Alan0521

博客访问： 712128
博文数量： 483
博客积分： 13297
博客等级： 上将
技术积分： 2895
用 户 组： 普通用户
注册时间： 2009-10-12 16:25

加关注 短消息
论坛 加好友

文章分类

全部博文（483）
Linux_drivers（9）
History（0）
News（0）
Miscellaneous（1）
Exam（19）
ARM（39）
C Sharp（36）
FPGA（11）
Windows（6）
Tools（104）
Hardware（25）
Life（6）
C（54）
Linux（173）
未分配的博文（0）

文章存档

2012年（9）
2011年（408）
2010年（66）

我的朋友



tekkaman xyl121 淘众福情

详细分析make uboot 最后的编译链接的具体执行过程

分类： LINUX 2011-08-15 11:28:25

此为转帖，向原作者表示感谢~~~

正常编译uboot的过程是，在make XXXX_config配置你的成你的板子之后，直接去make，就可以去编译出最后你需要的u-boot.bin了。此处，就是分析，在make之后，最后boot是如何生成的，去分析这个过程。
事先声明，由于知识有限，难免下面解释有误，希望懂行的不吝赐教。有关技术方面的讨论，欢迎mailto:
green-waste（At）163.com

好的，开始了。uboot中，make之后，最后的编译输出为：

```
UNDEF_SYM=`arm-linux-objdump -x board/ams/as3536/libas3536.a lib_generic/libgeneric.a
lib_generic/lzma/liblzma.a cpu/arm926ejs/libarm926ejs.a cpu/arm926ejs/ams/libams.a
lib_arm/libarm.a fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a
fs/jffs2/libjffs2.a fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a fs/yaffs2/libyaffs2.a
net/libnet.a disk/libdisk.a drivers/bios_emulator/libatibiosemu.a
drivers/block/libblock.a drivers/dma/libdma.a drivers/fpga/libfpga.a
drivers/gpio/libgpio.a drivers/hwmon/libhwmon.a drivers/i2c/libi2c.a
drivers/input/libinput.a drivers/misc/libmisc.a drivers/mmc/libmmc.a
drivers/mtd/libmtd.a drivers/mtd/nand/libnand.a
drivers/mtd/nand_legacy/libnand_legacy.a drivers/mtd/onenand/libonenand.a
drivers/mtd/ubi/libubi.a drivers/mtd/spi/libspi_flash.a drivers/net/libnet.a
drivers/net/phy/libphy.a drivers/net/sk98lin/libsk98lin.a drivers/pci/libpci.a
drivers/pcmcia/libpcmcia.a drivers/spi/libspi.a drivers/rtc/librtc.a
drivers/serial/libserial.a drivers/usb/libusb.a drivers/video/libvideo.a
common/libcommon.a libfdt/libfdt.a api/libapi.a post/libpost.a | \
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome && arm-linux-ld -
Bstatic -T /home/crifan/develop/uboot/uboot_as3536/u-boot-
2009.03_toHome/board/ams/as3536/u-boot.lds  -Ttext 0x00000000 $UNDEF_SYM
cpu/arm926ejs/start.o \
--start-group lib_generic/libgeneric.a lib_generic/lzma/liblzma.a
cpu/arm926ejs/libarm926ejs.a cpu/arm926ejs/ams/libams.a lib_arm/libarm.a
fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a
fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a fs/yaffs2/libyaffs2.a net/libnet.a
disk/libdisk.a drivers/bios_emulator/libatibiosemu.a drivers/block/libblock.a
drivers/dma/libdma.a drivers/fpga/libfpga.a drivers/gpio/libgpio.a
drivers/hwmon/libhwmon.a drivers/i2c/libi2c.a drivers/input/libinput.a
drivers/misc/libmisc.a drivers/mmc/libmmc.a drivers/mtd/libmtd.a
drivers/mtd/nand/libnand.a drivers/mtd/nand_legacy/libnand_legacy.a
drivers/mtd/onenand/libonenand.a drivers/mtd/ubi/libubi.a
drivers/mtd/spi/libspi_flash.a drivers/net/libnet.a drivers/net/phy/libphy.a
```

```
drivers/net/sk98lin/libsk98lin.a drivers/pci/libpci.a drivers/pcmcia/libpcmcia.a
drivers/spi/libspi.a drivers/rtc/librtc.a drivers/serial/libserial.a
drivers/usb/libusb.a drivers/video/libvideo.a common/libcommon.a libfdt/libfdt.a
api/libapi.a post/libpost.a board/ams/as3536/libas3536.a --end-group -L
/home/crifan/develop/buildroot/buildroot-2009.11/output/staging/usr/bin/../lib/gcc/arm-
linux-uclibcgnueabi/4.3.4 -lgcc \
-Map u-boot.map -o u-boot
arm-linux-objcopy -O srec u-boot u-boot.srec
arm-linux-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

[详细分析]
下面进行尽可能地分析具体含义：
(1)UNDEF_SYM变量赋值:

```
UNDEF_SYM=`arm-linux-objdump -x board/ams/as3536/libas3536.a ..... post/libpost.a | \
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`
```

其中UNDEF_SYM=XXX表示,将后面的XXX赋值给变量UNDEF_SYM.
而这里的是UNDEF_SYM=`XXXX`,下面要做两点说明:
A ,关于反引号`的作用
字符"`",叫做反引号,就是那个左Tab键上面,数字1左边的那个键.
这个反引号表示在shell脚本中,用于包含要执行的命令,否则不加反引号,后面的内容就视为普通字符串了.
因此,上面的意思就是,将下面命令:

```
arm-linux-objdump -x board/ams/as3536/libas3536.a ..... post/libpost.a | \
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq
```

执行的结果,送给UNDEF_SYM.
B.关于 空格加上反斜杠
即" \",意思是,一行内容太多了,分成多行书写,此时,就要用到上面一行的后面,加上空格再加一个反斜杠\,然后接下来写后面的内容,这样,对于系统处理,就可以把你的输入看成是一行的输入,
否则,不加空格和反斜杠,就视为两行,
像上面的内容,就成了

```
arm-linux-objdump -x board/ams/as3536/libas3536.a ..... post/libpost.a |
```

和

```
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq
```

签名objdump的内容,通过管道输出给当前终端了, 没有传递给后面的sed处理,就违背了我们的本意,打算去用sed处理的了.
因此这里的全部意思就是,
将反引号中间包含的命令执行的结果,赋值给变量UNDEF_SYM(以留作后面使用).
接下来看后面命令的具体含义:

(2)用objdump -x导出库文件中所有的头信息:
先来看看objdump -x参数的含义:

```
"  -x, --all-headers        Display the contents of all headers"
```

即显示(.a库文件)所包含全部的头(信息).因此上面的就是把所有的.a库文件中的头信息都导出来给那个变量.
而具体的headers是啥样子的,我们可以以第一个库libas3536.a为例,来看看结果:

```
arm-linux-objdump -x board/ams/as3536/libas3536.a
```

由于结果太长,此处只节选部分显示:

```
In archive board/ams/as3536/libas3536.a:

as3536.o:     file format elf32-littlearm
rw-r--r-- 1000/1000   5872 Feb 26 23:27 2010 as3536.o
architecture: arm, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
private flags = 600: [APCS-32] [VFP float format] [software FP]

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
0 .text         000001c8  00000000  00000000  00000034  2**2
```

```
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data         00000000  00000000  00000000  000001fc  2**0
CONTENTS, ALLOC, LOAD, DATA
2 .bss          00000000  00000000  00000000  000001fc  2**0
ALLOC
3 .debug_abbrev 00000155  00000000  00000000  000001fc  2**0
CONTENTS, READONLY, DEBUGGING
4 .debug_info   000002b7  00000000  00000000  00000351  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
5 .debug_line   00000127  00000000  00000000  00000608  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
6 .rodata.str1.1 00000011  00000000  00000000  0000072f  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rodata       0000000a  00000000  00000000  00000740  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
.............
SYMBOL TABLE:
00000000 l    df *ABS*    00000000 as3536.c
00000000 l    d  .text    00000000 .text
00000000 l    d  .data    00000000 .data
00000000 l    d  .bss     00000000 .bss
00000000 l    d  .debug_abbrev    00000000 .debug_abbrev
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .rodata.str1.1    00000000 .rodata.str1.1
00000000 l     O .rodata    0000000a __FUNCTION__.2915
00000000 l    d  .rodata    00000000 .rodata
..............
00000000         *UND*    00000000 i2cIsMasterBusy
00000000         *UND*    00000000 gpioInitialise
00000000         *UND*    00000000 uartInitialize
00000000         *UND*    00000000 serial_init
00000000         *UND*    00000000 mpmcInitialize
00000000         *UND*    00000000 mpmcDynamicConfig
00000000         *UND*    00000000 icache_enable
00000000         *UND*    00000000 dcache_enable
..............

as353x_nand.o:     file format elf32-littlearm
rw-r--r-- 1000/1000  24080 Feb 26 23:27 2010 as353x_nand.o
architecture: arm, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
private flags = 600: [APCS-32] [VFP float format] [software FP]

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
0 .text         00000df8  00000000  00000000  00000034  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data         00000000  00000000  00000000  00000e2c  2**0
CONTENTS, ALLOC, LOAD, DATA
2 .bss          00000000  00000000  00000000  00000e2c  2**0
ALLOC
3 .debug_abbrev 00000286  00000000  00000000  00000e2c  2**0
CONTENTS, READONLY, DEBUGGING
..............
6 .rodata.str1.1 00000093  00000000  00000000  00002aab  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rodata       00000185  00000000  00000000  00002b3e  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
.....................
SYMBOL TABLE:
00000000 l    df *ABS*    00000000 as353x_nand.c
00000000 l    d  .text    00000000 .text
00000000 l    d  .data    00000000 .data
.........................
00000ce4 l     F .text    00000114 as353x_nand_read_buf_swbch4
00000bd0 l     F .text    00000114 as353x_nand_write_buf_swbch4
00000a28 l     F .text    000001a8 as353x_enable_hwecc_swbch4
00000a0c l     F .text    0000001c as353x_calculate_ecc
00000944 l     F .text    000000c8 as353x_correct_data
00000460 l     F .text    00000034 as353x_nand_ready
.................
00000043 l     O .rodata    00000017 __FUNCTION__.3286
0000005a l     O .rodata    00000013 __FUNCTION__.3214
.......................
00000000         *UND*    00000000 bchDecode
00000000         *UND*    00000000 bchEncodeT4
00000000         *UND*    00000000 DebugClrRegBits8
00000000         *UND*    00000000 DebugRReg32
.............
```

可以看出来,里面包含了N个.o目标文件,而目标文件,就是我们之前用c文件编译出来的.
而上面的类似于:

```
00000a0c l     F .text    0000001c as353x_calculate_ecc
00000944 l     F .text    000000c8 as353x_correct_data
00000460 l     F .text    00000034 as353x_nand_ready
```

的.text类型的,都是该.o目标文件里面包含的,已经实现了的,某个函数.
而类似于:

```
00000000         *UND*    00000000 udelay
00000000         *UND*    00000000 bchDecode
00000000         *UND*    00000000 bchEncodeT4
```

都是*UND*类型的,即 未定义 类型,即,此文件里面调用了这个函数,而这个函数是在别的地方(其他c文件所对应的.o目前文件里面)实现的.
像这样的信息:

```
as353x_nand.o:     file format elf32-littlearm
rw-r--r-- 1000/1000  24080 Feb 26 23:27 2010 as353x_nand.o
architecture: arm, flags 0x00000011:
HAS_RELOC, HAS_SYMS
```

也大概可以看出来大概含义;
此代码是编译出来给arm的CPU执行的,是little endian,小端,elf32格式.HAS_RELOC表示可重定位,即需要最后的链接之后,才最终变成可执行文件.

好了,看懂一个,其他的.a输出的信息也是类似的:每个.a里面包含了多个.o, 每个.o对应一个.c或.s汇编文件.
对应的信息,就是那些函数,变量等等.

(3) 用sed处理字符串:
看懂了arm-linux-objdump之后,后面的字符"|",表示管道,即前面的内容,字节通过通过管道,送到后面的sed命令去处理.
因此,再来看后面的sed具体啥含义.

```
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
```

网上随便搜索一下"sed 语法",即可找到相关的资料,根据资料,一点点来对照,看看啥意思.
先看参数 -n,-e:
sed --help就可以找到:

```
-n, --quiet, --silent
取消自动打印模式空间
-e 脚本, --expression=脚本
添加“脚本”到程序的运行列表
```

http://www.xxlinux.com/linux/article/development/soft/20070508/8349.html
中解释了:
sed 默认会打印匹配的每一行，但是可以使用-n选项将其关闭。在这种情况下，只有用P命令显式说明的行才出现在输出中。如
sed -n ‘/模式/p’
sed -n ‘/模式/!p’

所以,-n的意思就是,默认后面如果有匹配的内容,比如要替换字符子类的,就会打印这些相关信息,此处加了-n,就no print了.
-e参数：

-e 的含义，说实话，之前一直没有完全看懂。现在经过一番探究，终于算是搞懂了。
-e的含义，简单的来说，就是把-e后面跟的内容，当做要执行的脚本（命令）放入到命令列表中（以待后面分别执行这些命令去处理字符流/串）。
对应的英文解释是：
http://www.gnu.org/software/sed/manual/html_node/Invoking-sed.html#Invoking-sed
-e script
--expression=script
Add the commands in script to the set of commands to be run while processing the input.
更简单的英文的解释是：
http://unixhelp.ed.ac.uk/CGI/man-cgi?sed
-e script, --expression=script
add the script to the commands to be executed
即，将-e参数后面的内容，加到要执行的命令列表中。
单独看这些英文解释，实在很难理解真正的意思，至少对于我，一直没搞懂具体是啥意思。
对此，还发现，经过尝试，对于不加-e参数：
arm-linux-objdump -x board/samsung/crl2440/libcrl2440.o 。。。post/libpost.o | sed -n 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
和加了-e参数的：
arm-linux-objdump -x board/samsung/crl2440/libcrl2440.o 。。。post/libpost.o | sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
的输出，都是类似于下面这样的结果:
-u__u_boot_cmd_start
。。。
-u__u_boot_cmd_env
即，加-e与不加-e，好像也没啥区别。
对此，继续去网上找资料探究。
后来，发现这里：
http://sed.sourceforge.net/sedlline_zh-CN.html
有很多例子，其中有些例子，就是每一个sed中，加了多个-e参数的。
同时，这里：
http://www.panix.com/~elflord/unix/sed.html
给出的例子中，也有个是同一个sed中，包含了2个-e参数的：
>cat file
I have three dogs and two cats
>sed -e 's/dog/cat/g' -e 's/cat/elephant/g' file
I have three elephants and two elephants
>
而对此例子，根据该帖子，其解释为：
上面的例子中，sed，对于字符流：
I have three dogs and two cats
会先用第一个-e后面的's/dog/cat/g'，即所谓的脚步或者叫做命令，来处理。
具体处理结果就是，g代表全局范围，s/dog/cat代表把字符串dog替换为cat，所以处理后，元字符流就变成：
I have three cats and two cats
然后对于处理后的这个字符流，继续用后面的's/cat/elephant/g'来处理，即将其中的cat都替换为elephant，所以处理后的结果为：
I have three elephants and two elephants

然后，看完了上面这个例子，才更加清楚，-e的参数的含义：
-e XXX，XXX是对应的脚本，即要执行的命令。
把XXX，加入到命令列表中。
而这个命令列表，就是一个个命令，比如上面例子中的，'s/dog/cat/g'和's/cat/elephant/g'。
然后sed在处理字符流之前，会把-e后面的所有的命令都一个个地加入到总的命令的列表中去。
然后一个个地调用这些命令，调用完前一个命令，处理后的结果，作为下一个命令的输入。
就这样一点点处理字符，知道处理完毕，所有命令列表中的命令，都执行完毕为止。
至此，才真正理解最开始的那些英文的解释，即-e后面，是要执行的命令，会把这个命令，放到命令列表中。以待后面去一个个地顺序地执行。

而对于前面提到的，加了-e和不加-e，执行结果也一样的现象，对应的解释在这里：
http://www.gnu.org/software/sed/manual/html_node/Invoking-sed.html#Invoking-sed
If no -e, -f, --expression, or --file options are given on the command-line, then the first non-option argument on the command line is taken to be the script to be executed.
即，如果sed中，没有-e，-f--expression或者--file等参数，那么就将第一个非选项的参数（即不是那种-n等以-开头的参数），视为要执行的脚本，即要执行的命令。
所以，才有上面的
sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
与
sed -n 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
两者执行的结果是一样的，因为对于第二个没有-e的参数的话，那么sed自动会把后面的第一个非选项的参数，即's/.*\(__u_boot_cmd_.*\)/-u\1/p'，视为要处理的脚本，
这和上面带-e参数的：
sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
所表达的意思，将's/.*\(__u_boot_cmd_.*\)/-u\1/p视为要执行的脚本，加入到将要执行的命令列表中，就完全是同一个意思了。
所以两者对于输入的字符串，所执行的结果是一样的。
其中对于像：
sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'
这样，sed中只有一个-e的参数，对应的sed的命令列表中，也只有一个命令，此处即's/.*\(__u_boot_cmd_.*\)/-u\1/p'。
至此，才算比较清楚，sed中-e参数的真正含义。

后面的,两个单引号中间的是具体的动作:
s/.*\(__u_boot_cmd_.*\)/-u\1/p
中间以斜杠/为分隔,属于 s/regexp/replacement/flag 共四部分:
第一部分 :
s,表示substitue,字符串替换.
第二部分:
regexpt为.*\(__u_boot_cmd_.*\),
这部分的意思,这些帖子说的相对清楚些:
http://www.grymoire.com/Unix/Regular.html#uh-4
和
http://www.grymoire.com/Unix/Sed.html#uh-4
对照来看,其中前面的.*表示单个字符以及任意字符,
\(.....\)的中间那部分就是__u_boot_cmd_.*
表示以字符串__u_boot_cmd_开头,后面有(单引号.表示)任意单个字符或(星号*表示)任意字符的字符串.
第三部分:
-u\1,-u就是普通的字符而已,\1表示前面用\(.....\)存储的那个字串,即,__u_boot_cmd_.*
第四部分: p,表示打印
所以,总体放在一起的意思就是,
用sed处理,-n表示不打印,而后面-p又表示,找到匹配的字符串,处理之后再打印
-e 此参数含义，上面已经解释过，此处不再多说。
对于前面送来的字符串流数据,
对于符合 .*\(__u_boot_cmd_.*\),即以单个任意字符或者多个任意字符开头,后面符合__u_boot_cmd_,后面再跟上单个任意字符或者多个任意字符的 字符串,
用-u再加上__u_boot_cmd_.*,即__u_boot_cmd_跟上单个任意字符或者多个任意字符,来代替
举例来说具体是如何执行的:
对于前面由

arm-linux-objdump -x board/ams/as3536/libas3536.a ..... post/libpost.a
获得的字符串流,其中有这些:
....
00000630 g   O .data   00000058 tbOffsetGPIO
00000000 g   O .u_boot_cmd   00000018 __u_boot_cmd_asdebug
....
sed对于获得的每一行,去判断,是否有
单个或多个任意字符开头,后面是__u_boot_cmd_,然后后面还有单个或多个任意字符
如果有的话,那么就用
-u,再加上__u_boot_cmd_.*,即从__u_boot_cmd_开始,以及后面所有的字符.
对照上面内容,
对于
00000630 g   O .data   00000058 tbOffsetGPIO
中间没有__u_boot_cmd_这种字符串,因此匹配失败,不处理,不打印
对于
00000000 g   O .u_boot_cmd   00000018 __u_boot_cmd_asdebug
符合上面说的,XXX__u_boot_cmd_XXXX类似的字符串,
所以用-u,加上原先的__u_boot_cmd_XXXX,即__u_boot_cmd_asdebug,组合起来就是
-u__u_boot_cmd_asdebug来代替,并且打印出来,即输出(留后面的sort命令去处理)
所以,就这样地,对于签名获得的字符串流,每一行都去找一下,有没有类似于XXX__u_boot_cmd_XXXX的字符串,有的话就处理并打印,没有就继续处理下一行,直至处理完毕.

最后所有的处理的结果,就是这样的了:
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_bdinfo
-u__u_boot_cmd_bdinfo
-u__u_boot_cmd_go
-u__u_boot_cmd_reset
-u__u_boot_cmd_go
-u__u_boot_cmd_reset
-u__u_boot_cmd_bootm
-u__u_boot_cmd_bootm
-u__u_boot_cmd_flinfo
-u__u_boot_cmd_erase
-u__u_boot_cmd_protect
-u__u_boot_cmd_flinfo
-u__u_boot_cmd_erase
....

(4)sort和uniq:
好了,得到这些结果后,注意到,上面的这些字符串,即没有按照字母数序排列,也有重复没去掉的.
所以,后面用|sort|uniq,即通过管道送给sort去排序,然后将排序后的结果用uniq去去掉重复的,
有人会问为何不直接调用uniq去掉重复,再去sort,回答是,经过我实际测试,uniq好像只能去掉前后两行有重复的,所以,如果直接先调用uniq,那么像这样的结果
-u__u_boot_cmd_flinfo
-u__u_boot_cmd_erase
-u__u_boot_cmd_protect
-u__u_boot_cmd_flinfo
中的-u__u_boot_cmd_flinfo,就无法去掉了,所以要先sort排序,再uniq去掉重复的,这样处理之后的结果,就保证了既是没有任何重复的,又是排好序的.
最后得到这样的结果:
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_base
-u__u_boot_cmd_bdinfo
-u__u_boot_cmd_bootm
-u__u_boot_cmd_bootp
-u__u_boot_cmd_cmp
-u__u_boot_cmd_cp
-u__u_boot_cmd_crc32

```
-u__u_boot_cmd_end
-u__u_boot_cmd_erase
-u__u_boot_cmd_flinfo
-u__u_boot_cmd_go
-u__u_boot_cmd_help
-u__u_boot_cmd_loadb
-u__u_boot_cmd_loads
......
```

(5)分号用于单行输入中,分隔多个命令:
在之行完

```
UNDEF_SYM=`arm-linux-objdump -x board/ams/as3536/libas3536.a ...... api/libapi.a
post/libpost.a | \
sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`
```

之后,后面加了个分号";"用于在一次单行输入中,分割多个命令,
这里的意思就是,
先UNDEF_SYM=XXXX,然后加上;,结束了前面的变量赋值,后面又去执行其他的命令,这里的是

```
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome
```

(6)&&，表示逻辑与，前面为真，后面的命令才执行：
后面的是:

```
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome && arm-linux-ld XXXX
```

意思是先cd切换目录,如果切换目录成功，接着执行arm-linux-ld XXXX,
＆＆此处也是将两个命令连在一起，就相当于在命令行中输入:

```
cd /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome
arm-linux-ld XXXX
```

意思是,切换到那个目录,然后执行ld.
此处如果没有&&,那么实际执行的cd切换目录的动作和后面的ld命令,就没有任何关系.
因此就相当于在当前目录执行ld命令了,就不是我们所期望的意思了.

(7)ld的执行:

```
arm-linux-ld -Bstatic -T /home/crifan/develop/uboot/uboot_as3536/u-boot-
2009.03_toHome/board/ams/as3536/u-boot.lds  -Ttext 0x00000000 $UNDEF_SYM
cpu/arm926ejs/start.o \
--start-group lib_generic/libgeneric.a ...... board/ams/as3536/libas3536.a --end-group
-L /home/crifan/develop/buildroot/buildroot-
2009.11/output/staging/usr/bin/../lib/gcc/arm-linux-uclibcgnueabi/4.3.4 -lgcc \
-Map u-boot.map -o u-boot
```

接下来详细分析各个ld参数的具体含义:
参考这个解释:
http://www.computerhope.com/unix/uld.htm
具体分析如下:
(A)-Bstatic

```
------------------------------
-B dynamic | static
------------------------------
```

Options governing library inclusion. -B dynamic is valid in dynamic mode only. These options may be specified any number of times on the command line as toggles: if the -B static option is given, no shared objects will be accepted until -B dynamic is seen. See also the -l option.

```
--------------------------------------------------------------------------------
--------------------------
```

意思是后面的接下来要处理的库,都是static,静态方式链接进来.
相对于dynamic,动态方式来说,要更加占用生成(目标或可执行)文件大小,但是,不会出现类似:
"还要再依赖执行环境中,系统要包含对应的库,否则就无法执行"的问题.

(B)-T /home/crifan/develop/uboot/uboot_as3536/u-boot-2009.03_toHome/board/ams/as3536/u-boot.lds

```
--------------------------------------------------------------------------------
--------------------------
```

```
The following SunOS 4.x.y options are not supported: -align datum, -A name, -D hex, -p,
-T[text] hex, -T data hex. Much of the functionality of these options can be achieved
using the -M mapfile option.
-------------------------------------------------------------------------------------
-------------------------
```

按照解释来看,这个 -T 加上hex的参数,已经不支持了, 取而代之的是,都可以用-M mapfile来实现.
但是 此处的-T 后面跟的是lds,load脚本,应该不属于hex吧? 具体意思,还真是看不懂....
后来在这里:
http://www.cublog.cn/u1/42111/showart.php?id=386864
找到了解释:
“-T FILE或—script FILE读链接描述文件名，以确定符号等的定位地址”
这里，意思就是，使用u-boot.lds，以确定符号等的定位地址，即根据这个lds中的指示，把对应的不同段的数据，放在指定的地方。

(C)u-boot.lds的简单分析:
对于u-boot.lds中的内容，这里贴出来看看：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
. = 0x00000000;
. = ALIGN(4);
.text      :
{
cpu/arm926ejs/start.o     (.text)
*(.text)
}
.rodata : { *(.rodata) }
. = ALIGN(4);
.data : { *(.data) }
. = ALIGN(4);
.got : { *(.got) }

. = .;
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;

. = ALIGN(4);
__bss_start = .;
.bss (NOLOAD) : { *(.bss) }
_end = .;
}
```

其中，里面的ENTRY(_start)，表示程序入口位置，是_start,而_start去查找后，
正是cpu\arm926ejs\start.S汇编代码里面的，程序的入口位置：

```
.globl _start
_start:
b    reset
ldr    pc, _undefined_instruction
ldr    pc, _software_interrupt
ldr    pc, _prefetch_abort
ldr    pc, _data_abort
ldr    pc, _not_used
ldr    pc, _irq
ldr    pc, _fiq
。。。。。
```

其他更多内容，参考：
http://blog.csdn.net/ling1874/archive/2010/01/04/5128269.aspx

(D) -Ttext 0x00000000
即，根据上面的解释，就是，指定text段的起始地址是0x0。就比如，如果不是0，比如是某开发板的SDRAM的起始地址0x40000000，那么，lds中所指定的，后面的那些.text段就要从0x40000000加上lds里面
“. = 0x00000000;”所指定的0x0,即还是0x40000000，然后接着后面放的是lds里面指定的其他段，即.rodata,.data,.got等等段。
总之一句话，ld命令会根据你这些相关变量：
ld时候参数中的text指定的地址，lds文件中指定的起始地址，
等去把对应的段的内容，放到对应的地方，生成对应的目标(可执行)代码。

(E)$UNDEF_SYM
就是上面，我们获得的，内容是这样的：
-u__u_boot_cmd_asdebug
-u__u_boot_cmd_base
-u__u_boot_cmd_bdinfo
-u__u_boot_cmd_bootm
-u__u_boot_cmd_bootp
-u__u_boot_cmd_cmp
。。。
去看ld参数-u的含义：

-u symname

Enter symname as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library, since initially the symbol table is empty, and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that will define the symbol.

总的意思，就是，在ld的时候，不定义，这些符号，即不定义__u_boot_cmd_asdebug，__u_boot_cmd_base等等这些符 号，我的理解是，因为开始需要从某个库中载入其他的符号，而这个时候，还没有载入到那个包含此符号定义的库，所以，暂时先加入这个-u说明，先不定义这些 符号，等到所有的库都加载完了，再去找这些符号的定义，此时已经加载完所有的库了，也就能找到这些符号的定义了。
(F)接下来的，就是要ld的，那一堆的目标文件，库文件了：
cpu/arm926ejs/start.o和
--start-group lib_generic/libgeneric.a lib_generic/lzma/liblzma.a cpu/arm926ejs/libarm926ejs.a 。。。。。。 post/libpost.a board/ams/as3536/libas3536.a --end-group
关于--start-group和 --end-group的含义：
-（archives）或者—start-group archives -end-group 参数中间的目标文件会被ld反复搜索，对相互交叉引用的目标文件很有用
所以，我的理解是，如果ld载入了一个库，发现该库中，有UNDF，未被定义的变量，有了这个参数的指示后，就会在这一堆.a和.o文件里面反复搜索，直至找到为止，否则，如果在已经加载的库中，找不到，就会报错（是不是这样，不确定，只是这么猜测而已）。
(G)-L加入搜索路径：
-L /home/crifan/develop/buildroot/buildroot-2009.11/output/staging/usr/bin/../lib/gcc/arm-linux-uclibcgnueabi/4.3.4
即把这个目录，加入到搜索路径中，如果在系统搜索路径中，找不到前面那些库，就去这个路径中找，如果都找不到，那么肯定会报错的。
不过官方的解释：

-L path

Add path to the library search directories. ld searches for libraries first in any directories
specified by the -L options and then in the standard directories. This option is useful only if it precedes the -l options to which it applies on the command line. The

environment variable LD_LIBRARY_PATH may be used to supplement the library search path.

说到，是-L必须要加载-l之前，否则无效。这里，也正是这么做的。接下来，就是-l.
(H)-l加入通用（命名的）的库：
-lgcc，即加载libgcc.a(静态库)或者libgcc.so(动态库)。
-l参数的含义：

-l x

Search a library libx.so or libx.a, the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the -B static option is in effect, ld searches each directory specified in the library search path for a libx.so or libx.a file. The directory search stops at the first directory containing either. ld chooses the file ending in .so if -lx expands to two files with names of the form libx.so and libx.a. If no libx.so is found, then ld accepts libx.a. In static mode, or when the -B static option is in effect, ld selects only the file ending in .a. ld searches a library when it encounters its name, so the placement of -l is significant.

因为通常我们去命名一个常用库的时候，都是以lib开头，加上库的名称，如果是静态库，后缀是.a，如果是动态库，后缀是.so。

(I)Map指定map文件：
-Map的含义：
-MAP FILE 把符号映射关系输出到文件
-Map u-boot.map，意思就是，将地址和符号的映射关系，输出到u-boot.map文件里面。

(J)-o指定输出文件
-o u-boot就是，指定输出文件为u-boot，这个文件，我后来才得知，
由于之前的编译各个源文件的时候，是加了-g的参数的，RVDS 3.0，是可以用这个u-boot，加上u-boot.map配合，来调试u-boot的。

8.objcopy输出srec格式的可执行文件：
arm-linux-objcopy -O srec u-boot u-boot.srec
将当前的elf32-littlearm格式的，转换成srec格式的，输出。

9.objcopy输出二进制的可执行文件：
arm-linux-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
其中，--gap-fill=0xFF，意思就是gap,中间空的地方，即段与段之间，如果你指定了特殊地址，中间还有空余空间的，或者由于字节对其，中间空余的，都用0xFF来填充。
之所以指定0xFF，而不是0x0，估计是为了nand flash特性，空的flash里面数据都是0xFF的缘故吧，这样烧写uboot到flash上，就不会浪费多余动作将nand 芯片里面的1都写0了。

----------azhgul----------

U-Boot的源码是通过GCC和Makefile组织编译的。顶层目录下的Makefile首先可以设置开发板的定义，然后递归地调用各级子目录下的Makefile，最后把编译过的程序链接成U-Boot映像。

1. 顶层目录下的Makefile

它负责U-Boot整体配置编译。按照配置的顺序阅读其中关键的几行。

每一种开发板在Makefile都需要有板子配置的定义。例如smdk2410开发板的定义如下。

```
smdk2410_config :   unconfig
    @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

执行配置U-Boot的命令make smdk2410_config，通过./mkconfig脚本生成include/config.mk的配置文件。文件内容正是根据Makefile对开发板的配置生成的。

```
ARCH   = arm
CPU    = arm920t
BOARD = smdk2410
```

```
SOC    = s3c24x0
```

上面的include/config.mk文件定义了ARCH、CPU、BOARD、SOC这些变量。这样硬件平台依赖的目录文件可以根据这些定义来确定。SMDK2410平台相关目录如下。

```
board/smdk2410/
cpu/arm920t/
cpu/arm920t/s3c24x0/
lib_arm/
include/asm-arm/
include/configs/smdk2410.h
```

再回到顶层目录的Makefile文件开始的部分，其中下列几行包含了这些变量的定义。

```
# load ARCH, BOARD, and CPU configuration
include include/config.mk
export      ARCH CPU BOARD VENDOR SOC
```

Makefile的编译选项和规则在顶层目录的config.mk文件中定义。各种体系结构通用的规则直接在这个文件中定义。通过ARCH、 CPU、BOARD、SOC等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在ppc_config.mk、arm_config.mk、 mips_config.mk等文件中。

顶层目录的Makefile中还要定义交叉编译器，以及编译U-Boot所依赖的目标文件。

```
ifeq ($(ARCH),arm)
CROSS_COMPILE = arm-linux-          //交叉编译器的前缀
#endif
export CROSS_COMPILE
…
# U-Boot objects....order is important (i.e. start must be first)
OBJS = cpu/$(CPU)/start.o                   //处理器相关的目标文件
…
LIBS = lib_generic/libgeneric.a             //定义依赖的目录，每个目录下先把目标文件连接成*.a文件。
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
…
```

然后还有U-Boot映像编译的依赖关系。

```
ALL = u-boot.srec u-boot.bin System.map
all:        $(ALL)
u-boot.srec:    u-boot
          $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
u-boot.bin: u-boot
          $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
……
u-boot:          depend $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
          UNDEF_SYM='$(OBJDUMP) -x $(LIBS) \
          |sed -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
          $(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) \
              --start-group $(LIBS) $(PLATFORM_LIBS) --end-group \
              -Map u-boot.map -o u-boot
```

Makefile缺省的编译目标为all，包括u-boot.srec、u-boot.bin、System.map。u-boot.srec和 u-boot.bin又依赖于U-Boot。U-Boot就是通过ld命令按照u-boot.map地址表把目标文件组装成u-boot。

其他Makefile内容就不再详细分析了，上述代码分析应该可以为阅读代码提供了一个线索。

2．开发板配置头文件

除了编译过程Makefile以外，还要在程序中为开发板定义配置选项或者参数。这个头文件是include/configs/.h。用相应的BOARD定义代替。

这个头文件中主要定义了两类变量。

一类是选项，前缀是CONFIG_，用来选择处理器、设备接口、命令、属性等。例如：

#define CONFIG_ARM920T 1

#define CONFIG_DRIVER_CS8900 1

另一类是参数，前缀是CFG_，用来定义总线频率、串口波特率、Flash地址等参数。例如：

#define CFG_FLASH_BASE 0x00000000

#define CFG_PROMPT "=>"

3．编译结果

根据对Makefile的分析，编译分为2步。第1步配置，例如：make smdk2410_config；第2步编译，执行make就可以了。

编译完成后，可以得到U-Boot各种格式的映像文件和符号表，如表6.3所示。

表6.3 U-Boot编译生成的映像文件

文件名称	说　明	文件名称	说　明
System.map	U-Boot映像的符号表	u-boot.bin	U-Boot映像原始的二进制格式
u-boot	U-Boot映像的ELF格式	u-boot.srec	U-Boot映像的S-Record格式

U-Boot的3种映像格式都可以烧写到Flash中，但需要看加载器能否识别这些格式。一般u-boot.bin最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到Flash中就可以了。U-Boot和u-boot.srec格式映像都自带定位信息。

4．U-Boot工具

在tools目录下还有些U-Boot的工具。这些工具有的也经常用到。表6.4说明了几种工具的用途。

表6.4 U-Boot的工具

工具名称	说　明	工具名称	说　明
bmp_logo	制作标记的位图结构体	img2srec	转换SREC格式映像
envcrc	校验u-boot内部嵌入的环境变量	mkimage	转换U-Boot格式映像
gen_eth_addr	生成以太网接口MAC地址	updater	U-Boot自动更新升级工具

这些工具都有源代码，可以参考改写其他工具。其中mkimage是很常用的一个工具，Linux内核映像和ramdisk文件系统映像都可以转换成U-Boot的格式。

====

http://blog.csdn.net/azhgul/article/details/6673166

上一篇：Bash Shell中Shift用法
下一篇：arm-linux源码分析之解压内核映像

0