# introduction to linux kernel modules under yocto

## goal

The goal of this Lab is to allow you to build a simple loadable kernel module using the Yocto build environment.

## prerequisites

To follow this Lab, you need:

1. Raspberry Pi 3 board full;
2. Micro USB cable;
3. 8 GB Micro SD card;
4. USB-to-Serial debug module for Raspberry Pi 3 or USB to TTL adapter;
5. A PC with Ubuntu Desktop 14.04 LTS, or a virtual machine hosting Ubuntu Desktop 14.04 LTS;
6. A Micro SD card reader attached to the PC/virtual machine;
7. (**Optional**) Micro HDMI cable.

## workplace setup

Assuming you completed the first lab, move to the directory ***raspberryPi3*** and prepare the build environment:

cd ~/raspberryPi3/

source sources/poky/oe-init-build-env rpi-build

You now have the system ready for building the embedded Linux distribution for the Raspberry Pi 3 board. We will update the source tree used in the first lab to include a simple device driver.

## Preparing the recipes

In this Lab, we are adding a new device driver to be compiled as Linux loadable kernel module, as you would have to do if you were asked to customize the board support package for the Raspberry Pi 3 to support a new I/O device attached to the system.

For this purpose, we will add a new Yocto recipe to the meta-raspberrypi layer. For this purpose, create the directory as follows:

cd ~/raspberryPi3/sources/meta-raspberrypi/recipes-kernel

mkdir -p hello-mod/files

In the ***hello-mod*** folder, put a file named ***hello\_1.0.bb*** with the following content:

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10: | DESCRIPTION = "hello driver"  LICENSE = "GPLv2+"  LIC\_FILES\_CHKSUM = "file://${BPN}.c;endline=19;md5=4866f9824d27c1cd5324fd5e84caeb6e"  inherit module  PR = "r0"  SRC\_URI = "file://Makefile file://${BPN}.c"  S = "${WORKDIR}" |

This file describes the property of the recipe, and in particular:

* Line 0 gives a simple description of the recipe purpose;
* Lines 1-3 give the reference to the license under which the recipe is distributed and provide the indication of the license file as well as its checksum;
* Line 4 indicates that the recipe inherits the properties of the recipes to build kernel modules;
* Line 6 tells for which machine layer the recipe is intended;
* Line 8 lists the source files needed to build the recipe;
* Line 10 sets the symbol pointing to the working directory used to perform the build operations.

In ***hello-mod/files***, create a file named ***hello.c*** with the following content:

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44: 45:46:47: | #include <linux/cdev.h>  #include <asm/uaccess.h>  #include <linux/module.h>  #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>  static dev\_t hello\_dev;  struct cdev hello\_cdev;  static char buffer[64];  ssize\_t hello\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)  {  printk(KERN\_INFO "Dummy read (count=%d, offser=%d)\n", (int)count, (int)\*f\_pos );  return 1;  }  struct file\_operations hello\_fops = {  .owner = THIS\_MODULE,  .read = hello\_read,  };  static int \_\_init hello\_module\_init(void)  {  printk(KERN\_INFO "Loading HelloWorld\_module\n");  alloc\_chrdev\_region(&hello\_dev, 0, 1, "hello\_dev");  printk(KERN\_INFO "%s\n", format\_dev\_t(buffer, hello\_dev));  cdev\_init(&hello\_cdev, &hello\_fops);  hello\_cdev.owner = THIS\_MODULE;  cdev\_add(&hello\_cdev, hello\_dev, 1);  return 0;  }  static void \_\_exit hello\_module\_cleanup(void)  {  printk(KERN\_INFO "Cleaning-up hello\_dev.\n");  cdev\_del(&hello\_cdev);  unregister\_chrdev\_region(hello\_dev, 1);  }  module\_init(hello\_module\_init);  module\_exit(hello\_module\_cleanup);  MODULE\_AUTHOR("Your Name");  MODULE\_LICENSE("GPL"); |

The file contains the source code of the kernel module;

* Lines 0-7 include the needed header files containing the function prototypes and the data structure needed to write a kernel module;
* Lines 9-42 are identical to what was discussed during lecture for module 5;
* Line 44 declares the function to be used as module initialization function;
* Line 45 declares the function to be used as module cleanup function;
* Lines 46 and 47 declare the author of the source code and the license used to distribute it.

Finally, in the same directory, place a file name ***Makefile*** with the following content (ensure that the lines are indented with tabs, not spaces, or else this will cause errors later in the lab - You will need to manually add the tabs to rows with indentation - copy and paste will cause problems):

|  |  |
| --- | --- |
| 0:  1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14: | obj-m := hello.o  SRC := $(shell pwd)  all:  $(MAKE) -C $(KERNEL\_SRC) M=$(SRC)  modules\_install:  $(MAKE) INSTALL\_MOD\_DIR=kernel/drivers/my\_mod -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 |

The file describes how the source code shall be built and where the output shall be placed, in particular:

* Lines 0-5 define which is the object module to be created, locate the current directory, and invokes the build command, which refers to the source and kernel source symbols Yocto maintains;
* Lines 7-9 define which operation to perform for installing the loadable kernel module resulting from the compilation. In this example, the object will be installed in the root file system in the /lib/modules/<kernel version>/kernel/drivers/my\_mod directory.
* Lines 11-14 define the operation to perform when cleaning the build outputs (i.e., removing the temporary files and the compilation outputs).

Once these operations are completed, you have to tell the machine layer configuration that the new driver is needed. For this purpose, edit the ***local.conf*** file

gedit ~/raspberryPi3/rpi-build/conf/local.conf

Add as last line of the file the following statement:

IMAGE\_INSTALL\_append += "hello"

These lines tell Yocto that when building Linux, the newly created device driver shall be built and that it shall be included in the root file system.

## Building and deploying the new system

You are now ready to build the new system as follows:

cd ～/raspberrypi3/rpi-build

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

During the build, the compiler will recognize a discrepancy in the license reference. Copy and paste the suggested reference (“the new md5 checksum is..”) into your hello\_1.0.bb file where it says "md5=...." and start the build again.

After a while, a new Micro SD card image would be available, which you can deploy in the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly, or use a flash program to automate the process.

While the SD card is still connected to the development host, use the following lines to navigate to the etc folder on the SD card (assuming the device has now been mounted).

cd /media/user/SD\_name/etc – Use the ls command to find the name of the SD card in the user folder.

Then, use a terminal text editor to open the shadow file.

sudo vi shadow

Or

sudo gedit shadow

Check that there are no characters between the first two colons in the first line. If there is, remove it so that the first line looks like this:

Root::17728:0:99999:7:::

Exit the text editor by entering :x or simply closing the application!

## Running the module

After booting the new Linux system, you can check whether the build process was completed successfully. After logging into the Raspberry Pi 3, you can type the following commands:

root@raspberrypi3:/# ls -la /lib/modules/4.1.21/kernel/drivers/my\_mod

drwxr-xr-x 3 root root 1024 May 15 08:47 .

drwxr-xr-x 16 root root 1024 May 10 12:09 ..

-rw-r--r-- 1 root root 5384 May 11 08:05 hello.ko

root@raspberrypi3:/#

The directory in the root file system contains ***hello.ko***, which is the kernel object containing the binary code for the loadable kernel module.

You can now insert the module in the kernel as follows:

root@raspberrypi3:/# insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hello.ko

[ 363.669107] Loading HelloWorld\_module

[ 363.672846] 243:0

The output messages indicate that the module has been loaded correctly, with major number 243 and minor number 0.

For removing the module, you can act as follows:

root@udooneo:/# rmmod hello

[ 475.900838] Cleaning-up hello\_dev.

The message indicates that the module cleanup function has been executed correctly.

# post-lab practice

From this Lab session and onward, we shall implement a bigger challenge using the Raspberry Pi 3 board to do something useful. The overall workload is evenly distributed in each Lab session and failing to complete this session will affect the next two sessions. For this session, we shall mainly practice on kernel modules and BitBake usage.

**Q: Configure the corresponding files to autoload the “hello” kernel module.**

**(Hint: Amend the raspberrypi3.conf file)**

A: Assuming you have completed the lab; take out the SD card from the Raspberry Pi 3 board, plug it into the SD card reader, and connect it to your host PC.

The most crucial step is to amend the file ***raspberrypi3.conf*** under the ***raspberryPi3/sources/meta-raspberrypi/conf/machine*** directory. Open it and add:

KERNEL\_MODULE\_AUTOLOAD += "g\_serial hello"

The “hello” stands for the name of the kernel module to be autoloaded. In case you want to autoload multiple modules, simply include them with a single space in between.

Then, switch to the directory “***raspberryPi3***” and setup the build environment using the following command:

MACHINE=raspberrypi3 source sources/poky/oe-init-build-env rpi-build

Next, make sure you are at the build directory and build the new system using:

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

Copy the new Micro SD card image as done in the labs.

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Now test if the module is autoloaded. Connect your Raspberry Pi 3 board to the host PC and open minicom.

Using the following command should return you an error since the module is already loaded:

insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hello.ko

Alternatively, you can remove the kernel module immediately after booting using the following command. This should be done successfully.

rmmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hello.ko

**Q: Run the “atomic tank” game (**<http://atanks.sourceforge.net/>**) on the Raspberry Pi 3 board.**

**(Hint: Use BitBake to build it from source code)**

A: For this challenge, we will use atomic tank under GPL license as an example.

Firstly, we need to make a piece of recipe for the game in the source folder of your Yocto Project folder. You can either write your own recipe or search and download from: <https://layers.openembedded.org/layerindex/branch/master/layers/>

A recipe can be downloaded using the following command:

git clone <git repository address>

You can download the ***meta-games*** layer by navigating to the ***sources*** folder and then running:

git clone git://github.com/cazfi/meta-games.git

If you are adding a layer, make sure you amend the ***bblayers.conf*** file at ***build/conf/*** to add corresponding path to the layer. In our case, we added a layer called ***meta-games***, and hence we include this line:

${BSPDIR}/sources/meta-games

You also need to make sure that the dependencies of such a layer are satisfied. In this example, the ***meta-games*** layer depends on the ***meta-openembedded/meta-filesystems***.

${BSPDIR}/sources/meta-openembedded/meta-filesystems \

Before we begin building anything - we need to get rid of some extra things that came with the meta-games layer that may cause errors if left in the recipe.

Start by deleting:

***/home/user/raspberryPi3/sources/meta-games/recipes-images/usbistic/***

and

***/home/user/raspberryPi3/sources/meta-games/recipes-images/bistic-devel/***

Then go to:

***/home/user/raspberryPi3/sources/meta-games/recipes-games***

and delete every folder apart from "atanks".

Then we need to alter the ***rpi-build/conf/local.conf*** file to append ***opengl*** and ***x11*** as there are dependencies for these packages:

MACHINE ?= "raspberrypi3"

PREFERRED\_VERSION\_linux-raspberrypi = "4.%"

DISTRO\_FEATURES\_remove = " wayland"

DISTRO\_FEATURES\_append = " systemd opengl x11"

VIRTUAL-RUNTIME\_init\_manager = "systemd"

ENABLE\_UART = "1"

IMAGE\_INSTALL\_append += "hello"

You can display all the recipes in your current build to make sure you have what you need. Once everything is ready and included, you can bitbake the packages of the game.

bitbake-layers show-recipes

bitbake atanks

After the build is finished, you can locate the packages at:

***rpi-build/tmp-glibc/deploy/rpm/cortexa7hf\_neon\_vfpv4/ - Note this may vary on your environment.***

The “atomic tank” game is built with 3 packages as the output:

***- atanks-5.7-r0.cortexa7hf\_neon\_vfpv4.rpm***

***- atanks-dbg-5.7-r0.cortexa7hf\_neon\_vfpv4.rpm***

***- atanks-dev-5.7-r0.cortexa7hf\_neon\_vfpv4.rpm***

Among these, the first package is the essential one to make the game run.

Now we have the packages ready; the next task is including all the dependencies needed by the game. If you downloaded the game recipe from openembedded.org, you can check the .bb file of your recipe to see the dependencies. In our case, ***atomic tank*** requires ***allegro4*** for the kernel image. To include this, amend the ***build/conf/local.conf*** file and add:

IMAGE\_INSTALL\_append= " allegro4 atanks"

Then, add rpm package manager to the image. For sufficient space to install the game, also add:

EXTRA\_IMAGE\_FEATURES= "package-management”

PACKAGE\_CLASSES ?= "package\_rpm"

IMAGE\_ROOTFS\_EXTRA\_SPACE = " 300000"

The build/conf/local.conf file should now look similar to this:

MACHINE ?= "raspberrypi3"

PREFERRED\_VERSION\_linux-raspberrypi = "4.%"

DISTRO\_FEATURES\_append = " systemd x11 wayland opengl allegro4"

VIRTUAL-RUNTIME\_init\_manager = "systemd"

IMAGE\_INSTALL\_append= " allegro4 atanks"

CORE\_IMAGE\_EXTRA\_INSTALL += "allegro4"

ENABLE\_UART = "1"

EXTRA\_IMAGE\_FEATURES= "package-management"

PACKAGE\_CLASSES ?= "package\_rpm"

IMAGE\_ROOTFS\_EXTRA\_SPACE = " 300000"

Before we build the image, we need to ensure our system is up-to-date with the latest packages. Run these commands:

sudo apt-get update

sudo apt-get upgrade

Since we are running a graphical game, we need a kernel image with desktop environment enabled. There is an option already available with the Yocto project called “***core-image-sato***”. When everything is ready, you can build your image:

bitbake core-image-sato

When your image is ready, populate it onto the SD card as done in the lab, the image should be found here:

/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/core-image-sato-raspberrypi3.rpi-sdimg

Now you can power up the Raspberry Pi 3 and copy the game packages to the root file system. You can do it with “scp” if you have network connected and configured, or simply copy it to the SD card with a card reader. Next, you need to install the package:

rpm -ivh atanks-5.7-r0.cortexa9hf\_vfp\_neon.rpm

It should install the game if all the dependencies are included in your kernel image. To start the game, simply enter the command:

atanks

If you do not connect a keyboard to the Raspberry Pi 3 board, you can still start the game by selecting its icon in GUI, under the “games” tab. If arrow and enter keys are mapped to buttons, then you can be completely “keyboard-free”.

For a better experience, go to game options, then graphics, and then enable the full screen option.