关于 arm-linux 工具链

1.arm-linux-ar
建立、修改、提取归档文件(类似于 tar 文件)。
2.arm-linux-ld
连接器，它把一些目标和归档文件结合在一起，重定位数据，并连接符号引用。通常是编译的最后一步。
3.arm-linux-objcopy
把一种目标文件中的内容复制到另一种类型的目标文件中。
4.arm-linux-objdump
显示一个或者更多目标文件的信息。使用选项来控制其显示的信息，它所显示的信息通常只有编写编译工具的人才感兴趣。

GCC 和 Makefile 基础知识

1.编译过程
预处理 gcc -E hello.c -o **hello.i** 预处理代码
编译 gcc -S hello.i -o **hello.s** 汇编代码
汇编 gcc -c hello.c -o **hello.o** 二进制码
连接 gcc hello.c -o **hello** 可执行文件

2.编译选项
gcc -g -Wall test.c -lm -o test
-o 只激活预处理,编译,和汇编,也就是他只把程序做成 obj 文件
-g 表示生成的目标文件中带调试信息
-Wall 表示开启警告
-O2 编译器对程序提供的编译优化选项，可以使生成的执行文件的执行效率提高
-lm 表示链接外部库函数的参数，或者将-lm 直接替换为库的路径
（查看函数所在头文件和连接库参数：**man 函数名**，查看连接的库：**ldd 可执行程序名**）
编译多个 c 文件依次列出即可：gcc -g -Wall test1.c test2.c -lm -o test

3.命令行调试工具 GBD
进入调试：gdb 可执行程序名
列出代码：l
设置断点：break 行号 或 break 函数名
查看断点信息：info break
运行：r
单步执行：n
查看变量：p 变量名
跳出函数：finish
继续运行：c
结束调试：q

4.GDBserver 远程调试
(1)先解压安装(略)
(2)再配置
cd /gdb/gdbserver
./configure --target=arm-linux
--host=arm-linux
make CC=arm-linux-gcc
(3)交叉编译：arm-linux-gcc -g test.c -o test

(4)目标机运行：./gbdserver ip 地址 test
(5)远程调试：arm-linux-gdb test

5.工程管理 Makefile 基本格式
在 makefile 文件中描述了整个工程所有文件的编译顺序、编译规则
规则的格式：
<target>:<depend>
    command(必须以[Tab]字符开始)

一条规则由三部分组成，test 是终极目标，也就是可执行程序名：
最简单的例子：
test（目标）：main.o xx.o x.o（依赖条件，永远会比目标文件旧）
    cc -o test main.o xx.o x.o(运行命令)
main.o : main.c xx.h
    cc -c main.c
xx.o : xx.c xxx.h
    cc -c xx.c
……
clean :
    rm main.o xx.o x.o（删除所有.o 文件）

编译时有三种情况：
(1)test 后的.o 文件不存在，会通过下面的规则生成.o 文件
(2)test 后的.o 文件存在，但是.o 所依赖的.c 或.h 文件被修改，也会重新生成.o 文件
(3)test 后的.o 文件存在，并且.o 文件比其依赖的文件新，则不需要做任何工作

设置依赖条件的作用：
如果依赖的某个文件比目标文件新，则目标认为是“过时的”，需要通过命令重新生成，也就是生成文件前会确保依赖文件是旧的，目标文件是新的。

6.Makefile 变量(文本字符串)用法
使用"="定义变量
调用时在变量名前加$，且多于一个字母的变量要加括号，即：$(变量名)

= 是最基本的赋值（结果为整个文件中最后被指定的那个值）
:= 是覆盖之前的值 (类似于变量的赋值，赋值后会立即生效）
?= 是如果没有被赋值过就赋予等号后面的值
+= 是添加等号后面的值

变量内字符的替换：
例：SRCS = fun1.c fun2.c
    OBJS = $(SRCS:.c=.o)
那么 OBJS 的值为 fun1.o fun2.o

7.Makefile 自动变量
$@ 表示**目标文件名**
$^ 表示**所有**依赖的名字
$< 表示**第一个**依赖的文件名

8.Makefile 常用函数

格式：$(函数名 参数 1，参数 2，参数 3)

1.$(subst ee, EE, feet)　　结果：fEEt

2.$(patsubst %.c, %.o, 1.c.c 2.c)　　结果：1.c.o 2.o

3.$(filter %.c %.s, 1.c 2.s 3.h)　　结果：1.c 2.s

4.$(filter-out %.c %.s, 1.c 2.s 3.h)　　结果：3.h

5.$(sort a c d b)　　结果：a b c d

6.$(if condition,then-part,[else-part])

Linux 内核的 Makefile 分为 5 个部分

1.顶层 Makefile：负责制作 vmlinux 和所有模块，制作的过程主要是通过递归向下访问子目录的形式完成。并根据内核配置文件确定访问哪些子目录。

2. .config 配置文件：由默认的配置文件拷贝得到，用 make menuconfig 也可产生并修改。

3.arch/$(ARCH)/Makefile：具体架构的 Makefile（其中在嵌入式中 ARCH = arm），向顶层 Makefile 提供其架构的特别信息。

4.scripts/Makefile.* 通用的规则：面向所有的 Kbuild Makefiles， 包含了所有的定义、规则等信息。（注意：kbuild Makefile 是指使用 kbuild 结构的 Makefile，内核中的大多数 Makefile 都是 kbuild Makefile）

5.Kbuild Makefiles：内核源代码每个文件夹下都有一个 Makefile ，大约有 500 个这样的文件。用来执行从其上层目录传递下来的命令，从.config 文件中提取信息，生成 Kbuild 编译内核需要的文件列表，它们负责生成 built-in.o 或模块化目标。

obj-y 的作用：

obj-y 用于生成 built-in.o，Kbuild 编译所有的$(obj-y)文件，并调用$(LD) -r 把所有这些文件合并到 built-in.o 文件。这个 built-in.o 会被上一级目录的 Makefile 使用，最终链接到 vmlinux 中。

类似的，lib-y 用于生成 lib.a 文件（对于 lib/和 arch/arm/lib/文件夹中的 Makefile）

向下递归：**obj-y += 子目录/**

驱动程序由多个 C 文件组成的情况：

如果一个驱动模块由多个 C 文件组成，则通过如下方式添加进来，yaffs-y 一般可以写为 yaffs-objs，最终只生成一个 yaffs.o 加入 obj-y 变量。

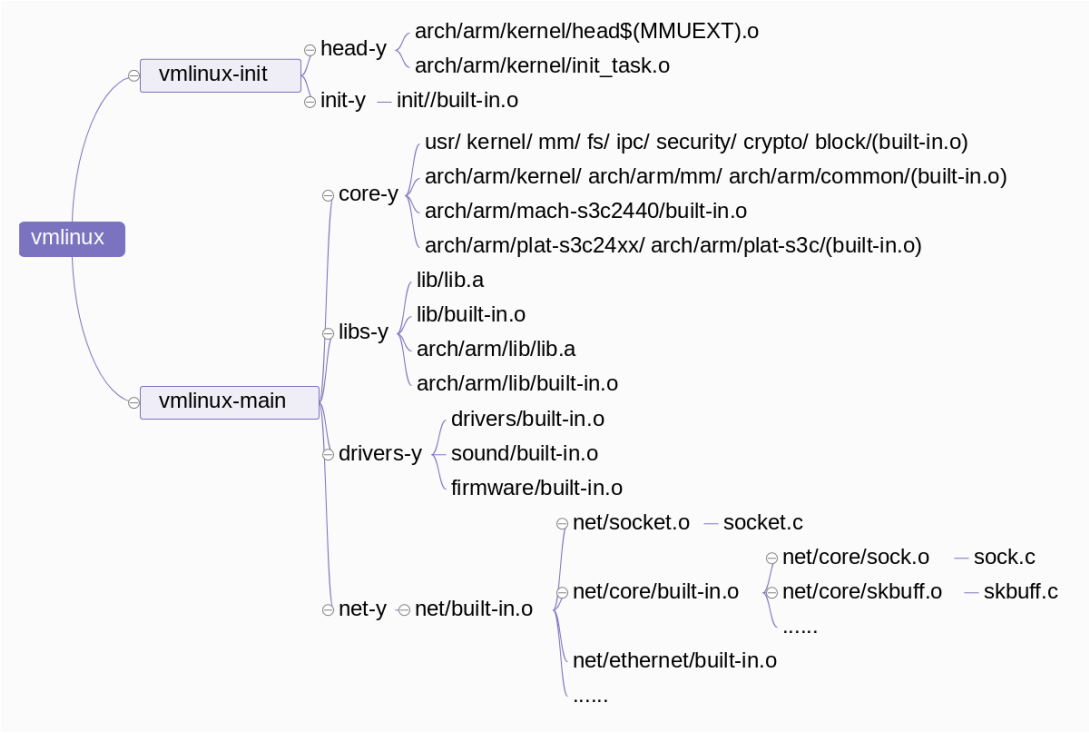
```
#
# Makefile for the linux YAFFS filesystem routines.
#

obj-$(CONFIG_YAFFS_FS) += yaffs.o

yaffs-y := yaffs_ecc.o yaffs_fs.o yaffs_guts.o yaffs_checkptrw.o
yaffs-y += yaffs_packedtags1.o yaffs_packedtags2.o yaffs_nand.o yaffs_qsort.o
yaffs-y += yaffs_tagscompat.o yaffs_tagsvalidity.o
yaffs-y += yaffs_mtdif.o yaffs_mtdif1.o yaffs_mtdif2.o
~
```

编译过程分析

将所有子目录分为 head.o、init-y、core-y、drivers-y、net-y、libs-y 这 6 类，再将子目录名替换子目录里的 built-in.o 文件名，这些 built-in.o 已经连接了更深子目录里的所有.o 文件。

然后将这 6 个变量合并为 2 个变量 vmlinux_init、vmlinux_main，这两个变量就是生成 vmlinux 的主要依赖条件。

整个编译过程如下，最终连接出 vmlinux 内核 elf 文件（再加上一些前缀生成镜像文件）：



对上图的解释：

上图中的 vmlinx-init、vmlinux-main 和后一级的 xxx-y 是**变量**，其他部分都是实际存在的**文件**。

最末端例如 sock.c 通过**.sock.o.cmd 编译**（内容是 arm-linux-gcc ...）产生 sock.o

接着 built-in.o 文件都是由**.built-in.o.cmd 连接**产生（内容是 arm-linux-ld -EL -r -o a.o b.o c.o ...）

最终 vmlinux 由所有最浅层子目录的 built-in.o 连接产生。

因此子目录里的 Makefile 写得非常简单（大部分都是 obj-y+=...），因为子目录的编译和连接操作基本都由 script 文件夹里的规则产生的.xxx.o.cmd 文件来完成了。

向下递归的核心代码分析

首先看顶层 Makefile 中:

```
$(vmlinux-dirs): prepare scripts #vmlinux-dirs 实际就是浅层子目录名且去掉了后面的斜杠
	$(Q)$(MAKE) $(build)=$@ #向下递归的关键入口
```

上句翻译过来就是 make -f **scripts/Makefile.build** obj= vmlinux-dirs

真正的目标其实是 vmlinux-dirs 目录下的 Makefile 文件，即$(obj)/Makefile。

首先会构建默认目标__build 的依赖：

__build 在 Makefile.build 中的构建规则为：

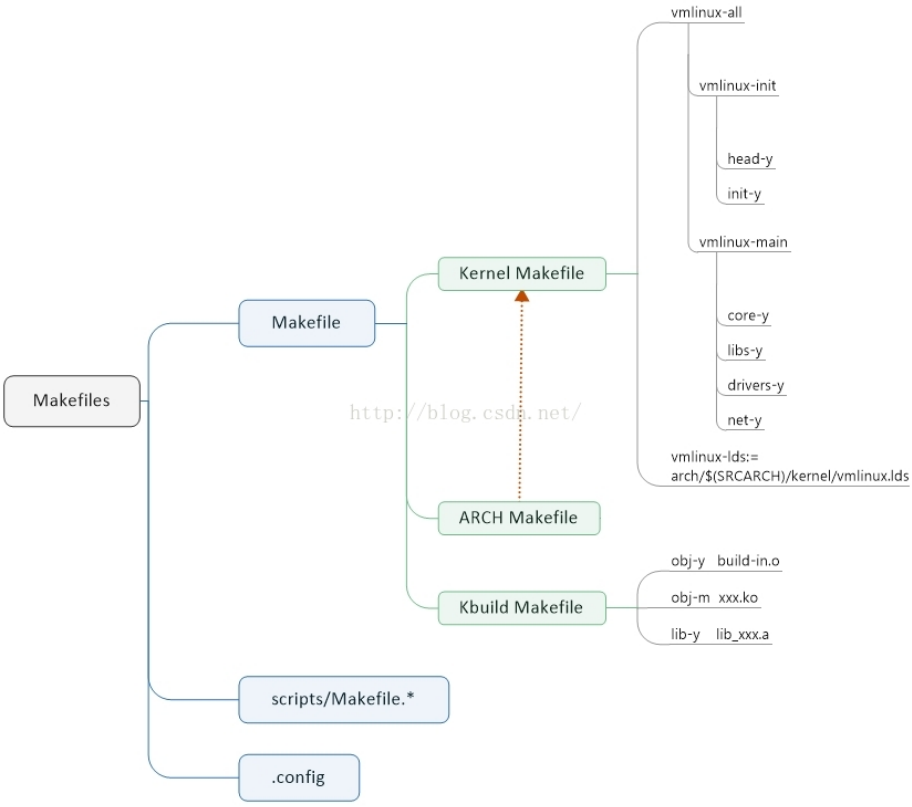
```
__build:$(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y))\
	$(if $(KBUILD_MODULES),$(obj-m) $(modorder-target))\
	$(subdir-ym)$(always)
	@:
```

因$(KBUILD_BUILTIN)被主目录设置为 1，且 export，所以将首先重建依赖$(builtin-target)。而依赖$(builtin-target)的重建规则在 Makefile.build 中：

```
$(builtin-target):$(obj-y)FORCE   #$(obj)/built-in.o,且先要构建依赖$(obj-y)
	$(call if_changed,link_o_target)
```

该规则要依赖$(obj-y)变量。$(obj-y)在$(obj)/Makefile 中定义且被赋值为各种.o 文件，这时编译器会查找$(obj-y)变量包含的.o 文件的构建规则（此时 obj-y 里的.o 文件还不存在）。同样地，该规则要么在 Makefile.build 中，要么在$(obj)/Makefile 中。.o 文件的构建规则在 Makefile.build 中被定义：

```
$(obj)/%.o:$(src)/%.c$(recordmcount_source)FORCE #$(obj-y)里的所有.o 都依赖于对应的.c 文件
	$(call cmd,force_checksrc)
	$(call if_changed_rule,cc_o_c)
```



最终 vmlinux 的连接代码分析

顶层 Makefile 中，如下图：

patsubst 函数将目录中的“/”替换为“/built-in.o”，为了将所有子目录里的 built-in.o 连接起来。

```
670 ifeq ($(KBUILD_EXTMOD),)
671 core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
672
673 vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
674                      $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
675                      $(net-y) $(net-m) $(libs-y) $(libs-m)))
676
677 vmlinux-alldirs := $(sort $(vmlinux-dirs) $(patsubst %/,%,$(filter %/, \
678                      $(init-n) $(init-) \
679                      $(core-n) $(core-) $(drivers-n) $(drivers-) \
680                      $(net-n)  $(net-)  $(libs-n)    $(libs-))))
681
682 init-y          := $(patsubst %/, %/built-in.o, $(init-y))
683 core-y          := $(patsubst %/, %/built-in.o, $(core-y))
684 drivers-y       := $(patsubst %/, %/built-in.o, $(drivers-y))
685 net-y           := $(patsubst %/, %/built-in.o, $(net-y))
686 libs-y1         := $(patsubst %/, %/lib.a, $(libs-y))
687 libs-y2         := $(patsubst %/, %/built-in.o, $(libs-y))
688 libs-y          := $(libs-y1) $(libs-y2)
689
690 # Build vmlinux
691 # -------------------------------------------------------------------------
692 # vmlinux is built from the objects selected by $(vmlinux-init) and
693 # $(vmlinux-main). Most are built-in.o files from top-level directories
694 # in the kernel tree, others are specified in arch/$(ARCH)/Makefile.
695 # Ordering when linking is important, and $(vmlinux-init) must be first.
696 #
697 # vmlinux
698 #   ^
699 #   |
700 #   +-< $(vmlinux-init)
701 #   |   +--< init/version.o + more
702 #   |
703 #   +--< $(vmlinux-main)
704 #   |    +--< driver/built-in.o mm/built-in.o + more
705 #   |
706 #   +-< kallsyms.o (see description in CONFIG_KALLSYMS section)
```

cmd_vmlinux__：定义了连接生成 vmlinux 的命令和规则，下面只要满足了 if_changed_rule，会用被调用到。（在 scripts/Kbuild.include 中定义了 if_changed_rule，实际上就是个 if 函数）

```
717 vmlinux-init := $(head-y) $(init-y)
718 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
719 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
720 vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
721 export KBUILD_VMLINUX_OBJS := $(vmlinux-all)
722
723 # Rule to link vmlinux - also used during CONFIG_KALLSYMS
724 # May be overridden by arch/$(ARCH)/Makefile
725 quiet_cmd_vmlinux__ ?= LD      $@
726       cmd_vmlinux__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
727       -T $(vmlinux-lds) $(vmlinux-init)                          \
728       --start-group $(vmlinux-main) --end-group                  \
729       $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)
730
```

vmlinux-modpost：定义了连接生成 vmlinux 的命令和规则，在连接 vmlinux 时会调用。

```
853 # Do modpost on a prelinked vmlinux. The finally linked vmlinux has
854 # relevant sections renamed as per the linker script.
855 quiet_cmd_vmlinux-modpost = LD      $@
856       cmd_vmlinux-modpost = $(LD) $(LDFLAGS) -r -o $@                      \
857         $(vmlinux-init) --start-group $(vmlinux-main) --end-group          \
858         $(filter-out $(vmlinux-init) $(vmlinux-main) FORCE ,$^)
859 define rule_vmlinux-modpost
860         :
861         +$(call cmd,vmlinux-modpost)
862         $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost $@
863         $(Q)echo 'cmd_$@ := $(cmd_vmlinux-modpost)' > $(dot-target).cmd
864 endef
865
866 # vmlinux image - including updated kernel symbols
867 vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE
868 ifdef CONFIG_HEADERS_CHECK
869         $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
870 endif
871 ifdef CONFIG_SAMPLES
872         $(Q)$(MAKE) $(build)=samples
873 endif
874 ifdef CONFIG_BUILD_DOCSRC
875         $(Q)$(MAKE) $(build)=Documentation
876 endif
877         $(call vmlinux-modpost)
878         $(call if_changed_rule,vmlinux__)
879         $(Q)rm -f .old_version
```

vmlinux 的连接过程见如上图最后三行。

vmlinux-init 和 vmlinux-main 的分析方法:

由上图 vmlinux 的依赖变量主要有 vmlinux-init，vmlinux-main，在此对 vmlinux-init 进行简单分析：

在上上张图中：

vmlinux-init := $(head-y) $(init-y)

vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)

```
553 include $(srctree)/arch/$(SRCARCH)/Makefile
```

由以上 include 信息得到 head-y 的位置在 arch/arm/Makefile 中：

打开 arch/arm 目录下的 Makefile，就能找到 head-y 变量的实际值：

```
110 head-y          := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o
```

对于 init-y，回到顶层 Makefile：

```
682 init-y          := $(patsubst %/, %/built-in.o, $(init-y))
```

上图的意思就是 init-y := init/built-in.o

vmlinux-main 的分析过程和 vmlinux-init 类似。

详细讲解请参考 GNU Makefile 中文手册.pdf：http://vdisk.weibo.com/s/u8nUXA3JjZ5j0