## Kbuild: the Linux Kernel Build System

**Javier Martinez Canillas**

Issue #222, October 2012

The first step to contributing to a project is to know how its build system works. Here, I cover the kernel build system and show you how to use it to add a feature to a kernel.

One amazing thing about Linux is that the same code base is used for a different range of computing systems, from supercomputers to very tiny embedded devices. If you stop for a second and think about it, Linux is probably the only OS that has a unified code base. For example, Microsoft and Apple use different kernels for their desktop and mobile OS versions (Windows NT/Windows CE and OS X/iOS). Two of the reasons this is possible on Linux are that the kernel has many abstraction layers and levels of indirection and because its build system allows for creating highly customized kernel binary images.

The Linux kernel has a monolithic architecture, which means that the whole kernel code runs in kernel space and shares the same address space. Because of this architecture, you have to choose the features your kernel will include at compile time. Technically, Linux is not a pure monolithic kernel, because it can be extended at runtime using loadable kernel modules. To load a module, the kernel must contain all the kernel symbols used in the module. If those symbols were not included in the kernel at compile time, the module will not be loaded due to missing dependencies. Modules are only a way to defer compilation (or execution) of a specific kernel feature. Once a kernel module is loaded, it is part of the monolithic kernel and shares the same address space of the code that was included at kernel compile time. Even when Linux supports modules, you still need to choose at kernel compile time most of the features that will be built in the kernel image and the ones that will allow you to load specific kernel modules once the kernel is executing.

For this reason, it is very important to be able to choose what code you want to compile (or not) in a Linux kernel. The approach for achieving this is using conditional compilation. There are tons of configuration options for choosing whether a specific feature will be included. This is translated to deciding whether a specific C file, code segment or data structure will be included in the kernel image and its modules.

So, an easy and efficient way to manage all these compilation options is needed. The infrastructure to manage this—building the kernel image and its modules—is known as the Kernel Build System (kbuild).

I don't explain the kbuild infrastructure in too much detail here, because the Linux kernel documentation provides a good explanation (Documentation/kbuild). Instead, I discuss the kbuild basics and show how to use it to include your own code in a Linux kernel tree, such as a device driver.

The Linux Kernel Build System has four main components:

- Config symbols: compilation options that can be used to compile code conditionally in source files and to decide which objects to include in a kernel image or its modules.
- Kconfig files: define each config symbol and its attributes, such as its type, description and dependencies. Programs that generate an option menu tree (for example, `make menuconfig`) read the menu entries from these files.
- .config file: stores each config symbol's selected value. You can edit this file manually or use one of the many `make` configuration targets, such as menuconfig and xconfig, that call specialized programs to build a tree-like menu and automatically update (and create) the .config file for you.
- Makefiles: normal GNU makefiles that describe the relationship between source files and the commands needed to generate each make target, such as kernel images and modules.

Now, let's look at each of these components in more detail.

## Compilation Options: Configuration Symbols

Configuration symbols are the ones used to decide which features will be included in the final Linux kernel image. Two kinds of symbols are used for conditional compilation: boolean and tristate. They differ only in the number of values that each one can take. But, this difference is more important than it seems. Boolean symbols (not surprisingly) can take one of two values: true or false. Tristate symbols, on the other hand, can take three different values: yes, no or module.

Not everything in the kernel can be compiled as a module. Many features are so intrusive that you have to decide at compilation time whether the kernel will support them. For example, you can't add Symmetric Multi-Processing (SMP) or kernel preemption support to a running kernel. So, using a boolean config symbol makes sense for those kinds of features. Most features that can be compiled as modules also can be added to a kernel at compile time. That's the reason tristate symbols exist—to decide whether you want to compile a feature built-in (y), as a module (m) or not at all (n).

There are other config symbol types besides these two symbols, such as strings and hex. But, because they are not used for conditional compilation, I don't cover those here. Read the Linux kernel documentation for a complete discussion of config symbols, types and uses.

## Defining Configuration Symbols: Kconfig Files

Configuration symbols are defined in files known as Kconfig files. Each Kconfig file can describe an arbitrary number of symbols and can also include (source) other Kconfig files. Compilation targets that construct configuration menus of kernel compile options, such as `make menuconfig`, read these files to build the tree-like structure. Every directory in the kernel has one Kconfig that includes the Kconfig files of its subdirectories. On top of the kernel source code directory, there is a Kconfig file that is the root of the options tree. The menuconfig (scripts/kconfig/mconf), gconfig (scripts/kconfig/gconf) and other compile targets invoke programs that start at this root Kconfig and recursively read the Kconfig files located in each subdirectory to build their menus. Which subdirectory to visit also is defined in each Kconfig file and also depends on the config symbol values chosen by the user.

## Storing Symbol Values: .config File

All config symbol values are saved in a special file called .config. Every time you want to change a kernel compile configuration, you execute a make target, such as menuconfig or xconfig. These read the Kconfig files to create the menus and update the config symbols' values using the values defined in the .config file. Additionally, these tools update the .config file with the new options you chose and also can generate one if it didn't exist before.

Because the .config file is plain text, you also can change it without needing any specialized tool. It is very convenient for saving and restoring previous kernel compilation configurations as well.

## Compiling the Kernel: Makefiles

The last component of the kbuild system is the Makefiles. These are used to build the kernel image and modules. Like the Kconfig files, each subdirectory has a Makefile that compiles only the files in its directory. The whole build is done recursively—a top Makefile descends into its subdirectories and executes each subdirectory's Makefile to generate the binary objects for the files in that directory. Then, these objects are used to generate the modules and the Linux kernel image.

## Putting It All Together: Adding the Coin Driver

Now that you know more about kbuild system basics, let's consider a practical example —adding a device driver to a Linux kernel tree. The example driver is for a very simple character device called coin. The driver's function is to mimic a coin flipping and returning on each read one of two values: head or tail. The driver has an optional feature that exposes previous flip statistics using a special debugfs virtual file. Listing 1 shows an example interaction with the coin device.

**Listing 1. Coin Character Device Semantics**

```
root@localhost:~# cat /dev/coin
tail
root@localhost:~# cat /dev/coin
head
root@sauron:/# cat /sys/kernel/debug/coin/stats
head=6 tail=4
```

To add a feature to a Linux kernel (such as the coin driver), you need to do three things:

1. Put the source file(s) in a place that makes sense, such as drivers/net/wireless for Wi-Fi devices or fs for a new filesystem.
2. Update the Kconfig for the subdirectory (or subdirectories) where you put the files with config symbols that allow you to choose to include the feature.
3. Update the Makefile for the subdirectory where you put the files, so the build system can compile your code conditionally.

Because this driver is for a character device, put the coin.c source file in drivers/char.

The next step is to give the user the option to compile the coin driver. To do this, you need to add two configuration symbols to the drivers/char/Kconfig file: one to choose to add the driver to the kernel and a second to decide whether the driver statistics will be available.

Like most drivers, coin can be built in the kernel, included as a module or not included at all.

So, the first config symbol, called `COIN`, is of type tristate (y/n/m). The second symbol, `COIN_STAT`, is used to decide whether you want to expose the statistics. Clearly this is a binary decision, so the symbol type is bool (y/n). Also, it doesn't make sense to add the coin statistics to the kernel if you choose not to include the coin driver itself. This behavior is very common in the kernel—for example, you can't add a block-based filesystem, such as ext3 or fat32, if you didn't enable the block layer first. Obviously, there is some kind of dependency between symbols, and you should model this. Fortunately, you can describe config symbols' relationships in Kconfig files using the "depends on" keyword. When, for example, the `make menuconfig` target generates the compilation options menu tree, it hides all the options whose symbol dependencies are not met. This is just one of many keywords available for describing symbols in a Kconfig file. For a complete description of the Kconfig language, refer to kbuild/kconfig-language.txt in the Linux kernel Documentation directory.

Listing 2 shows a segment of the drivers/char/Kconfig file with the symbols added for the coin driver.

**Listing 2. Kconfig Entries for the Coin Driver**

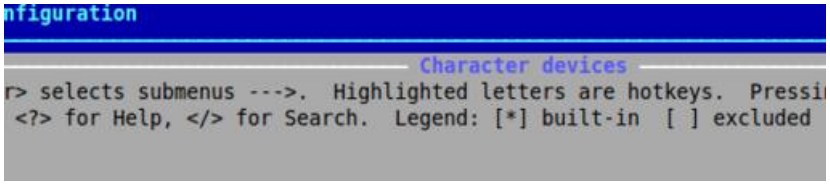
```
#
# Character device configuration
#

menu "Character devices"

config COIN
       tristate "Coin char device support"
       help
         Say Y here if you want to add support for the
         coin char device.

         If unsure, say N.

         To compile this driver as a module, choose M here:
         the module will be called coin.

config COIN_STAT
        bool "flipping statistics"
        depends on COIN
       help
        Say Y here if you want to enable statistics about
        the coin char device.
```

So, how can you use your recently added symbols?

As mentioned previously, `make` targets that build a tree menu with all the compilation options use this config symbol, so you can choose what to compile in your kernel and its modules. For example, when you execute:

```
$ make menuconfig
```

the command-line utility scripts/kconfig/mconf will start and read all the Kconfig files to build a menu-based interface. You then use these programs to update the values of your `COIN` and `COIN_STAT` compilation options. Figure 1 shows how the menu looks when you navigate to Device Drivers→Character devices; see how the options for the coin driver can be set.



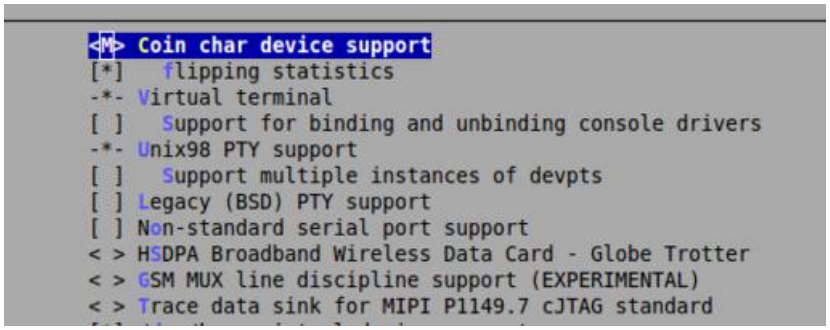



Figure 1. Menu Example

Once you are done with the compilation option configuration, exit the program, and if you made some changes, you will be asked to save your new configuration. This saves the configuration options to the .config file. For every symbol, a CONFIG_ prefix is appended in the .config file. For example, if the symbol is of type boolean and you chose it, in the .config file, the symbol will be saved like this:

```
CONFIG_COIN_STAT=y
```

On the other hand, if you didn't choose the symbol, it won't be set in the .config file, and you will see something like this:

```
# CONFIG_COIN_STAT is not set
```

Tristate symbols have the same behavior as bool types when chosen or not. But, remember that tristate also has the third option of compiling the feature as a module. For example, you can choose to compile the COIN driver as a module and have something like this in the .config file:

```
CONFIG_COIN=m
```

The following is a segment of the .config file that shows the values chosen for the coin driver symbols:

```
CONFIG_COIN=m
CONFIG_COIN_STAT=y
```

Here you are telling kbuild that you want to compile the coin driver as a module and activate the flipping statistics. If you have chosen to compile the driver built-in and without the flipping statistics, you will have something like this:

```
CONFIG_COIN=y
# CONFIG_COIN_STAT is not set
```

Once you have your .config file, you are ready to compile your kernel and its modules. When you execute a compile target to compile the kernel or the modules, it first executes a binary that reads all the Kconfig files and .config:

```
$ scripts/kconfig/conf Kconfig
```

This binary updates (or creates) a C header file with the values you chose for all the configuration symbols. This file is include/generated/autoconf.h, and every gcc compile

instruction includes it, so the symbols can be used in any source file in the kernel.

The file is composed of thousands of #define macros that describe the state for each symbol. Let's look at the conventions for the macros.

Bool symbols with the value true and tristate symbols with the value yes are treated equally. For both of them, three macros are defined.

For example, the bool `CONFIG_COIN_STAT` symbol with the value true and the tristate `CONFIG_COIN` symbol with the value yes will generate the following:

```
#define __enabled_CONFIG_COIN_STAT 1
#define __enabled_CONFIG_COIN_STAT_MODULE 0
#define CONFIG_COIN_STAT 1

#define __enabled_CONFIG_COIN 1
#define __enabled_CONFIG_COIN_MODULE 0
#define CONFIG_COIN 1
```

In the same way, bool symbols with the value false and tristate symbols with the value no have the same semantics. For both of them, two macros are defined. For example, the `CONFIG_COIN_STAT` with the value false and the `CONFIG_COIN` with the value no will generate the following group of macros:

```
#define __enabled_CONFIG_COIN_STAT 0
#define __enabled_CONFIG_COIN_STAT_MODULE 0

#define __enabled_CONFIG_COIN 0
#define __enabled_CONFIG_COIN_MODULE 0
```

For tristate symbols with the value module, three macros are defined. For example, the `CONFIG_COIN` with the value module will generate the following:

```
#define __enabled_CONFIG_COIN 0
#define __enabled_CONFIG_COIN_MODULE 1
#define CONFIG_COIN_MODULE 1
```

Curious readers probably will ask why are those `__enabled_option` macros needed? Wouldn't it be sufficient to have only the `CONFIG_option` and `CONFIG_option_MODULE`? And, why is `_MODULE` declared even for symbols that are of type bool?

Well, the `__enabled_` constants are used by three macros:

```
#define IS_ENABLED(option) \
        (__enabled_ ## option || __enabled_ ## option ## _MODULE)

#define IS_BUILTIN(option) __enabled_ ## option

#define IS_MODULE(option) __enabled_ ## option ## _MODULE
```

So, the `__enabled_option` and `__enabled_option_MODULE` always are defined, even for bool symbols to make sure that this macro will work for any configuration option.

The third and last step is to update the Makefiles for the subdirectories where you put your source files, so kbuild can compile your driver if you chose it.

But, how do you instruct kbuild to compile your code conditionally?

The kernel build system has two main tasks: creating the kernel binary image and the kernel modules. To do that, it maintains two lists of objects: obj-y and obj-m, respectively. The former is a list of all the objects that will be built in the kernel image, and the latter is the list of the objects that will be compiled as modules.

The configuration symbols from .config and the macros from autoconf.h are used along with some GNU make syntax extensions to fill these lists. Kbuild recursively enters each directory and builds the lists adding the objects defined in each subdirectory's Makefile. For more information about the GNU make extensions and the objects list, read Documentation/kbuild/makefiles.txt.

For the coin driver, the only thing you need to do is add a line in drivers/char/Makefile:

```
obj-$(CONFIG_COIN)    += coin.o
```

This tells kbuild to create an object from the source file coin.c and to add it to an object list. Because `CONFIG_COIN`'s value can be y or m, the coin.o object will be added to the obj-y or obj-m list depending on the symbol value. It then will be built in the kernel or as a module. If you didn't choose the `CONFIG_COIN` option, the symbol is undefined, and coin.o will not be compiled at all.

Now you know how to include source files conditionally. The last part of the puzzle is how to compile source code segments conditionally. This can be done easily by using the macros defined in autoconf.h. Listing 3 shows the complete coin character device driver.

**Listing 3. Coin Character Device Driver Example**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/device.h>
#include <linux/random.h>
#include <linux/debugfs.h>

#define DEVNAME "coin"
#define LEN  20
enum values {HEAD, TAIL};

struct dentry *dir, *file;
int file_value;
int stats[2] = {0, 0};
char *msg[2] = {"head\n", "tail\n"};

static int major;
static struct class *class_coin;
static struct device *dev_coin;

static ssize_t r_coin(struct file *f, char __user *b,
                      size_t cnt, loff_t *lf)
{
        char *ret;
        u32 value = random32() % 2;
        ret = msg[value];
        stats[value]++;
        return simple_read_from_buffer(b, cnt,
                                       lf, ret,
                                       strlen(ret));
}

static struct file_operations fops = { .read = r_coin };

#ifdef CONFIG_COIN_STAT
static ssize_t r_stat(struct file *f, char __user *b,
                        size_t cnt, loff_t *lf)
```

```
{
        char buf[LEN];
        snprintf(buf, LEN, "head=%d tail=%d\n",
                 stats[HEAD], stats[TAIL]);
        return simple_read_from_buffer(b, cnt,
                                       lf, buf,
                                       strlen(buf));
}

static struct file_operations fstat = { .read = r_stat };
#endif

int init_module(void)
{
        void *ptr_err;
        major = register_chrdev(0, DEVNAME, &fops);
        if (major < 0)
                return major;

        class_coin = class_create(THIS_MODULE,
                                  DEVNAME);
        if (IS_ERR(class_coin)) {
                ptr_err = class_coin;
                goto err_class;
        }

        dev_coin = device_create(class_coin, NULL,
                                 MKDEV(major, 0),
                                 NULL, DEVNAME);
        if (IS_ERR(dev_coin))
                goto err_dev;

#ifdef CONFIG_COIN_STAT
        dir = debugfs_create_dir("coin", NULL);
        file = debugfs_create_file("stats", 0644,
                                   dir, &file_value,
                                   &fstat);
#endif

        return 0;
err_dev:
        ptr_err = class_coin;
        class_destroy(class_coin);
err_class:
        unregister_chrdev(major, DEVNAME);
        return PTR_ERR(ptr_err);
}

void cleanup_module(void)
{
#ifdef CONFIG_COIN_STAT
        debugfs_remove(file);
        debugfs_remove(dir);
#endif

        device_destroy(class_coin, MKDEV(major, 0));
        class_destroy(class_coin);
        return unregister_chrdev(major, DEVNAME);
}
```

In Listing 3, you can see that the `CONFIG_COIN_STAT` configuration option is used to register (or not) a special debugfs file that exposes the coin-flipping statistics to userspace.

Figure 2 summarizes the kernel build process, and the output of the `git diff --stat` command shows the files you have modified to include the driver:

```
drivers/char/Kconfig  |   16 +++++++++
drivers/char/Makefile |    1 +
drivers/char/coin.c   |   89 ++++++++++++++++++++++++++++++++++++++++
3 files changed, 106 insertions(+), 0 deletions(-)
```

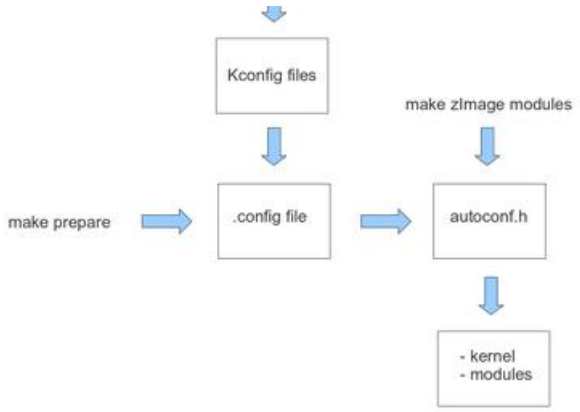



Figure 2. Kernel Build Process

## Conclusion

Linux, despite being a monolithic kernel, is highly modular and customizable. You can use the same kernel in a varied range of devices from high-performance clusters to desktops all the way to mobile phones. This makes the kernel a very big and complex piece of software. But, even when the kernel has millions of lines of code, its build system allows you to extend it with new features easily. In the past, to have access to an operating system's source code, you had to work for a big company and sign large NDA agreements. Nowadays, the source of probably the most modern operating system is publicly available. You can use it, study its internals and modify it in any creative way you want. The best part is that you even can share your work and get feedback from an active community. Happy hacking!

Javier Martinez Canillas is a longtime Linux user, administrator and open-source advocate developer. He has an MS from the Universitat Autònoma de Barcelona and works as a Linux kernel engineer. Besides hacking, he enjoys spending as much time as possible with his wife Tami, running, reading and photography. He can be reached at javier@dowhile0.org.

Archive Index Issue Table of Contents Article