Working with Kernel Modules Lab

© 2013 Wind River Systems, Inc

WIND RIVER

Working with Kernel Modules Lab

Objective

In this lab, you will learn how to manage kernel modules in your projects, as well as how to develop new modules.

NOTE: This lab should take approximately 60 minutes.

Lab Overview

Loadable kernel modules are the equivalent of shared libraries in user-space. They provide additional functionality to the kernel when required, without having to pay the penalty in terms of memory (and boot-time) when the module is not in use. They are particularly useful for implementing device drivers and file systems, where compiling support for every type of file system or device statically into the kernel would be prohibitive.

In this lab, you will learn how to do the following:

- *configure the kernel build to add/remove modules*
- *load/unload modules on the target*
- *develop new modules*

Before You Begin

This lab modifies the kernel source of an existing platform project. Use the pre-built project provided with your lab environment. Refer to the *Getting Started* lab to identify where to find this project in your lab environment. The location of this project will be referenced as **$TARGET_BASE** throughout this lab.

This lab uses a command-line environment. Take the time now to open a command-line terminal and navigate to this directory.

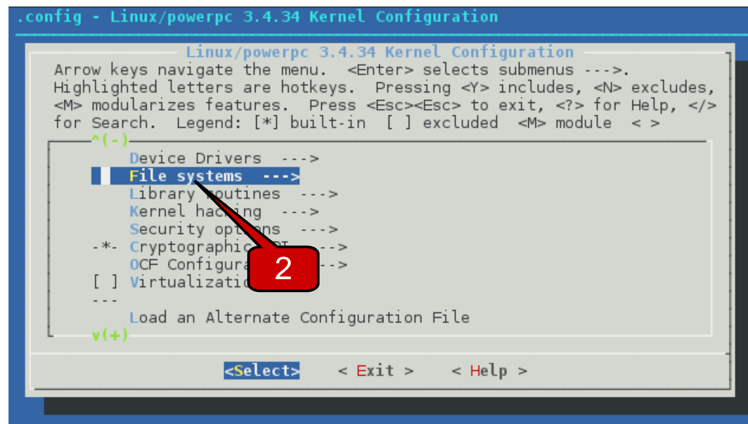
Managing Modules in the Kernel Build

In this section, you will modify your kernel configuration to include a new feature which was previously disabled. This feature will be enabled as a module. The feature chosen for this exercise is the **NILFS2** file system, which would enable your target to mount a file system formatted using **NILFS2**. You may choose others if you wish.

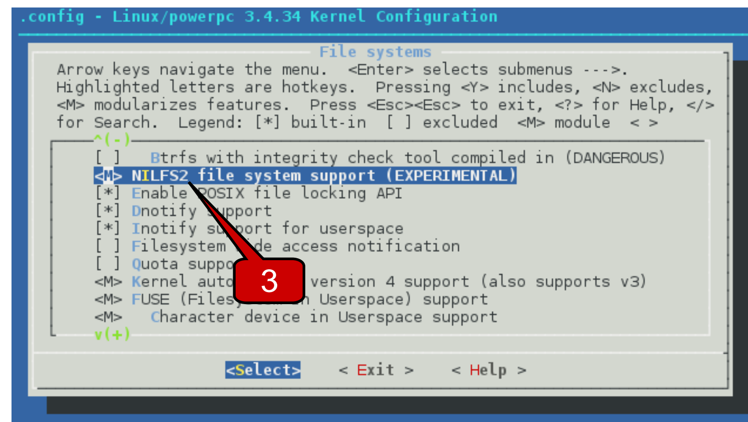
1. Execute the following command to invoke the **menuconfig** interface to configure your kernel.

```
make -C build linux-windriver.menuconfig
```

2. In the **Kernel Configuration** dialog, select the **File systems** group.

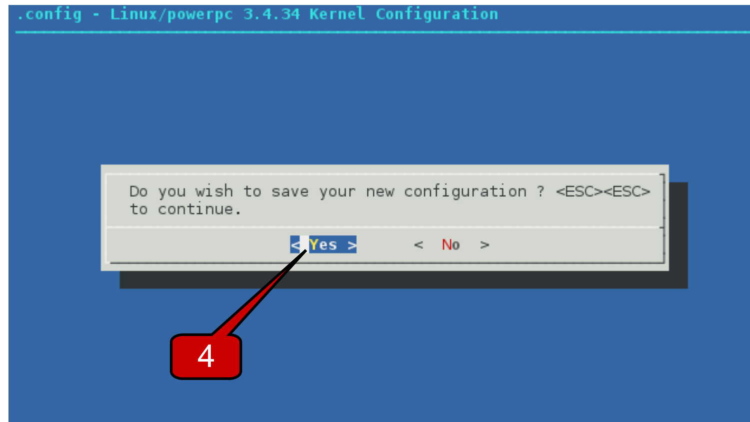


3. Use the down arrow key to scroll and select the **NILFS2 file system support (EXPERIMENTAL)** menu (**NILFS2_FS**), then press **M** to enable this feature to be built as a module.



NOTE: Some features cannot be compiled as modules; they can only be enabled (statically compiled into the kernel image) or disabled.

4. Exit from **Device Drivers** group and **Kernel Configuration** window. Select **<Yes>** to save the changes.



5. Execute the following command to rebuild your kernel to incorporate the configuration changes you made in the previous step.

```
make -C build linux-windriver.rebuild
```

Locating Modules

Enabling an option as a module results in a corresponding module getting built. Kernel module files (**.ko**) are built by the kernel build system, and contain the binary code implementing a particular module. Although they live in the file system, only the kernel knows how to load and link them into its run-time environment.

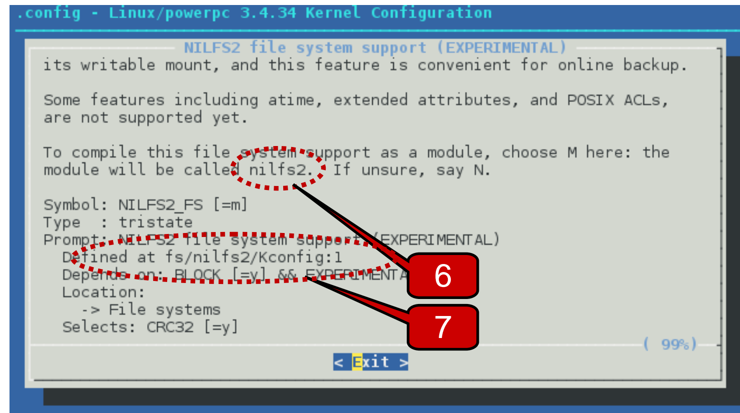
Unfortunately there is no one-to-one mapping between the name of the option (as it appears in the configuration interface) and the resulting module file name. This can be problematic if you wish to audit the module, or manually install it at a later time.

This section will provide some techniques on how to locate the module corresponding to a particular option. But if you wish to audit the module or manually install it later, you will need to know the name of the actual file implementing that module.

6. Most kernel configuration interfaces provide the ability to display information associated with a particular option. In the **menuconfig**, place the cursor on the option and press the **<?>** key to display the information.

   Look for a name disclosure within the text. Most options provide this information, but it's not strictly mandatory.

7. If the option you're working with is not well documented, take a note of the location where the option is defined, as this will provide you with the information you need to track the module name down.



8. Refer to the makefile in the same directory containing the Kconfig file noted in the previous step. The makefile will be found in the extracted source directory, **build/linux-windriver-3.4-r0/linux/fs/nilfs2/Makefile**.

   Locate the target within the file which switches on the configuration option (**CONFIG_NILFS2_FS**). The object on the right hand side of the expression tells you the name of the **.o** file that will be generated. The **.ko** file will inherit this name as well, providing you with your answer.

```
obj-$(CONFIG_NILFS2_FS) += nilfs2.o
nilfs2-y := inode.o file.o dir.o super.o namei.o page.o mdt.o \
        btnode.o bmap.o btree.o direct.o dat.o recovery.o \
        the_nilfs.o segbuf.o segment.o cpfile.o sufile.o \
        ifile.o alloc.o gcinode.o ioctl.o
```

9. To verify that the module is getting built, scan the kernel build directory for signs of the module file. Note that the command below should be adjusted depending on the board selection in your project.

```
find build/linux-windriver-3.4-r0/linux-$BSP-standard-build \
     -name nilfs2.ko
…/fs/nilfs2/nilfs2.ko
```

10. With the new kernel module built, rebuild your image and deploy it to the target in preparation for the next section. Please refer to the *Getting Started* lab for more details, if needed.

Managing Modules on the Target

Kernel modules and be inserted and removed on the fly. Although the actual loading and linking of the module is handled by the kernel, the request is initiated in user-space using standard user-space utilities. Furthermore, the kernel module image resides in user-space, on an attached file system.

Unless otherwise noted, perform the steps in this section in a shell running on the target.

11. *On the target*, execute the following command to show which modules are currently loaded.

```
lsmod
    Not tainted
```

It may surprise you to see that no modules are loaded. Remember that crucial functionality (functionality required to boot) is compiled statically into the kernel. Drivers which are compiled statically into the kernel do not show up in **lsmod** output.

The string **not tainted** refers to the **taint state** of the kernel (which can also be read directly from the file **/proc/sys/kernel/tainted**). Taint state is triggered when untrusted modules are loaded, and has an impact on the reliability of kernel problem reports, and how effective source code analysis can help solve the problem. Common situations that can trigger the taint state include:

- loading a module with a license that is not GPL-compatible (source code is not available)
- loading an out-of-tree module
- forcibly loading a module (bypassing normal sanity checks for version information, etc)
- forcibly removing a module without applying proper dependency checks

12. Execute the following command to load your module into the kernel:

```
modprobe nilfs2
NILFS version 2 loaded
```

NOTE: Modules can also be loaded using a utility called **insmod**. However, **modprobe** is recommend since it provides additional important functionality:

- Supports policies which can be configured via files and the boot line
- Satisfies module dependencies by loading additional modules if required
- Provides addition verbosity options

13. Execute the command **lsmod** to verify that your module is inserted.

```
lsmod
    Not tainted
nilfs2 172327 0 - Live 0xc92fc000
```

14. Execute the following command to unload your module.

```
modprobe -r nilfs2
```

NOTE: You can also using the counterpart to **insmod**, **rmod** to unload modules. However for the reasons listed above, it is recommend to use **modprobe** instead.

Developing Out-of-Tree Modules

In these exercises you will develop a dummy network driver. The code will consist of a very basic kernel module; it will not actually drive any packets. The approach outlined here will apply for kernel modules of any level of complexity, including real drivers.

When developing new drivers or other kernel functionality, it's usually more convenient to begin development in your own personal working directory, outside of the kernel source tree (which is where the name out-of-tree comes from). This lets you focus your effort on getting your driver going, rather than integration issues.

The approach outlined here will involve developing your module within the local layer, where your code will be readily accessible by the build system without further configuration. A recipe will then build your module as a package which you can add or remove from your image, just like any other package.

The following list highlights the workflow for creating your new module:

- Create a working directory to group all your files.
- Create a new source file, populated with code that implements your module's functionality.
- Create a makefile to drive the build of your module.
- Create a bitbake recipe to integrate the module to the build system.

After the module builds cleanly, you can add it to your image, deploy, and test.

15. Establish a working directory to contain your source, within the local layer.

```
mkdir -p layers/local/recipes-kernel/mynet
```

The meaning of the path components are as follows:

- **layers/local** represents the top-level directory of the local layer. This path component already exists in every platform project.
- **recipes-kernel** is a new directory in the local layer which can be used to collect any configuration information (recipes, source, etc) related to the kernel. The naming of this directory is somewhat arbitrary; provided the directory name begins with **recipes-**, any standard layer will recognize it as a directory providing such configuration.
- The **mynet** component will house the source for your module. The name does not have to match the name of your module, but it's not a bad idea for them to match.

16. Within the **mynet** directory, create **mynet.c** and populate it with the source for your module. The code consists of:

    - An **init()** function, which merely prints a message to the kernel log buffer. The kernel will call this function immediately after your module is loaded. If you ever end up building this code statically into the kernel, then your **init** function will be called during kernel initialization.
    - An **exit()** function, which the kernel calls when unloading your module. If your code ever ends up getting statically built into the kernel, the **exit** function will not be called.
    - The **module_init()** and **module_exit()** macros publish your module's entry points.
    - The **MODULE_LICENSE()** macro publishes the license for your module. The license your module publishes has a legal implication as to how you treat the source code, and it also has a run-time and support implications; GPL-incompatible modules have access to a restricted set of kernel APIs, and loading them will "taint" the kernel (discussed in the previous section).

    A copy of the source can be found in **/Labs/Kernel/answer/mynet/mynet.c**.

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

static int __init mynet_init(void)
{
   printk("Hello from mynet\n");
   return 0;
}

static void __exit mynet_exit(void)
{
   printk("Goodbye from mynet\n");
}

module_init(mynet_init);
module_exit(mynet_exit);
MODULE_LICENSE("GPL");
```

17. Within the same directory, create the makefile for your module, which consists of:
    - An **obj-m** expression which tells the build system which objects to build into this module; the module will be named according to the object name.
    - A macro definition which caches the current working directory into a variable named **SRC**.
    - A default rule **all** which invokes the main kernel makefile, passing the current working directory as a variable named **M**. This is the standard way to tell the kernel build system to build an out-of-tree module located in a specified directory.
    - A **modules_install** rule which operates identically to the **all** target.
    - A **clean** rule which deletes any temporary or build-related files.

    A copy of the source can be found in **/Labs/Kernel/answer/mynet/Makefile**.

NOTE: If you copy and paste from this document, be aware that Makefiles are notoriously picky with regards to formatting. The lines below must be indented with a single **TAB** character, not spaces.

```
obj-m := mynet.o

SRC := $(shell pwd)

all:
	$(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
	$(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
	rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
	rm -f Module.markers Module.symvers modules.order
	rm -rf .tmp_versions Modules.symvers
```

18. Before proceeding, you must address the requirement of a license file, because all bitbake packages require a license file. It might be tempting to short-circuit this requirement by using a source file or makefile as the license file – and this approach is perfectly valid the file actually contains the license text. But don't be tempted to use a plain .c file (or similar) as your "license file". There are two good reasons to avoid this:

    - From the point of view of being an active contributor in the Yocto Project, falsifying license information in a package is a bad habit to develop
    - If the .c file (or whichever file you choose) is changed – which is likely during development – it forces you to update the license checksum value in the recipe which, in the end, produces more work for you.

    Invest the short amount of time now to set up a license file. Many are already included and ready to use, but be aware that some might require minor edits (replacing company names, deleting instructions, etc).

```
cp layers/oe-core/meta/files/common-licenses/GPL-2.0 \
   layers/local/recipes-kernel/mynet/COPYING
```

    Review the file, and make any edits needed to the parts of the document outside the actual license text. You should never change the text within the license.

19. Within the same directory, create the file **mynet_0.1.bb** and populate it with the recipe to build your module. The recipe contains:
    - Package information, including the **LICENSE** and **LIC_FILES_CHKSUM**
    - The statement **inherit module**, which leverages the a ready-to-use bitbake class that implements all functionality required to build a kernel module package
    - **PR** and **PV** settings that denote package revision and version numbers, respectively
    - **SRC_URI** which lists the files required to build the package
    - Set **S**, telling bitbake where to copy files from

    A copy of the source can be found in **/Labs/Kernel/answer/mynet/mynet_0.1.bb**.

```
DESCRIPTION = "Dummy network driver."
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5="

inherit module

PR = "r0"
PV = "0.1"

SRC_URI = "file://Makefile \
           file://mynet.c \
           file://COPYING \
          "

S = "${WORKDIR}"
```

20. Populate the md5 portion of **LIC_FILES_CHKSUM** by obtaining the checksum of the license file.

```
md5sum layers/local/recipes-kernel/mynet/COPYING
801f80980d171dd6425610833a22dbe6  …/COPYING
```

    Transfer the value you obtain into your recipe:

```
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=801f80980d171dd642561…"

inherit module
…
```

21. With all the pieces in place, attempt the first build of your module.

```
make -C build mynet
```

22. Once the build succeeds, add the **mynet** package to your image:

```
make -C build mynet.addpkg
```

23. Now do the following to test your module:

- Build your image with the command **make**.
- Deploy your image to the target, as outlined in the *Building and Customizing a Wind River Linux Platform* lab.
- On the target, insert the **mynet** module as outlined in the *Managing Modules on the Target* section.

Migrating Modules into the Tree

In the previous section, you learned how to develop modules outside the kernel tree. Although you may choose to keep your modules out-of-tree, moving them into the tree carries some benefits:

- you can easily share your module with others (or push upstream) in the form of a kernel patch
- accessibility to a wider audience, since a kernel patch can be applied to any kernel source tree, whereas the module in its current form can only be shared with bitbake users
- support for configuration parameters within your module
- you can build your module statically into the kernel image
- loading a module will not taint the kernel

In this section, you will migrate the module you developed earlier into the kernel tree.

24. Begin by removing the module package from your image, as well as any traces left over in the build environment.

```
make -C build mynet.rmpkg mynet.distclean
```

25. Next, ensure that the kernel source is cleaned and re-checked out, with any applicable patches applied. Performing a **distclean** at this stage is important, since objects in the sstate cache may interfere when rebuilding the kernel later on in this section.

```
make -C build linux-windriver.distclean linux-windriver.patch
```

26. Copy the contents of your module development directory into the appropriate place within the kernel source. Since the module you developed was modeling a network device, **drivers/net** is an appropriate place.

```
cp -a layers/local/recipes-kernel/mynet \
   build/linux-windriver-3.4-r0/linux/drivers/net
```

Not all files are required — you can omit the following:

- **mynet_0.1.bb**, since the code now falls under the domain of the kernel build system, not bitbake
- **COPYING**, since all in-tree source is subject to the kernel license, GPLv2

27. Create a **Kconfig** file within the **mynet** directory which you created in the previous step.

The file contains a single **tristate** entry. Tristate options allow the feature to be compiled statically, as a module, or disabled.

```
config NET_MYNET
        tristate "Mynet example"
        ---help---
          This is a kernel module example.

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

28. The makefile for your module can now be greatly simplified. In an out-of-tree module, the makefile must do some basic set-up and then invoke the kernel build system. In this scenario, the situation is reversed; the build of your module is driven by the kernel build system. Essentially, the makefile can be boiled down to a single line which switches on **CONFIG_NET_MYNET**:

```
obj-$(CONFIG_NET_MYNET) := mynet.o
```

This concludes the work that must be done to your module in order to integrate it into the rest of the kernel. The actual integration consists of linking your new module into the kernel configuration and build systems, which you will perform next.

29. Incorporate your module into the kernel configuration system by adding a reference to your module's **Kconfig** file to the **Kconfig** file in the module's parent directory; that is, **build/linux-windriver-3.4-r0/linux/drivers/net/Kconfig**.

Until you fully understand the structure of the **Kconfig** files, it's advisable to add the entry near the bottom as shown:

```
source "drivers/net/hyperv/Kconfig"
source "drivers/net/mynet/Kconfig"
endif # NETDEVICES
```

30. Now, perform a similar task with respect to the kernel build system. In this case, you will add a reference to your module's directory to the makefile in the module's parent directory; that is, **build/linux-windriver-3.4-r0/linux/drivers/net/Makefile**.

    Notice that the reference is conditional on **CONFIG_NET_MYNET**; structuring the reference this way means that the build system will not even enter into your module's directory if **CONFIG_NET_MYNET** is not set (to **Yes** or **Module**).

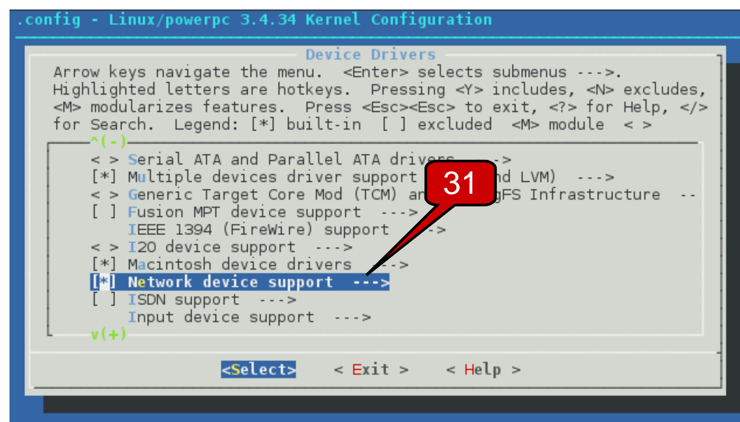
    As before, add the entry to the bottom of the file.

```
obj-$(CONFIG_USB_CDC_PHONET)    += usb/
obj-$(CONFIG_HYPERV_NET) += hyperv/
obj-$(CONFIG_NET_MYNET) += mynet/
```

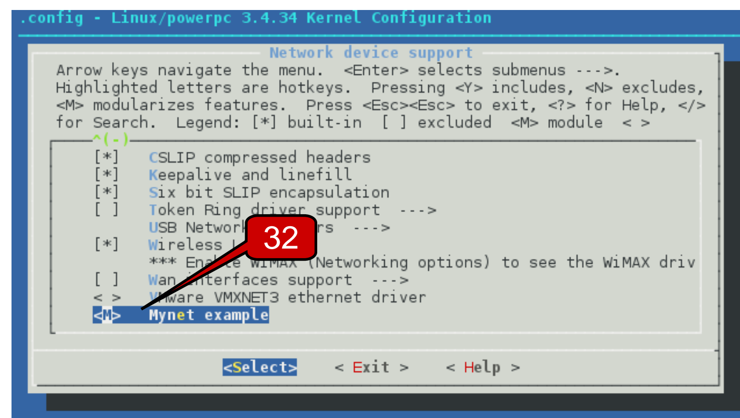
31. With your new module fully integrated, you can now configure your module into the kernel build.

```
make -C build linux-windriver.menuconfig
```

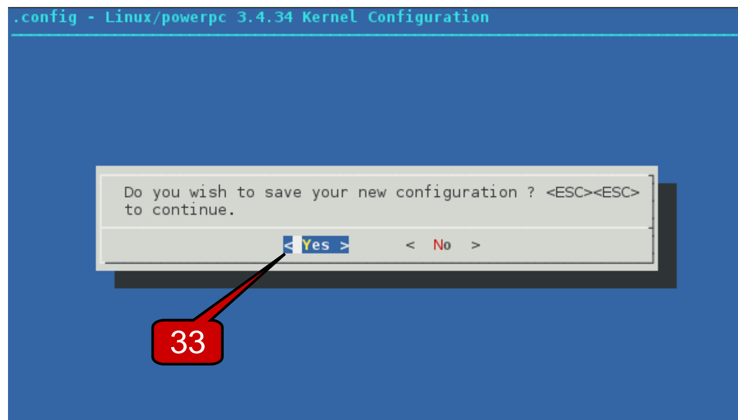
    In the **Kernel Configuration** window, select the **Device Drivers** group. Continue scrolling down and select the **Network device support** subgroup.



32. Scroll down and locate the cursor on **Mynet example** and press **M**.



33. Exit from **Network device support** subgroup, **Device Drivers** group, **and Kernel Configuration** window. Select **<Yes>** to save the changes.



34. Execute the following command to rebuild the kernel with your changes.

```
make -C build linux-windriver.rebuild
```

35. You may now test your new in-tree module:

- Build your image with the command **make**
- Deploy your image to the target, as outlined in the *Getting Started* lab.
- On the target, insert the **mynet** module as outlined in the *Managing Modules on the Target* section.

Conclusion

You will lose the work you have done here if you run **distclean** (or **cleansstate**) on your kernel. If you want to capture your in-tree module permanently, you must capture a patch containing your additions and modifications:

- the modifications to **drivers/net/Kconfig** and **drivers/net/Makefile**
- the addition of the directory **drivers/net/mynet**, and all files contained within

Refer to the *Generating Kernel Patches* lab for instructions on how to capture your changes into a patch. Refer to the *Configuring and Patching the Kernel* lab for steps on how roll your patch into a layer.

This concludes the lab. Do not proceed.