目录

[1.总的步骤 2](#_Toc353464742)

[2.setup的编译 2](#_Toc353464743)

[2.1platform.setp 2](#_Toc353464744)

[2.2 product.Setup 5](#_Toc353464745)

[3. make clean 6](#_Toc353464746)

[4.Uboot的编译 6](#_Toc353464747)

[5. make all 7](#_Toc353464748)

[5.1 linux内核的编译 7](#_Toc353464749)

[5.2 software的编译 8](#_Toc353464750)

[5.3 roofts的编译 10](#_Toc353464751)

[5.4 make app 10](#_Toc353464752)

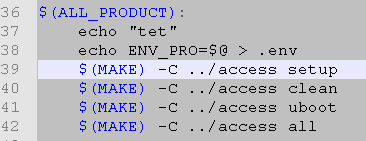
[5.5 image制作过程 11](#_Toc353464753)

EPN204 的Makefile 编译流程：

## 1.总的步骤

顶层makefile位于/Snmp\_develop/trunk/product/access目录下。

开始的切入点为如下：总结了下make其实总共就执行四步 setup、clean、uboot、all



ALL\_PRODUCT定义成为：



其实也就是指定哪个产品，我们在执行make时就指定 如：make EPN204

## 2.setup的编译

一开始的时候会把产品名赋值给ENV\_PRO并写入.env文件



在make的时候一开始就会输出信息为如下,给ENV\_PRO赋值并写入.env文件



接着执行 $(MAKE) -C ../access setup ，开始setup的编译。Setup被定义成伪目标，用.PHONY

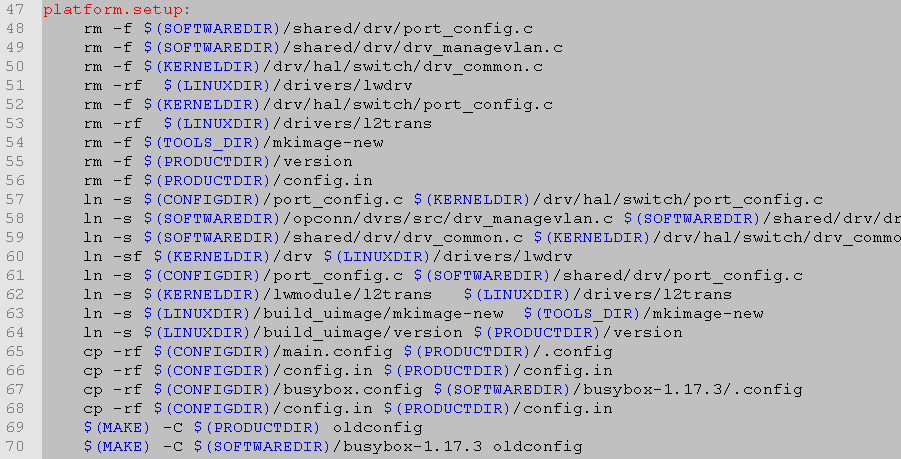


往下我们可以看到setup又分为platform.setp和product.setup



### 2.1platform.setp

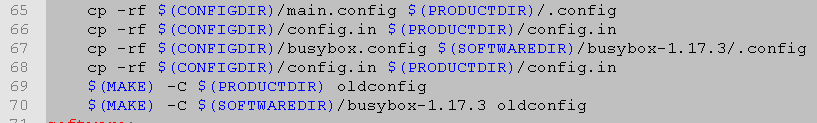
所以首先会去执行执行的是platform.Setup，如下图，platform.Setup包含了那么多的步骤



在这里会一一执行这些语句，首先是用rm命令清除一些目录下的原有文件，包括config文件和mkimage文件，然后在从其他目录链接过来或者拷贝过来，工具mkimage链接如下，在制作zImage的时候会用到。



还有下面的这些config文件的拷贝，



我们可以看到CONFIGDIR 表示的路径是





也就是产品的目录/ trunk/product/access/config/EPN204/ 对这里的一些config文件进行操作：

1. main.config文件并命名为.config保存在access目录，
2. 拷贝config.in到access目录，
3. 拷贝busybox.config到app/busybox-1.17.3目录下

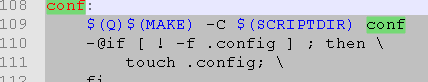
接着在进入access 与busybox-1.17.3目录下去编译



首先进入到目录access目录下，oldconfig这个伪目标依赖于conf 与$(INIT\_KCONFIG)，所以分两步运行



首先执行conf ，跳转到



会进入SCRIPTDIR 指定的目录

下编译生成conf，假如找不到.config就生成 touch .config 。 前面加个 “-”减号表示如果找不到忽略错误继续make

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

（这有个问题：就是前面不是从EPN204目录下拷贝了.config过来了么？为什么还要花费功夫又要编译生成.config？）我自己的理解：为了防止.config拷贝不成功或者没有.config文件。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

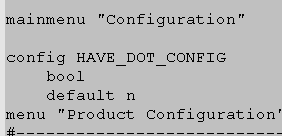
上面的步骤是为了生成conf文件，其实也就是个命令在下一步用到。编译完后跳转到 $(INIT\_KCONFIG) 执行 而我们发现INIT\_KCONFIG如下，其实在就是 指向config.in文件。

接下来的一句

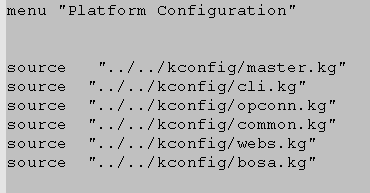
就是用前面生成的conf命令去运行config.in。在make的时候输出的打印为：



运行config.in就是去读取生成的文件里模块是否要编译。打开config.in会发现里面都是些模块定义，如以下就是模块HAVE\_DOT\_CONFIG的定义，default n表示模块没有被选上，也就没有被编译



在文件的底部还有诸如以下信息：



所以会跳转到kconfig目录下去执行这几个config文件，

(其实我们也可以用make menuconfig命令去配置执行生成.config， config.in里定义的模块选项我们都可以用make menuconfig生成界面形式去选择)

编译最后一个bosa.kg完成后会执行



在make的时候输出的信息为：



这时也就完成了platform.setup: 里的

这条语句的编译，然后转到下条语句

，这里也的步骤也跟上面的步骤类似，主要目的是进入busybox-1.17.3目录下去执行该目录下的Config.in文件，并生成.config文件。在make的时候输出的打印信息为:



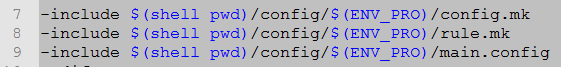
在编译完了busybox-1.17.3目录下的config.in文件后也就完成了setup里的第一步platform.Setup的编译。

### 2.2 product.Setup

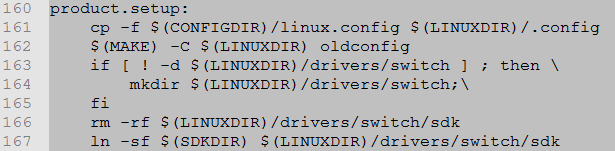
编译完platform.Setup后，接着执行setup步骤的第二步product.Setup



而在当前的目录也就是access目录下的makefile我们找不到product.Setup的定义，但我们注意到顶层makefile里一开始前面就定义了：



Makefile把config目录下产品EPN204目录下的这几个文件包含了进来，而product.Setup就是在rule.mk文件里，所以编译到这里就会转到产品目录下的rule.mk里去编译：



在这里会把产品目录下的linux.config复制到Linux内核下并更名为.config，LINUXDIR指向的目录就是linux内核的目录



接着进入到kernel/linux-2.6.21.5目录下去执行oldconfig伪目标，其实就是为了生成目录下的.config。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这里也有个问题，下面这句是make的时候打印的信息,找不到在哪定义的这句，但作用是一样都是为了生成.config



在哪里执行的oldconfig，目前还不清楚，不是在access目录下的oldconfig \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

接着判断kernel/linux-2.6.21.5/drivers目录下是否有目录switch，没有的话新建一个。然后删除switch目录下的sdk目录，从Snmp\_develop\trunk\product\access\sdk连接sdk目录过来：



SDKIR定义在access目录的makefile里。 然后结束product.Setup的编译，也就结束setup的编译。

## 3. make clean

在编译完成setup后，接着执行第二大步：



找到clean：如下：也是在当前顶层makefile里，执行make clean 其实是为了后面uboot、linux等的编译，在编译前先清除一下环境。



然后一一执行uboot-clean 、 software-clean比如：



在执行这些clean完后，在make的时候会打印出一下信息：



## 4.Uboot的编译

Clean完成后就执行第三大步

编译uboot，可以看到打印信息为：



前面说过因为包含了rule.mk文件，uboot是在rule.mk里定义的，所以进入到rule.mk去完成uboot的编译：



进入boot/u-boot目录编译uboot，在这里没有详细的给出编译的过程，有兴趣者可根据rule.mk里uboot的定义追踪到目录下的makefile等文件区分析。

## 5. make all

编译完uboot后，就开始了all的编译，在这里也是最重要的，这里详细分析下。



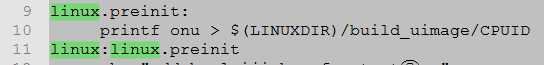
在顶层的makefile下找到all的定义：



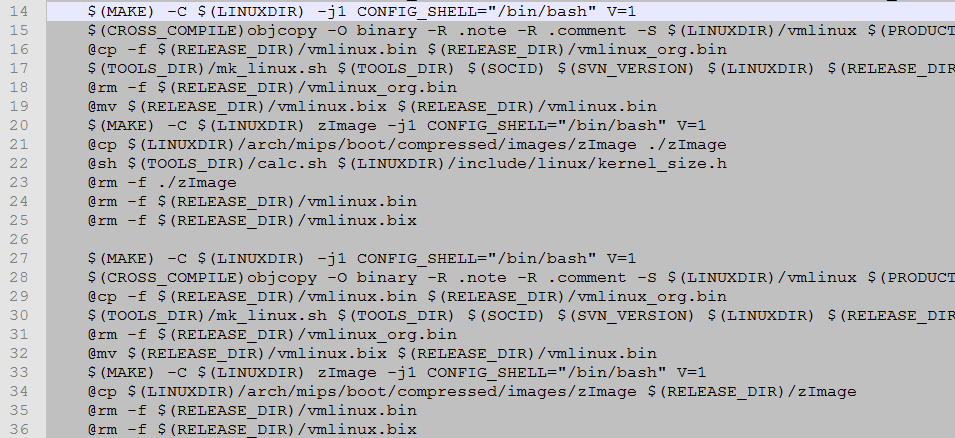
包含了蛮多内容如：linux linux-modules software、app、image等

### 5.1 linux内核的编译

首先是linux 内核的编译，其也是进入在产品目录下的rule.mk里完成的：



先跳转到linux.preinit去执行，打印onu到CPUID文件内。接着执行以下内容：



这里对linux内核进行两次编译，以中间分开，上边为第一次编译，下边为第二次编译，两次编译操作一样但完成的功能不一样。

1. 用object文件link生成的vmlinux来objcopy生成vmlinux.bin文件：



再用mkimage工具来生成vmlinux.bix ：



打开mk\_linux.sh 我们发现其实就是利用工具mkimage，然后对其进行压缩再加上个自解压头，制作成变成zImage



这个解压缩头涉及到uboot传递命令参数之类，在拷贝kernel到RAM内uboot会传递参数运行自解压程序解压kernel。在这次的编译中还计算了kernel的大小：



1. 在这次的编译中根据大小动态分配RAM ，一般在系统中，kernel之后就紧跟fs，假如这两个分配的空间是3.5M，一开始的时候kernel是0M，fs就是3.5-0 =3.5M整个空间分配给fs，在第一次编译结束后得到kernel的大小，，然后在第二次编译的时候，就把这个大小填充到0M的地方，假如是1M，而fs就会是3.5-1=2.5M，这就是第二次编译时的动态分配ram。

编译linux完后接着去编译linux-modoules：如下



会进入到kernel/drv/hal/cpu/mips/opulan/opl6750/module目录下去编译各个模块。

这里也不一一详细介绍了。

### 5.2 software的编译

编译完后就开始software的编译了



在linux-modules末尾里我加了一句add by leijinbao for test6 用于跟踪编译，可以看到结束后就开始software的编译，而应用程序的编译主要在编译software的时候，顶层makefile下我们可以看到software如下定义



其实SOFTWAREDIR 这个路径其实就是指向app，所以我们在开始的时候看到那条输出信息是进入目录app。



进入到/trunk/app目录下看makefile 以watchdog应用程序来看

首先先把目录编译进来

其次我们可以发现在makefile的底部有伪目标all，而我们在/product/access主目录下的makefile在编译software时



这个目标all在编译到software的时候指向/trunk/app目录下makefile ：



我们在这里往上一步步推发现

而

在这里说一下函数foreach在makefile的用法与作用，Foreach这个函数是用来做循环用的，语法如下：

$(foreach <var>,<list>,<text>)

意思是把，把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。在这里就是把$(obj-y) $(obj-n)放到obj中，再去执行$(obj)-build，以我们举的例子watchdog程序来看，因为前面有所以把watchdog放到obj中，这时再去执行$(obj)-build也就是执行watchdog-build。上面的等式最后就是obj-build = watchdog-build。

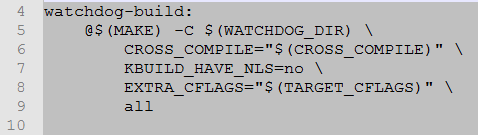
再看下这两行：



最后结果就是include< rules/ watchdog.mk >,include在makefile理就是包含进来文件

我们转到app/rules下找到watchdog.mk

会发现目标文件就是watchdog-build

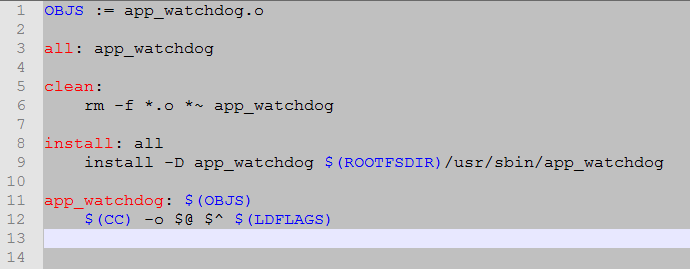


还发现为了生成这个目标文件执行

进入路径WATCHDOG\_DIR，而这个刚好有是watchdog源文件的目录



进入watchdog源文件目录下找到makefile：



就是对源文件的编译安装。

OBJS依赖于app\_watchdog.o ，最后生成app\_watchdog可执行程序



然后安装到$(ROOTFSDIR)/usr/sbin/app\_watchdog下





其他应用程序类似。

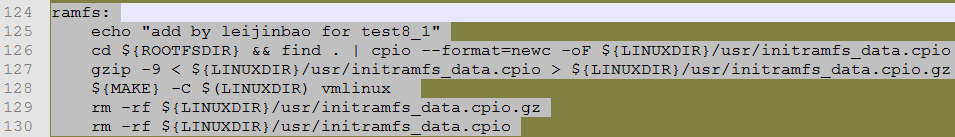
### 5.3 roofts的编译

编译完app目录下的应用程序后就开始了文件系统的编译，rootfs 的编译，在这也不详细说了，在rule.mk里实现的，可以在这个文件里找到roofts跟踪分析。

### 5.4 make app

接下来就到了app的编译制作过程了，也是在产品目录下的rule.mk文件内：

首先去执行ramfs，内存文件系统



在这里其实是为了生成opvos.bin文件，就是直接可以在RAM上跑的可执行文件，进入ramfs里，发现又一次编译了内核：

但这里并没有完成制作opvos.bin文件，而是结合了app里的三条语句：

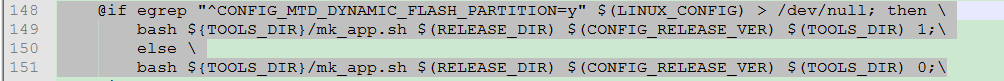


其实只要我们把这三条语句放在ramfs里然后在顶层makefile里all伪目标添加ramfs，完全就可以把ramfs从app中去掉，另成一分支，因为make app 本来就不包括opvos.bin的制作。

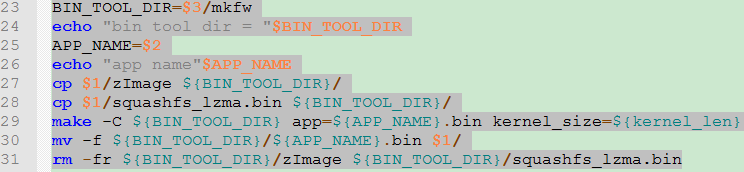
Make app 过程就是负责把squashfs.bin 、zImage、mkfw 制作成临时的产品.bin比如EPN204.bin，但这只是临时的而已。squashfs.bin就是利用工具squashfs对rootfs制作的：



而mkfw则是如下在mk\_app.sh中实现的，我们发现如下语句会利用mk\_app.sh文件。

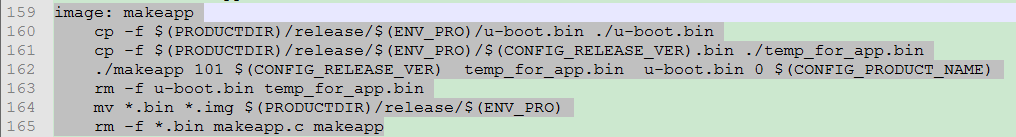


打开mk\_app.sh我们发现其实就是mkfw和app最后的生成.bin文件的过程。



### 5.5 image制作过程

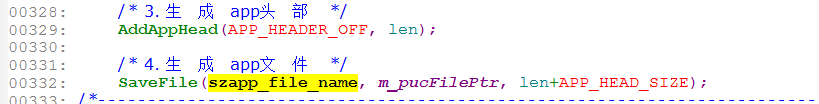
Make app结束后接着就到了image的制作过程了



首先会去执行makeapp，我们发现在makeapp里其实就是去编译执行makeapp.c文件生成makeapp，路径trunk\product\access\tools。然后利用makeapp制作image。



打开这个文件我们发现：



在这里会给make app过程中生成的临时的bin文件加上个app头部，最后替换掉原来的临时的bin文件，生成最终的bin文件比如：EPN204.bin文件。

Image的制作过程其实就是把u-boot.bin和前面make app过程生成的比如EPN204.bin文件再合成.img文件：

首先会拷贝u-boot.bin 和 临时的app制作生成的bin文件：

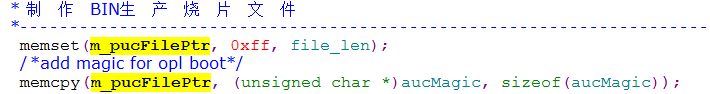


最后用makeapp工具生成.img文件：



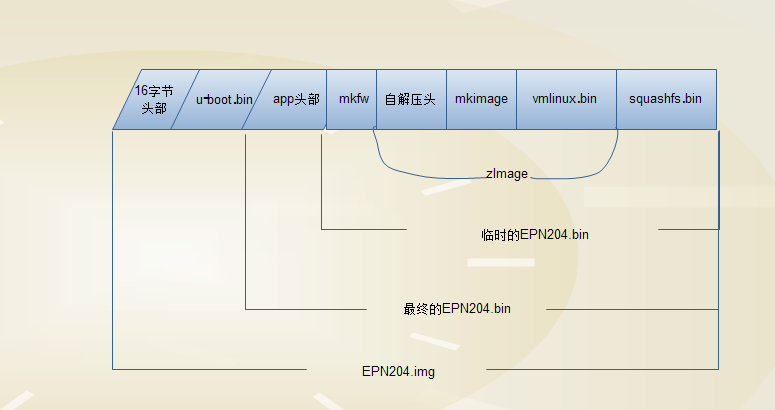


在makeapp.c里还对其加上了16字节的头部，



自此，img文件如EPN204.img全部制作完成。

如下图：



Makefile中常出现export (…)，这个是用来将变量传递到下一级的makefile

$< 表示所有依赖目标集

$@ 目标集