Linux Kernel Makefiles

This document describes the Linux kernel Makefiles.

目录

1. Overview

The Makefiles have five parts:

Makefile the top Makefile.

.config the kernel configuration file.

arch/$(ARCH)/Makefile the arch Makefile.

scripts/Makefile.* common rules etc. for all kbuild Makefiles.

kbuild Makefiles there are about 500 of these.

The top Makefile reads the .config file, which comes from the kernel configuration process.

The top Makefile is responsible for building two major products: vmlinux (the resident kernel image) and modules (any module files). It builds these goals by recursively descending into the subdirectories of the kernel source tree. The list of subdirectories which are visited depends upon the kernel configuration. The top Makefile textually includes an arch Makefile with the name arch/$(ARCH)/Makefile. The arch Makefile supplies architecture-specific information to the top Makefile.

Each subdirectory has a kbuild Makefile which carries out the commands passed down from above. The kbuild Makefile uses information from the .config file to construct various file lists used by kbuild to build any built-in or modular targets.

scripts/Makefile.* contains all the definitions/rules etc. that are used to build the kernel based on the kbuild makefiles.

2. Who does what

People have four different relationships with the kernel Makefiles.

Users are people who build kernels. These people type commands such as "make menuconfig" or "make". They usually do not read or edit any kernel Makefiles (or any other source files).

Normal developers are people who work on features such as device drivers, file systems, and network protocols. These people need to maintain the kbuild Makefiles for the subsystem they are working on. In order to do this effectively, they need some overall knowledge about the kernel Makefiles, plus detailed knowledge about the public interface for kbuild.

Arch developers are people who work on an entire architecture, such as sparc or ia64. Arch developers need to know about the arch Makefile as well as kbuild Makefiles.

Kbuild developers are people who work on the kernel build system itself. These people need to know about all aspects of the kernel Makefiles.

This document is aimed towards normal developers and arch developers.

3. The kbuild files

Most Makefiles within the kernel are kbuild Makefiles that use the kbuild infrastructure. This chapter introduces the syntax used in the kbuild makefiles. The preferred name for the kbuild files are 'Makefile' but 'Kbuild' can be used and if both a 'Makefile' and a 'Kbuild' file exists, then the 'Kbuild' file will be used.

Section 3.1 "Goal definitions" is a quick intro, further chapters provide more details, with real examples.

3.1. Goal definitions

Goal definitions are the main part (heart) of the kbuild Makefile. These lines define the files to be built, any special compilation options, and any subdirectories to be entered recursively.

The most simple kbuild makefile contains one line:

Example: obj-y += foo.o

This tells kbuild that there is one object in that directory, named

foo.o. foo.o will be built from foo.c or foo.S.

If foo.o shall be built as a module, the variable obj-m is used. Therefore the following pattern is often used:

Example: obj-$(CONFIG_FOO) += foo.o

$(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled

nor linked.

3.2. Built-in object goals - obj-y

The kbuild Makefile specifies object files for vmlinux
in the $(obj-y) lists. These lists depend on the kernel
configuration.
Kbuild compiles all the $(obj-y) files. It then calls
"$(LD) -r" to merge these files into one built-in.o file.
built-in.o is later linked into vmlinux by the parent Makefile.
The order of files in $(obj-y) is significant. Duplicates in
the lists are allowed: the first instance will be linked into
built-in.o and succeeding instances will be ignored.
Link order is significant, because certain functions
(module_init() / __initcall) will be called during boot in the
order they appear. So keep in mind that changing the link
order may e.g. change the order in which your SCSI
controllers are detected, and thus your disks are renumbered.
Example: #drivers/isdn/i4l/Makefile

```
# Makefile for the kernel ISDN subsystem and device drivers.
# Each configuration option enables a list of files.
obj-$(CONFIG_ISDN)                      += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

3.3. Loadable module goals - obj-m

$(obj-m) specify object files which are built as loadable
kernel modules.
A module may be built from one source file or several source
files. In the case of one source file, the kbuild makefile
simply adds the file to $(obj-m).
Example: #drivers/isdn/i4l/Makefile

```
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

Note: In this example $(CONFIG_ISDN_PPP_BSDCOMP) evaluates to 'm'

If a kernel module is built from several source files, you specify
that you want to build a module in the same way as above.
Kbuild needs to know which the parts that you want to build your
module from, so you have to tell it by setting an
$(<module_name>-objs) variable.
Example: #drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN) += isdn.o
isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
In this example, the module name will be isdn.o. Kbuild will
compile the objects listed in $(isdn-objs) and then run
"$(LD) -r" on the list of these files to generate isdn.o.
Kbuild recognises objects used for composite objects by the suffix
-objs, and the suffix -y. This allows the Makefiles to use
the value of a CONFIG_ symbol to determine if an object is part
of a composite object.
Example: #fs/ext2/Makefile obj-$(CONFIG_EXT2_FS) += ext2.o ext2-
y := balloc.o bitmap.o ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
In this example, xattr.o is only part of the composite object
ext2.o if $(CONFIG_EXT2_FS_XATTR) evaluates to 'y'.
Note: Of course, when you are building objects into the kernel,
the syntax above will also work. So, if you have CONFIG_EXT2_FS=y, kbuild will build an
ext2.o file for you out of the individual
parts and then link this into built-in.o, as you would expect.

3.4. Objects which export symbols

No special notation is required in the makefiles for
modules exporting symbols.

3.5. Library file goals - lib-y

Objects listed with obj-* are used for modules, or

combined in a built-in.o for that specific directory.

There is also the possibility to list objects that will

be included in a library, lib.a.

All objects listed with lib-y are combined in a single

library for that directory.

Objects that are listed in obj-y and additionally listed in

lib-y will not be included in the library, since they will

be accessible anyway.

For consistency, objects listed in lib-m will be included in lib.a.

Note that the same kbuild makefile may list files to be built-in

and to be part of a library. Therefore the same directory

may contain both a built-in.o and a lib.a file.

Example: #arch/i386/lib/Makefile

```
lib-y        := checksum.o delay.o
```

This will create a library lib.a based on checksum.o and delay.o.

For kbuild to actually recognize that there is a lib.a being built,

the directory shall be listed in libs-y.

See also "6.3 List directories to visit when descending".

Use of lib-y is normally restricted to lib/ and arch/*/lib.

3.6. Descending down in directories

A Makefile is only responsible for building objects in its own directory. Files in subdirectories should be taken care of by

Makefiles in these subdirs. The build system will automatically

invoke make recursively in subdirectories, provided you let it know of them.

To do so, obj-y and obj-m are used.

ext2 lives in a separate directory, and the Makefile present in fs/ tells kbuild to descend down using the following assignment.

Example: #fs/Makefile

```
obj-$(CONFIG_EXT2_FS) += ext2/
```

If CONFIG_EXT2_FS is set to either 'y' (built-in) or 'm' (modular)

the corresponding obj- variable will be set, and kbuild will descend down in the ext2 directory.

Kbuild only uses this information to decide that it needs to visit

the directory, it is the Makefile in the subdirectory that

specifies what is modules and what is built-in.

It is good practice to use a CONFIG_ variable when assigning directory names. This allows kbuild to totally skip the directory if the corresponding CONFIG_ option is neither 'y' nor 'm'.

3.7. Compilation flags

ccflags-y, asflags-y and ldflags-y The three flags listed above applies only to the kbuild makefile

where they are assigned. They are used for all the normal

cc, as and ld invocation happenign during a recursive build.

Note: Flags with the same behaviour were previously named:

EXTRA_CFLAGS, EXTRA_AFLAGS and EXTRA_LDFLAGS.

They are yet supported but their use are deprecated.

ccflags-y specifies options for compiling C files with $(CC).

Example: # drivers/sound/emu10k1/Makefile

```
ccflags-y += -I$(obj)
ccflags-$(DEBUG) += -DEMU10K1_DEBUG
```

This variable is necessary because the top Makefile owns the

variable $(KBUILD_CFLAGS) and uses it for compilation flags for the entire tree.

asflags-y is a similar string for per-directory options

when compiling assembly language source.

Example: #arch/x86_64/kernel/Makefile

```
asflags-y := -traditional
```

ldflags-y is a string for per-directory options to $(LD).

Example: #arch/m68k/fpsp040/Makefile

```
ldflags-y := -x
```

subdir-ccflags-y, subdir-asflags-y The two flags listed above are similar to ccflags-y and

as-falgs-y.

The difference is that the subdir- variants has effect for the kbuild file where tey are present and all subdirectories.

Options specified using subdir-* are added to the commandline before the options specified using the non-subdir variants.

Example: subdir-ccflags-y := -Werror

CFLAGS_$@, AFLAGS_$@

CFLAGS_$@ and AFLAGS_$@ only apply to commands in current

kbuild makefile.

$(CFLAGS_$@) specifies per-file options for $(CC). The $@

part has a literal value which specifies the file that it is for.

Example: # drivers/scsi/Makefile

CFLAGS_aha152x.o = -DAHA152X_STAT -DAUTOCONF

CFLAGS_gdth.o = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \

-DGDTH_STATISTICS CFLAGS_seagate.o = -DARBITRATE -DPARITY -DSEAGATE_USE_ASM

These three lines specify compilation flags for aha152x.o,

gdth.o, and seagate.o

$(AFLAGS_$@) is a similar feature for source files in assembly languages.

Example: # arch/arm/kernel/Makefile

AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional

AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional

3.8. Dependency tracking

Kbuild tracks dependencies on the following: 1) All prerequisite files (both *.c and *.h) 2) CONFIG_ options used in all prerequisite files 3) Command-line used to compile target

Thus, if you change an option to $(CC) all affected files will

be re-compiled.

3.9. Special Rules

Special rules are used when the kbuild infrastructure does
not provide the required support. A typical example is
header files generated during the build process.
Another example are the architecture-specific Makefiles which
need special rules to prepare boot images etc.
Special rules are written as normal Make rules.
Kbuild is not executing in the directory where the Makefile is
located, so all special rules shall provide a relative
path to prerequisite files and target files.
Two variables are used when defining special rules:
$(src) $(src) is a relative path which points to the directory
where the Makefile is located. Always use $(src) when
referring to files located in the src tree.
$(obj) $(obj) is a relative path which points to the directory
where the target is saved. Always use $(obj) when
referring to generated files.
Example: #drivers/scsi/Makefile

$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl $(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl

This is a special rule, following the normal syntax
required by make.
The target file depends on two prerequisite files. References
to the target file are prefixed with $(obj), references
to prerequisites are referenced with $(src) (because they are not generated files).
$(kecho) echoing information to user in a rule is often a good practice
but when execution "make -s" one does not expect to see any output except for warnings/errors.
To support this kbuild define $(kecho) which will echo out the
text following $(kecho) to stdout except if "make -s" is used.

Example: #arch/blackfin/boot/Makefile

```
$(obj)/vmImage: $(obj)/vmlinux.gz $(call if_changed,uimage)
    @$(kecho) 'Kernel: $@ is ready'
```

3.10. $(CC) support functions

The kernel may be built with several different versions of
$(CC), each supporting a unique set of features and options.
kbuild provide basic support to check for valid options for $(CC).
$(CC) is usually the gcc compiler, but other alternatives are
available.

as-option as-option is used to check if $(CC) -- when used to compile
assembler (*.S) files -- supports the given option. An optional
second option may be specified if the first option is not supported.

Example: #arch/sh/Makefile

```
cflags-y += $(call as-option,-Wa$(comma)-isa=$(isa-y),)
```

In the above example, cflags-y will be assigned the option
-Wa$(comma)-isa=$(isa-y) if it is supported by $(CC).
The second argument is optional, and if supplied will be used
if first argument is not supported.

cc-ldoption cc-ldoption is used to check if $(CC) when used to link object files supports the given option. An optional second option may be
specified if first option are not supported.

Example: #arch/i386/kernel/Makefile

```
vsyscall-flags += $(call cc-ldoption, -Wl$(comma)--hash-style=sysv)
```

In the above example, vsyscall-flags will be assigned the option
-Wl$(comma)--hash-style=sysv if it is supported by $(CC).
The second argument is optional, and if supplied will be used
if first argument is not supported.

as-instr as-instr checks if the assembler reports a specific instruction
and then outputs either option1 or option2
C escapes are supported in the test instruction

Note: as-instr-option uses KBUILD_AFLAGS for $(AS) options

cc-option cc-option is used to check if $(CC) supports a given option, and not supported to use an optional second option.

Example: #arch/i386/Makefile

cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)

In the above example, cflags-y will be assigned the option

-march=pentium-mmx if supported by $(CC), otherwise -march=i586.

The second argument to cc-option is optional, and if omitted,

cflags-y will be assigned no value if first option is not supported. Note: cc-option uses KBUILD_CFLAGS for $(CC) options

cc-option-yn cc-option-yn is used to check if gcc supports a given option

and return 'y' if supported, otherwise 'n'.

Example: #arch/ppc/Makefile

biarch := $(call cc-option-yn, -m32)

aflags-$(biarch) += -a32

cflags-$(biarch) += -m32

In the above example, $(biarch) is set to y if $(CC) supports the -m32 option. When $(biarch) equals 'y', the expanded variables $(aflags-y) and $(cflags-y) will be assigned the values -a32 and -m32,

respectively.

Note: cc-option-yn uses KBUILD_CFLAGS for $(CC) options

cc-option-align gcc versions >= 3.0 changed the type of options used to specify alignment of functions, loops etc. $(cc-option-align), when used

as prefix to the align options, will select the right prefix:

gcc < 3.00 cc-option-align = -malign gcc >= 3.00 cc-option-align = -falign

Example: KBUILD_CFLAGS += $(cc-option-align)-functions=4

In the above example, the option -falign-functions=4 is used for

gcc >= 3.00. For gcc < 3.00, -malign-functions=4 is used.

Note: cc-option-align uses KBUILD_CFLAGS for $(CC) options

cc-version cc-version returns a numerical version of the $(CC) compiler version. The format is <major><minor> where both are two digits. So for example gcc 3.41 would return 0341.

cc-version is useful when a specific $(CC) version is faulty in one area, for example -mregparm=3 was broken in some gcc versions

even though the option was accepted by gcc.

Example: #arch/i386/Makefile

cflags-y += $(shell \

if [$(call cc-version) -ge 0300] ; then \ echo "-mregparm=3"; fi ;)

In the above example, -mregparm=3 is only used for gcc version greater than or equal to gcc 3.0.

cc-ifversion cc-ifversion tests the version of $(CC) and equals last argument if version expression is true.

Example: #fs/reiserfs/Makefile

ccflags-y := $(call cc-ifversion, -lt, 0402, -O1)

In this example, ccflags-y will be assigned the value -O1 if the

$(CC) version is less than 4.2.

cc-ifversion takes all the shell operators:

-eq, -ne, -lt, -le, -gt, and -ge

The third parameter may be a text as in this example, but it may also be an expanded variable or a macro.

cc-fullversion cc-fullversion is useful when the exact version of gcc is needed.

One typical use-case is when a specific GCC version is broken.

cc-fullversion points out a more specific version than cc-version does.

Example: #arch/powerpc/Makefile

$(Q)if test "$(call cc-fullversion)" = "040200" ; then \ echo -n '*** GCC-4.2.0 cannot compile the 64-bit powerpc ' ; \

false ; \ fi

In this example for a specific GCC version the build will error out explaining to the user why it stops.

cc-cross-prefix cc-cross-prefix is used to check if there exists a $(CC) in path with one of the listed prefixes. The first prefix where there exist a

prefix$(CC) in the PATH is returned - and if no prefix$(CC) is found then nothing is returned.

Additional prefixes are separated by a single space in the

call of cc-cross-prefix.

This functionality is useful for architecture Makefiles that try

to set CROSS_COMPILE to well-known values but may have several

values to select between.

It is recommended only to try to set CROSS_COMPILE if it is a cross build (host arch is different from target arch). And if CROSS_COMPILE

is already set then leave it with the old value.

Example: #arch/m68k/Makefile

ifneq ($(SUBARCH),$(ARCH)) ifeq ($(CROSS_COMPILE),) CROSS_COMPILE := $(call cc-cross-prefix, m68k-linux-gnu-) endif endif

3.11. $(LD) support functions

ld-option ld-option is used to check if $(LD) supports the supplied option.

ld-option takes two options as arguments.

The second argument is an optional option that can be used if the

first option is not supported by $(LD).

Example: #Makefile

LDFLAGS_vmlinux += $(call really-ld-option, -X)

4. Host Program support

Kbuild supports building executables on the host for use during the compilation stage. Two steps are required in order to use a host executable.

The first step is to tell kbuild that a host program exists. This is done utilising the variable hostprogs-y.

The second step is to add an explicit dependency to the executable. This can be done in two ways. Either add the dependency in a rule, or utilise the variable $(always). Both possibilities are described in the following.

4.1. Simple Host Program

In some cases there is a need to compile and run a program on the computer where the build is running.

The following line tells kbuild that the program bin2hex shall be

built on the build host.

Example: hostprogs-y := bin2hex

Kbuild assumes in the above example that bin2hex is made from a single c-source file named bin2hex.c located in the same directory as

the Makefile.

4.2. Composite Host Programs

Host programs can be made up based on composite objects.

The syntax used to define composite objects for host programs is similar to the syntax used for kernel objects.

$(<executable>-objs) lists all objects used to link the final executable.

Example: #scripts/lxdialog/Makefile

```
hostprogs-y     := lxdialog
lxdialog-objs := checklist.o lxdialog.o
```

Objects with extension .o are compiled from the corresponding .c

files. In the above example, checklist.c is compiled to checklist.o

and lxdialog.c is compiled to lxdialog.o.

Finally, the two .o files are linked to the executable, lxdialog.

Note: The syntax <executable>-y is not permitted for host-programs.

4.3. Defining shared libraries

Objects with extension .so are considered shared libraries, and

will be compiled as position independent objects.

Kbuild provides support for shared libraries, but the usage

shall be restricted.

In the following example the libkconfig.so shared library is used

to link the executable conf.

Example: #scripts/kconfig/Makefile

```
hostprogs-y     := conf
conf-objs       := conf.o libkconfig.so
libkconfig-objs := expr.o type.o
```

Shared libraries always require a corresponding -objs line, and

in the example above the shared library libkconfig is composed by

the two objects expr.o and type.o.

expr.o and type.o will be built as position independent code and

linked as a shared library libkconfig.so. C++ is not supported for shared libraries.

4.4. Using C++ for host programs

kbuild offers support for host programs written in C++. This was introduced solely to support kconfig, and is not recommended

for general use.

Example: #scripts/kconfig/Makefile

```
hostprogs-y     := qconf
qconf-cxxobjs := qconf.o
```

In the example above the executable is composed of the C++ file

qconf.cc - identified by $(qconf-cxxobjs).

If qconf is composed by a mixture of .c and .cc files, then an additional line can be used to identify this.

Example: #scripts/kconfig/Makefile

```
hostprogs-y     := qconf
qconf-cxxobjs := qconf.o
qconf-objs      := check.o
```

4.5. Controlling compiler options for host programs

When compiling host programs, it is possible to set specific flags.

The programs will always be compiled utilising $(HOSTCC) passed
the options specified in $(HOSTCFLAGS).
To set flags that will take effect for all host programs created
in that Makefile, use the variable HOST_EXTRACFLAGS.
Example: #scripts/lxdialog/Makefile

    HOST_EXTRACFLAGS += -I/usr/include/ncurses

To set specific flags for a single file the following construction
is used:
Example: #arch/ppc64/boot/Makefile

    HOSTCFLAGS_piggyback.o := -DKERNELBASE=$(KERNELBASE)

It is also possible to specify additional options to the linker.
Example: #scripts/kconfig/Makefile

    HOSTLOADLIBES_qconf := -L$(QTDIR)/lib

When linking qconf, it will be passed the extra option
"-L$(QTDIR)/lib".

4.6. When host programs are actually built

Kbuild will only build host-programs when they are referenced
as a prerequisite.
This is possible in two ways:
(1) List the prerequisite explicitly in a special rule.
Example: #drivers/pci/Makefile

    hostprogs-y := gen-devlist
    $(obj)/devlist.h: $(src)/pci.ids $(obj)/gen-devlist (cd $(obj); ./gen-devlist) < $<

The target $(obj)/devlist.h will not be built before
$(obj)/gen-devlist is updated. Note that references to
the host programs in special rules must be prefixed with $(obj).
(2) Use $(always)
When there is no suitable special rule, and the host program
shall be built when a makefile is entered, the $(always)
variable shall be used.

Example: #scripts/lxdialog/Makefile

```
hostprogs-y     := lxdialog
always          := $(hostprogs-y)
```

This will tell kbuild to build lxdialog even if not referenced in any rule.

4.7. Using hostprogs-$(CONFIG_FOO)

A typical pattern in a Kbuild file looks like this:

Example: #scripts/Makefile

```
hostprogs-$(CONFIG_KALLSYMS) += kallsyms
```

Kbuild knows about both 'y' for built-in and 'm' for module. So if a config symbol evaluate to 'm', kbuild will still build the binary. In other words, Kbuild handles hostprogs-m exactly like hostprogs-y. But only hostprogs-y is recommended to be used when no CONFIG symbols are involved.

5. Kbuild clean infrastructure

"make clean" deletes most generated files in the obj tree where the kernel is compiled. This includes generated files such as host programs. Kbuild knows targets listed in $(hostprogs-y), $(hostprogs-m), $(always), $(extra-y) and $(targets). They are all deleted during "make clean". Files matching the patterns "*.[oas]", "*.ko", plus some additional files generated by kbuild are deleted all over the kernel src tree when "make clean" is executed.

Additional files can be specified in kbuild makefiles by use of $(clean-files).

Example: #drivers/pci/Makefile

```
clean-files := devlist.h classlist.h
```

When executing "make clean", the two files "devlist.h classlist.h" will be deleted. Kbuild will assume files to be in same relative directory as the Makefile except if an absolute path is specified (path starting with '/').

To delete a directory hierarchy use:

Example: #scripts/package/Makefile

clean-dirs := $(objtree)/debian/

This will delete the directory debian, including all subdirectories. Kbuild will assume the directories to be in the same relative path as the Makefile if no absolute path is specified (path does not start with '/').

Usually kbuild descends down in subdirectories due to "obj-* := dir/", but in the architecture makefiles where the kbuild infrastructure is not sufficient this sometimes needs to be explicit.

Example: #arch/i386/boot/Makefile

subdir- := compressed/

The above assignment instructs kbuild to descend down in the directory compressed/ when "make clean" is executed.

To support the clean infrastructure in the Makefiles that builds the final bootimage there is an optional target named archclean:

Example: #arch/i386/Makefile

archclean: (Q)(MAKE) $(clean)=arch/i386/boot

When "make clean" is executed, make will descend down in arch/i386/boot, and clean as usual. The Makefile located in arch/i386/boot/ may use the subdir- trick to descend further down.

Note 1: arch/$(ARCH)/Makefile cannot use "subdir-", because that file is included in the top level makefile, and the kbuild infrastructure is not operational at that point.

Note 2: All directories listed in core-y, libs-y, drivers-y and net-y will be visited during "make clean".

6. Architecture Makefiles

The top level Makefile sets up the environment and does the preparation, before starting to descend down in the individual directories. The top level makefile contains the generic part, whereas arch/$(ARCH)/Makefile contains what is required to set up kbuild for said architecture. To do so, arch/$(ARCH)/Makefile sets up a number of variables and defines a few targets.

When kbuild executes, the following steps are followed (roughly):

1) Configuration of the kernel => produce .config

2) Store kernel version in include/linux/version.h

3) Symlink include/asm to include/asm-$(ARCH)

4) Updating all other prerequisites to the target prepare:

- Additional prerequisites are specified in arch/$(ARCH)/Makefile

5) Recursively descend down in all directories listed in init-* core* drivers-* net-* libs-* and build all targets.

- The values of the above variables are expanded in arch/$(ARCH)/Makefile.

6) All object files are then linked and the resulting file vmlinux is located at the root of the obj tree.

The very first objects linked are listed in head-y, assigned by

arch/$(ARCH)/Makefile.

7) Finally, the architecture-specific part does any required post processing and builds the final bootimage.

- This includes building boot records

- Preparing initrd images and the like

6.1. Set variables to tweak the build to the architecture

LDFLAGS Generic $(LD) options

Flags used for all invocations of the linker.

Often specifying the emulation is sufficient.

Example: #arch/s390/Makefile

LDFLAGS := -m elf_s390 Note: ldflags-y can be used to further customise the flags used. See chapter 3.7.

LDFLAGS_MODULE Options for $(LD) when linking modules

LDFLAGS_MODULE is used to set specific flags for $(LD) when

linking the .ko files used for modules.

Default is "-r", for relocatable output.

LDFLAGS_vmlinux Options for $(LD) when linking vmlinux

LDFLAGS_vmlinux is used to specify additional flags to pass to

the linker when linking the final vmlinux image.

LDFLAGS_vmlinux uses the LDFLAGS_$@ support.

Example: #arch/i386/Makefile

LDFLAGS_vmlinux := -e stext

OBJCOPYFLAGS objcopy flags

When $(call if_changed,objcopy) is used to translate a .o file,

the flags specified in OBJCOPYFLAGS will be used.

$(call if_changed,objcopy) is often used to generate raw binaries on vmlinux.

Example: #arch/s390/Makefile

OBJCOPYFLAGS := -O binary

#arch/s390/boot/Makefile

$(obj)/image: vmlinux FORCE $(call if_changed,objcopy)

In this example, the binary $(obj)/image is a binary version of

vmlinux. The usage of $(call if_changed,xxx) will be described later.

KBUILD_AFLAGS $(AS) assembler flags

Default value - see top level Makefile

Append or modify as required per architecture.

Example: #arch/sparc64/Makefile

KBUILD_AFLAGS += -m64 -mcpu=ultrasparc

KBUILD_CFLAGS $(CC) compiler flags

Default value - see top level Makefile

Append or modify as required per architecture.

Often, the KBUILD_CFLAGS variable depends on the configuration.

Example: #arch/i386/Makefile

cflags-$(CONFIG_M386) += -march=i386

KBUILD_CFLAGS += $(cflags-y)

Many arch Makefiles dynamically run the target C compiler to

probe supported options:

#arch/i386/Makefile

...

cflags-$(CONFIG_MPENTIUMII) += $(call cc-option,\

-march=pentium2,-march=i686) ...

Disable unit-at-a-time mode ...

KBUILD_CFLAGS += $(call cc-option,-fno-unit-at-a-time)

...

The first example utilises the trick that a config option expands

to 'y' when selected.

CFLAGS_KERNEL $(CC) options specific for built-in

$(CFLAGS_KERNEL) contains extra C compiler flags used to compile resident kernel code.

CFLAGS_MODULE $(CC) options specific for modules

$(CFLAGS_MODULE) contains extra C compiler flags used to compile code for loadable kernel modules.

6.2. Add prerequisites to archprepare:

The archprepare: rule is used to list prerequisites that need to be built before starting to descend down in the subdirectories.

This is usually used for header files containing assembler constants.

Example:

#arch/arm/Makefile

archprepare: maketools

In this example, the file target maketools will be processed

before descending down in the subdirectories.

See also chapter XXX-TODO that describe how kbuild supports

generating offset header files.

6.3. List directories to visit when descending

An arch Makefile cooperates with the top Makefile to define variables which specify how to build the vmlinux file. Note that there is no corresponding arch-specific section for modules; the module-building machinery is all architecture-independent.

head-y, init-y, core-y, libs-y, drivers-y, net-y

$(head-y) lists objects to be linked first in vmlinux.

$(libs-y) lists directories where a lib.a archive can be located.

The rest list directories where a built-in.o object file can be

located.

$(init-y) objects will be located after $(head-y).

Then the rest follows in this order:

$(core-y), $(libs-y), $(drivers-y) and $(net-y).

The top level Makefile defines values for all generic directories,

and arch/$(ARCH)/Makefile only adds architecture-specific directories.

Example: #arch/sparc64/Makefile

```
core-y += arch/sparc64/kernel/
libs-y += arch/sparc64/prom/ arch/sparc64/lib/
drivers-$(CONFIG_OPROFILE)    += arch/sparc64/oprofile/
```

6.4. Architecture-specific boot images

An arch Makefile specifies goals that take the vmlinux file, compress it, wrap it in bootstrapping code, and copy the resulting files somewhere. This includes various kinds of installation commands.

The actual goals are not standardized across architectures.

It is common to locate any additional processing in a boot/

directory below arch/$(ARCH)/.

Kbuild does not provide any smart way to support building a

target specified in boot/. Therefore arch/$(ARCH)/Makefile shall

call make manually to build a target in boot/.

The recommended approach is to include shortcuts in

arch/$(ARCH)/Makefile, and use the full path when calling down

into the arch/$(ARCH)/boot/Makefile.

Example: #arch/i386/Makefile

```
boot := arch/i386/boot
bzImage: vmlinux $(Q)$(MAKE) $(build)=$(boot) $(boot)/$@
```

"(Q)(MAKE) $(build)=<dir>" is the recommended way to invoke

make in a subdirectory.

There are no rules for naming architecture-specific targets,

but executing "make help" will list all relevant targets.

To support this, $(archhelp) must be defined.

Example: #arch/i386/Makefile

define archhelp echo '* bzImage - Image (arch/$(ARCH)/boot/bzImage)'

endif

When make is executed without arguments, the first goal encountered will be built. In the top level Makefile the first goal present

is all:.

An architecture shall always, per default, build a bootable image.

In "make help", the default goal is highlighted with a '*'.

Add a new prerequisite to all: to select a default goal different

from vmlinux.

Example: #arch/i386/Makefile

all: bzImage

When "make" is executed without arguments, bzImage will be built.

6.5. Building non-kbuild targets

extra-y

extra-y specify additional targets created in the current

directory, in addition to any targets specified by obj-*.

Listing all targets in extra-y is required for two purposes: 1) Enable kbuild to check changes in command lines

- When $(call if_changed,xxx) is used 2) kbuild knows what files to delete during "make clean"

Example: #arch/i386/kernel/Makefile

extra-y := head.o init_task.o

In this example, extra-y is used to list object files that

shall be built, but shall not be linked as part of built-in.o.

6.6. Commands useful for building a boot image

Kbuild provides a few macros that are useful when building a

boot image.

if_changed

if_changed is the infrastructure used for the following commands.

Usage: target: source(s) FORCE $(call if_changed,ld/objcopy/gzip)

When the rule is evaluated, it is checked to see if any files

need an update, or the command line has changed since the last invocation. The latter will force a rebuild if any options

to the executable have changed.

Any target that utilises if_changed must be listed in $(targets), otherwise the command line check will fail, and the target will

always be built.

Assignments to $(targets) are without $(obj)/ prefix.

if_changed may be used in conjunction with custom commands as

defined in 6.7 "Custom kbuild commands".

Note: It is a typical mistake to forget the FORCE prerequisite.

Another common pitfall is that whitespace is sometimes

significant; for instance, the below will fail (note the extra space after the comma):

target: source(s) FORCE #WRONG!# $(call if_changed, ld/objcopy/gzip)

ld Link target. Often, LDFLAGS_$@ is used to set specific options to ld.

objcopy Copy binary. Uses OBJCOPYFLAGS usually specified in

arch/$(ARCH)/Makefile.

OBJCOPYFLAGS_$@ may be used to set additional options.

gzip Compress target. Use maximum compression to compress target.

Example: #arch/i386/boot/Makefile

```
LDFLAGS_bootsect := -Ttext 0x0 -s --oformat binary
LDFLAGS_setup    := -Ttext 0x0 -s --oformat binary -e begtext
targets += setup setup.o bootsect bootsect.o
$(obj)/setup $(obj)/bootsect: %: %.o FORCE $(call if_changed,ld)
```

In this example, there are two possible targets, requiring different options to the linker. The linker options are specified using the LDFLAGS_$@ syntax - one for each potential target.

$(targets) are assigned all potential targets, by which kbuild knows the targets and will:

1) check for commandline changes 2) delete target during make clean

The ": %: %.o" part of the prerequisite is a shorthand that

free us from listing the setup.o and bootsect.o files.

Note: It is a common mistake to forget the "target :=" assignment, resulting in the target file being recompiled for no

obvious reason.

6.7. Custom kbuild commands

When kbuild is executing with KBUILD_VERBOSE=0, then only a shorthand of a command is normally displayed.

To enable this behaviour for custom commands kbuild requires

two variables to be set:

quiet_cmd_<command>- what shall be echoed cmd_<command> - the command to execute

Example: #

quiet_cmd_image = BUILD $@ cmd_image = $(obj)/tools/build $(BUILDFLAGS) \ $(obj)/vmlinux.bin > $@

targets += bzImage

$(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE $(call if_changed,image)

@echo 'Kernel: $@ is ready'

When updating the $(obj)/bzImage target, the line

BUILD arch/i386/boot/bzImage

will be displayed with "make KBUILD_VERBOSE=0".

6.8. Preprocessing linker scripts

When the vmlinux image is built, the linker script

arch/$(ARCH)/kernel/vmlinux.lds is used.

The script is a preprocessed variant of the file vmlinux.lds.S

located in the same directory.

kbuild knows .lds files and includes a rule *lds.S -> *lds.

Example: #arch/i386/kernel/Makefile

always := vmlinux.lds

#Makefile

export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)

The assignment to $(always) is used to tell kbuild to build the

target vmlinux.lds.

The assignment to $(CPPFLAGS_vmlinux.lds) tells kbuild to use the specified options when building the target vmlinux.lds.

When building the *.lds target, kbuild uses the variables: KBUILD_CPPFLAGS : Set in top-level Makefile

cppflags-y : May be set in the kbuild makefile

CPPFLAGS_$(@F) : Target specific flags. Note that the full filename is used in this assignment.

The kbuild infrastructure for *lds file are used in several architecture-specific files.

7. Kbuild syntax for exported headers

The kernel include a set of headers that is exported to userspace. Many headers can be exported as-is but other headers requires a minimal pre-processing before they are ready for user-space. The pre-processing does:

- drop kernel specific annotations
- drop include of compiler.h
- drop all sections that is kernel internat (guarded by ifdef __KERNEL__)

Each relevant directory contain a file name "Kbuild" which specify the headers to be exported. See subsequent chapter for the syntax of the Kbuild file.

7.1. header-y

header-y specify header files to be exported.

Example: #include/linux/Kbuild

header-y += usb/

header-y += aio_abi.h

The convention is to list one file per line and

preferably in alphabetic order.

header-y also specify which subdirectories to visit.

A subdirectory is identified by a trailing '/' which

can be seen in the example above for the usb subdirectory.

Subdirectories are visited before their parent directories.

7.2. objhdr-y

objhdr-y specifies generated files to be exported.

Generated files are special as they need to be looked

up in another directory when doing 'make O=...' builds.

Example: #include/linux/Kbuild

objhdr-y += version.h

7.3. destination-y

When an architecture have a set of exported headers that needs to be exported to a different directory destination-y is used.

destination-y specify the destination directory for all exported headers in the file where it is present.

Example: #arch/xtensa/platforms/s6105/include/platform/Kbuild

destination-y := include/linux

In the example above all exported headers in the Kbuild file

will be located in the directory "include/linux" when exported.

7.4. unifdef-y (deprecated)

unifdef-y is deprecated. A direct replacement is header-y.

8. Kbuild Variables

The top Makefile exports the following variables:

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

These variables define the current kernel version. A few arch Makefiles actually use these values directly; they should use

$(KERNELRELEASE) instead.

$(VERSION), $(PATCHLEVEL), and $(SUBLEVEL) define the basic

three-part version number, such as "2", "4", and "0". These three values are always numeric.

$(EXTRAVERSION) defines an even tinier sublevel for pre-patches

or additional patches. It is usually some non-numeric string

such as "-pre4", and is often blank.

KERNELRELEASE

$(KERNELRELEASE) is a single string such as "2.4.0-pre4", suitable

for constructing installation directory names or showing in

version strings. Some arch Makefiles use it for this purpose.

ARCH

This variable defines the target architecture, such as "i386",

"arm", or "sparc". Some kbuild Makefiles test $(ARCH) to

determine which files to compile.

By default, the top Makefile sets $(ARCH) to be the same as the

host system architecture. For a cross build, a user may

override the value of $(ARCH) on the command line:

```
make ARCH=m68k ...
```

INSTALL_PATH

This variable defines a place for the arch Makefiles to install

the resident kernel image and System.map file.

Use this for architecture-specific install targets.

INSTALL_MOD_PATH, MODLIB

$(INSTALL_MOD_PATH) specifies a prefix to $(MODLIB) for module installation. This

variable is not defined in the Makefile but

may be passed in by the user if desired.

$(MODLIB) specifies the directory for module installation.

The top Makefile defines $(MODLIB) to

$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE). The user may override this value on the command line if desired.

INSTALL_MOD_STRIP

If this variable is specified, will cause modules to be stripped

after they are installed. If INSTALL_MOD_STRIP is '1', then the default option --strip-debug will be used. Otherwise,

INSTALL_MOD_STRIP will used as the option(s) to the strip command.

9. Makefile language

The kernel Makefiles are designed to be run with GNU Make. The Makefiles use only the documented features of GNU Make, but they do use many GNU extensions.

GNU Make supports elementary list-processing functions. The kernel Makefiles use a novel style of list building and manipulation with few "if" statements.

GNU Make has two assignment operators, ":=" and "=". ":=" performs immediate evaluation of the right-hand side and stores an actual string into the left-hand side. "=" is like a formula definition; it stores the right-hand side in an unevaluated form and then evaluates this form each time the left-hand side is used.

There are some cases where "=" is appropriate. Usually, though, ":=" is the right choice.

10. Credits

Original version made by Michael Elizabeth Chastain, <mailto:mec@shout.net> Updates by Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de> Updates by Sam Ravnborg <sam@ravnborg.org> Language QA by Jan Engelhardt <jengelh@gmx.de>

11. TODO

- Describe how kbuild supports shipped files with _shipped.

- Generating offset header files.

- Add more variables to section 7?