Linux Kernel Makefiles

This document describes the Linux kernel Makefiles.

1 Overview

The Makefiles have five parts:

```
Makefile                    the top Makefile.
.config                     the kernel configuration file.
arch/$(SRCARCH)/Makefile    the arch Makefile.
scripts/Makefile.*          common rules etc. for all kbuild Makefiles.
kbuild Makefiles            exist in every subdirectory
```

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/$(SRCARCH)/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 $(obj-y) files. It then calls "$(AR) rcSTP" to merge these files into one built-in.a file. This is a thin archive without a symbol table. It will be later linked into vmlinux by scripts/link-vmlinux.sh

The order of files in $(obj-y) is significant. Duplicates in the lists are allowed: the first instance will be linked into built-in.a 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-$(CONFIG_ISDN_I4L)         += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

3.3 Loadable module goals - obj-m

$(obj-m) specifies 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; however, kbuild needs to know which object files you want to build your module from, so you have to tell it by setting a $(<module_name>-y) variable.

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_I4L) += isdn.o
isdn-y := 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 $(isdn-y) and then run "$(LD) -r" on the list of these files to generate isdn.o.

Due to kbuild recognizing $(<module_name>-y) for composite objects, you can use the value of a *CONFIG_* symbol to optionally include an object file as part of a composite object.

Example:

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o dir.o file.o ialloc.o inode.o ioctl.o \
          namei.o super.o symlink.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o xattr_user.o \
                                xattr_trusted.o
```

In this example, xattr.o, xattr_user.o and xattr_trusted.o are 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.a, as you would expect.

3.5 Library file goals - lib-y

Objects listed with obj-* are used for modules, or combined in a built-in.a 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.a and a lib.a file.

Example:

```
#arch/x86/lib/Makefile
lib-y    := delay.o
```

This will create a library lib.a based on 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 "7.4 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 uses this information not only to decide that it needs to visit the directory, but also to decide whether or not to link objects from the directory into vmlinux.

When Kbuild descends into the directory with 'y', all built-in objects from that directory are combined into the built-in.a, which will be eventually linked into vmlinux.

When Kbuild descends into the directory with 'm', in contrast, nothing from that directory will be linked into vmlinux. If the Makefile in that directory specifies obj-y, those objects will be left orphan. It is very likely a bug of the Makefile or of dependencies in Kconfig.

Kbuild also supports dedicated syntax, subdir-y and subdir-m, for descending into subdirectories. It is a good fit when you know they do not contain kernel-space objects at all. A typical usage is to let Kbuild descend into subdirectories to build tools.

Examples:

```
# scripts/Makefile
subdir-$(CONFIG_GCC_PLUGINS) += gcc-plugins
subdir-$(CONFIG_MODVERSIONS) += genksyms
subdir-$(CONFIG_SECURITY_SELINUX) += selinux
```

Unlike obj-y/m, subdir-y/m does not need the trailing slash since this syntax is always used for directories.

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 Non-builtin vmlinux targets - extra-y

extra-y specifies targets which are needed for building vmlinux, but not combined into built-in.a.

Examples are:

1. head objects

   Some objects must be placed at the head of vmlinux. They are directly linked to vmlinux without going through built-in.a A typical use-case is an object that contains the entry point.

   arch/$(SRCARCH)/Makefile should specify such objects as head-y.

   Discussion:
   Given that we can control the section order in the linker script, why do we need head-y?

2. vmlinux linker script

   The linker script for vmlinux is located at arch/$(SRCARCH)/kernel/vmlinux.lds

Example:

```
# arch/x86/kernel/Makefile
extra-y := head_$(BITS).o
extra-y += head$(BITS).o
extra-y += ebda.o
```

```
extra-y += platform-quirks.o
extra-y += vmlinux.lds
```

$(extra-y) should only contain targets needed for vmlinux.

Kbuild skips extra-y when vmlinux is apparently not a final goal. (e.g. 'make modules', or building external modules)

If you intend to build targets unconditionally, always-y (explained in the next section) is the correct syntax to use.

3.8 Always built goals - always-y

always-y specifies targets which are literally always built when Kbuild visits the Makefile.

Example::

./Kbuild offsets-file := include/generated/asm-offsets.h always-y += $(offsets-file)

3.9 Compilation flags

ccflags-y, asflags-y and ldflags-y

These three flags apply only to the kbuild makefile in which they are assigned. They are used for all the normal cc, as and ld invocations happening during a recursive build. Note: Flags with the same behaviour were previously named: EXTRA_CFLAGS, EXTRA_AFLAGS and EXTRA_LDFLAGS. They are still supported but their usage is deprecated.

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

Example:

```
# drivers/acpi/acpica/Makefile
ccflags-y                       := -Os -D_LINUX -DBUILDING_ACPICA
ccflags-$(CONFIG_ACPI_DEBUG)    += -DACPI_DEBUG_OUTPUT
```

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 specifies assembler options.

Example:

```
#arch/sparc/kernel/Makefile
asflags-y := -ansi
```

ldflags-y specifies options for linking with $(LD).

Example:

```
#arch/cris/boot/compressed/Makefile
ldflags-y += -T $(srctree)/$(src)/decompress_$(arch-y).lds
```

subdir-ccflags-y, subdir-asflags-y

The two flags listed above are similar to ccflags-y and asflags-y. The difference is that the subdir- variants have effect for the kbuild file where they 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
```

ccflags-remove-y, asflags-remove-y

These flags are used to remove particular flags for the compiler, assembler invocations.

Example:

```
ccflags-remove-$(CONFIG_MCOUNT) += -pg
```

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.

CFLAGS_$@ has the higher priority than ccflags-remove-y; CFLAGS_$@ can re-add compiler flags that were removed by ccflags-remove-y.

Example:

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
```

This line specify compilation flags for aha152x.o.

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

AFLAGS_$@ has the higher priority than asflags-remove-y; AFLAGS_$@ can re-add assembler flags that were removed by asflags-remove-y.

Example:

```
# arch/arm/kernel/Makefile
AFLAGS_head.o        := -DTEXT_OFFSET=$(TEXT_OFFSET)
AFLAGS_crunch-bits.o := -Wa,-mcpu=ep9312
AFLAGS_iwmmxt.o      := -Wa,-mcpu=iwmmxt
```

3.10 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.

3.11 Custom Rules

Custom 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 custom rules to prepare boot images etc.

Custom rules are written as normal Make rules. Kbuild is not executing in the directory where the Makefile is located, so all custom rules shall use a relative path to prerequisite files and target files.

Two variables are used when defining custom 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 custom 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 defines $(kecho) which will echo out the text following $(kecho) to stdout except if "make -s" is used.

Example:

```
# arch/arm/Makefile
$(BOOT_TARGETS): vmlinux
        $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
        @$(kecho) '  Kernel: $(boot)/$@ is ready'
```

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:

```
# lib/Makefile
quiet_cmd_crc32 = GEN     $@
      cmd_crc32 = $< > $@
```

```
$(obj)/crc32table.h: $(obj)/gen_crc32table
        $(call cmd,crc32)
```

When updating the $(obj)/crc32table.h target, the line:

GEN lib/crc32table.h

will be displayed with "make KBUILD_VERBOSE=0".

3.12 Command change detection

When the rule is evaluated, timestamps are compared between the target and its prerequisite files. GNU Make updates the target when any of the prerequisites is newer than that.

The target should be rebuilt also when the command line has changed since the last invocation. This is not supported by Make itself, so Kbuild achieves this by a kind of meta-programming.

if_changed is the macro used for this purpose, in the following form:

```
quiet_cmd_<command> = ...
      cmd_<command> = ...

<target>: <source(s)> FORCE
        $(call if_changed,<command>)
```

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

If the target is already listed in the recognized syntax such as obj-y/m, lib-y/m, extra-y/m, always-y/m, hostprogs, userprogs, Kbuild automatically adds it to $(targets). Otherwise, the target must be explicitly added to $(targets).

Assignments to $(targets) are without $(obj)/ prefix. if_changed may be used in conjunction with custom rules as defined in "3.11 Custom Rules".

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, objcopy)

Note:

if_changed should not be used more than once per target. It stores the executed command in a corresponding .cmd file and multiple calls would result in overwrites and unwanted results when the target is up to date and only the tests on changed commands trigger execution of commands.

3.13 $(CC) support functions

The kernel may be built with several different versions of $(CC), each supporting a unique set of features and options. kbuild provides 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.

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 assembler options

cc-option

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

Example:

```
#arch/x86/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-disable-warning

cc-disable-warning checks if gcc supports a given warning and returns the commandline switch to disable it. This special function is needed, because gcc 4.4 and later accept any unknown -Wno-* option and only warn about it if there is another warning in the source file.

Example:

```
KBUILD_CFLAGS += $(call cc-disable-warning, unused-but-set-variable)
```

In the above example, -Wno-unused-but-set-variable will be added to KBUILD_CFLAGS only if gcc really accepts it.

cc-ifversion

cc-ifversion tests the version of $(CC) and equals the fourth parameter if version expression is true, or the fifth (if given) if the version expression is false.

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-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
```

3.14 $(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 ld-option, -X)
```

3.15 Script invocation

Make rules may invoke scripts to build the kernel. The rules shall always provide the appropriate interpreter to execute the script. They shall not rely on the execute bits being set, and shall not invoke the script directly. For the convenience of manual script invocation, such as invoking ./scripts/checkpatch.pl, it is recommended to set execute bits on the scripts nonetheless.

Kbuild provides variables $(CONFIG_SHELL), $(AWK), $(PERL), and $(PYTHON3) to refer to interpreters for the respective scripts.

Example:

```
#Makefile
cmd_depmod = $(CONFIG_SHELL) $(srctree)/scripts/depmod.sh $(DEPMOD) \
             $(KERNELRELEASE)
```

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".

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-y". 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 := 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     := 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 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     := 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 of a mixture of .c and .cc files, then an additional line can be used to identify this.

Example:

```
#scripts/kconfig/Makefile
hostprogs     := qconf
```

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

4.4 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 $(KBUILD_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
HOSTLDLIBS_qconf := -L$(QTDIR)/lib
```

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

4.5 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 custom rule.

Example:

```
#drivers/pci/Makefile
hostprogs := 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 custom rules must be prefixed with $(obj).

2. Use always-y

When there is no suitable custom rule, and the host program shall be built when a makefile is entered, the always-y variable shall be used.

Example:

```
#scripts/lxdialog/Makefile
hostprogs     := lxdialog
always-y      := $(hostprogs)
```

Kbuild provides the following shorthand for this:

```
hostprogs-always-y := lxdialog
```

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

5 Userspace Program support

Just like host programs, Kbuild also supports building userspace executables for the target architecture (i.e. the same architecture as you are building the kernel for).

The syntax is quite similar. The difference is to use "userprogs" instead of "hostprogs".

5.1 Simple Userspace Program

The following line tells kbuild that the program bpf-direct shall be built for the target architecture.

Example:

```
userprogs := bpf-direct
```

Kbuild assumes in the above example that bpf-direct is made from a single C source file named bpf-direct.c located in the

same directory as the Makefile.

5.2 Composite Userspace Programs

Userspace programs can be made up based on composite objects. The syntax used to define composite objects for userspace programs is similar to the syntax used for kernel objects. $(<executable>-objs) lists all objects used to link the final executable.

Example:

```
#samples/seccomp/Makefile
userprogs      := bpf-fancy
bpf-fancy-objs := bpf-fancy.o bpf-helper.o
```

Objects with extension .o are compiled from the corresponding .c files. In the above example, bpf-fancy.c is compiled to bpf-fancy.o and bpf-helper.c is compiled to bpf-helper.o.

Finally, the two .o files are linked to the executable, bpf-fancy. Note: The syntax <executable>-y is not permitted for userspace programs.

5.3 Controlling compiler options for userspace programs

When compiling userspace programs, it is possible to set specific flags. The programs will always be compiled utilising $(CC) passed the options specified in $(KBUILD_USERCFLAGS). To set flags that will take effect for all userspace programs created in that Makefile, use the variable userccflags.

Example:

```
# samples/seccomp/Makefile
userccflags += -I usr/include
```

To set specific flags for a single file the following construction is used:

Example:

```
bpf-helper-userccflags += -I user/include
```

It is also possible to specify additional options to the linker.

Example:

```
# net/bpfilter/Makefile
bpfilter_umh-userldflags += -static
```

When linking bpfilter_umh, it will be passed the extra option -static.

From command line, :ref:`USERCFLAGS and USERLDFLAGS <userkbuildflags>` will also be used.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kbuild\(linux-master)(Documentation)(kbuild)makefiles.rst`, **line 985**); *backlink*
>
> Unknown interpreted text role "ref".

5.4 When userspace programs are actually built

Kbuild builds userspace programs only when told to do so. There are two ways to do this.

1. Add it as the prerequisite of another file

Example:

```
#net/bpfilter/Makefile
userprogs := bpfilter_umh
$(obj)/bpfilter_umh_blob.o: $(obj)/bpfilter_umh
```

$(obj)/bpfilter_umh is built before $(obj)/bpfilter_umh_blob.o

2. Use always-y

Example:

```
userprogs := binderfs_example
always-y := $(userprogs)
```

Kbuild provides the following shorthand for this:

```
userprogs-always-y := binderfs_example
```

This will tell Kbuild to build binderfs_example when it visits this Makefile.

6 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), $(always-y), $(always-m), $(always-), $(extra-y), $(extra-) 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 source tree when "make clean" is executed.

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

Example:

```
#lib/Makefile
clean-files := crc32table.h
```

When executing "make clean", the file "crc32table.h" will be deleted. Kbuild will assume files to be in the same relative directory as the Makefile, except if prefixed with $(objtree).

To exclude certain files or directories from make clean, use the $(no-clean-files) variable.

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/x86/boot/Makefile
subdir- := compressed
```

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

Note 1: arch/$(SRCARCH)/Makefile cannot use "subdir-", because that file is included in the top level makefile. Instead, arch/$(SRCARCH)/Kbuild can use "subdir-".

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

7 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/$(SRCARCH)/Makefile contains what is required to set up kbuild for said architecture. To do so, arch/$(SRCARCH)/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. Updating all other prerequisites to the target prepare: - Additional prerequisites are specified in arch/$(SRCARCH)/Makefile
4. Recursively descend down in all directories listed in init-* core* drivers-* net-* libs-* and build all targets. - The values of the above variables are expanded in arch/$(SRCARCH)/Makefile.
5. 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/$(SRCARCH)/Makefile.
6. 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

7.1 Set variables to tweak the build to the architecture

KBUILD_LDFLAGS

Generic $(LD) options

Flags used for all invocations of the linker. Often specifying the emulation is sufficient.

Example:

```
#arch/s390/Makefile
KBUILD_LDFLAGS         := -m elf_s390
```

Note: ldflags-y can be used to further customise the flags used. See section 3.7.

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/x86/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

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/x86/boot/compressed/Makefile
cflags-$(CONFIG_X86_32) := -march=i386
cflags-$(CONFIG_X86_64) := -mcmodel=small
KBUILD_CFLAGS += $(cflags-y)
```

Many arch Makefiles dynamically run the target C compiler to probe supported options:

```
#arch/x86/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.

KBUILD_AFLAGS_KERNEL

Assembler options specific for built-in

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

KBUILD_AFLAGS_MODULE

Assembler options specific for modules

$(KBUILD_AFLAGS_MODULE) is used to add arch-specific options that are used for assembler.

From commandline AFLAGS_MODULE shall be used (see kbuild.rst).

KBUILD_CFLAGS_KERNEL

$(CC) options specific for built-in

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

KBUILD_CFLAGS_MODULE

Options for $(CC) when building modules

$(KBUILD_CFLAGS_MODULE) is used to add arch-specific options that are used for $(CC). From commandline CFLAGS_MODULE shall be used (see kbuild.rst).

KBUILD_LDFLAGS_MODULE

Options for $(LD) when linking modules

$(KBUILD_LDFLAGS_MODULE) is used to add arch-specific options used when linking modules. This is often a linker script.

From commandline LDFLAGS_MODULE shall be used (see kbuild.rst).

KBUILD_LDS

The linker script with full path. Assigned by the top-level Makefile.

KBUILD_LDS_MODULE

The module linker script with full path. Assigned by the top-level Makefile and additionally by the arch Makefile.

KBUILD_VMLINUX_OBJS

All object files for vmlinux. They are linked to vmlinux in the same order as listed in KBUILD_VMLINUX_OBJS.

KBUILD_VMLINUX_LIBS

All .a "lib" files for vmlinux. KBUILD_VMLINUX_OBJS and KBUILD_VMLINUX_LIBS together specify all the object files used to link vmlinux.

7.2 Add prerequisites to archheaders

The archheaders: rule is used to generate header files that may be installed into user space by "make header_install".

It is run before "make archprepare" when run on the architecture itself.

7.3 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 describes how kbuild supports generating offset header files.

7.4 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, core-y, libs-y, drivers-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.a object file can be located.

Then the rest follows in this order:

$(core-y), $(libs-y), $(drivers-y)

The top level Makefile defines values for all generic directories, and arch/$(SRCARCH)/Makefile only adds architecture-specific directories.

Example:

```
# arch/sparc/Makefile
core-y                  += arch/sparc/

libs-y                  += arch/sparc/prom/
```

```
libs-y                  += arch/sparc/lib/

drivers-$(CONFIG_PM) += arch/sparc/power/
```

7.5 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/$(SRCARCH)/.

Kbuild does not provide any smart way to support building a target specified in boot/. Therefore arch/$(SRCARCH)/Makefile shall call make manually to build a target in boot/.

The recommended approach is to include shortcuts in arch/$(SRCARCH)/Makefile, and use the full path when calling down into the arch/$(SRCARCH)/boot/Makefile.

Example:

```
#arch/x86/Makefile
boot := arch/x86/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/x86/Makefile
define archhelp
  echo  '* bzImage      - Compressed kernel image (arch/x86/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/x86/Makefile
all: bzImage
```

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

7.7 Commands useful for building a boot image

Kbuild provides a few macros that are useful when building a boot image.

ld

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

Example:

```
#arch/x86/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 frees us from listing the setup.o and bootsect.o files.

Note:

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

objcopy

Copy binary. Uses OBJCOPYFLAGS usually specified in arch/$(SRCARCH)/Makefile. OBJCOPYFLAGS_$@ may be used to set additional options.

gzip

Compress target. Use maximum compression to compress target.

Example:

```
#arch/x86/boot/compressed/Makefile
$(obj)/vmlinux.bin.gz: $(vmlinux.bin.all-y) FORCE
        $(call if_changed,gzip)
```

dtc

Create flattened device tree blob object suitable for linking into vmlinux. Device tree blobs linked into vmlinux are placed in an init section in the image. Platform code *must* copy the blob to non-init memory prior to calling unflatten_device_tree().

To use this command, simply add **.dtb* into obj-y or targets, or make some other target depend on *%.dtb*

A central rule exists to create *$(obj)/%.dtb* from *$(src)/%.dts*; architecture Makefiles do no need to explicitly write out that rule.

Example:

```
targets += $(dtb-y)
DTC_FLAGS ?= -p 1024
```

7.9 Preprocessing linker scripts

When the vmlinux image is built, the linker script arch/$(SRCARCH)/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/x86/kernel/Makefile
extra-y := vmlinux.lds
```

The assignment to extra-y 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* files is used in several architecture-specific files.

7.10 Generic header files

The directory include/asm-generic contains the header files that may be shared between individual architectures. The recommended approach how to use a generic header file is to list the file in the Kbuild file. See "8.2 generic-y" for further info on syntax etc.

7.11 Post-link pass

If the file arch/xxx/Makefile.postlink exists, this makefile will be invoked for post-link objects (vmlinux and modules.ko) for architectures to run post-link passes on. Must also handle the clean target.

This pass runs after kallsyms generation. If the architecture needs to modify symbol locations, rather than manipulate the kallsyms, it may be easier to add another postlink target for .tmp_vmlinux? targets to be called from link-vmlinux.sh.

For example, powerpc uses this to check relocation sanity of the linked vmlinux file.

8 Kbuild syntax for exported headers

The kernel includes a set of headers that is exported to userspace. Many headers can be exported as-is but other headers require 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 are kernel internal (guarded by *ifdef __KERNEL__*)

All headers under include/uapi/, include/generated/uapi/, arch/<arch>/include/uapi/ and arch/<arch>/include/generated/uapi/ are exported.

A Kbuild file may be defined under arch/<arch>/include/uapi/asm/ and arch/<arch>/include/asm/ to list asm files coming from asm-generic. See subsequent chapter for the syntax of the Kbuild file.

8.1 no-export-headers

no-export-headers is essentially used by include/uapi/linux/Kbuild to avoid exporting specific headers (e.g. kvm.h) on architectures that do not support it. It should be avoided as much as possible.

8.2 generic-y

If an architecture uses a verbatim copy of a header from include/asm-generic then this is listed in the file arch/$(SRCARCH)/include/asm/Kbuild like this:

Example:

```
#arch/x86/include/asm/Kbuild
generic-y += termios.h
generic-y += rtc.h
```

During the prepare phase of the build a wrapper include file is generated in the directory:

```
arch/$(SRCARCH)/include/generated/asm
```

When a header is exported where the architecture uses the generic header a similar wrapper is generated as part of the set of exported headers in the directory:

```
usr/include/asm
```

The generated wrapper will in both cases look like the following:

Example: termios.h:

```
#include <asm-generic/termios.h>
```

8.3 generated-y

If an architecture generates other header files alongside generic-y wrappers, generated-y specifies them.

This prevents them being treated as stale asm-generic wrappers and removed.

Example:

```
#arch/x86/include/asm/Kbuild
generated-y += syscalls_32.h
```

8.4 mandatory-y

mandatory-y is essentially used by include/(uapi/)asm-generic/Kbuild to define the minimum set of ASM headers that all architectures must have.

This works like optional generic-y. If a mandatory header is missing in arch/$(SRCARCH)/include/(uapi/)/asm, Kbuild will automatically generate a wrapper of the asm-generic one.

9 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 ...
```

SRCARCH

This variable specifies the directory in arch/ to build.

ARCH and SRCARCH may not necessarily match. A couple of arch directories are biarch, that is, a single *arch/*/* directory supports both 32-bit and 64-bit.

For example, you can pass in ARCH=i386, ARCH=x86_64, or ARCH=x86. For all of them, SRCARCH=x86 because arch/x86/ supports both i386 and x86_64.

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, it 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, the INSTALL_MOD_STRIP value will be used as the option(s) to the strip command.

10 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.

11 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>

12 TODO

- Describe how kbuild supports shipped files with _shipped.
- Generating offset header files.
- Add more variables to chapters 7 or 9?