REF:

https://opensource.com/article/18/10/kbuild-and-kconfig

See also:

https://stackoverflow.com/questions/18841101/adding-support-for-menuconfig-kconfig-in-my-project

# Exploring the Linux kernel: The secrets of Kconfig/kbuild

## Dive into understanding how the Linux config/build system works.

11 Oct 2018 [Cao Jin](https://opensource.com/users/pinocchio)



The Linux kernel config/build system, also known as Kconfig/kbuild, has been around for a long time, ever since the Linux kernel code migrated to Git. As supporting infrastructure, however, it is seldom in the spotlight; even kernel developers who use it in their daily work never really think about it.

To explore how the Linux kernel is compiled, this article will dive into the Kconfig/kbuild internal process, explain how the .config file and the vmlinux/bzImage files are produced, and introduce a smart trick for dependency tracking.

## Kconfig

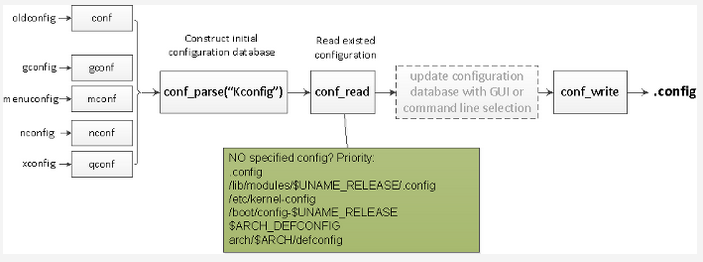
The first step in building a kernel is always configuration. Kconfig helps make the Linux kernel highly modular and customizable. Kconfig offers the user many config targets:

|  |  |
| --- | --- |
| config | Update current config utilizing a line-oriented program |
| nconfig | Update current config utilizing a ncurses menu-based program |
| menuconfig | Update current config utilizing a menu-based program |
| xconfig | Update current config utilizing a Qt-based frontend |
| gconfig | Update current config utilizing a GTK+ based frontend |
| oldconfig | Update current config utilizing a provided .config as base |
| localmodconfig | Update current config disabling modules not loaded |
| localyesconfig | Update current config converting local mods to core |
| defconfig | New config with default from Arch-supplied defconfig |
| savedefconfig | Save current config as ./defconfig (minimal config) |
| allnoconfig | New config where all options are answered with 'no' |
| allyesconfig | New config where all options are accepted with 'yes' |
| allmodconfig | New config selecting modules when possible |
| alldefconfig | New config with all symbols set to default |
| randconfig | New config with a random answer to all options |
| listnewconfig | List new options |
| olddefconfig | Same as oldconfig but sets new symbols to their default value without prompting |
| kvmconfig | Enable additional options for KVM guest kernel support |
| xenconfig | Enable additional options for xen dom0 and guest kernel support |
| tinyconfig | Configure the tiniest possible kernel |

I think menuconfig is the most popular of these targets. The targets are processed by different host programs, which are provided by the kernel and built during kernel building. Some targets have a GUI (for the user's convenience) while most don't. Kconfig-related tools and source code reside mainly under scripts/kconfig/ in the kernel source. As we can see from scripts/kconfig/Makefile, there are several host programs, including conf, mconf, and nconf. Except for conf, each of them is responsible for one of the GUI-based config targets, so, conf deals with most of them.

Logically, Kconfig's infrastructure has two parts: one implements a [new language](https://github.com/torvalds/linux/blob/master/Documentation/kbuild/kconfig-language.txt) to define the configuration items (see the Kconfig files under the kernel source), and the other parses the Kconfig language and deals with configuration actions.

Most of the config targets have roughly the same internal process (shown below):



Note that all configuration items have a default value.

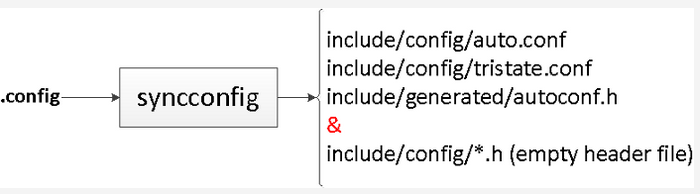
The first step reads the Kconfig file under source root to construct an initial configuration database; then it updates the initial database by reading an existing configuration file according to this priority:

.config  
/lib/modules/$(shell,uname -r)/.config  
/etc/kernel-config  
/boot/config-$(shell,uname -r)  
ARCH\_DEFCONFIG  
arch/$(ARCH)/defconfig

If you are doing GUI-based configuration via menuconfig or command-line-based configuration via oldconfig, the database is updated according to your customization. Finally, the configuration database is dumped into the .config file.

But the .config file is not the final fodder for kernel building; this is why the syncconfig target exists. syncconfig used to be a config target called silentoldconfig, but it doesn't do what the old name says, so it was renamed. Also, because it is for internal use (not for users), it was dropped from the list.

Here is an illustration of what syncconfig does:



syncconfig takes .config as input and outputs many other files, which fall into three categories:

* auto.conf & tristate.conf are used for makefile text processing. For example, you may see statements like this in a component's makefile:

obj-$(CONFIG\_GENERIC\_CALIBRATE\_DELAY) += calibrate.o

* autoconf.h is used in C-language source files.
* Empty header files under include/config/ are used for configuration-dependency tracking during kbuild, which is explained below.

After configuration, we will know which files and code pieces are not compiled.

## kbuild

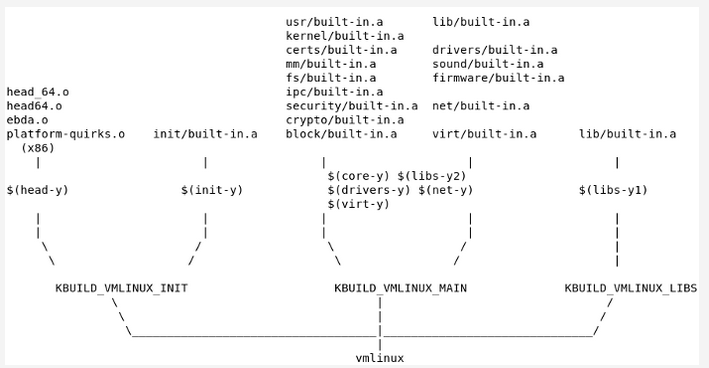
Component-wise building, called *recursive make*, is a common way for GNU make to manage a large project. Kbuild is a good example of recursive make. By dividing source files into different modules/components, each component is managed by its own makefile. When you start building, a top makefile invokes each component's makefile in the proper order, builds the components, and collects them into the final executive.

Kbuild refers to different kinds of makefiles:

* Makefile is the top makefile located in source root.
* .config is the kernel configuration file.
* arch/$(ARCH)/Makefile is the arch makefile, which is the supplement to the top makefile.
* scripts/Makefile.\* describes common rules for all kbuild makefiles.
* Finally, there are about 500 kbuild makefiles.

The top makefile includes the arch makefile, reads the .config file, descends into subdirectories, invokes make on each component's makefile with the help of routines defined in scripts/Makefile.\*, builds up each intermediate object, and links all the intermediate objects into vmlinux. Kernel document [Documentation/kbuild/makefiles.txt](https://github.com/torvalds/linux/blob/master/Documentation/kbuild/makefiles.txt) describes all aspects of these makefiles.

As an example, let's look at how vmlinux is produced on x86-64:



(The illustration is based on Richard Y. Steven's [blog](https://blog.csdn.net/richardysteven/article/details/52502734). It was updated and is used with the author's permission.)

https://blog.csdn.net/richardysteven/article/details/52502734

All the .o files that go into vmlinux first go into their own built-in.a, which is indicated via variables KBUILD\_VMLINUX\_INIT, KBUILD\_VMLINUX\_MAIN, KBUILD\_VMLINUX\_LIBS, then are collected into the vmlinux file.

Take a look at how recursive make is implemented in the Linux kernel, with the help of simplified makefile code:

# In top Makefile

vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps)

+$(call if\_changed,link-vmlinux)

# Variable assignments

vmlinux-deps := $(KBUILD\_LDS) $(KBUILD\_VMLINUX\_INIT) $(KBUILD\_VMLINUX\_MAIN) $(KBUILD\_VMLINUX\_LIBS)

export KBUILD\_VMLINUX\_INIT := $(head-y) $(init-y)

export KBUILD\_VMLINUX\_MAIN := $(core-y) $(libs-y2) $(drivers-y) $(net-y) $(virt-y)

export KBUILD\_VMLINUX\_LIBS := $(libs-y1)

export KBUILD\_LDS := arch/$(SRCARCH)/kernel/vmlinux.lds

init-y := init/

drivers-y := drivers/ sound/ firmware/

net-y := net/

libs-y := lib/

core-y := usr/

virt-y := virt/

# Transform to corresponding built-in.a

init-y := $(patsubst %/, %/built-in.a, $(init-y))

core-y := $(patsubst %/, %/built-in.a, $(core-y))

drivers-y := $(patsubst %/, %/built-in.a, $(drivers-y))

net-y := $(patsubst %/, %/built-in.a, $(net-y))

libs-y1 := $(patsubst %/, %/lib.a, $(libs-y))

libs-y2 := $(patsubst %/, %/built-in.a, $(filter-out %.a, $(libs-y)))

virt-y := $(patsubst %/, %/built-in.a, $(virt-y))

# Setup the dependency. vmlinux-deps are all intermediate objects, vmlinux-dirs

# are phony targets, so every time comes to this rule, the recipe of vmlinux-dirs

# will be executed. Refer "4.6 Phony Targets" of `info make`

$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;

# Variable vmlinux-dirs is the directory part of each built-in.a

vmlinux-dirs := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \

$(core-y) $(core-m) $(drivers-y) $(drivers-m) \

$(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))

# The entry of recursive make

$(vmlinux-dirs):

$(Q)$(MAKE) $(build)=$@ need-builtin=1

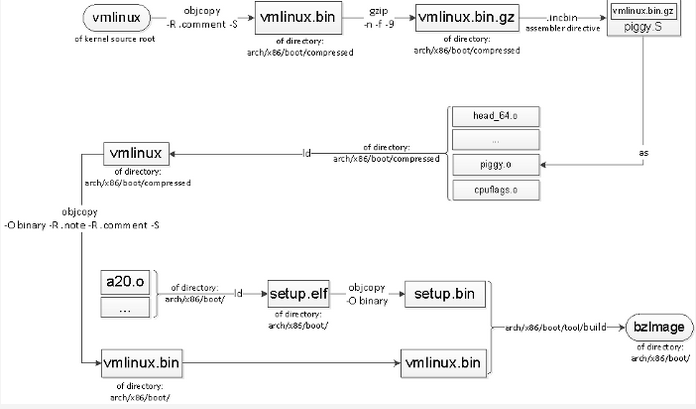
The recursive make recipe is expanded, for example:

make -f scripts/Makefile.build obj=init need-builtin=1

This means make will go into scripts/Makefile.build to continue the work of building each built-in.a. With the help of scripts/link-vmlinux.sh, the vmlinux file is finally under source root.

### Understanding vmlinux vs. bzImage

Many Linux kernel developers may not be clear about the relationship between vmlinux and bzImage. For example, here is their relationship in x86-64:



The source root vmlinux is stripped, compressed, put into piggy.S, then linked with other peer objects into arch/x86/boot/compressed/vmlinux. Meanwhile, a file called setup.bin is produced under arch/x86/boot. There may be an optional third file that has relocation info, depending on the configuration of CONFIG\_X86\_NEED\_RELOCS.

A host program called build, provided by the kernel, builds these two (or three) parts into the final bzImage file.

### Dependency tracking

Kbuild tracks three kinds of dependencies:

1. All prerequisite files (both \*.c and \*.h)
2. CONFIG\_ options used in all prerequisite files
3. Command-line dependencies used to compile the target

The first one is easy to understand, but what about the second and third? Kernel developers often see code pieces like this:

#ifdef CONFIG\_SMP

\_\_boot\_cpu\_id = cpu;

#endif

When CONFIG\_SMP changes, this piece of code should be recompiled. The command line for compiling a source file also matters, because different command lines may result in different object files.

When a .c file uses a header file via a #include directive, you need write a rule like this:

main.o: defs.h

recipe...

When managing a large project, you need a lot of these kinds of rules; writing them all would be tedious and boring. Fortunately, most modern C compilers can write these rules for you by looking at the #include lines in the source file. For the GNU Compiler Collection (GCC), it is just a matter of adding a command-line parameter: -MD depfile

# In scripts/Makefile.lib

c\_flags = -Wp,-MD,$(depfile) $(NOSTDINC\_FLAGS) $(LINUXINCLUDE) \

-include $(srctree)/include/linux/compiler\_types.h \

$(\_\_c\_flags) $(modkern\_cflags) \

$(basename\_flags) $(modname\_flags)

This would generate a .d file with content like:

init\_task.o: init/init\_task.c include/linux/kconfig.h \

include/generated/autoconf.h include/linux/init\_task.h \

include/linux/rcupdate.h include/linux/types.h \

...

Then the host program [**fixdep**](https://github.com/torvalds/linux/blob/master/scripts/basic/fixdep.c) takes care of the other two dependencies by taking the depfile and command line as input, then outputting a .<target>.cmd file in makefile syntax, which records the command line and all the prerequisites (including the configuration) for a target. It looks like this:

# The command line used to compile the target

cmd\_init/init\_task.o := gcc -Wp,-MD,init/.init\_task.o.d -nostdinc ...

...

# The dependency files

deps\_init/init\_task.o := \

$(wildcard include/config/posix/timers.h) \

$(wildcard include/config/arch/task/struct/on/stack.h) \

$(wildcard include/config/thread/info/in/task.h) \

...

include/uapi/linux/types.h \

arch/x86/include/uapi/asm/types.h \

include/uapi/asm-generic/types.h \

...

A .<target>.cmd file will be included during recursive make, providing all the dependency info and helping to decide whether to rebuild a target or not.

The secret behind this is that fixdep will parse the depfile (.d file), then parse all the dependency files inside, search the text for all the CONFIG\_ strings, convert them to the corresponding empty header file, and add them to the target's prerequisites. Every time the configuration changes, the corresponding empty header file will be updated, too, so kbuild can detect that change and rebuild the target that depends on it. Because the command line is also recorded, it is easy to compare the last and current compiling parameters.

## Looking ahead

Kconfig/kbuild remained the same for a long time until the new maintainer, Masahiro Yamada, joined in early 2017, and now kbuild is under active development again. Don't be surprised if you soon see something different from what's in this article.

