Building External Modules

This document describes how to build an out-of-tree kernel module.

=== Table of Contents

=== 1. Introduction

"kbuild" is the build system used by the Linux kernel. Modules must use kbuild to stay compatible with changes in the build infrastructure and to pick up the right flags to "gcc." Functionality for building modules both in-tree and out-of-tree is provided. The method for building either is similar, and all modules are initially developed and built out-of-tree.

Covered in this document is information aimed at developers interested in building out-of-tree (or "external") modules. The author of an external module should supply a makefile that hides most of the complexity, so one only has to type "make" to build the module. This is easily accomplished, and a complete example will be presented in section 3.

=== 2. How to Build External Modules

To build external modules, you must have a prebuilt kernel available that contains the configuration and header files used in the build. Also, the kernel must have been built with modules enabled. If you are using a distribution kernel, there will be a package for the kernel you are running provided by your distribution.

An alternative is to use the "make" target "modules_prepare." This will make sure the kernel contains the information required. The target exists solely as a simple way to prepare a kernel source tree for building external modules.

NOTE: "modules_prepare" will not build Module.symvers even if

CONFIG_MODVERSIONS is set; therefore, a full kernel build needs to be
executed to make module versioning work.

--- 2.1 Command Syntax

        The command to build an external module is:

                $ make -C <path_to_kernel_src> M=$PWD

        The kbuild system knows that an external module is being built
        due to the "M=<dir>" option given in the command.

        To build against the running kernel use:

                $ make -C /lib/modules/`uname -r`/build M=$PWD

        Then to install the module(s) just built, add the target
        "modules_install" to the command:

                $ make -C /lib/modules/`uname -r`/build M=$PWD modules_install

--- 2.2 Options

        ($KDIR refers to the path of the kernel source directory.)

        make -C $KDIR M=$PWD

        -C $KDIR
                The directory where the kernel source is located.
                "make" will actually change to the specified directory
                when executing and will change back when finished.

        M=$PWD
                Informs kbuild that an external module is being built.
                The value given to "M" is the absolute path of the
                directory where the external module (kbuild file) is
                located.

--- 2.3 Targets

        When building an external module, only a subset of the "make"
        targets are available.

        make -C $KDIR M=$PWD [target]

        The default will build the module(s) located in the current
        directory, so a target does not need to be specified. All
        output files will also be generated in this directory. No
        attempts are made to update the kernel source, and it is a
        precondition that a successful "make" has been executed for the
        kernel.

        modules
                The default target for external modules. It has the
                same functionality as if no target was specified. See
                description above.

        modules_install
                Install the external module(s). The default location is
                /lib/modules/<kernel_release>/extra/, but a prefix may
                be added with INSTALL_MOD_PATH (discussed in section 5).

        clean
                Remove all generated files in the module directory only.

```
        help
                List the available targets for external modules.

--- 2.4 Building Separate Files

        It is possible to build single files that are part of a module.
        This works equally well for the kernel, a module, and even for
        external modules.

        Example (The module foo.ko, consist of bar.o and baz.o):
                make -C $KDIR M=$PWD bar.lst
                make -C $KDIR M=$PWD baz.o
                make -C $KDIR M=$PWD foo.ko
                make -C $KDIR M=$PWD /


=== 3. Creating a Kbuild File for an External Module

In the last section we saw the command to build a module for the
running kernel. The module is not actually built, however, because a
build file is required. Contained in this file will be the name of
the module(s) being built, along with the list of requisite source
files. The file may be as simple as a single line:

        obj-m := <module_name>.o

The kbuild system will build <module_name>.o from <module_name>.c,
and, after linking, will result in the kernel module <module_name>.ko.
The above line can be put in either a "Kbuild" file or a "Makefile."
When the module is built from multiple sources, an additional line is
needed listing the files:

        <module_name>-y := <src1>.o <src2>.o ...

NOTE: Further documentation describing the syntax used by kbuild is
located in Documentation/kbuild/makefiles.txt.

The examples below demonstrate how to create a build file for the
module 8123.ko, which is built from the following files:

        8123_if.c
        8123_if.h
        8123_pci.c
        8123_bin.o_shipped      <= Binary blob

--- 3.1 Shared Makefile

        An external module always includes a wrapper makefile that
        supports building the module using "make" with no arguments.
        This target is not used by kbuild; it is only for convenience.
        Additional functionality, such as test targets, can be included
        but should be filtered out from kbuild due to possible name
        clashes.

        Example 1:
                --> filename: Makefile
                ifneq ($(KERNELRELEASE),)
                # kbuild part of makefile
                obj-m  := 8123.o
                8123-y := 8123_if.o 8123_pci.o 8123_bin.o

                else
                # normal makefile
```

```
KDIR ?= /lib/modules/`uname -r`/build

default:
	$(MAKE) -C $(KDIR) M=$$PWD

# Module specific targets
genbin:
	echo "X" > 8123_bin.o_shipped

endif
```

The check for KERNELRELEASE is used to separate the two parts of the makefile. In the example, kbuild will only see the two assignments, whereas "make" will see everything except these two assignments. This is due to two passes made on the file: the first pass is by the "make" instance run on the command line; the second pass is by the kbuild system, which is initiated by the parameterized "make" in the default target.

--- 3.2 Separate Kbuild File and Makefile

In newer versions of the kernel, kbuild will first look for a file named "Kbuild," and only if that is not found, will it then look for a makefile. Utilizing a "Kbuild" file allows us to split up the makefile from example 1 into two files:

Example 2:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
	$(MAKE) -C $(KDIR) M=$$PWD

# Module specific targets
genbin:
	echo "X" > 8123_bin.o_shipped
```

The split in example 2 is questionable due to the simplicity of each file; however, some external modules use makefiles consisting of several hundred lines, and here it really pays off to separate the kbuild part from the rest.

The next example shows a backward compatible version.

Example 3:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
include Kbuild

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
	$(MAKE) -C $(KDIR) M=$$PWD
```

```
# Module specific targets
genbin:
        echo "X" > 8123_bin.o_shipped

endif
```

Here the "Kbuild" file is included from the makefile. This allows an older version of kbuild, which only knows of makefiles, to be used when the "make" and kbuild parts are split into separate files.

--- 3.3 Binary Blobs

Some external modules need to include an object file as a blob. kbuild has support for this, but requires the blob file to be named <filename>_shipped. When the kbuild rules kick in, a copy of <filename>_shipped is created with _shipped stripped off, giving us <filename>. This shortened filename can be used in the assignment to the module.

Throughout this section, 8123_bin.o_shipped has been used to build the kernel module 8123.ko; it has been included as 8123_bin.o.

```
8123-y := 8123_if.o 8123_pci.o 8123_bin.o
```

Although there is no distinction between the ordinary source files and the binary file, kbuild will pick up different rules when creating the object file for the module.

--- 3.4 Building Multiple Modules

kbuild supports building multiple modules with a single build file. For example, if you wanted to build two modules, foo.ko and bar.ko, the kbuild lines would be:

```
obj-m := foo.o bar.o
foo-y := <foo_srcs>
bar-y := <bar_srcs>
```

It is that simple!

=== 4. Include Files

Within the kernel, header files are kept in standard locations according to the following rule:

* If the header file only describes the internal interface of a module, then the file is placed in the same directory as the source files.
* If the header file describes an interface used by other parts of the kernel that are located in different directories, then the file is placed in include/linux/.

  NOTE: There are two notable exceptions to this rule: larger subsystems have their own directory under include/, such as include/scsi; and architecture specific headers are located under arch/$(ARCH)/include/.

--- 4.1 Kernel Includes

To include a header file located under include/linux/, simply

use:

```
#include <linux/module.h>
```

kbuild will add options to "gcc" so the relevant directories are searched.

--- 4.2 Single Subdirectory

External modules tend to place header files in a separate include/ directory where their source is located, although this is not the usual kernel style. To inform kbuild of the directory, use either ccflags-y or CFLAGS_<filename>.o.

Using the example from section 3, if we moved 8123_if.h to a subdirectory named include, the resulting kbuild file would look like:

```
--> filename: Kbuild
obj-m := 8123.o

ccflags-y := -Iinclude
8123-y := 8123_if.o 8123_pci.o 8123_bin.o
```

Note that in the assignment there is no space between -I and the path. This is a limitation of kbuild: there must be no space present.

--- 4.3 Several Subdirectories

kbuild can handle files that are spread over several directories. Consider the following example:

```
.
|__ src
|   |__ complex_main.c
|   |__ hal
|       |__ hardwareif.c
|       |__ include
|           |__ hardwareif.h
|__ include
    |__ complex.h
```

To build the module complex.ko, we then need the following kbuild file:

```
--> filename: Kbuild
obj-m := complex.o
complex-y := src/complex_main.o
complex-y += src/hal/hardwareif.o

ccflags-y := -I$(src)/include
ccflags-y += -I$(src)/src/hal/include
```

As you can see, kbuild knows how to handle object files located in other directories. The trick is to specify the directory relative to the kbuild file's location. That being said, this is NOT recommended practice.

For the header files, kbuild must be explicitly told where to look. When kbuild executes, the current directory is always the root of the kernel tree (the argument to "-C") and therefore an absolute path is needed. $(src) provides the absolute path by pointing to the directory where the currently executing kbuild

file is located.

=== 5. Module Installation

Modules which are included in the kernel are installed in the
directory:

/lib/modules/$(KERNELRELEASE)/kernel/

And external modules are installed in:

/lib/modules/$(KERNELRELEASE)/extra/

--- 5.1 INSTALL_MOD_PATH

Above are the default directories but as always some level of
customization is possible. A prefix can be added to the
installation path using the variable INSTALL_MOD_PATH:

```
$ make INSTALL_MOD_PATH=/frodo modules_install
=> Install dir: /frodo/lib/modules/$(KERNELRELEASE)/kernel/
```

INSTALL_MOD_PATH may be set as an ordinary shell variable or,
as shown above, can be specified on the command line when
calling "make." This has effect when installing both in-tree
and out-of-tree modules.

--- 5.2 INSTALL_MOD_DIR

External modules are by default installed to a directory under
/lib/modules/$(KERNELRELEASE)/extra/, but you may wish to
locate modules for a specific functionality in a separate
directory. For this purpose, use INSTALL_MOD_DIR to specify an
alternative name to "extra."

```
$ make INSTALL_MOD_DIR=gandalf -C $KDIR \
       M=$PWD modules_install
=> Install dir: /lib/modules/$(KERNELRELEASE)/gandalf/
```

=== 6. Module Versioning

Module versioning is enabled by the CONFIG_MODVERSIONS tag, and is used
as a simple ABI consistency check. A CRC value of the full prototype
for an exported symbol is created. When a module is loaded/used, the
CRC values contained in the kernel are compared with similar values in
the module; if they are not equal, the kernel refuses to load the
module.

Module.symvers contains a list of all exported symbols from a kernel
build.

--- 6.1 Symbols From the Kernel (vmlinux + modules)

During a kernel build, a file named Module.symvers will be
generated. Module.symvers contains all exported symbols from
the kernel and compiled modules. For each symbol, the
corresponding CRC value is also stored.

The syntax of the Module.symvers file is:

```
<CRC>       <Symbol>           <module>

0x2d036834  scsi_remove_host   drivers/scsi/scsi_mod
```

For a kernel build without CONFIG_MODVERSIONS enabled, the CRC would read 0x00000000.

Module.symvers serves two purposes:
1) It lists all exported symbols from vmlinux and all modules.
2) It lists the CRC if CONFIG_MODVERSIONS is enabled.

--- 6.2 Symbols and External Modules

When building an external module, the build system needs access to the symbols from the kernel to check if all external symbols are defined. This is done in the MODPOST step. modpost obtains the symbols by reading Module.symvers from the kernel source tree. If a Module.symvers file is present in the directory where the external module is being built, this file will be read too. During the MODPOST step, a new Module.symvers file will be written containing all exported symbols that were not defined in the kernel.

--- 6.3 Symbols From Another External Module

Sometimes, an external module uses exported symbols from another external module. kbuild needs to have full knowledge of all symbols to avoid spitting out warnings about undefined symbols. Three solutions exist for this situation.

NOTE: The method with a top-level kbuild file is recommended but may be impractical in certain situations.

Use a top-level kbuild file
If you have two modules, foo.ko and bar.ko, where foo.ko needs symbols from bar.ko, you can use a common top-level kbuild file so both modules are compiled in the same build. Consider the following directory layout:

```
./foo/ <= contains foo.ko
./bar/ <= contains bar.ko
```

The top-level kbuild file would then look like:

```
#./Kbuild (or ./Makefile):
        obj-y := foo/ bar/
```

And executing

```
        $ make -C $KDIR M=$PWD
```

will then do the expected and compile both modules with full knowledge of symbols from either module.

Use an extra Module.symvers file
When an external module is built, a Module.symvers file is generated containing all exported symbols which are not defined in the kernel. To get access to symbols from bar.ko, copy the Module.symvers file from the compilation of bar.ko to the directory where foo.ko is built. During the module build, kbuild will read the Module.symvers file in the directory of the external module, and when the build is finished, a new Module.symvers file is created containing the sum of all symbols defined and not part of the kernel.

Use "make" variable KBUILD_EXTRA_SYMBOLS
	If it is impractical to copy Module.symvers from
	another module, you can assign a space separated list
	of files to KBUILD_EXTRA_SYMBOLS in your build file.
	These files will be loaded by modpost during the
	initialization of its symbol tables.

=== 7. Tips & Tricks

--- 7.1 Testing for CONFIG_FOO_BAR

Modules often need to check for certain CONFIG_ options to
decide if a specific feature is included in the module. In
kbuild this is done by referencing the CONFIG_ variable
directly.

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o

ext2-y := balloc.o bitmap.o dir.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

External modules have traditionally used "grep" to check for
specific CONFIG_ settings directly in .config. This usage is
broken. As introduced before, external modules should use
kbuild for building and can therefore use the same methods as
in-tree modules when testing for CONFIG_ definitions.