Building Custom Kernel Modules

Chris Jenkins - Genesi USA, Inc

Terminology

- “Kernel Modules” are pieces of software (usually device drivers) that are not a part of the Linux kernel proper but are executed in kernel space.
- “Built in” modules are compiled into the kernel image. They are loaded when the system boots
  - e.g. the MMC driver for the Efika MX53 (or the Drónov Slimbook)
- “Dynamically loaded” modules are loaded / removed during system run-time, either as needed or manually using commands from `kmod` package (e.g. `insmod`, `rmmod`)
- `*.ko`: kernel object files; the compiled kernel module

Prerequisites

- The same software tools needed to compile the Linux kernel (`make`, `gcc`)
  - I will assume you are cross-compiling modules, but in theory you should be able to build them on an Efika MX53 system (such as the Drónov Slimbook) itself
- `kmod`: package with utilities for loading / unloading kernel modules.
  - `insmod`: insert module
  - `rmmod`: remove module
  - `lsmod`: show loaded modules
  - `modinfo ${KO_FILE}`: print metadata of module

If you are building on an Efika MX53 system, you can just download the Linux kernel headers (much more feasible than downloading the kernel source) and build against those

```
$ sudo apt-get install linux-headers-`uname -r`
```

“Hello world” Module[1]

- Create a directory called “hello-module” and change to it
  - This example will create it in the root of the kernel source repo itself
- Create files `hello.c` and `Makefile` and open both with your favorite text editor
- Code for both files in following slides

1: Code taken from “The Linux Kernel Programming Guide” http://www.tldp.org/LDP/lkmpg/2.6/html/

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "\"Hello world\" #1\n");

        /*
         * Non-zero means kernel init failed, could not be loaded
         */
        return 0;
}

void cleanup_module(void)
{
        printk("Goodbye, cruel world #1\n");
}
```

`hello-1.c`: Concepts

- `init` and `exit` functions
- Log levels (`KERN_INFO`)
- `printk`

```
1 obj-m += hello-1.o
2
3 all:
4     make -C .. M=$(PWD) ARCH=arm CROSS_COMPILE=arm-none-eabi- modules
5
6 clean:
7     make -C .. M=$(PWD) ARCH=arm CROSS_COMPILE=arm-none-eabi- clean
```

`Makefile`: Concepts

- `obj-y`, `obj-n`, `obj-m`: include object into kernel image, exclude object from kernel image, and compile object as kernel module
  - This is what `obj-$(CONFIG_MYOPTION)` is all about. We'll worry about kernel configuration another time.

- arguments to `make`:
  - `-C ${DIR}`: change directory before executing. This should be the directory with the kernel headers
  - `M=$(PWD)`: Location of module
  - `ARCH=${arch}`, `CROSS_COMPILE=${ccc}`: Same as with building Linux kernel

Compile & Insert `hello-1`

- In your module directory, type `make`
- If successful, your directory will have the following files:

  ```
  hello-1.c   hello-1.mod.c  hello-1.o  Module.symvers
  hello-1.ko  hello-1.mod.o  modules.order
  ```

- Check the module meta-data:

  ```
  modinfo hello-1.ko
  ```

- Move `hello-1.ko` to your Efika MX53 system (such as a Drónov Slimbook), and insert (from terminal): `sudo insmod hello-1.ko`
- There will be no console output. Instead, check kernel logs

  ```
  dmesg | tail
  ```

- And remove:

  ```
  sudo rmmod hello_1
  ```

  - Note that the dash was converted to an underscore!

Extended example

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

#define DRIVER_AUTHOR   "Chris Jenkins"
#define DRIVER_DESC     "Example module"

static int hello_extended_data __initdata = 1337;

static int __init init_hello_extended(void)
{
        printk(KERN_INFO "Hello world!\n");
        printk(KERN_INFO "Here's a number for you: %d\n", hello_extended_data);
        return 0;
}

static void __exit exit_hello_extended(void)
{
        printk(KERN_INFO "Goodbye world\n");
}

module_init(init_hello_extended);
module_exit(exit_hello_extended);

MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);

MODULE SUPPORTED DEVICE("testdevice");
```

`hello-ext.c`: Concepts

- `init`: macro definitions for special init and exit code
  - `__init` marks your function as an initializer only. After boot the kernel will delete it from active memory. `__init_data` is similar, but for data
  - `__exit` marks your function as cleanup only. If it is a built-in module, this code is also deleted from memory
  - `module_init()`, `module_exit()`: macros marking which functions should be used for init and exit of the module, resp. Using these with more descriptive function names than "`init_module`" and "`exit_module`" is preferred style.

- Licensing and Module Info
  - `MODULE_LICENSE`: Module software license. You will get compile complaints if not GPL-compatible, but only complaints (does not affect code)
  - `MODULE_AUTHOR()`, `MODULE_DESCRIPTION()`: Self explanatory.
  - `MODULE_SUPPORTED_DEVICE()`: Used to document which devices the module is intended for
  - You can test this was configured correctly with `modinfo`

Advanced Example

Here is the code snippet we use to build the wireless driver modules:

```
#build sagrad compat-wireless
kernelrelease=$(cat $currentpath/source/include/config/kernel.release)
cd sagrad-wifi/compat-wireless
make KLIB_BUILD=$currentpath/source -j8 # similar to our -C option
cd $currentpath
mkdir -p modules/lib/modules/$kernelrelease/updates
rsync -avr `find sagrad-wifi/compat-wireless -name '*.ko'`
        modules/lib/modules/$kernelrelease/updates
mkdir -p modules/lib/firmware
rsync -avr sagrad-wifi/FIRMWARE/ modules/lib/firmware
cd modules
tar cjvf ../extra-modules-$kernelrelease.tar.bz2 .
cd $currentpath
rm -rf modules
```

genesi