	1 KB
cmd_sound.c	2018/1/2 16:32	C 文件	2 KB
cmd_source.c	2018/1/2 16:32	C 文件	5 KB
cmd_spi.c	2018/1/2 16:32	C 文件	4 KB
cmd_spibootldr.c	2018/1/2 16:32	C 文件	1 KB

一个命令有一个描述该命令的结构体和对应的 do_xxx 开头的对应函数。若干个命令的在内存中存放靠的不是链表，也不是结构体数据，而是上次在连接脚本时说的自定义的段。下图是一个命令结构体的内容。include/command.h

```
30: struct cmd_tbl_s {
31:     char        *name;      /* Command Name         */
32:     int     maxargs;    /* maximum number of arguments  */
33:     int     repeatable; /* autorepeat allowed?      */
34:                     /* Implementation function  */
35:     int     (*cmd)(struct cmd_tbl_s *, int, int, char * const []);
36:     char        *usage;     /* Usage message    (short) */
37: #ifdef  CONFIG_SYS_LONGHELP
38:     char        *help;      /* Help  message    (long)  */
39: #endif
40: #ifdef CONFIG_AUTO_COMPLETE
41:     /* do auto completion on the arguments */
42:     int     (*complete)(int argc, char * const argv[], char last_char, int maxv, char *cmdv[]);
43: #endif
44: };
45:
46: typedef struct cmd_tbl_s    cmd_tbl_t;
```

name 是命令的名字。

maxargs 是设定的最大的传参个数

repeatable 是设定这个命令是否可以重复执行，就是当你在命令行输入一行命令执行后，当你接着在一次不输入命令，只按回车（就是输入空）他会执行上一次执行的那个命令。

(*cmd)(struct cmd_tbl_s *, int, int, char * const []);这个就是命令本体了，是一个函数指针，指向 do_开头的执行函数。

*usage; 是简短的使用帮助。

*help; 是长使用帮助说明

(*complete)(int argc, char * const argv[], char last_char, int maxv, char *cmdv[]);这个是自动补全命令的一个函数，跟 linux 中那个效果一样的。

****

每一个命令都有这样一个结构体的，让我们来看一下定义。这就是 saveenv 命令的所有身家了。

```
689: static int do_env_save(cmd_tbl_t *cmdtp, int flag, int argc,
690:                char * const argv[])
691: {
692:     printf("Saving Environment to %s...\n", env_name_spec);
693:
694:     return saveenv() ? 1 : 0;
695: }
696:
697: U_BOOT_CMD(
698:     saveenv, 1, 0,  do_env_save,
699:     "save environment variables to persistent storage",
700:     ""
701: );
```

U_BOOT_CMD 是一个宏定义。在 include/command.h

```
175:
176: #define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help)      \
177:     U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, NULL)
178:
```

这个宏调用了 U_BOOT_CMD_COMPLETE 的宏，有调用了两个宏，晕。。。

```
171: #define U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, _comp) \
172:     ll_entry_declare(cmd_tbl_t, _name, cmd) =           \
173:         U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,  \
174:                     _usage, _help, _comp);
```

11_entry_declare 在 include/linker_lists.h 文件中，

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

_type 是命令的结构体类型，_name 是命令的名字，cmd 是函数指针。##两个这个在 C 中是连接符的意思，整个这个的意思就是定义结构体并把这个结构体放到了内存中的.u_boot_list 开头的那一段，并且有新的标号名也是结构体名_u_boot_list_2_cmd_2_name,如果是 save 命令就是_u_boot_list_2_cmd_2_saveenv 接下来是这个宏 U_BOOT_CMD_MKENT_COMPLETE，这个其实就是在给刚定义的结构体赋值。_CMD_HELP 和_CMD_COMPLETE 是两个宏，根据条件编译来确定用

不用包含 help 和 complete 内容。

```
162: #define U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,      \
163:                   _usage, _help, _comp)                \
164:         { #_name, _maxargs, _rep, _cmd, _usage,          \
165:             _CMD_HELP(_help) _CMD_COMPLETE(_comp) }
```

```
#ifdef CONFIG_AUTO_COMPLETE
# define _CMD_COMPLETE(x) x,
#else
# define _CMD_COMPLETE(x)
#endif
#ifdef CONFIG_SYS_LONGHELP
# define _CMD_HELP(x) x,
#else
# define _CMD_HELP(x)
#endif
```

总体来说呢，就是通过宏定义来定义了一个命令的结构体，并且添加属性连接到.u_boot_list 的那一段去了。

```
38      . = ALIGN(4);
39      .u_boot_list : {
40          KEEP(*(SORT(.u_boot_list*)));
41      }
```

KEEP 关键字是为了保证所有的段多被加进来，不要被链接器自作聪明的把某些它认为没有的段舍弃，事实上我们确实是定义了一些会让它看起来没用的段，这个我们后面会提到；而 SORT 关键字如其名字就是根据段名字串进行排序然后存放。*********现在来看一下 System.map 中的分布图吧。

```
1726 8085bc64 D _u_boot_list_2_cmd_2_base
1727 8085bc80 D _u_boot_list_2_cmd_2_bdinfo
1728 8085bc9c D _u_boot_list_2_cmd_2_boot
1729 8085bcb8 D _u_boot_list_2_cmd_2_bootd
1730 8085bcd4 D _u_boot_list_2_cmd_2_bootm
1731 8085bcf0 D _u_boot_list_2_cmd_2_bootp
1732 8085bd0c D _u_boot_list_2_cmd_2_bootz
1733 8085bd28 D _u_boot_list_2_cmd_2_chpart
1734 8085bd44 D _u_boot_list_2_cmd_2_cmp
1735 8085bd60 D _u_boot_list_2_cmd_2_coninfo
```

我只是截图截了一部分，还有挺多的。可能现在还是有疑问，虽然都放在了一块但是并没有标明存放的其实地址和长度啊，但是来查找还是无法下手啊。还是之前版本的好理解一点啊。之前的版本是这样的，看直接标明了命令段的其实和结束地址在代码中可以直接在这一段内存中遍历查询就可以。

```
56      __u_boot_cmd_start = .;
57      .u_boot_cmd : { *(.u_boot_cmd) }
58      __u_boot_cmd_end = .;
```

这种确实比较好理解。

不过咱们接着看下去。

接着 main_loop 那里说到，最后都执行到了 common/command.c 文件中 cmd_process 函数。find_cmd 函数就是来在内存中寻找命令结构体的，你看他的返回值也是一个命令结构体。

```
503: enum command_ret_t cmd_process(int flag, int argc, char * const argv[],
504:                   int *repeatable, ulong *ticks)
505: {
506:     enum command_ret_t rc = CMD_RET_SUCCESS;
507:     cmd_tbl_t *cmdtp;
508:
509:     /* Look up command in command table */
510:     cmdtp = find_cmd(argv[0]);
511:     if (cmdtp == NULL) {
512:         printf("Unknown command '%s' - try 'help'\n", argv[0]);
513:         return 1;
514:     }
```

在 find_cmd 中，11_entry_start 和 11_entry_count 都是和刚才呢类似的宏定义啊。

```
122: cmd_tbl_t *find_cmd (const char *cmd)
123: {
124:     cmd_tbl_t *start = ll_entry_start(cmd_tbl_t, cmd);
125:     const int len = ll_entry_count(cmd_tbl_t, cmd);
126:     return find_cmd_tbl(cmd, start, len);
127: }
```

继续往下找，先定义了一个字符数组，很奇芭的是 0 个元素的数组，而且同样是声明在了和命令结构体的同一段内，但是人家_list_后面是 1 啊，这样在脚本中的 SORT 排序命令就用上了，而且因为这个数组 0 元素不占内存，所以为了避免编译器的优化，KEEP 命令就用上了。定义完成后，就是取地址并且强制类型转化为命令结构体的地址了。

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

让我们去 System.map 中看一下

```
1718 8085bc38 d TftpBlkSize
1719 8085bc48 D _u_boot_list_2_cmd_2_askenv
1720 8085bc48 d start.6243
1721 8085bc48 d start.6247
1722 8085bc48 d start.6292
1723 8085bc48 d start.6296
1724 8085bc48 d start.6323
1725 8085bc48 d start.6327
1726 8085bc64 D _u_boot_list_2_cmd_2_base
1727 8085bc80 D _u_boot_list_2_cmd_2_bdinfo
```

可以看出来 start 并不占内存而且是第一个命令的地址。

```
#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;                                     \
    })
```

在 11_entry_count 中标明了起始地址和结束地址，然后计算了长度。

真正在查找命令的是 find_cmd_tbl();函数，通过一个 for 循环来遍历那一段内存

```
104:    for (cmdtp = table;
105:         cmdtp != table + table_len;
106:         cmdtp++) {
107:        if (strncmp (cmd, cmdtp->name, len) == 0) {
108:            if (len == strlen (cmdtp->name))
109:                return cmdtp;   /* full match */
110:
111:            cmdtp_temp = cmdtp; /* abbreviated command ? */
112:            n_found++;
113:        }
114:    }
115:    if (n_found == 1) {         /* exactly one match */
116:        return cmdtp_temp;
117:    }
```

如果没有找到命令则就是返回 NULL 了

```
510:    cmdtp = find_cmd(argv[0]);
511:    if (cmdtp == NULL) {
512:        printf("Unknown command '%s' - try 'help'\n", argv[0]);
513:        return 1;
514:    }
515:
```

```
U-Boot# sdf
Unknown command 'sdf' - try 'help'
U-Boot# sd;
Unknown command 'sd' - try 'help'
U-Boot#
```

如果找到的话就是检查参数然后接调用 cmd_call 函数执行命令了，就是通过结构体的方式调用的啊

```
532:    /* If OK so far, then do the command */
533:    if (!rc) {
534:        if (ticks)
535:            *ticks = get_timer(0);
536:        rc = cmd_call(cmdtp, flag, argc, argv);
537:        if (ticks)
538:            *ticks = get_timer(*ticks);
539:        *repeatable &= cmdtp->repeatable;
540:    }
541:    if (rc == CMD_RET_USAGE)
542:        rc = cmd_usage(cmdtp);
543:    return rc;
544: } « end cmd_process »
```

```
493: static int cmd_call(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
494: {
495:     int result;
496:
497:     result = (cmdtp->cmd)(cmdtp, flag, argc, argv);
498:     if (result)
499:         debug("Command failed, result=%d", result);
500:     return result;
501: }
```

当然，当你调用有问题是他会打印出来 usage 的简短的使用说明来提醒你，如果使用 help 命令则打印长 help 帮助。

命令的执行大致先这样，具体每个命令时怎么实现的，可以去看源码，源码就在那摆着，随时等你去看。而且确实感觉如果能把 UBOOT 的源码全部看懂了，不用说全部，就单单一个字符串处理你如果整会了那就很屌了。加油加油。。。

十三 、UBOOT 的环境变量

环境变量这个东西，可以理解为对于这个系统来说就是一些全局的变量，当然这些都是以字符串的形式存储的，所以又涉及到了字符串的处理，那 256 个 ascii 码的字符感觉构成整个软件行业啊。

环境变量让我们可以不用修改 uboot 的源代码，而是通过修改环境变量来影响 uboot 运行时的一些数据和特性。比如说通过修改 bootdelay 环境变量就可以更改系统开机自动启动时倒数的秒数。环境变量相关的代码主要在 common 目录中以 env_xxx.c 文件中，因为我们指定的是在 mmc 中保存环境变量，所以在 env_mmc.c 文件中。在 include 目录下也有相关的 env_xxx.h 的头文件。

前面多少也涉及到了环境变量的初始化和从 SD 卡中读取 u-boot.env 的环境变量。这个是 u-boot.env 内容的一部分，是一个整的字符串，以 NULL 也就是‘\0’来隔开的，剩下的全部是‘\0’

在代码中初始化 env 时使用的是默认的环境变量，然后在环境变量的 relocate 时读取 SD 卡中的 u-boot.env，如果失败就是使用默认了。

在 include 的 environment.h 文件中定义了环境变量的结构体，包含 crc 校验码和一个是否有效的判断位和一个包含真正数据的字符数组。环境变量的大小在配置文件中也定义大概就是 64KB 然后减掉了头部那两个变量的大小差不多就是 64KB 减去 5 个字节的大小。

```
156: typedef struct environment_s {
157:     uint32_t    crc;        /* CRC32 over data bytes    */
158: #ifdef CONFIG_SYS_REDUNDAND_ENVIRONMENT
159:     unsigned char   flags;      /* active/obsolete flags    */
160: #endif
161:     unsigned char   data[ENV_SIZE]; /* Environment data     */
162: } env_t
```

在 include 目录下的 env_default.h 文件中定义了默认的环境量，如下图所示。

****

环境变量的优先级是比内部默认的值要高的，如果有环境变量是以环境变量的结果为主要参考的。

```
#ifdef DEFAULT_ENV_INSTANCE_EMBEDDED
env_t environment __PPCENV__ = {
    ENV_CRC,    /* CRC Sum */
#ifdef CONFIG_SYS_REDUNDAND_ENVIRONMENT
    1,      /* Flags: valid */
#endif
    {
#elif defined(DEFAULT_ENV_INSTANCE_STATIC)
static char default_environment[] = {
#else
const uchar default_environment[] = {
#endif
#ifdef  CONFIG_ENV_CALLBACK_LIST_DEFAULT
    ENV_CALLBACK_VAR "=" CONFIG_ENV_CALLBACK_LIST_DEFAULT "\0"
#endif
#ifdef  CONFIG_ENV_FLAGS_LIST_DEFAULT
    ENV_FLAGS_VAR "=" CONFIG_ENV_FLAGS_LIST_DEFAULT "\0"
#endif
#ifdef  CONFIG_BOOTARGS
    "bootargs=" CONFIG_BOOTARGS         "\0"
#endif
```

```
#ifdef  CONFIG_BOOTCOMMAND
    "bootcmd="  CONFIG_BOOTCOMMAND      "\0"
#endif
#ifdef  CONFIG_RAMBOOTCOMMAND
    "ramboot="  CONFIG_RAMBOOTCOMMAND       "\0"
#endif
#ifdef  CONFIG_NFSBOOTCOMMAND
    "nfsboot="  CONFIG_NFSBOOTCOMMAND       "\0"
#endif
#if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
    "bootdelay="    __stringify(CONFIG_BOOTDELAY)   "\0"
#endif
#if defined(CONFIG_BAUDRATE) && (CONFIG_BAUDRATE >= 0)
    "baudrate=" __stringify(CONFIG_BAUDRATE)    "\0"
#endif
#ifdef  CONFIG_LOADS_ECHO
    "loads_echo="   __stringify(CONFIG_LOADS_ECHO)  "\0"
```

中间的那些是一些宏这些宏又是在配置的头文件中定义的一堆宏，其实说到底真正的指定默认环境变量的地方还是配置头文件，在 include/configs 目录下的。将一个宏变为字符串就是前面加一个#号就行

```
#define __stringify_1(x...) #x
#define __stringify(x...)   __stringify_1(x)
```

和环境变量相关的几个命令主要是 printenv、setenv、saveenv 主要就是打印所有的环境变量、设置环境变量和保存环境变量。

这些个函数全部定义在 common/cmd_nvedit.c 文件中。

一个一个看一下这个是判断-a 选项的，就是打印所有的环境变量。

```

    if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'a') {
        argc--;
        argv++;
        env_flag &= ~H_HIDE_DOT;
    }
```

新版的 uboot 好像在内存中存储环境变量使用哈希表表示的，怪不得我是一点没看懂。

```
126:     if (argc == 1) {
127:         /* print all env vars */
128:         rcode = env_print(NULL, env_flag);
129:         if (!rcode)
130:             return 1;
131:         printf("\nEnvironment size: %d/%ld bytes\n",
132:             rcode, (ulong)ENV_SIZE);
133:         return 0;
134:     }
```

这个是打印所有的环境变量的，并且在最后输出了已经使用的和总共的空间。

主要还是利用 for 循环来实现打印所有的环境变量。

```
136:     /* print selected env vars */
137:     env_flag &= ~H_HIDE_DOT;
138:     for (i = 1; i < argc; ++i) {
139:         int rc = env_print(argv[i], env_flag);
140:         if (!rc) {
141:             printf("## Error: \"%s\" not defined\n", argv[i]);
142:             ++rcode;
143:         }
144:     }
```

这个是打印指定的环境变量的。也是先遍历后找到了在打印的。

setenv 就是用来设置环境变量的，如果原来有就直接改原来的如果没有则新建一个。其实我是不大明白新建这个有啥用，程序里面可定用不上你后加的还是自己随便叫的东西吧，只是保存一些只看的数据感觉。

如果设置是名字后面为空，就是删除这个环境变量了。

```
244:     /* Delete only ? */
245:     if (argc < 3 || argv[2] == NULL) {
246:         int rc = hdelete_r(name, &env_htab, env_flag);
247:         return !rc;
248:     }
249:
```

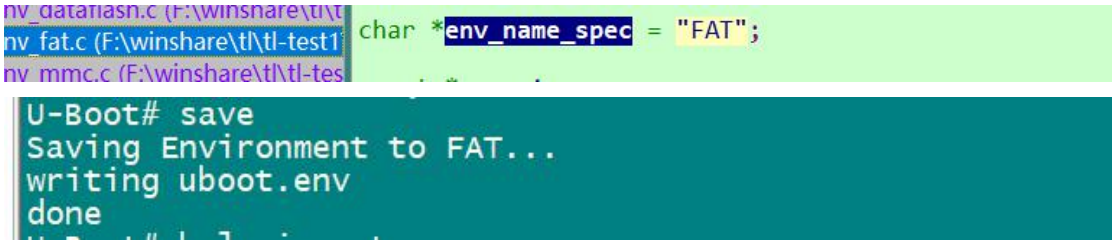
这就是用来删除的，传参个数小于 3 或者第三个参数是空就是删除了。

在往下执行的就是真正的修改某个环境变量了。不过我看不懂。。。。好像每个环境变量都会排序的，不再是像原来那种的最新修改的都是在最后一个，现在的是更人性化了，会排序了，但是就是代码更复杂了。

saveenv 是保存环境变量到 SD 卡中

```
689: static int do_env_save(cmd_tbl_t *cmdtp, int flag, int argc,
690:                char * const argv[])
691: {
692:     printf("Saving Environment to %s...\n", env_name_spec);
693:
694:     return saveenv() ? 1 : 0;
695: }
696:
```

保存在哪里是可以设置的，我这是保存在了 fat 格式的 sd 卡上了



在 common/env_fat.c 中调用 file_fat_write();完成了在 SD 卡写入 uboot.env 的文件中

common/env_common.c 文件中的这两个函数感觉是字符串和哈希表之间的桥梁啊，从内存往 sd 卡写的时候就先 export 导出来成字符串并且计算了 crc，等开机从 SD 卡往内存写的时候就是 import 导入，先计算 crc，如果没问题就直接换成哈希表形式。

env_export();

env_import();

读取环境变量

char *getenv(const char *name);

这个函数返回的是直接的内存中的地址，属于不可重入的函数

int getenv_f(const char *name, char *buf, unsigned len);

这个是把环境变量读取到一个 buf 中，是可重入函数，就是比较安全

可重入函数就是可以被并发执行的任务调用的函数。

十四 、UBOOT 的启动内核及传参

当 bootdelay 没有被打断时就是直接根据环境变量“bootcmd”中的值去启动内核了，而“bootcmd”的值其实就是几个启动内核的命令。看一下，

```
U-Boot# print bootcmd
bootcmd=run findfdt; run mmcboot;run usbboot;run nandboot;
U-Boot#
```

先寻找了 fdt 设备树，然后从 mmc 启动，如果失败就 usb 启动，再就是 nand 启动。这是一个寻找 kernel 的顺序。

do_run 在 common/cli.c 文件中定义。先读取了环境变量然后去执行了。

```
112: int do_run(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
113: {
114:     int i;
115:
116:     if (argc < 2)
117:         return CMD_RET_USAGE;
118:
119:     for (i = 1; i < argc; ++i) {
120:         char *arg;
121:
122:         arg = getenv(argv[i]);
123:         if (arg == NULL) {
124:             printf("## Error: \"%s\" not defined\n", argv[i]);
125:             return 1;
126:         }
127:
128:         if (run_command(arg, flag) != 0)
129:             return 1;
130:     }
131:     return 0;
132: } « end do_run »
```

run 这个命令是用来方便自己使用的，你可以添加一个环境变量这个环境变量的值是想要执行的一堆的操作，然后 run 这个环境变量，就可以执行上面所定义的一系列命令了。比如 mmcboot 就是一些列的命令，zxc 那个是我临时测试的，你看就是打印出来了 bootdelay。这回算是知道自定义的环境变量用来干啥了。

```
U-Boot#
mmcboot=mmc dev ${mmcdev}; setenv devnum ${mmcdev}; setenv devtype mmc; if mmc rescan; then echo SD/MMC
 ${devnum};if run loadbootenv; then echo Loaded environment from ${bootenv};run importbootenv;fi;if test
hen echo Running uenvcmd ...;run uenvcmd;fi;if run loadimage; then run loadfdt; echo Booting from mmc${m
 mmcargs; bootz ${loadaddr} - ${fdtaddr}; fi;fi;
U-Boot# set zxc print bootdelay
U-Boot# run zxc
bootdelay=3
U-Boot#
```

感觉跟写了个脚本一样。

这一系列的判断查找 sd 卡然后还干了一些其他的事情打印一些 mmc 相关的信息然后就是 bootz 命令真正干事的来了。

bootz 命令就是来启动 zImage 格式的 linux 内核的。原来看过的是用 bootm 来启动 zImage 镜像的。

https://www.cnblogs.com/chenfulin5/p/6937334.html

```
U-Boot# run mmcboot
switch to partitions #0, OK
mmc0 is current device
SD/MMC found on device 0
reading uEnv.txt
** Unable to read file uEnv.txt **
4574640 bytes read in 427 ms (10.2 MiB/s)
50712 bytes read in 23 ms (2.1 MiB/s)
Booting from mmc0 ...
Kernel image @ 0x82000000 [ 0x000000 - 0x45cdb0 ]
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Device Tree to 8fff0000, end 8ffff617 ... OK

Starting kernel ...
```

****

当时开了个会，心烦意乱了，看 bootz 看不下去了。 2018-01-23 周二

****

vmlinuz 和 zImage 和 uImage，*******这段是来自于朱友鹏老师的讲义

(1)uboot 经过编译直接生成的 elf 格式的可执行程序是 u-boot，这个程序类似于 windows 下的 exe 格式，在操作系统下是可以直接执行的。但是这种格式不能用来烧录下载。我们用来烧录下载的是 u-boot.bin，这个东西是由 u-boot 使用 arm-linux-objcopy 工具进行加工（主要目的是去掉一些无用的）得到的。这个 u-boot.bin 就叫镜像（image），镜像就是用来烧录到 sd 卡中执行的。

(2)linux 内核经过编译后也会生成一个 elf 格式的可执行程序，叫 vmlinux 或 vmlinuz，这个就是原始的未经任何处理加工的原版内核 elf 文件；嵌入式系统部署时烧录的一般不是这个 vmlinuz/vmlinux，而是要用 objcopy 工具去制作成烧录镜像格式（就是 u-boot.bin 这种，但是内核没有.bin 后缀），经过制作加工成烧录镜像的文件就叫 Image（制作把 78M 大的精简成了 7.5M，因此这个制作烧录镜像主要目的就是缩减大小，节省磁盘）。

(3)原则上 Image 就可以直接被烧录到 Flash 上进行启动执行(类似于 u-boot.bin)，但是实际上并不是这么简单。实际上 linux 的作者们觉得 Image 还是太大了所以对 Image 进行了压缩，并且在 image 压缩后的文件的前端附加了一部分解压缩代码。构成了一个压缩格式的镜像就叫 zImage。（因为当年 Image 大小刚好比一张软盘（软盘有 2 种，1.2M 的和 1.44MB 两种）大，为了节省 1 张软盘的钱于是乎设计了这种压缩 Image 成 zImage 的技术）。

(4)uboot 为了启动 linux 内核，还发明了一种内核格式叫 uImage。uImage 是由 zImage 加工得到的，uboot 中有一个工具，可以将 zImage 加工生成 uImage。注

意：uImage 不关 linux 内核的事，linux 内核只管生成 zImage 即可，然后 uboot 中的 mkimage 工具再去由 zImage 加工生成 uImage 来给 uboot 启动。这个加工过程其实就是在 zImage 前面加上 64 字节的 uImage 的头信息即可。

(4)原则上 uboot 启动时应该给他 uImage 格式的内核镜像，但是实际上 uboot 中也可以支持 zImage 定义了 LINUX_ZIMAGE_MAGIC 这个宏。所以大家可以看出：有些 uboot 是支持 zImage 启动的，有些则不支持。但是所有的 uboot 肯定都支持 uImage 启动。

****

主要来看一下 do_bootz 函数执行 mmcboot 的前面命令已经把 zImage 拷贝到内存指定位置了啊。

```
583: int do_bootz(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
584: {
585:     int ret;
586:
587:     /* Consume 'bootz' */
588:     argc--; argv++;
589:
590:     if (bootz_start(cmdtp, flag, argc, argv, &images))
591:         return 1;
592:
593:     /*
594:      * We are doing the BOOTM_STATE_LOADOS state ourselves, so must
595:      * disable interrupts ourselves
596:      */
597:     bootm_disable_interrupts();
598:
599:     images.os.os = IH_OS_LINUX;
600:     ret = do_bootm_states(cmdtp, flag, argc, argv,
601:                   BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
602:                   BOOTM_STATE_OS_GO,
603:                   &images, 1);
604:
605:     return ret;
606: } « end do_bootz »
```

bootz_start()；函数主要是 start 么初始化一些东西。

do_bootm_states();这个函数会根据传参来做一些不同状态时做的事情。如图下面这些就是不同的状态。

```
#define BOOTM_STATE_START   (0x00000001)
#define BOOTM_STATE_FINDOS  (0x00000002)
#define BOOTM_STATE_FINDOTHER   (0x00000004)
#define BOOTM_STATE_LOADOS  (0x00000008)
#define BOOTM_STATE_RAMDISK (0x00000010)
#define BOOTM_STATE_FDT     (0x00000020)
#define BOOTM_STATE_OS_CMDLINE  (0x00000040)
#define BOOTM_STATE_OS_BD_T (0x00000080)
#define BOOTM_STATE_OS_PREP (0x00000100)
#define BOOTM_STATE_OS_FAKE_GO  (0x00000200)    /* 'Almost' run the OS */
#define BOOTM_STATE_OS_GO   (0x00000400)
```

do_bootm_states();这个函数会多次调用，但传参不同，执行的动作也不同。

最后调用 common/boom.c 文件中的 bootm_start()函数。主要初始化了头部信息。

```
static int bootm_start(cmd_tbl_t *cmdtp, int flag, int argc,
               char * const argv[])
{
    memset((void *)&images, 0, sizeof(images));
    images.verify = getenv_yesno("verify");

    boot_start_lmb(&images);

    bootstage_mark_name(BOOTSTAGE_ID_BOOTM_START, "bootm_start");
    images.state = BOOTM_STATE_START;
```

然后回到 bootz_start()函数中因为 argc 不是 0 所以讲传入 argv[0]作为入口地址。也就是 0x82000000。

```
556:    /* Setup Linux kernel zImage entry point */
557:    if (!argc) {
558:        images->ep = load_addr;
559:        debug("*  kernel: default image load address = 0x%08lx\n",
560:                load_addr);
561:    } else {
562:        images->ep = simple_strtoul(argv[0], NULL, 16);
563:        debug("*  kernel: cmdline image address = 0x%08lx\n",
564:            images->ep);
565:    }
```

接下来执行 bootz_setup();函数，

```
567:     ret = bootz_setup(images->ep, &zi_start, &zi_end);
568:     if (ret != 0)
569:         return 1;
570:
```

位于 arch/arm/lib 目录下的 bootm.c 文件中

```
322: #ifdef CONFIG_CMD_BOOTZ
323:
324: struct zimage_header {
325:     uint32_t    code[9];
326:     uint32_t    zi_magic;
327:     uint32_t    zi_start;
328:     uint32_t    zi_end;
329: };
330:
331: #define LINUX_ARM_ZIMAGE_MAGIC  0x016f2818
332:
333: int bootz_setup(ulong image, ulong *start, ulong *end)
334: {
335:     struct zimage_header *zi;
336:
337:     zi = (struct zimage_header *)map_sysmem(image, 0);
338:     if (zi->zi_magic != LINUX_ARM_ZIMAGE_MAGIC) {
339:         puts("Bad Linux ARM zImage magic!\n");
340:         return 1;
341:     }
342:
343:     *start = zi->zi_start;
344:     *end = zi->zi_end;
345:
346:     printf("Kernel image @ %#08lx [ %#08lx - %#08lx ]\n", image, *start,
347:             *end);
348:
349:     return 0;
350: }
351:
352: #endif  /* CONFIG_CMD_BOOTZ */
```

那个宏 LINUX_ARM_ZIMAGE_MAGIC 0x016f2818 就是 zImage 的魔数身份证。

kernel 的信息也是这里打印出来的。

执行完 bootz_setup 后回到 bootz_start 中执行寻找 fdt 设备，同时相关的信息也是这里面打印出来的。

```
577:     if (bootm_find_ramdisk_fdt(flag, argc, argv))
578:         return 1;
579:
580:     return 0;
581: } « end bootz_start »
```

执行完 bootz_start 后回到 do_bootz 中先关闭了中断后就开始有执行了 do_bootm_states 函数。状态有_PREP、 _FAKE_GO、 _GO

```
599:     images.os.os = IH_OS_LINUX;
600:     ret = do_bootm_states(cmdtp, flag, argc, argv,
601:                   BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
602:                   BOOTM_STATE_OS_GO,
603:                   &images, 1);
604:
```

有上面知悉，images->os.os 是 IH_OS_LINUX

```
597:     boot_fn = bootm_os_get_boot_func(images->os.os);
598:     need_boot_fn = states & (BOOTM_STATE_OS_CMDLINE |
599:             BOOTM_STATE_OS_BD_T | BOOTM_STATE_OS_PREP |
600:             BOOTM_STATE_OS_FAKE_GO | BOOTM_STATE_OS_GO);
```

这个函数中主要是获取 os 的加载程序，就是加载 kernel 的函数。

```
463: boot_os_fn *bootm_os_get_boot_func(int os)
464: {
465: #ifdef CONFIG_NEEDS_MANUAL_RELOC
466:     static bool relocated;
467:
468:     if (!relocated) {
469:         int i;
470:
471:         /* relocate boot function table */
472:         for (i = 0; i < ARRAY_SIZE(boot_os); i++)
473:             if (boot_os[i] != NULL)
474:                 boot_os[i] += gd->reloc_off;
475:
476:         relocated = true;
477:     }
478: #endif
479:     return boot_os[os];
480: }
```

```
407: static boot_os_fn *boot_os[] = {
408:     [IH_OS_U_BOOT] = do_bootm_standalone,
409: #ifdef CONFIG_BOOTM_LINUX
410:     [IH_OS_LINUX] = do_bootm_linux,
411: #endif
```

咱们是 linux 就是执行的 do_bootm_linux 函数。arch/arm/lib 目录下的 bootm.c boot_fn 就是刚查找到的 do_bootm_linux 函数，下面这句也就是执行了这个函数。

```
615:     if (!ret && (states & BOOTM_STATE_OS_PREP))
616:         ret = boot_fn(BOOTM_STATE_OS_PREP, argc, argv, images);
```

进入 do_bootm_linux 函数中了。

```
301: int do_bootm_linux(int flag, int argc, char *argv[], bootm_headers_t *images)
302: {
303:     /* No need for those on ARM */
304:     if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
305:         return -1;
306:
307:     if (flag & BOOTM_STATE_OS_PREP) {
308:         boot_prep_linux(images);
309:         return 0;
310:     }
311:
312:     if (flag & (BOOTM_STATE_OS_GO | BOOTM_STATE_OS_FAKE_GO)) {
313:         boot_jump_linux(images, flag);
314:         return 0;
315:     }
316:
317:     boot_prep_linux(images);
318:     boot_jump_linux(images, flag);
319:     return 0;
320: } « end do_bootm_linux »
```

因为有 BOOTM_STATE_OS_PREP 所致执行了 boot_prep_linux 函数。这个函数其实就是用来准备给 kernel 传参数的。分为 atags 传参和 fdt 传参。咱们用的是 fdt

****

tag 方式传参**********朱老师讲义

(1)struct tag，tag 是一个数据结构，在 uboot 和 linux kernel 中都有定义 tag 数据机构，而且定义是一样的。

(2)tag_header 和 tag_xxx。tag_header 中有这个 tag 的 size 和类型编码，kernel 拿到一个 tag 后先分析 tag_header 得到 tag 的类型和大小，然后将 tag 中剩余部分当作一个 tag_xxx 来处理。

(3)tag_start 与 tag_end。kernel 接收到的传参是若干个 tag 构成的，这些 tag 由 tag_start 起始，到 tag_end 结束。

(4)tag 传参的方式是由 linux kernel 发明的，kernel 定义了这种向我传参的方式，uboot 只是实现了这种传参方式从而可以支持给 kernel 传参。

****

bootargs 也是我们要传给 kernel 的参数，这个是在 mmcboot 的那一堆命令中定义的，有的 uboot 是在环境变量中直接就有这个值。

```
210: static void boot_prep_linux(bootm_headers_t *images)
211: {
212:     char *commandline = getenv("bootargs");
213:
214:     if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len) {
215: #ifdef CONFIG_OF_LIBFDT
216:         debug("using: FDT\n");
217:         if (image_setup_linux(images)) {
218:             printf("FDT creation failed! hanging...");
219:             hang();
220:         }
221: #endif
222:     } else if (BOOTM_ENABLE_TAGS) {
223:         debug("using: ATAGS\n");
224:         setup_start_tag(gd->bd);
225:         if (BOOTM_ENABLE_SERIAL_TAG)
226:             setup_serial_tag(&params);
227:         if (BOOTM_ENABLE_CMDLINE_TAG)
228:             setup_commandline_tag(gd->bd, commandline);
```

用的是 fdt 传参，这个设备树我是换没搞明白是个什么东东

回到 do_bootm_states 中继续执行 boot_selected_os，看注释我们不希望你在回来了啊。

```
636:     /* Now run the OS! We hope this doesn't return */
637:     if (!ret && (states & BOOTM_STATE_OS_GO))
638:         ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_GO,
639:                 images, boot_fn);
640:
```

```
446: int boot_selected_os(int argc, char * const argv[], int state,
447:             bootm_headers_t *images, boot_os_fn *boot_fn)
448: {
449:     arch_preboot_os();
450:     boot_fn(state, argc, argv, images);
451:
```

再一次执行 boot_fn 也就是 do_bootm_linux 函数这回就不回来了。因为有 BOOTM_STATE_GO 所致直接执行 boot_jump_linux 函数了

```
311:
312:     if (flag & (BOOTM_STATE_OS_GO | BOOTM_STATE_OS_FAKE_GO)) {
313:         boot_jump_linux(images, flag);
314:         return 0;
315:     }
316:
```

这是将入口地址直接转为了一个函数指针，

```
266:     unsigned long machid = gd->bd->bi_arch_number;
267:     char *s;
268:     void (*kernel_entry)(int zero, int arch, uint params);
269:     unsigned long r2;
270:     int fake = (flag & BOOTM_STATE_OS_FAKE_GO);
271:
272:     kernel_entry = (void (*)(int, int, uint))images->ep;
273:
```

这个函数打印出了 Starting kernel ...的最后一句话，并且清除了一些 uboot 中的设置。

```
static void announce_and_cleanup(int fake)
{
    printf("\nStarting kernel ...%s\n\n", fake ?
        "(fake run for tracing)" : "");
```

Starting kernel ... 这个是 uboot 中最后一句打印出来的东西。

这句如果能出现，说明 uboot 整个是成功的，也成功的加载了内核镜像，也校验通过了，也找到入口地址了，也试图去执行了。如果这句后串口就没输出了，说明内核并没有被成功执行。原因一般是：传参（80%）、内核在 DDR 中的加载地址・・・・・・・

这个就是直接调用函数指针跳转了啊，传的参数是一个 0，machid 和参数的存放的开始地址

```
    if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
        r2 = (unsigned long)images->ft_addr;
    else
        r2 = gd->bd->bi_boot_params;

    if (!fake)
        kernel_entry(0, machid, r2);
```

移植时注意事项

(1)uboot 移植时一般只需要配置相应的宏即可

(2)kernel 启动不成功，注意传参是否成功。传参不成功首先看 uboot 中 bootargs 设置是否正确，其次看 uboot 是否开启了相应宏以支持传参。

2018-01-24 周三

十五 、UBOOT 的硬件驱动

**********朱老师课件

1、uboot 本身是裸机程序

(1)裸机本来是没有驱动的概念的（狭义的驱动的概念就是操作系统中用来具体操控硬件的那部分代码叫驱动）

(2)裸机程序中是直接操控硬件的，操作系统中必须通过驱动来操控硬件。这两个有什么区别？本质区别就是分层。

2、uboot 的虚拟地址对硬件操作的影响

(1)操作系统（指的是 linux）下 MMU 肯定是开启的，也就是说 linux 驱动中肯定都使用的是虚拟地址。而纯裸机程序中根本不会开 MMU，全部使用的是物理地址。这是裸机下和驱动中操控硬件的一个重要区别。

(2)uboot 早期也是纯物理地址工作的，但是现在的 uboot 开启了 MMU 做了虚拟地址映射，这个东西驱动也必须考虑。

3、uboot 借用（移植）了 linux 驱动

(1)linux 驱动本身做了模块化设计。linux 驱动本身和 linux 内核不是强耦合的，这是 linux 驱动可以被 uboot 借用（移植）的关键。

(2)uboot 移植了 linux 驱动源代码。uboot 是从源代码级别去移植 linux 驱动的，这就是 linux 系统的开源性。

(3)uboot 中的硬件驱动比 linux 简单。linux 驱动本身有更复杂的框架，需要实现更多的附带功能，而 uboot 本质上只是个裸机程序，uboot 移植 linux 驱动时只是借用了 linux 驱动的一部分而已。

****

同样也看一下我当前版本的 mmc 驱动。之前看就是看见带_init 的就觉得初始化了。单片机搞习惯了，就一直觉得那一堆来回的操作都是在玩蛇啊，只有真真看到了往寄存器所在的内存写东西了才觉得：哦，你确实初始化了。

MMC 初始化，想象一下写单片机程序是初始化 SD 卡时都干了啥？

1）用啥外设就先开啥外设的时钟，然后配置内部寄存器

2）用到了 gpio 所以 gpio 也得配置

3）给 SD 卡发一些命令让其初始化

4）如果有文件系统还得完成里面的底层 io 读写接口，然后挂载上去

来看一下源码吧，从 UBOOT 的 board_init_r 函数的一堆函数队列中有 initr_mmc 的初始化。

```
404: #ifdef CONFIG_GENERIC_MMC
405: int initr_mmc(void)
406: {
407:     puts("MMC:   ");
408:     mmc_initialize(gd->bd);
409:     return 0;
410: }
411: #endif
```

在 drivers/mmc/mmc.c 文件中有，mmc_devices 是一个链表头，初始化是指向自己的，uboot 中的同一类设备都是用一个链表连起来的。

```
1438: int mmc_initialize(bd_t *bis)
1439: {
1440:     INIT_LIST_HEAD (&mmc_devices);
1441:     cur_dev_num = 0;
1442:
1443:     if (board_mmc_init(bis) < 0)
1444:         cpu_mmc_init(bis);
1445:
1446: #ifndef CONFIG_SPL_BUILD
1447:     print_mmc_devices(',');
1448: #endif
1449:
1450:     do_preinit();
1451:     return 0;
1452: }
```

board_mmc_init();这个函数也在 mmc.c 文件中，（一定要仔细找啊，不要上了 Source Insight 的坑，它自动帮你定位的是一个错的），看在 mmc.c 中给人起了一个别名叫__def_mmc_init 还有一个 weak 属性，weak 就是你定义这个函数我就啥也是不是了，就以你定义的为准，你没定义的话，那就不好意思了，就只能是我了。当然在这里，我是这么理解的。

```
1383: static int __def_mmc_init(bd_t *bis)
1384: {
1385:     return -1;
1386: }
1387:
1388: int cpu_mmc_init(bd_t *bis) __attribute__((weak, alias("__def_mmc_init")));
1389: int board_mmc_init(bd_t *bis) __attribute__((weak, alias("__def_mmc_init")));
```

直接返回-1 了。调用 cpu_mmc_init 这个函数自定是定义的了。在 arch/arm/cpu/armv7/am33xx 的 board.c 文件中，（还一位驱动都在 drivers 文件夹中呢，这就是分离和分层的思想了吧）

cpu_mmc_init()　在 arch/arm/cpu/armv7/am33xx 的 board.c 文件中

调用 omap_mmc_init()函数

omap_mmc_init()　在 drivers/mmc/omap_hsmmc.c 文件中，这个函数主要是获取了相关的寄存器、时钟配置和 gpio 配置信息全部存在一个结构体中。包括一些 mmc 的操作函数指向

mmc_creat()　在 drivers/mmc/mmc.c 文件中

将这个包含了各种信息的结构体添加到了链表中

同样包括一些 mmc 的 block 的操作函数的指向

```
687:     cfg->name = "OMAP SD/MMC";
688:     cfg->ops = &omap_hsmmc_ops;
```

在 driver/mmc/omap_hsmmc.c 文件中定义

```
626: static const struct mmc_ops omap_hsmmc_ops = {
627:     .send_cmd    = omap_hsmmc_send_cmd,
628:     .set_ios     = omap_hsmmc_set_ios,
629:     .init        = omap_hsmmc_init_setup,
630: #ifdef OMAP_HSMMC_USE_GPIO
631:     .getcd       = omap_hsmmc_getcd,
632:     .getwp       = omap_hsmmc_getwp,
633: #endif
634: };
```

在 mmc_creat()函数中，也是初始化了一些操作 mmc 的 block 的函数，这些函数也同样是穿给 fs 的。

```
1245:     mmc->block_dev.if_type = IF_TYPE_MMC;
1246:     mmc->block_dev.dev = cur_dev_num++;
1247:     mmc->block_dev.removable = 1;
1248:     mmc->block_dev.block_read = mmc_bread;
1249:     mmc->block_dev.block_write = mmc_bwrite;
1250:     mmc->block_dev.block_erase = mmc_berase;
1251:
```

一个结构体包括了这个设备所有你想知道的东西和使用的东西

譬如 MMC 驱动的结构体就是 struct mmc 这些结构体中包含一些变量和一些函数指针，变量用来记录驱动相关的一些属性，函数指针用来记录驱动相关的操作方法。这些变量和函数指针加起来就构成了驱动。

驱动就被抽象为这个结构体。

一个驱动工作时主要就分几部分：驱动构建（构建一个 struct mmc 然后填充它）、驱动运行时（调用这些函数指针指针的函数和变量）

print_mmc_devices(',');在 drivers/mmc/mmc.c 文件中，这个函数主要是导引信息，比如启动时打印的名字就是在这打印出来的

```
1393: void print_mmc_devices(char separator)
1394: {
1395:     struct mmc *m;
1396:     struct list_head *entry;
1397:
1398:     list_for_each(entry, &mmc_devices) {
1399:         m = list_entry(entry, struct mmc, link);
1400:
1401:         printf("%s: %d", m->cfg->name, m->block_dev.dev);
1402:
1403:         if (entry->next != &mmc_devices)
1404:             printf("%c ", separator);
1405:     }
1406:
1407:     printf("\n");
1408: }
```

遍历 mmc_decices 这个链表，然后打印出来的设备的名字。

struct mmc *mmc;这个结构体包含了跟 mmc 相关的所有信息和操作函数，所以说一个设备就是一个链表中的结构体。

do_preinit(); 在 drivers/mmc/mmc.c 文件中遍历链表开始初始化了。

```
1424: static void do_preinit(void)
1425: {
1426:     struct mmc *m;
1427:     struct list_head *entry;
1428:
1429:     list_for_each(entry, &mmc_devices) {
1430:         m = list_entry(entry, struct mmc, link);
1431:
1432:         if (m->preinit)
1433:             mmc_start_init(m);
1434:     }
1435: }
```

mmc_start_init 函数 在 drivers/mmc/mmc.c 这里面主要就是发送 mmc/SD 卡相关的命令来初始化 SD 卡或者 MMC 卡了。比如一些命令码

mmc_go_idle(mmc);

mmc_send_cmd(mmc, &cmd, NULL);

mmc_send_if_cond(mmc);

mmc_send_cmd(mmc, &cmd, NULL);

mmc_send_op_cond(mmc);

mmc_send_cmd(mmc, &cmd, NULL);

.....还有一些类似的

mmc_send_cmd(mmc, &cmd, NULL);这个函数最后调用的就是 mmc 结构体里面的那个函数指针。

```
104: #else
105:     ret = mmc->cfg->ops->send_cmd(mmc, cmd, data);
106: #endif
```

****

分离思想

(1)分离思想就是说在驱动中将操作方法和数据分开。

(2)操作方法就是函数，数据就是变量。所谓操作方法和数据分离的意思就是：在不同的地方来存储和管理驱动的操作方法和变量，这样的优势就是驱动便于移植。

分层思想

(1)分层思想是指一个整个的驱动分为好多个层次。简单理解就是驱动分为很多个源文件，放在很多个文件夹中。本次的在 drivers/mmc 目录和 arch/arm/cpu/armv7/am33xx 目录。

(2)以 mmc 驱动为例来分析各个文件的作用：

drivers/mmc/mmc.c：这种毫无特色的文件命名法，肯定是说这个文件是通用的，本文件的主要内容是和 MMC 卡操作有关的方法，MMC 卡设置空闲状态的、卡读写数据等。但是本文件中并没有具体的硬件操作函数，操作最终指向的是 struct mmc 结构体中的函数指针，这些函数指针是在驱动构建的时候和真正硬件操作的函数挂接的（真正的硬件操作的函数在别的文件中）。

drivers/mmc/omap_hsmmc.c：这个文件就完成了和当前 SoC 相关的一些列的设置和操作函数的具体实现。比如 omap_send_command 等这些函数就是具体操作硬件的函数，也就是 mmc.c 中需要的那些硬件操作函数。

arch/arm/cpu/armv7/am33xx/board.c：这个文件中有个 cpu_mmc_init(bd_t *bis)函数。这个函数其实啥也没干，而是直接调用了 drivers/mmc/omap_hsmmc.c 中的 omap_mmc_init 函数。就是兜了一圈了。完全没必要啊。

```
55: int cpu_mmc_init(bd_t *bis)
56: {
57:     int ret;
58:
59:     ret = omap_mmc_init(0, 0, 0, -1, -1);
60:     if (ret)
61:         return ret;
```

****

如果我们要把这一套 mmc 驱动移植到别的 SoC 上 mmc.c 就不用动，omap_hsmmc.c 动就可以了；

如果 SoC 没变但是 SD 卡升级了，这时候只需要更换 mmc.c，不需要更换 omao_hsmmc.即可。

****

十六 、总结

之前一直玩个裸机，后来才意识到 arm+linux 这种的威力，于是乎马上开整，正好项目上也涉及到了 arm+linux，也算是学有所用。

通过这一周多的学习，算是对 uboot 简单走了一个过场，不知道为什么心里不是会很高兴，感觉越看越发现有太多太多东西还不会，感觉压力很大。真的是刷新了认知，真是体验到了朱老师说过的，你看这种代码的时候你就发现，诶，这特么 C 语言还能这样玩，这跟我学的不一样啊....

安装 ubuntu 系统时装了个 64 位，结果交叉编译工具链装完后真个不能使用，后来一通查，才知道是 32 位库的问题，虽然解决了，但为了后续再出现这种问题，我还是装回 32 位版本的了，一开始也只是照着开发板的用户手册在简单配置编译烧写试了一下，后来才开始研究源码，最起码知道它的执行流程啊。再后来你得会用 uboot 啊，用就用吧，还老想知道这是咋实现的，就继续看源码。看的过程也发想由于这个工程挺大的，你可能当时知道了，等再去看的时候就又要在翻一通，所以我才想着简单的写一下这个流程。

通过写出来，确实能加深自己的理解啊，所以后面再做移植的时候我也打算在一边学一边记下来。

路漫漫其修远兮.....

壮士一去兮不复还.....

2018-01-25 周四