Module Build

In this document you will find information about:

- how to build external modules

- how to make your module use the kbuild infrastructure

- how kbuild will install a kernel

- how to install modules in a non-standard location

目录

1. Introduction

kbuild includes functionality for building modules both within the kernel source tree and outside the kernel source tree. The latter is usually referred to as external or "out-of-tree" modules and is used both during development and for modules that are not planned to be included in the kernel tree.

What is covered within this file is mainly information to authors of 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. A complete example will be presented in chapter 4, "Creating a kbuild file for an external module".

2. How to build external modules

kbuild offers functionality to build external modules, with the prerequisite that there is a pre-built kernel available with full source. A subset of the targets available when building the kernel is available when building an external module.

2.1. Building external modules

Use the following command to build an external module:

```
make -C <path-to-kernel> M=`pwd`
```

For the running kernel use:

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

For the above command to succeed, the kernel must have been
built with modules enabled.

To install the modules that were just built:

```
make -C <path-to-kernel> M=`pwd` modules_install
```

More complex examples will be shown later, the above should
be enough to get you started.

2.2. Available targets

$KDIR refers to the path to the kernel source top-level directory

make -C $KDIR M=`pwd` Will build the module(s) located in current directory.

All output files will be located in the same directory

as the module source.

No attempts are made to update the kernel source, and it is

a precondition that a successful make has been executed

for the kernel.

make -C $KDIR M=`pwd` modules The modules target is implied when no target is given.

Same functionality as if no target was specified.

See description above.

make -C $KDIR M=`pwd` modules_install Install the external module(s).

Installation default is in /lib/modules/<kernel-version>/extra,

but may be prefixed with INSTALL_MOD_PATH - see separate

chapter.

make -C $KDIR M=`pwd` clean Remove all generated files for the module - the kernel

source directory is not modified.

make -C $KDIR M=`pwd` help help will list the available target when building external

modules.

2.3. Available options:

$KDIR refers to the path to the kernel source top-level directory

make -C $KDIR Used to specify where to find the kernel source.

'$KDIR' represent the directory where the kernel source is.

Make will actually change directory to the specified directory

when executed but change back when finished.

make -C $KDIR M=`pwd` M= is used to tell kbuild that an external module is

being built.

The option given to M= is the directory where the external

module (kbuild file) is located.

When an external module is being built only a subset of the

usual targets are available.

make -C $KDIR SUBDIRS=`pwd` Same as M=. The SUBDIRS= syntax is kept for backwards compatibility.

2.4. Preparing the kernel tree for module build

To make sure the kernel contains the information required to

build external modules the target 'modules_prepare' must be used.

'modules_prepare' 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.

2.5. Building separate files for a module It is possible to build single files which are part of a module.

This works equally well for the kernel, a module and even for

external modules.

Examples (module foo.ko, consist of bar.o, baz.o): make -C $KDIR M=`pwd` bar.lst

make -C $KDIR M=`pwd` bar.o

make -C $KDIR M=`pwd` foo.ko

make -C $KDIR M=`pwd` /

3. Example commands

This example shows the actual commands to be executed when building an external module for the currently running kernel. In the example below, the distribution is supposed to use the facility to locate output files for a kernel compile in a different directory than the kernel source - but the examples will also work when the source and the output files are mixed in the same

directory.

Kernel source /lib/modules/<kernel-version>/source -> /usr/src/linux-<version>

Output from kernel compile /lib/modules/<kernel-version>/build -> /usr/src/linux-<version>-up

Change to the directory where the kbuild file is located and execute the following commands to build the module:

```
cd /home/user/src/module
make -C /usr/src/`uname -r`/source \ O=/lib/modules/`uname-r`/build \
M=`pwd`
```

Then, to install the module use the following command:

```
make -C /usr/src/`uname -r`/source \ O=/lib/modules/`uname-r`/build \
M=`pwd` \ modules_install
```

If you look closely you will see that this is the same command as listed before - with the directories spelled out.

The above are rather long commands, and the following chapter lists a few tricks to make it all easier.

4. Creating a kbuild file for an external module

kbuild is the build system for the kernel, and external modules must use kbuild to stay compatible with changes in the build system and to pick up the right flags to gcc etc.

The kbuild file used as input shall follow the syntax described in Documentation/kbuild/makefiles.txt. This chapter will introduce a few more tricks to be used when dealing with external modules.

In the following a Makefile will be created for a module with the following files:

8123_if.c

8123_if.h

8123_pci.c

8123_bin.o_shipped <= Binary blob

4.1. Shared Makefile for module and kernel

An external module always includes a wrapper Makefile supporting building the module using 'make' with no arguments.

The Makefile provided will most likely include additional
functionality such as test targets etc. and this part shall
be filtered away from kbuild since it may impact kbuild if
name clashes occurs.

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
KERNELDIR := /lib/modules/`uname -r`/build
all:: $(MAKE) -C $(KERNELDIR) M=`pwd` $@
# Module specific targets
genbin: echo "X" > 8123_bin.o_shipped
endif
```

In example 1, the check for KERNELRELEASE is used to separate
the two parts of the Makefile. kbuild will only see the two
assignments whereas make will see everything except the two
kbuild assignments.

In recent versions of the kernel, kbuild will look for a file named Kbuild and as second option look for a file named Makefile.

Utilising the Kbuild file makes us split up the Makefile in example 1 into two files as shown in example 2:

Example 2:

```
--> filename: Kbuild
```

```
obj-m    := 8123.o 8123-y := 8123_if.o 8123_pci.o 8123_bin.o
--> filename: Makefile
KERNELDIR := /lib/modules/`uname -r`/build
all:: $(MAKE) -C $(KERNELDIR) M=`pwd` $@
# Module specific targets
genbin: echo "X" > 8123_bin.o_shipped
```

In example 2, we are down to two fairly simple files and for simple files as used in this example the split is questionable. But some external modules use Makefiles of several hundred lines and here it really pays off to separate the kbuild part from the rest.

Example 3 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),)
include Kbuild
else
# Normal Makefile
KERNELDIR := /lib/modules/`uname -r`/build
all:: $(MAKE) -C $(KERNELDIR) M=`pwd` $@
# Module specific targets
genbin: echo "X" > 8123_bin.o_shipped
endif
```

The trick here is to include the Kbuild file from Makefile, so
if an older version of kbuild picks up the Makefile, the Kbuild
file will be included.

4.2. Binary blobs included in a module

Some external modules needs to include a .o as a blob. kbuild
has support for this, but requires the blob file to be named
<filename>_shipped. In our example the blob is named

8123_bin.o_shipped and when the kbuild rules kick in the file

8123_bin.o is created as a simple copy off the 8213_bin.o_shipped file with the _shipped part stripped of the filename.

This allows the 8123_bin.o filename to be used in the assignment to

the module.

Example 4: obj-m := 8123.o 8123-y := 8123_if.o 8123_pci.o 8123_bin.o

In example 4, there is no distinction between the ordinary .c/.h files and the binary file. But kbuild will pick up different rules to create the .o file.

5. Include files

Include files are a necessity when a .c file uses something from other .c files (not strictly in the sense of C, but if good programming practice is used). Any module that consists of more than one .c file will have a .h file for one of the .c files.

- If the .h file only describes a module internal interface, then the .h file shall be placed in the same directory as the .c files.

- If the .h files describe an interface used by other parts of the kernel located in different directories, the .h files shall be located in include/linux/ or other include/ directories as appropriate.

One exception for this rule is larger subsystems that have their own directory under include/ such as include/scsi. Another exception is arch-specific .h files which are located under include/asm-$(ARCH)/*.

External modules have a tendency to locate include files in a separate include/ directory and therefore need to deal with this in their kbuild file.

5.1. How to include files from the kernel include dir

When a module needs to include a file from include/linux/, then one just uses:

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

kbuild will make sure to add options to gcc so the relevant

directories are searched.

Likewise for .h files placed in the same directory as the .c file.

#include "8123_if.h"

will do the job.

5.2. External modules using an include/ dir

External modules often locate their .h files in a separate include/ directory although this is not usual kernel style. When an external module uses an include/ dir then kbuild needs to be told so.

The trick here is to use either EXTRA_CFLAGS (take effect for all .c files) or CFLAGS_$F.o (take effect only for a single file).

In our example, if we move 8123_if.h to a subdirectory named include/ the resulting Kbuild file would look like:

--> filename: Kbuild

obj-m := 8123.o

EXTRA_CFLAGS := -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 kbuild limitation: there must be no space present.

5.3. External modules using several directories

If an external module does not follow the usual kernel style, but decides to spread files over several directories, then kbuild can

handle this too.

Consider the following example:

|

+- src/complex_main.c

| +- hal/hardwareif.c

| +- hal/include/hardwareif.h

+- include/complex.h

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

Kbuild: obj-m := complex.o

complex-y := src/complex_main.o

complex-y += src/hal/hardwareif.o

EXTRA_CFLAGS := -I$(src)/include

EXTRA_CFLAGS += -I$(src)src/hal/include

kbuild knows how to handle .o files located in another directory - although this is NOT recommended practice. The syntax is to specify

the directory relative to the directory where the Kbuild file is located.

To find the .h files, we have to explicitly tell kbuild where to look for the .h files. When kbuild executes, the current directory is always the root of the kernel tree (argument to -C) and therefore we have to tell kbuild how to find the .h files using absolute paths.

$(src) will specify the absolute path to the directory where the

Kbuild file are located when being build as an external module. Therefore -I$(src)/ is used to point out the directory of the Kbuild file and any additional path are just appended.

6. Module installation

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

/lib/modules/$(KERNELRELEASE)/kernel

External modules are installed in the directory:

/lib/modules/$(KERNELRELEASE)/extra

6.1. INSTALL_MOD_PATH

Above are the default directories, but as always, some level of customization is possible. One can prefix the 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 in the example above, can be specified on the command line when calling make. INSTALL_MOD_PATH has effect both when installing modules included in

the kernel as well as when installing external modules.

6.2. INSTALL_MOD_DIR

When installing external modules they are by default installed to a directory under /lib/modules/$(KERNELRELEASE)/extra, but one may wish

to locate modules for a specific functionality in a separate

directory. For this purpose, one can use INSTALL_MOD_DIR to specify an alternative name to 'extra'.

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

7. Module versioning & Module.symvers

Module versioning is enabled by the CONFIG_MODVERSIONS tag.

Module versioning is used as a simple ABI consistency check. The Module versioning creates a CRC value of the full prototype for an exported symbol and when a module is loaded/used then the CRC values contained in the kernel are compared with similar values in the module. If they are not equal, then the kernel refuses to load the module.

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

7.1. 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 symbols, the corresponding CRC value

is stored too.

The syntax of the Module.symvers file is: <CRC> <Symbol> <module> Sample: 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 both from vmlinux and all modules 2) It lists the CRC if CONFIG_MODVERSIONS is enabled

7.2. 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 and to obtain all

symbols, modpost reads Module.symvers from the kernel.

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.

7.3. Symbols from another external module

Sometimes, an external module uses exported symbols from another external module. Kbuild needs to have full knowledge on all symbols

to avoid spitting out warnings about undefined symbols.

Three solutions exist to let kbuild know all symbols of more than

one external module.

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' and 'bar', and 'foo' needs

symbols from 'bar', then one can use a common top-level kbuild

file so both modules are compiled in same build.

Consider following directory layout:

./foo/ <= contains the foo module

./bar/ <= contains the bar module

The top-level Kbuild file would then look like:

#./Kbuild: (this file may also be named Makefile) obj-y := foo/ bar/

Executing: make -C $KDIR M=`pwd`

will then do the expected and compile both modules with full

knowledge on symbols from both modules.

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 module 'bar', one can copy the

Module.symvers file from the compilation of the 'bar' module

to the directory where the 'foo' module 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 in the Makefile 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 Makfile. These files will be

loaded by modpost during the initialisation of its symbol

tables.

8. Tips & Tricks

8.1. Testing for CONFIG_FOO_BAR

Modules often need to check for certain CONFIG_ options to decide if

a specific feature shall be included in the module. When kbuild is used 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 shall use kbuild when building and therefore can use the same methods as in-kernel modules when

testing for CONFIG_ definitions.