Hands-on Kernel Lab

Based on the Poky 2.6 release (thud)

October 2018

Tom Zanussi <tom.zanussi@linux.intel.com>
Darren Hart <dvhart@linux.intel.com>
Saul Wold <sgw@linux.intel.com>
Richard Griffiths <richard.a.griffiths@linux.intel.com>

Introduction

Welcome to the Yocto Project Hands-on Kernel Lab! During this session you will learn how to work effectively with the Linux kernel within the Yocto Project.

The 'Hands-on Kernel Lab' is actually a series of labs that will cover the following topics:

- Creating and using a traditional kernel recipe (**lab1**)
- Using 'bitbake -c menuconfig' to modify the kernel configuration and replace the **defconfig** with the new configuration (**lab1**)
- Adding a kernel module to the kernel source and configuring it as a built-in module by adding options to the kernel **defconfig** (**lab1**)
- Creating and using a linux-yocto-based kernel (**lab2**)
- Adding a kernel module to the kernel source and configuring it as a built-in module using linux-yocto 'config fragments' (**lab2**)
- Using the linux-yocto kernel as an LTSI kernel (configuring in an item added by the LTSI kernel which is merged into linux-yocto) (**lab2**)
- Using an arbitrary git-based kernel via the linux-yocto-custom kernel recipe (**lab3**)
- Adding a kernel module to the kernel source of an arbitrary git-based kernel and configuring it as a loadable module using 'config fragments' (**lab3**)
- Actually getting the module into the image and autoloading it on boot (**lab3**)
- Using a local clone of an arbitrary git-based kernel via the linux-yocto-custom kernel recipe to demonstrate a typical development workflow (**lab4**)
- Modifying the locally cloned custom kernel source and verifying the changes in the new image (**lab4**)
- Using a local clone of a linux-yocto kernel recipe to demonstrate a typical development workflow (**lab4**)
- Adding and using an external kernel module via a module recipe (**lab4**)

This lab assumes you have a basic understanding of the Yocto Project and bitbake, and are comfortable navigating a UNIX filesystem from the shell and issuing shell commands. If you need help in this area, please consult the introductory material which you can find on the Yocto website and/or Google for whatever else you need to know to get started.

All of the material covered in this lab is documented in the Yocto Project Linux Kernel Development Manual:

http://www.yoctoproject.org/docs/latest/kernel-dev/kernel-dev.html

Please consult the kernel development manual for more detailed information and background on the topics covered in this lab.

Tip: Throughout the lab you will need to edit various files. Sometimes the pathnames to these files are long. It is critical that you enter them exactly. Remember you can use the **Tab** key to help autocomplete path names from the shell. You may also copy and paste the paths from the PDF version of this lab which you can find at the same location that this document was found.

Tip: Each lab is independent of the others and doesn't depend on the results of any previous lab, so feel free to jump right to any lab that's of interest to you.

Build System Basic Setup

This hands-on lab was designed to be completed on a computer running the Ubuntu 16.04"Xenial Xerus" operating system. While this specific release of Ubuntu is recommended to avoid unforeseen incompatibilities, you can generally use a recent release of Ubuntu, Fedora, or OpenSUSE to complete this hands-on lab. This hands-on lab was developed and therefore also tested on a Fedora 22 system.

Before starting these exercises, please ensure that your system has the necessary software prerequisites installed as described in the Yocto Project Quick Build Guide (see the subsection entitled "Build Host Packages"):

https://www.yoctoproject.org/docs/2.6/brief-yoctoprojectqs/brief-yoctoprojectqs.html

This hands-on lab assumes you have a network connection and have a working version of 'git' installed. You'll need a good network connection for the initial setup and download of the source packages built by the recipes. 'git' is required for creating and testing kernel patches for the git-based kernel recipes used in lab4, but isn't required by the

other labs.

Preparing Your Build Environment

Please log in to your system as a normal user and once logged in, launch a terminal by simultaneously pressing the following keys:

Ctrl + Alt + T

Alternatively, you can open a terminal by clicking the 'Dash' icon and typing 'terminal' in the entry field. When the 'Terminal' icon appears, click on it to open a terminal.

Throughout the lab you may find it useful to work with more than one tab in your terminal. To create additional tabs:

File ‣ Open Tab

In order to run the labs, you'll first need to checkout the Poky release 2.6 'thud' sources into your home directory. From your terminal shell, type:

```
$ git clone git://git.yoctoproject.org/poky/
$ cd poky
$ git checkout thud-20.0.0
```

Once you've gotten the sources, you should cd into the the directory that was created:

```
$ cd poky
```

Listing the files in that directory should show the following files and subdirectories:

```
$ ls
bitbake       documentation LICENSE        meta meta-poky
meta-selftest meta-skeleton meta-yocto-bsp oe-init-build-env README.hardware
README.LSB    README.poky   README.qemu    scripts
```

You also need to get and unpack the instructional layers for this lab:

```
$ wget https://www.yoctoproject.org/sites/yoctoproject.org/files/kernel-lab-2.6-
layers.tar.bz2
$ bunzip2 -c kernel-lab-2.6-layers.tar.bz2 | tar xvf -
```

Listing the files in the current directory, which should still be , should now show the following files and subdirectories:

```
$ ls
bitbake                       meta-selftest
documentation                 meta-skeleton
kernel-lab-2.6-layers.tar.bz2 meta-yocto
LICENSE                       meta-yocto-bsp
meta                          oe-init-build-env
meta-lab1-qemux86             oe-init-build-env-memres
meta-lab2-qemux86             patches
meta-lab3-qemux86             README
meta-lab4-qemux86             README.hardware
meta-poky                       scripts
```

Lab 1: Traditional Kernel Recipe

In this lab you will modify a stock 4.18 Linux kernel recipe to make it boot on a qemux86 machine. You will then apply a patch and modify the configuration to add a simple kernel module which prints a message to the console. This will familiarize you with the basic bitbake workflow for working with and modifying simple kernel recipes. We will use the editor "vi" to modify files in this lab material, but you should be able to use any text editor you prefer.

Set up the Environment

```
$ cd ~/poky/
$ source oe-init-build-env
```

Open local.conf:

```
$ vi conf/local.conf
```

Add the following line just above the line that says 'MACHINE ??= "qemux86":

```
MACHINE ?= "lab1-qemux86"
```

Save you changes and close vi.

Now open bblayers.conf:

```
$ vi conf/bblayers.conf
```

and add the 'meta-lab1-qemux86' layer to the BBLAYERS variable. The final result should look like this, assuming your account is called 'myacct' (simply copy the line containing 'meta-yocto-bsp' and replace 'meta-yocto-bsp' with 'meta-lab1-qemux86'):

```
BBLAYERS ?= " ¥
  /home/myacct/poky/meta ¥
  /home/myacct/poky/meta-yocto ¥
  /home/myacct/poky/meta-yocto-bsp ¥
  /home/myacct/poky/meta-lab1-qemux86 ¥
  "
```

You should not need to make any further changes. Save your changes and close vi.

The meta-lab1-qemux86 layer provides a very simple Linux 4.18 recipe. Open it in vi for review:

```
$ vi ~/poky/meta-lab1-qemux86/recipes-kernel/linux/linux-korg_4.18.bb
```

This is a bare-bones simple Linux kernel recipe. It inherits all of the logic for configuring and building the kernel from **kernel.bbclass** (the 'inherit kernel' line) which is can be found in the meta/classes/ directory. It specifies the Linux kernel sources in the SRC_URI variable. It is mostly empty for now, so the Linux kernel configuration system will use defaults for most options. By default, the build system will look for the package source in a directory having the same name as the recipe's package name, or 'PN' (recipe names are generally of the form 'PN-PV', where 'PN' refers to 'Package Name' and 'PV' refers to 'Package Version'). In the case of the linux-korg_4.18.bb recipe, the package name and thus the source directory would be linux-korg/. Because the kernel tarball extracts into a different directory, linux/, we need to make the build system aware of this non-default name, which is the purpose of the 'S = ${WORKDIR}/linux-${PV}' line in the recipe. You will also notice a commented out patch on another SRC_URI line - leave it commented out for now, we will come back to that.

The meta-lab1-qemux86 layer also provides a fairly standard machine configuration whose purpose is to define a group of machine-specific settings for the 'lab1-qemux86' machine. These settings provide machine-specific values for a number of variables (all documented in the Yocto Project Reference Manual) which allow us to boot the 'lab1-qemux86' machine into a graphical qemu environment. Open it in vi for review:

```
$ vi ~//meta-lab1-qemux86/conf/machine/lab1-qemux86.conf
```

Without going into too much detail, there are a few things to note about this file. The first is the file name itself; note that the base filename matches the machine name, in this case 'lab1-qemux86', which is also the same as the machine name specified in the MACHINE setting in local.conf.

Secondly, note that other than the 'require' statements, which essentially just implement a file inclusion mechanism, the configuration consists almost entirely of variable assignments. The various assignment operators are documented

elsewhere and are relatively obvious, but for now we'll just mention that the '?=' assignment operator implements conditional assignment: if the variable hasn't already been set, it takes on the value specified on the right-hand-side.

Finally, a word of explanation about the PREFERRED_PROVIDER assignments in the machine configuration file. Many components of the build system have multiple implementations available. The build system will normally choose a default implementation and version for a particular component, but sometimes it makes sense for a machine to explicitly specify another implementation and/or version if it knows it doesn't want to use the defaults. It may also want to specify some values in order to 'pin down' a particular implementation and version regardless of what the defaults are, or how they may change in the future.

In the case of the 'lab1-qemux86' machine, you see that it specifies a PREFERRED_PROVIDER for the virtual/kernel component:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-korg"
```

The reason it does this is that if it didn't, it would pick up the default linux-yocto-4.18 kernel (which is the version specified in the qemu.inc file included following that line). Also, because there's only a single linux-korg_* recipe, there's no ambiguity about which version to choose and therefore no specific version specified. If you needed to, you could do that using a PREFERRED_VERSION directive – you'll see an example of that in Lab 2.

Note: The reason the layers and the machines have slightly unwieldy names e.g. 'lab1-qemux86' rather than just the simpler 'lab1' is that there's a known problem with the runqemu script in that it will only recognize machine names that end with one of the base qemu machine names (see Yocto Bug #2890 for details). Keep this in mind if you decide to create your own qemu-machine based BSP layers.

It is necessary to supply a value for LIC_FILES_CHKSUM for the kernel license file. Use md5sum to get the checksum.

```
$ md5sum ./tmp/work/lab1_qemux86-poky-linux/linux-korg/4.18-r0/license-destdir/linux-korg/COPYING
```

Then add the result to meta-lab1-qemux86/recipes-kernel/linux/linux-korg_4.18.bb:

```
LIC_FILES_CHKSUM = "file://COPYING;md5=bbea815ee2795b2f4230826c0c6b8814"
```

Build the Image

Now you will build the kernel and assemble it into a qemu bootable image. This first build may take a long time, perhaps up to an hour, so go have lunch! (the first build will take the longest, since in addition to building, the system will download all the packages it needs).

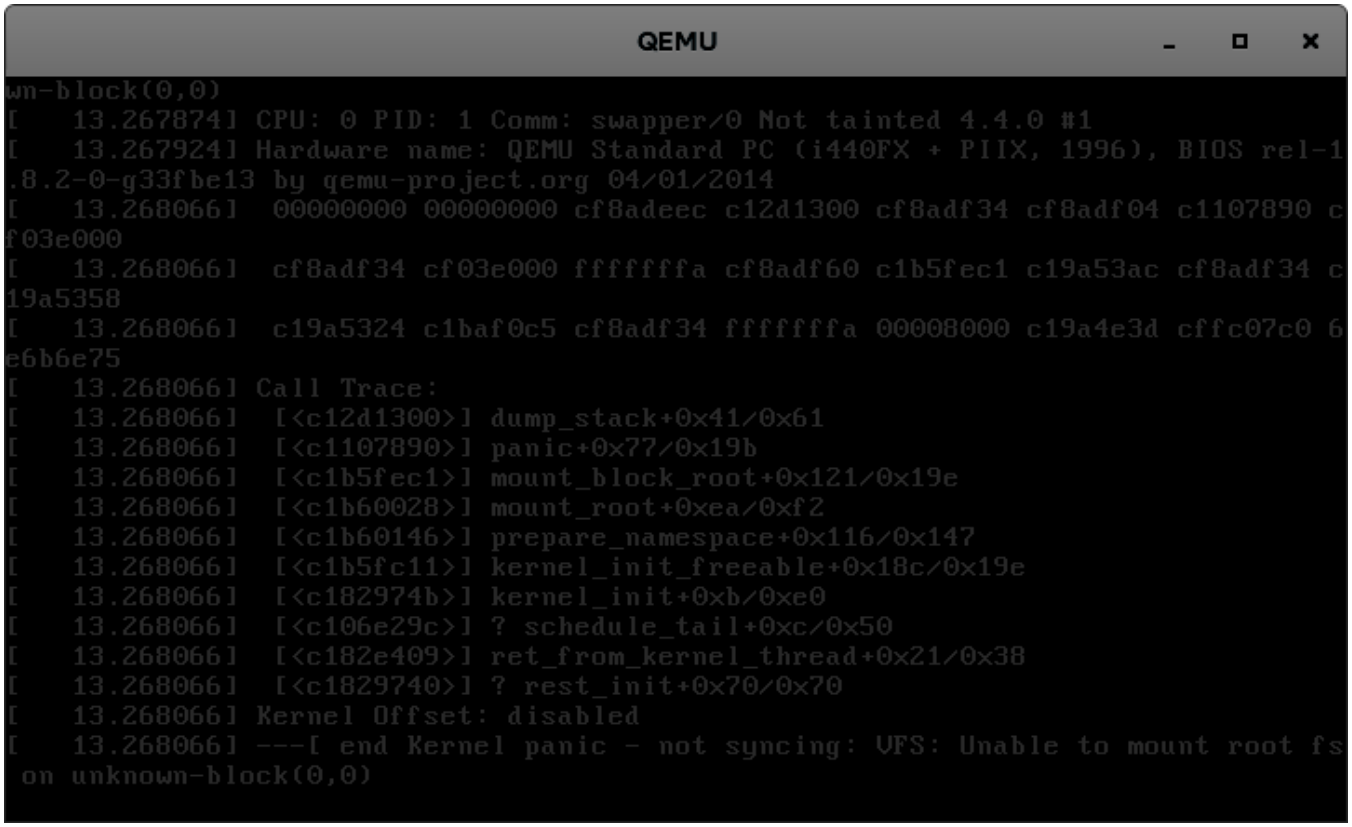
```
$ bitbake core-image-minimal
```

Note: For this lab, there will be a number of warning messages of the form 'WARNING: Failed to fetch ...'. You can safely ignore those.

Now boot the image with QEMU:

```
$ runqemu tmp/deploy/images/lab1-qemux86/bzImage-lab1-qemux86.bin tmp/deploy/images/lab1-qemux86/core-image-minimal-lab1-qemux86.ext4
```

A black QEMU window should appear and immediately start printing the Linux kernel boot messages... followed by a kernel panic:



The kernel failed to load a root filesystem. Note that under the "**List of all partitions:**" there are no devices. This means that the kernel did not find a driver for any of the block devices provided for the qemu machine. And we also need to add some virtual drivers to support qemu.

Reconfigure the Linux Kernel

QEMU provides an Intel PIIX IDE controller. Use the Linux kernel **menuconfig** command to configure this into your kernel:

```
$ bitbake virtual/kernel -c menuconfig
```

A new window will appear that allows you to enable various Linux kernel configuration options. Use the following keys to navigate the menu:

- Up/Down arrows: move up and down
- Left/Right arrows: Choose a command <Select> <Exit> or <Help>
- Enter: Execute a command
  - <Select> Descends into a menu
  - <Exit> Backs out of a menu, or exits menuconfig
- Space: toggle a configuration option

Note that before descending into a menu that is itself configurable, you will need to check the menu item or its contents will be empty.

Enable the following options:

```
Device Drivers --->
  [*] ATA/ATAPI/MFM/RLL support (DEPRECATED)  --->
    [*] Intel PIIX/ICH chipsets support
  [*] Virtualization drivers  -----
    [*] Virtio drivers --->
```

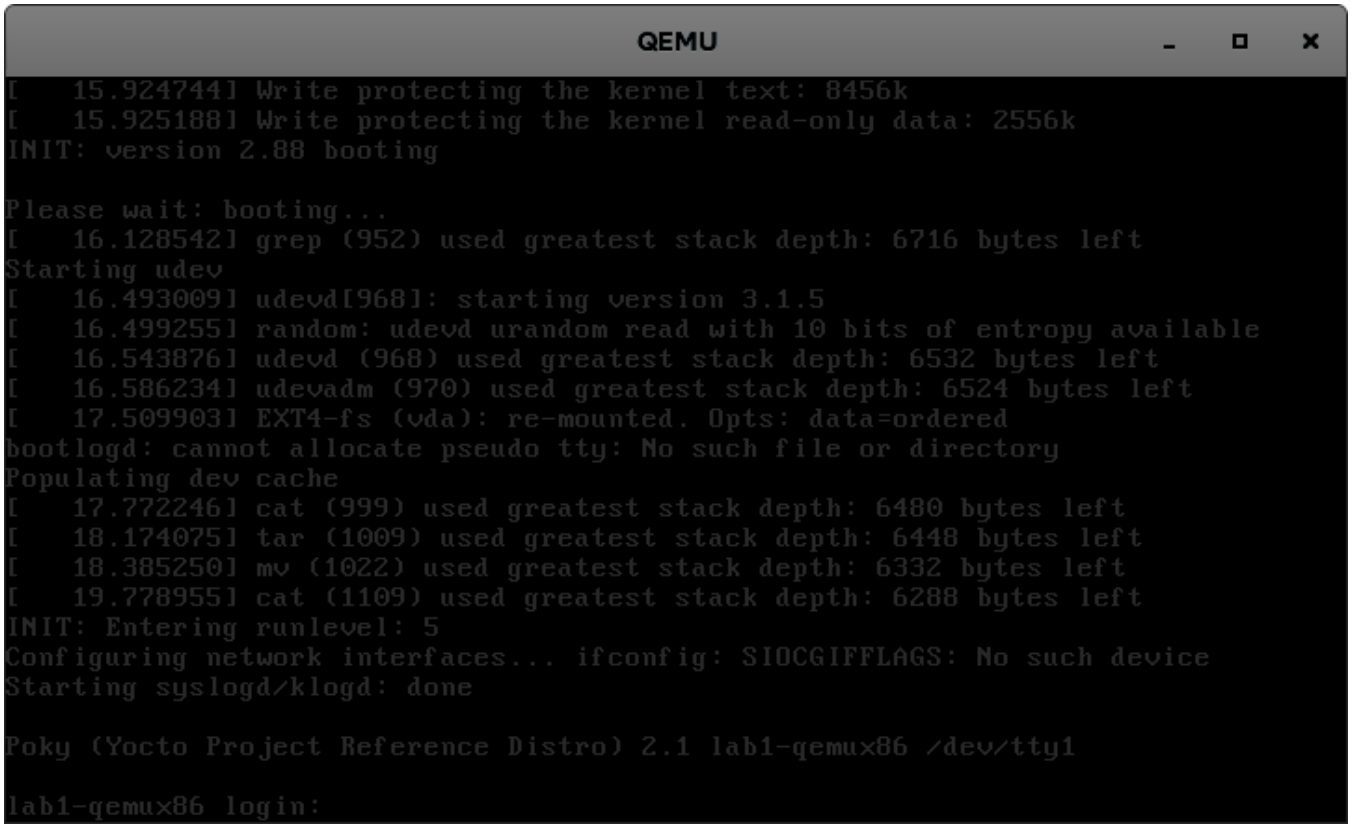
```
      [*] PCI driver for virtio devices
        [*] Support for legacy virtio draft 0.9.X and older devices
  [*] Block devices  --->
    [*] Virtio block driver
  Generic Driver Options  --->
    [*] Maintain a devtmpfs filesystem to mount at /dev
    [*]   Automount devtmpfs at /dev, after the kernel mounted the rootfs
File systems  --->
  [*] The Extended 4 (ext4) filesystem
```

Exit and save your changes by selecting **Exit** and pressing **Enter**, repeat until it prompts you to save your changes.

Now rebuild and deploy only the kernel. This avoids having to rebuild the image itself, which has not changed, saving you a few minutes. Then try to boot it in QEMU again:

```
$ bitbake virtual/kernel -c compile -f
$ bitbake virtual/kernel -c deploy
$ runqemu tmp/deploy/images/lab1-qemux86/bzImage-lab1-qemux86.bin tmp/deploy/images/lab1-qemux86/core-image-minimal-lab1-qemux86.ext4
```

QEMU will start as before, but this time will boot all the way to a login prompt. As you can see, there are a number of scary-looking errors and warnings on the console. This is due to the fact that you're starting with a bare-bones configuration and simply trying to get to a functional boot prompt, without bothering to worry about anything more at this point. In this respect, you have a successful outcome, and you can should now be able to log in as **root** with no password if you want to poke around.



Up to this point, if you were to share the meta-lab1-qemux86 layer with someone else, the kernel they build would still fail to boot, because the fixes only exist in your system's **WORKDIR**. You need to update the **defconfig** in the layer with the one you modified with **menuconfig**. Copy the **.config** over the **defconfig** in the layer:

```
$ cp tmp/work/lab1_qemux86-poky-linux/linux-korg/4.18-r0/build/.config ~/thud/meta-lab1-qemux86/recipes-kernel/linux/linux-korg/defconfig
```

Patching the Kernel

Now that you have the Linux kernel booting on your machine, you can start modifying it. Here you will apply a patch to add a simple driver which prints a message to the console during boot.

Open and review the patch so you know what to expect once it is applied:

```
$ vi ~/thud/meta-lab1-qemux86/recipes-kernel/linux/linux-korg/yocto-testmod.patch
```

Look for the **printk** statement in the **yocto_testmod_init()** function. This is the message you will look for to verify the changes have taken effect.

Instruct the layer to apply the patch by adding it to the SRC_URI. Edit the Linux kernel recipe:

```
$ vi ~/thud/meta-lab1-qemux86/recipes-kernel/linux/linux-korg_4.18.bb
```

Uncomment the following line (remove the leading '#' character):

```
#SRC_URI += "file://yocto-testmod.patch"
```

Save your changes and close vi.

Now use **menuconfig** to enable the driver. Bitbake will detect that the recipe file has changed and start by fetching the new sources and apply the patch.

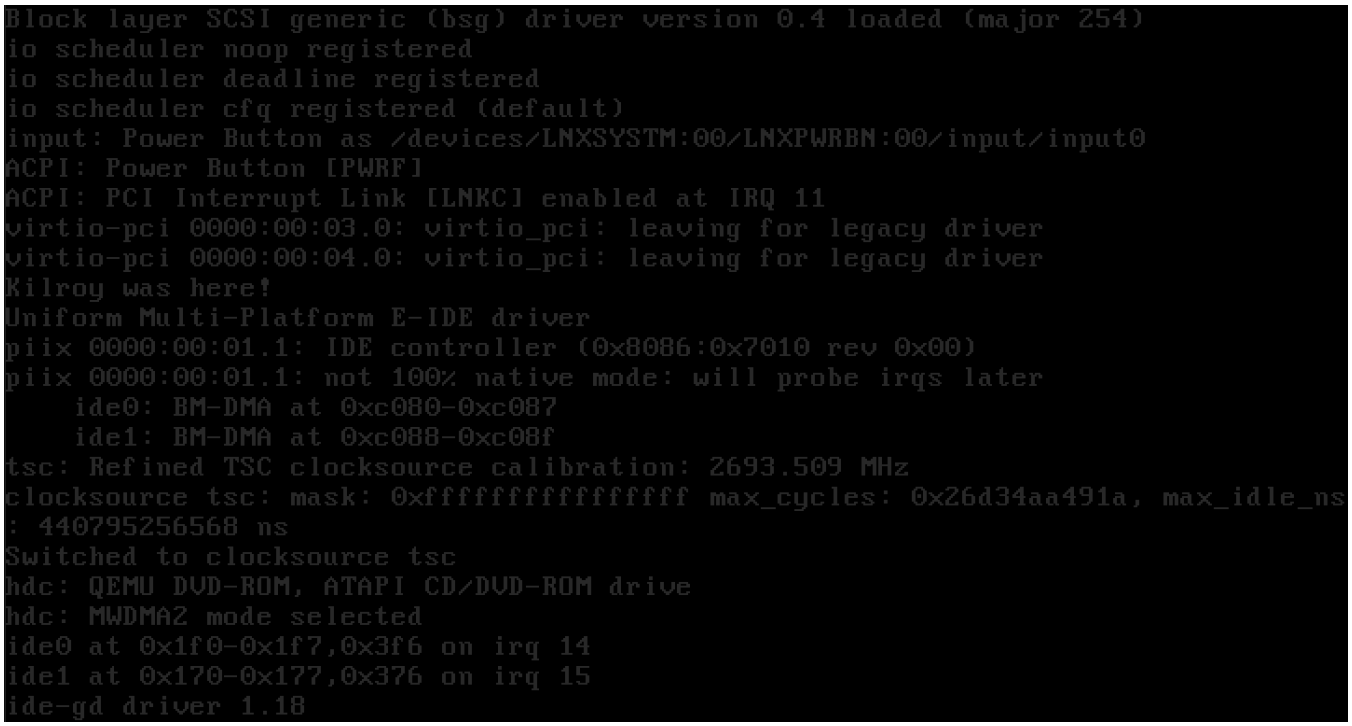
```
$ bitbake virtual/kernel -c menuconfig

Device Drivers  --->
  [*] Misc devices  --->
    [*] Yocto Test Driver (NEW)
```

Exit and save your changes as before.

Now rebuild the kernel and boot it in QEMU.

```
$ bitbake virtual/kernel -c deploy
$ runqemu tmp/deploy/images/lab1-qemux86/bzImage-lab1-qemux86.bin tmp/deploy/images/lab1-qemux86/core-image-minimal-lab1-qemux86.ext4
```

You can scroll back through the boot log using **Shift+PgUp**. You should find the Yocto test driver message in there or just grep for it:




```
$ dmesg |grep Kilroy
[    7.162747] Kilroy was here!
```

Finally, as before, if you want to save the configuration for posterity, you need to update the **defconfig** in the layer with the one you modified for the new driver. To do so, you can copy the **.config** over the **defconfig** in the layer (but it's not required at this point, as the lab is essentially finished and the results aren't required for any later labs):

```
$ cp tmp/work/lab1_qemux86-poky-linux/linux-korg/4.18-r0/build/.config ~/thud/meta-lab1-qemux86/recipes-kernel/linux/linux-korg/defconfig
```

Lab 1 Conclusion

Congratulations! You have modified and configured the Linux kernel using a traditional bitbake Linux kernel recipe. You also updated the layer itself so that your changes can be shared. This concludes Lab 1.

Lab 2: Linux-Yocto Kernel Recipe

In this lab you will work towards the same end goal as in Lab 1. This time you will use the **linux-yocto** recipe and tooling. This simplifies the process of configuring the kernel and makes reusing your work much easier.

Setup the Environment

```
$ cd ~/thud/
$ source oe-init-build-env
```

Open local.conf:

```
$ vi conf/local.conf
```

Add the following line just above the line that says 'MACHINE ??= “qemux86”:

```
MACHINE ?= "lab2-qemux86"
```

Save your changes and close vi.

Now open bblayers.conf:

```
$ vi conf/bblayers.conf
```

and add the 'meta-lab2-qemux86' layer to the BBLAYERS variable. The final result should look like this, assuming your account is called 'myacct' (simply copy the line containing 'meta-yocto-bsp' and replace 'meta-yocto-bsp' with 'meta-lab2-qemux86'):

```
BBLAYERS ?= " ¥
  /home/myacct/thud/meta ¥
  /home/myacct/thud/meta-yocto ¥
  /home/myacct/thud/meta-yocto-bsp ¥
  /home/myacct/thud/meta-lab2-qemux86 ¥
  "
```

You should not need to make any further changes. Save your changes and close vi.

Review the Lab 2 Layer

This layer differs from meta-lab1 only in the Linux kernel recipes. This layer contains the following files for the kernel:

```
recipes-kernel/
  linux /
    linux-yocto_4.18.bbappend
    files /
      mtd-nand-benand.cfg
      lab2.cfg
      yocto-testmod.patch
```

Open the 4.18 kernel recipe:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_4.18.bbappend
```

Note that this is not a complete recipe, but rather an extension of the **linux-yocto** recipe provided by the poky sources. It adds the layer path for additional files and sets up some machine-specific variables. Notice that instead of a **defconfig** file, the recipe adds **lab2.cfg** to the SRC_URI. This is a Linux kernel config fragment. Rather than a complete **.config** file, a config fragment lists only the config options you specifically want to change. To start out, this fragment is commented out, and the linux-yocto sources will provide a default **.config** compatible with common PC hardware.

The **lab2.cfg** config fragment is an example of a config fragment that's both defined and specified in 'recipe-space', in other words defined as a file under the recipe's (in this case) files/ directory and added via the SRC_URI. Config fragments can also be defined in the kernel repository's 'meta' branch and added to the BSP via KERNEL_FEATURES statements in the kernel recipe:

```
KBRANCH_lab2-qemux86 = "standard/base"
```

```
KMACHINE_lab2-qemux86  = "common-pc"

KERNEL_FEATURES_append_lab2-qemux86 += " cfg/smp.scc"
```

In the recipe fragment above, the 'cfg/smp.scc' kernel feature, which maps to the kernel's CONFIG_SMP configuration setting, is added to the machine's kernel configuration to turn on SMP capabilities for the BSP. Kernel features as well as the KBRANCH and KMACHINE settings referenced above, which essentially specify the source branch for the BSP, are all described in detail in the Yocto Linux Kernel Development Manual.

The meta-lab2-qemux86 machine configuration is very similar to the meta-lab1-qemux86 in Lab 1. Open it in vi for review:

```
$ vi ~/thud/meta-lab2-qemux86/conf/machine/lab2-qemux86.conf
```

The main difference from the Lab 1 machine configuration is that it specifies not only a PREFERRED_PROVIDER for the virtual/kernel component, but a PREFERRED_VERSION as well:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.18%"
```

Because the Lab 2 layer has multiple kernel implementations available to it (linux-yocto_4.14 and linux-yocto_4.18), there is in this case some ambiguity about which implementation and version to choose. The above lines choose a 'linux-yocto' recipe as the PREFERRED_PROVIDER, and explicitly select the linux-yocto_4.18 version via the PREFERRED_VERSION setting (the trailing '%' serves as a wildcard, meaning in this case to ignore any minor version in the package version when doing the match).

In this case, the build system would have chosen the same implementation and version via defaults (linux-yocto by virtue of the included qemu.inc, and 4.18 simply because it's the highest version number available for the linux-yocto recipes – this is contained in the logic treating package selection in the build system), but again, sometimes it makes sense to avoid surprises and explicitly 'pin down' specific providers and versions.

Build the Image

OK, you have done this before:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-qemux86/core-image-minimal-lab2-qemux86.ext4
```

The **linux-yocto** repository meta-data already provides the driver support needed to boot in QEMU – the missing configuration settings for PIIX/ICH, ext4, virtio and devtmpfs that you explicitly added to the configuration in Lab 1 are already included in the basic configuration inherited via the meta-data in Lab 2. This meta-data is reused across several Board Support Packages (BSPs), reducing the tedium of managing a complete kernel config for every BSP. In the case of Lab 2, it means that you don't have to go through an unproductive boot failure and configuration update cycle as you did in Lab 1.

Tip: If you are pressed for time, you can take our word that this will boot without configuration changes and move on to modifying the kernel. Be sure to run **bitbake core-image-minimal** before you run the **runqemu** command though, or it will fail to find a disk image for the lab2-qemux86 machine.

Modify the Kernel

Now you can apply the driver patch and configure the kernel to use it.

Edit the linux-yocto kernel recipe:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_4.18.bbappend
```

and uncomment the line including the patch and the line including the lab2 config fragment:

```
SRC_URI += "file://yocto-testmod.patch"
SRC_URI += "file://lab2.cfg"
```

This accomplishes the same thing, adding and enabling the 'yocto-testmod' module, that you accomplished in Lab 1. The difference here is that instead of using **menuconfig** to enable the new option in the monolithic .config file as in Lab 1, here you add the patch in the same way but enable the test module using the standalone **lab2.cfg** config fragment.

Save your changes and close vi.

Configure the Kernel

You could use **menuconfig** to enable the option, but since you already know what it is, you can simply add it to the **lab2.cfg** file.

Open the file:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/files/lab2.cfg
```

and examine the following lines, which enable the module as a built-in kernel module:

```
# Enable the testmod
CONFIG_YOCTO_TESTMOD=y
```

Close vi.

Tip: You know what you need to add now, but if you are not sure exactly which config option you need, you can save off the original **.config** (after an initial **linux-yocto** build), then run **menuconfig** and take a **diff** of the two files. You can then easily deduce what your config fragment should contain.

Now you can rebuild and boot the new kernel. Bitbake will detect the recipe file has changed and start by fetching the new sources and apply the patch:

```
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-qemux86/core-image-minimal-lab2-qemux86.ext4
```

Like before, QEMU will open a new window and boot to a login prompt. You can use **Shift+PgUp** to scroll up and find the new driver message.

Modify the Kernel to Make Use of an LTS Kernel Option

Note: This exercise shows how to enables support for software BCH error correction **mtd-nand-bch.cfg** config option. This is very similar to the previous exercise in which you enabled the Yocto 'testmod' using the **lab2.cfg** fragment.

We first need to switch to 4.14 kernel. Open the machine configuration file for lab2 in vi:

```
$ vi ~/thud/meta-lab2-qemux86/conf/machine/lab2-qemux86.conf
```

Change the preferred version of the linux-yocto kernel to 4.1 by commenting out the 4.18 line and uncommenting the 4.14 line as such:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
#PREFERRED_VERSION_linux-yocto ?= "4.18%"
PREFERRED_VERSION_linux-yocto ?= "4.14%"
```

Now you can rebuild and boot the new kernel:

```
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-qemux86/core-image-minimal-lab2-qemux86.ext4
```

Verify that you are in fact now running the 4.14 kernel:

```
# uname -a
```

Configure the Kernel

You could use **menuconfig** to enable the option, but since you already know what it is, you can simply add it and its dependencies to the **mtd-nand-bch.cfg** file.

Open the **mtd-nand-benand.cfg** file:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/files/mtd-nand-benand.cfg
```

and examine the following lines, which enable the built-in ECC NAND config item:

```
## Enable BCH ECC NAND
CONFIG_MTD=y
CONFIG_MTD_NAND=y
CONFIG_MTD_NAND_BCH=y
CONFIG_MTD_NAND_ECC_BCH=y
```

Close vi. Note that in addition to the CONFIG_MTD_NAND_ECC_BCH option itself, there are a few others which are required in order for that option to be enabled. For example, CONFIG_MTD needs to be enabled in order for CONFIG_MTD_NAND_BCH to be enabled and so on, as you would expect. (The complete set of dependent options required for a given option can be generated by taking the diff between the kernel .configs before and after the option of interest was enabled via '*bitbake -c menuconfig*' as demonstrated in previous labs.)

Because this option doesn't really produce an easily visible effect such as a line in the qemu machine's kernel log, we'll just verify that the config fragment actually ends up taking effect in the kernel build.

Open the kernel config file generated by the previous build:

```
$ vi ~/thud/build/tmp/work/lab2_qemux86-poky-linux/linux-
yocto/4.14.76+gitAUTOINC+3435617380_2c5caa7e84-r0/linux-lab2_qemux86-standard-build/.config
```

and use the 'Search | Find' menu item to search for the following config item:

```
CONFIG_MTD
```

What you should find is that that config item is not set:

```
# CONFIG_MTD is not set
```

Also, you shouldn't see a CONFIG_MTD_NAND_BCH option at all, because it depends on CONFIG_MTD, which is disabled.

Edit the linux-yocto kernel recipe:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_4.14.bbappend
```

and uncomment the line including the built-in ECC NAND config fragment:

```
SRC_URI += "file://mtd-nand-benand.cfg"
```

Save your changes and close vi.

Rebuild the Kernel

Now you can rebuild and boot the new kernel:

```
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-
qemux86/core-image-minimal-lab2-qemux86.ext4
```

Your kernel should boot without problem, and if you ran this on real hardware, you'd expect to see support for built-in ECC NAND enabled and available. For the purposes of this lab, however, it's sufficient to verify that the option actually took effect in the final kernel configuration. You can verify that the ECC NAND support has been enabled in the kernel by again looking at the final kernel config file after enabling the **mtd-nand-bch.cfg** config fragment:

Open the config file:

```
$ vi ~/thud/build/tmp/work/lab2_qemux86-poky-linux/linux-
yocto/4.14.76+gitAUTOINC+3435617380_2c5caa7e84-r0/linux-lab2_qemux86-standard-build/.config
```

and use the 'Search | Find' menu item to search for the following config items:

```
CONFIG_MTD
CONFIG_MTD_NAND_BCH
```

What you should find is that that those config items are now set:

```
CONFIG_MTD=y
CONFIG_MTD_NAND_BCH=y
```

Close vi.

Lab 2 Conclusion

In this lab you applied a patch and modified the configuration of the Linux kernel using a config fragment, which is a feature provided by the `linux-yocto` kernel tooling. You also switched kernel versions and enabled a kernel option using a config fragment This concludes Lab 2.

Extra Credit: Iterative Development

Should you need to modify the kernel further at this point, perhaps it failed to compile or you want to experiment with the new driver, you can do that directly using the sources in:

```
$ cd ~/thud/build/tmp/work-shared/lab2-qemux86/kernel-source
```

Tip: This is a great time to make use of that `Tab` completion!

After making changes to the source, you can rebuild and test those changes, just be careful not to run a clean, fetch, unpack or patch task or you will lose your changes:

```
$ cd ~/thud/build
$ bitbake linux-yocto -c compile -f
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-qemux86/core-image-minimal-lab2-qemux86.ext4
```

You can repeat this cycle as needed until you are happy with the kernel changes.

The `linux-yocto` recipe creates a git tree here, so once you are done making your changes, you can easily save them off into a patch using standard git commands:

```
$ git add path/to/file/you/change
$ git commit -signoff
$ git format-patch -1
```

You can then integrate these patches into the layer by copying them alongside the **`yocto-testmod.patch`** and adding them to the `SRC_URI`.

Lab 3: Custom Kernel Recipe

In this lab you will use use the `linux-yocto-custom` recipe and tooling to make use of a non-linux-yocto git-based kernel of your choosing, while still retaining the ability to reuse your work via config fragments. You'll also learn what you need to do to build, install, and automatically load a loadable kernel module instead of as a built-in module as you did in lab2.

Set up the Environment

```
$ cd ~/thud/
$ source oe-init-build-env
```

Open local.conf:

```
$ vi conf/local.conf
```

Add the following line just above the line that says 'MACHINE ??= “qemux86”:

```
MACHINE ?= "lab3-qemux86"
```

Save your changes and close vi.

Now open bblayers.conf:

```
$ vi conf/bblayers.conf
```

and add the 'meta-lab3-qemux86' layer to the BBLAYERS variable. The final result should look like this, assuming your account is called 'myacct' (simply copy the line containing 'meta-yocto-bsp' and replace 'meta-yocto-bsp' with 'meta-lab3-qemux86'):

```
BBLAYERS ?= " ¥
  /home/myacct/thud/meta ¥
  /home/myacct/thud/meta-yocto ¥
  /home/myacct/thud/meta-yocto-bsp ¥
  /home/myacct/thud/meta-lab3-qemux86 ¥
  "
```

You should not need to make any further changes. Save your changes and close vi.

Review the Lab 3 Layer

This layer differs from meta-lab2-qemux86 only in the Linux kernel recipe. This layer contains the following files for the kernel:

```
recipes-kernel /
  linux /
    linux-yocto-custom /
      defconfig
      lab3.cfg
      yocto-testmod.patch
    linux-yocto-custom.bb
```

Open the kernel recipe:

```
$ vi ~/thud/meta-lab3-qemux86/recipes-kernel/linux/linux-yocto-custom.bb
```

Note that this is a complete recipe rather an extension as in lab2. In fact it was derived from the linux-yocto-custom.bb recipe found in thud/meta-skeleton/recipes-kernel/linux. Notice that it uses a `defconfig` file and additionally adds `lab3.cfg` to the SRC_URI. The **defconfig** is required because this is not a linux-yocto kernel as used in lab2, but rather an arbitrary kernel wrapped by the linux-yocto-custom recipe.

An arbitrary kernel doesn't contain all the metadata present in the linux-yocto kernel and therefore doesn't have a mapping to any of the base configuration items associated with the set of BSP types available in the linux-yocto kernel. In the case of the linux-yocto kernel, this mapping is responsible for assembling the .config from a collection of fragments, but since a custom kernel doesn't have access to these, a **defconfig** that provides the basic set of options needed to boot the machine is explicitly required.

However, because this is a linux-yocto-custom kernel, it does have the ability to specify and reuse config fragments, which is the major difference between this setup and the simple kernel used in lab1, which also used a **defconfig**.

The lab3.cfg fragment is a Linux kernel config fragment. Rather than a complete `.config` file, a config fragment lists only the config options you specifically want to change. To start out, this fragment is commented out, and the linux-yocto-custom sources will use only the `defconfig` specified, which is compatible with common PC hardware.

Build the Image

OK, you have done this before:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab3-qemux86/bzImage-lab3-qemux86.bin tmp/deploy/images/lab3-qemux86/core-image-minimal-lab3-qemux86.ext4
```

Modify the Kernel

Now you can apply the driver patch and configure the kernel to use it.

Edit the linux-yocto-custom kernel recipe:

```
$ vi ~/thud/meta-lab3-qemux86/recipes-kernel/linux/linux-yocto-custom.bb
```

and uncomment the lines including the patch and the lab3 config fragment:

```
SRC_URI += "file://yocto-testmod.patch"
SRC_URI += "file://lab3.cfg"
```

Save your changes and close vi.

Configure the Kernel

You could use `menuconfig` to enable the option, but since you already know what it is, you can simply add it to the `lab3.cfg` file.

Open the file:

```
$ vi ~/thud/meta-lab3-qemux86/recipes-kernel/linux/linux-yocto-custom/lab3.cfg
```

and examine the following lines, which enable the module as a built-in kernel module:

```
# Enable the testmod
CONFIG_YOCTO_TESTMOD=m
```

This configures the yocto-testmod as a module this time instead of as a built-in module as in lab2. In order to actually get the module into the image and loaded, you'll need to add a couple additional items to the kernel recipe and machine configuration, but we'll cover that in the following step.

Save your changes and close vi.

> **Tip:** You know what you need to add now, but if you are not sure exactly which config option you need, you can save off the original `.config` (after an initial `linux-yocto` build), then run `menuconfig` and take a `diff` of the two files. You can then easily deduce what your config fragment should contain.

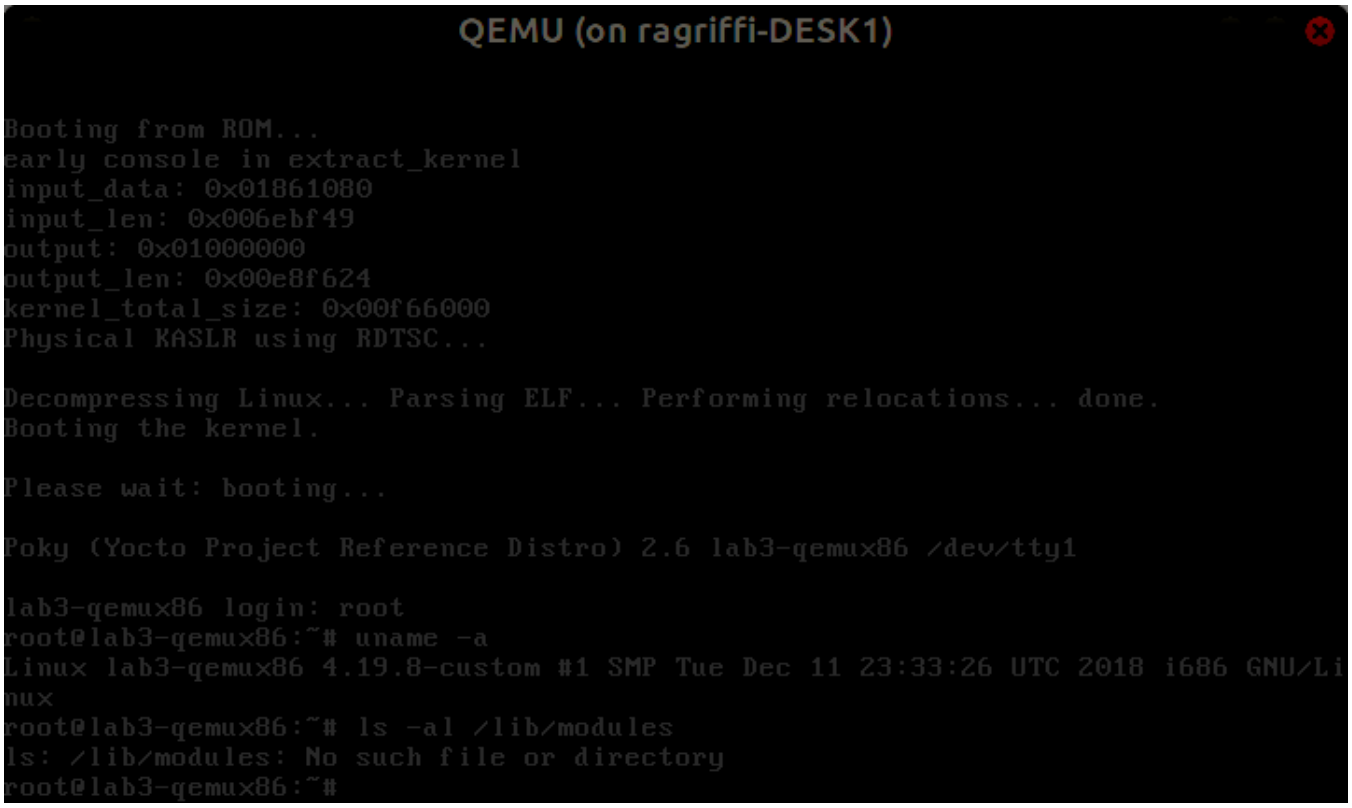
Rebuild the Image

Now you can rebuild and boot the new image. You're rebuilding the new image rather than just the kernel in this case because the module is no longer included in the kernel image but rather in the /lib/modules directory of the filesystem image, which requires us to build a new root filesystem. Bitbake will detect the recipe file has changed and start by fetching the new sources and apply the patch:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab3-qemux86/bzImage-lab3-qemux86.bin tmp/deploy/images/lab3-qemux86/core-image-minimal-lab3-qemux86.ext4
```

This time, you won't look for a message, but rather see whether your module was loaded. To do that, you'll use 'lsmod' to get a list of loaded modules. Not seeing what you expect (you'd expect to see 'yocto_testmod' in the output of 'lsmod', but won't), you should then check whether your modules were installed in the root filesystem (you should see a 'kernel/drivers/misc' directory in /lib/modules/4.19.8-custom, containing your yocto-testmod.ko kernel module

binary):



Obviously, you're not seeing what you'd expect, so let's verify whether your module was in fact built. You can do that by looking in the deploy directory on the build system:

```
$ ls ~/thud/build/tmp/deploy/rpm/lab3_qemux86/ | grep yocto-testmod
```

You should in fact see an RPM file that was created for the hello-testmod module – you should see something similar to the following output from the previous command:

```
kernel-module-yocto-testmod-4.19.8-custom-4.19.8+git0+178574b665-r1.lab3_qemux86.rpm
```

So, your module was built, it just wasn't added to the image. One way of making that happen is to add it to the machine configuration.

Add the Module to the Image and Have it Autoload on Boot

Open the machine configuration file:

```
$ vi ~/thud/meta-lab3-qemux86/conf/machine/lab3-qemux86.conf
```

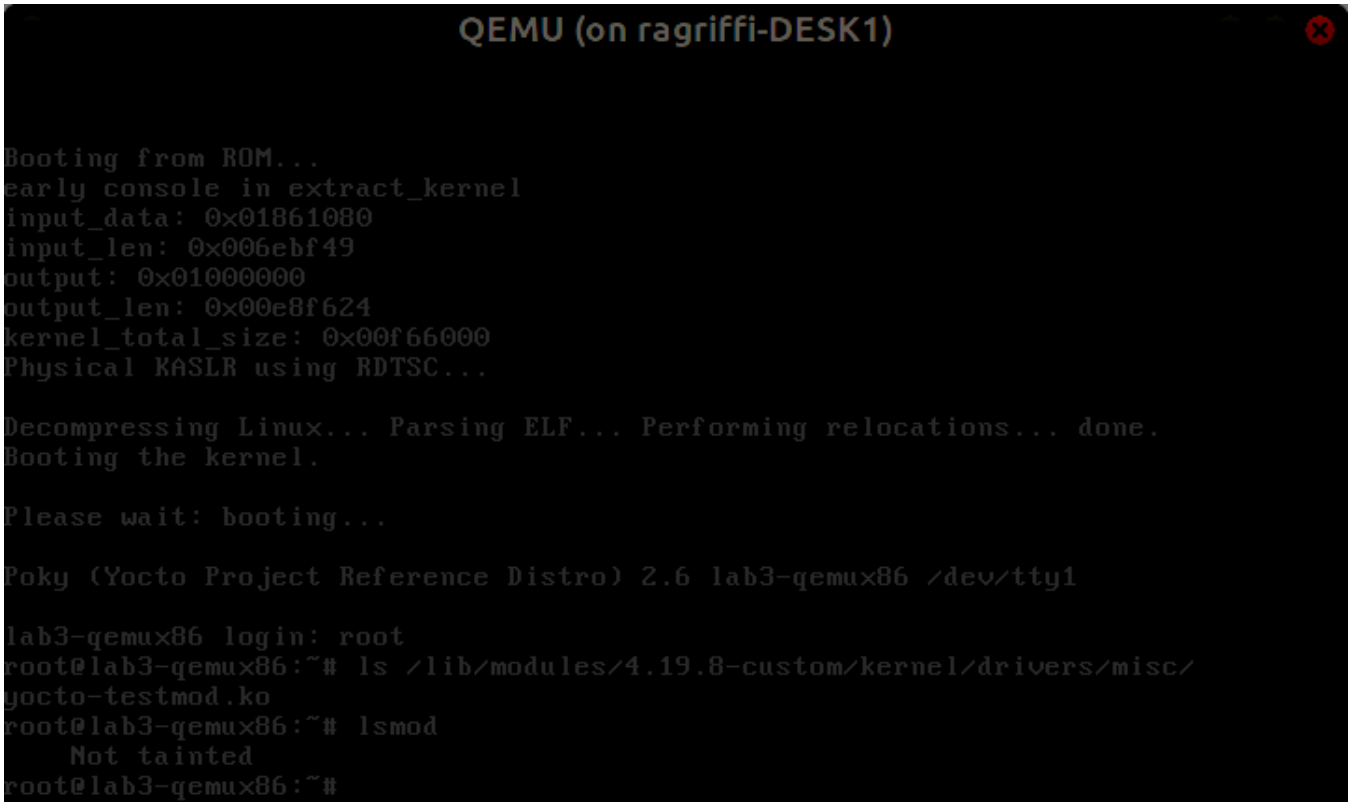
and uncomment the following line at the end of the file:

```
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-module-yocto-testmod"
```

This will cause the yocto-testmod module to be included in the minimal image, but it won't cause the module to be loaded on boot. Build the minimal image again and boot it:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab3-qemux86/bzImage-lab3-qemux86.bin tmp/deploy/images/lab3-
qemux86/core-image-minimal-lab3-qemux86.ext4
```

Notice that the '/lib/modules/4.19.8-custom' directory containing the yocto-testmod module is now present on the booted system, but the module hasn't been loaded:



To have yocto-testmod loaded on boot, you'll uncomment the following line to the linux-yocto-custom recipe:

```
KERNEL_MODULE_AUTOLOAD += "yocto-testmod"
```

Open the linux-yocto-custom.bb file and uncomment that line:

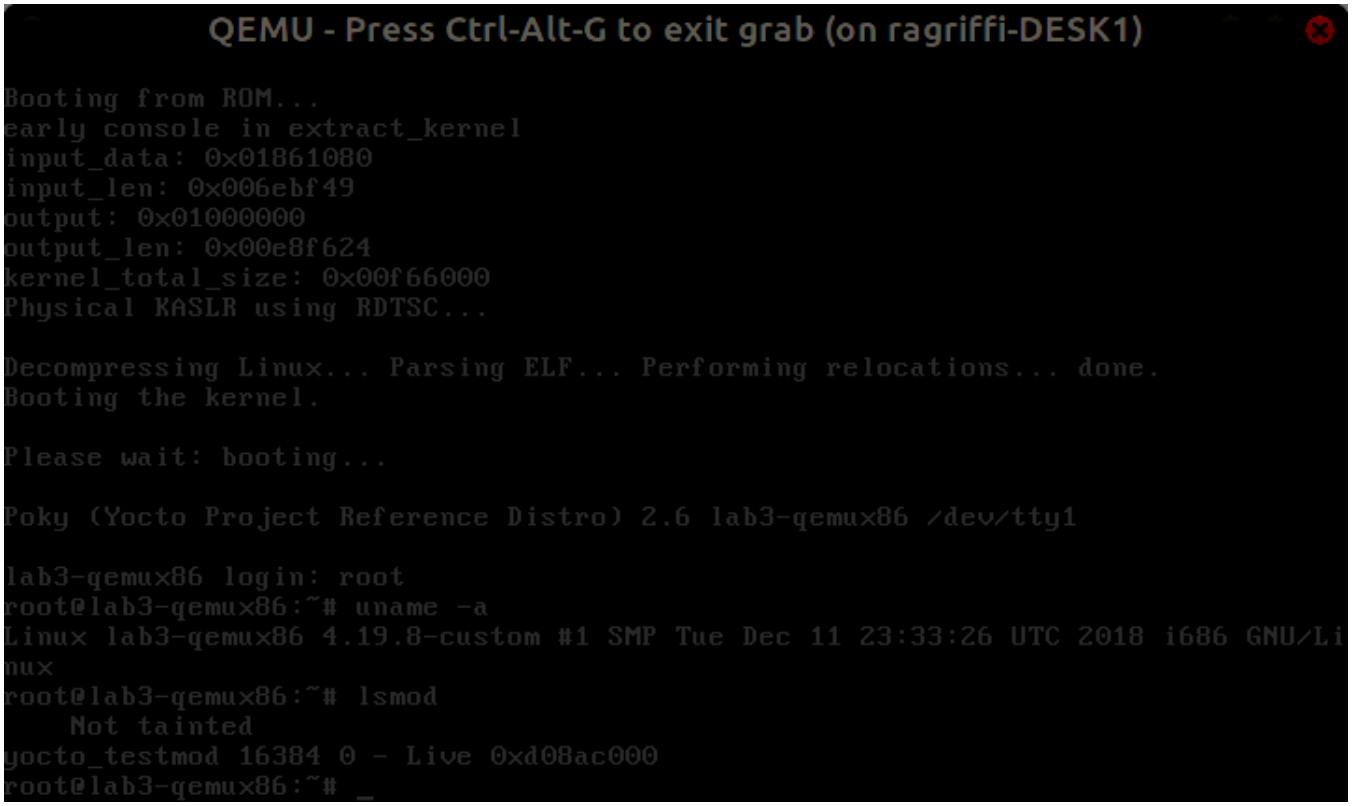
```
$ vi ~/thud/meta-lab3-qemux86/recipes-kernel/linux/linux-yocto-custom.bb
```

Note: Your module isn't exactly 'essential' and you'd normally use MACHINE_EXTRA_RRECOMMENDS, but this is the variable you need to use with the minimal image since it doesn't include the base package that includes the latter variable.

Now, let's build the minimal image again and boot it:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab3-qemux86/bzImage-lab3-qemux86.bin tmp/deploy/images/lab3-qemux86/core-image-minimal-lab3-qemux86.ext4
```

This time, lsmod shows yocto-testmod loaded, as expected:



Like before, QEMU will open a new window and boot to a login prompt. You can use **Shift+PgUp** to scroll up and find the new driver message. You can also type 'dmesg | less' at the prompt to look for the module init message.

Lab 3 Conclusion

In this lab you applied a patch and modified the configuration of an arbitrary git-based non-linux-yocto Linux kernel using a config fragment. You also added and autoloaded a module as a loadable module. This concludes Lab 3.

Lab 4: Custom Kernel Recipe With Local Repository

In this lab you will use the **linux-yocto-custom** recipe and tooling to make use of a local non-linux-yocto git-based kernel of your choosing, while still retaining the ability to reuse your work via config fragments. This makes for an easier workflow when changing the kernel code, which this lab will also demonstrate. This lab will also show how to create and use a recipe to build and install an external kernel module.

Set up the Environment

```
$ cd ~/thud/
$ source oe-init-build-env
```

Open local.conf:

```
$ vi conf/local.conf
```

Add the following line just above the line that says 'MACHINE ??= “qemux86”:

```
MACHINE ?= "lab4-qemux86"
```

Save your changes and close vi.

Now open bblayers.conf:

```
$ vi conf/bblayers.conf
```

and add the 'meta-lab4-qemux86' layer to the BBLAYERS variable. The final result should look like this, assuming your account is called 'myacct' (simply copy the line containing 'meta-yocto-bsp' and replace 'meta-yocto-bsp' with 'meta-lab4-qemux86'):

```
BBLAYERS ?= " ¥
  /home/myacct/thud/meta ¥
  /home/myacct/thud/meta-yocto ¥
  /home/myacct/thud/meta-yocto-bsp ¥
  /home/myacct/thud/meta-lab4-qemux86 ¥
  "
```

You should not need to make any further changes. Save your changes and close vi.

Review the Lab 4 Layer

This layer differs from meta-lab3-qemux86 in that instead of yocto-testmod patch in the Linux kernel recipe itself, you'll add an external kernel module called hello-mod. It also contains a change to the SRC_URI in the linux-yocto-custom.bb that points it to a local kernel repo, which you'll need to modify, and a KBRANCH variable that will point to a 'working branch' in the local repo, which we'll describe in more detail later. This layer contains the following files for the kernel:

```
recipes-kernel /
  hello-mod /
    files /
      COPYING
      hello.c
      Makefile
     hello-mod_0.1.bb
  linux /
    linux-yocto-custom /
      defconfig
    linux-yocto-custom.bb
```

Open the kernel recipe:

```
$ vi ~/thud/meta-lab4-qemux86/recipes-kernel/linux/linux-yocto-custom.bb
```

Note that as in lab3, this is a complete recipe rather an extension as in lab2. In fact it was derived from the linux-yocto-custom.bb recipe found in thud/meta-skeleton/recipes-kernel/linux. Notice that it uses a **defconfig** file but

doesn't add any additional **.cfg** file to the SRC_URI as in lab3.

Because you're adding an external module, you don't have a config option in the kernel to define – the module will be included in the image by virtue of the BSP configuration directives we'll describe in a later step rather than via changes to the kernel configuration itself.

The **defconfig** is required because this is not a linux-yocto kernel as used in lab2, but rather an arbitrary kernel wrapped by the linux-yocto-custom recipe. An arbitrary kernel doesn't contain all the metadata present in the linux-yocto kernel and therefore doesn't have a mapping to any of the base configuration items associated with the set of BSP types available in the linux-yocto kernel. In the case of the linux-yocto kernel, this mapping is responsible for assembling the .config from a collection of fragments, but since a custom kernel doesn't have access to these, a **defconfig** that provides the basic set of options needed to boot the machine is explicitly required.

However, because this is a linux-yocto-custom kernel, it does have the ability to specify and reuse config fragments, which is the major difference between this setup and the simple tarball-based kernel used in lab1. To start out, the linux-yocto-custom sources will use the **defconfig** specified, which is compatible with common PC hardware.

Moving on to the external module, open the hello-mod recipe and examine it:

```
$ vi ~/thud/meta-lab4-qemux86/recipes-kernel/hello-mod/hello-mod_0.1.bb
```

The recipe itself is very simple – it names the files that make up the module in the SRC_URI and inherits the module bbclass, which enables the build system to build the code listed as a kernel module. The hello-mod/files directory contains the hello.c kernel source file and a module Makefile, which you can also examine.

Because in this lab you're building the kernel from a local repository, you first need to create a local clone of the kernel you want to use. To do this, cd into the thud directory and create a local clone of the linux-stable kernel:

```
$ cd ~/thud
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git linux-stable-work.git
```

You should see something like the following as output:

```
Cloning into 'linux-stable-work.git'...
remote: Counting objects: 3815662, done.
remote: Compressing objects: 100% (582382/582382), done.
remote: Total 3815662 (delta 3216909), reused 3801382 (delta 3202730)
Receiving objects: 100% (3815662/3815662), 816.82 MiB | 791.00 KiB/s, done.
Resolving deltas: 100% (3216909/3216909), done.
Checking out files: 100% (46788/46788), done.
```

Note: Cloning the kernel can take a long time. You can speed up the clone if you already have a local clone that you can base the new one off of – see 'git-clone –reference' for details).

Now cd into the cloned kernel and check out a branch named 'work-branch':

```
$ cd ~/thud/linux-stable-work.git
$ git checkout -b work-branch remotes/origin/linux-4.18.y
```

You should see something like the following as output:

```
Checking out files: 100% (11240/11240), done.
Branch work-branch set up to track remote branch linux-4.18.y from origin.
Switched to a new branch 'work-branch'
```

Edit the linux-yocto-custom kernel recipe:

```
$ vi ~/thud/meta-lab4-qemux86/recipes-kernel/linux/linux-yocto-custom.bb
```

and change the SRC_URI to point to the local clone you just created. If you've done it as instructed, you should only need to change home/myacct to your home directory:

```
SRC_URI = "git:///home/myacct/thud/linux-stable-work.git"
```

Note also the KBRANCH line in the same file:

```
KBRANCH = "work-branch"
```

The KBRANCH variable names the branch that will be used to build the kernel. If you've checked out and want to work with a different branch, you should change the KBRANCH variable to that branch.

Save your changes and close vi.

Build the Image

OK, you have done this before (don't forget to cd back into the build directory):

```
$ cd ~/thud/build
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab4-qemux86/bzImage-lab4-qemux86.bin tmp/deploy/images/lab4-qemux86/core-image-minimal-lab4-qemux86.ext4
```

Add the External Kernel Module

Now that you have a working kernel, you can add the hello-mod external module to the image. Recall that you don't need to change the kernel configuration to add the module because it won't be made part of the kernel source via a SRC_URI addition as in the previous lab, but will be built as an 'external' module.

To do that, first open the machine configuration file:

```
$ vi ~/thud/meta-lab4-qemux86/conf/machine/lab4-qemux86.conf
```

and uncomment the following line at the end of the file:

```
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "hello-mod"
```

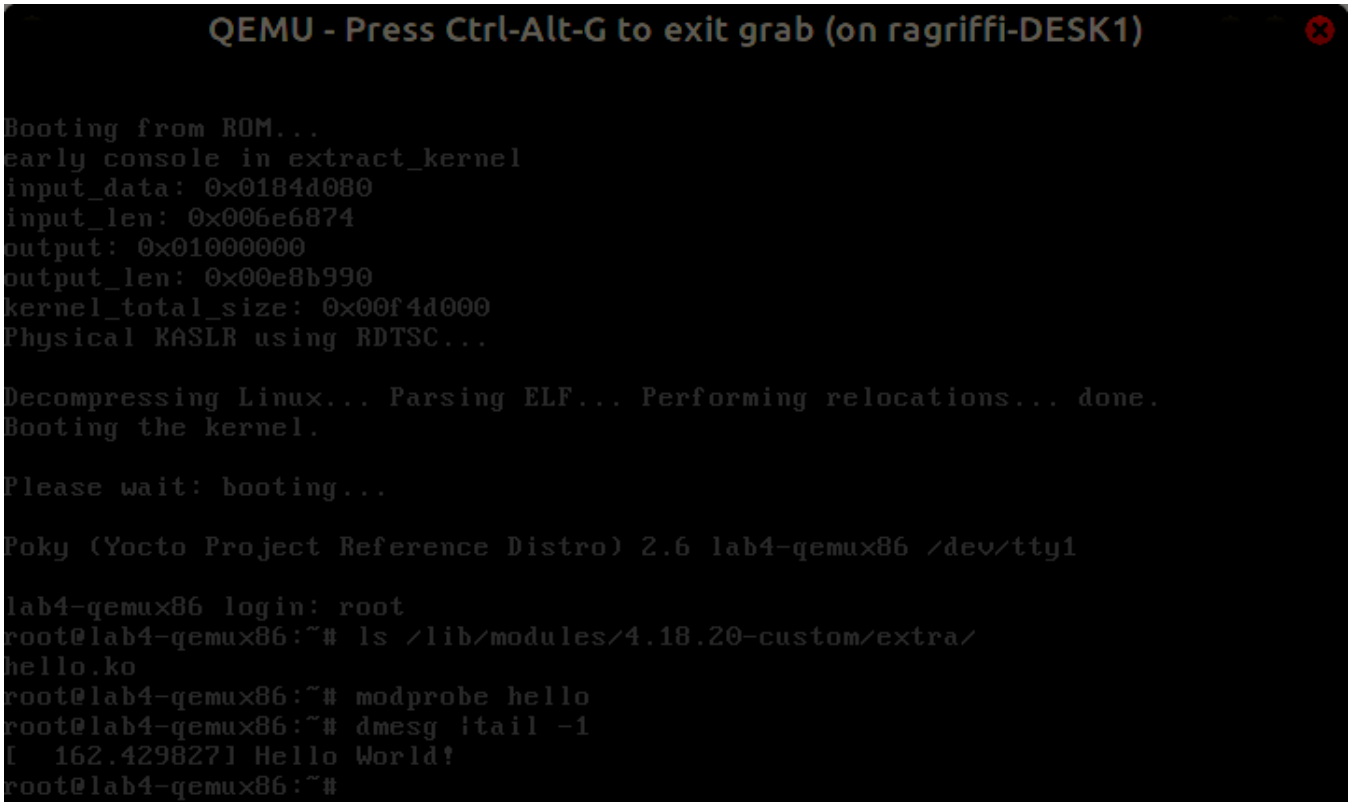
Note that in the MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS line, you used the name of the hello-mod package directly instead of prepending it with 'kernel-module-' as you did in lab3. That's because it has its own package created for it by virtue of the fact that it's a standalone recipe, rather than the synthesized package created by the kernel recipe in the case of lab3.

Note: Your module isn't exactly 'essential' and you'd normally use MACHINE_EXTRA_RRECOMMENDS, but this is the variable you need to use with the minimal image since it doesn't include the base package that includes the latter variable.

Now you can rebuild and boot the new image. You're rebuilding the new image rather than just the kernel in this case because the module is not included in the kernel image but instead is added to the /lib/modules directory of the filesystem image, which requires us to build a new root filesystem. Bitbake will detect the machine configuration has changed and will build and add the new module:

```
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/lab4-qemux86/bzImage-lab4-qemux86.bin tmp/deploy/images/lab4-qemux86/core-image-minimal-lab4-qemux86.ext4
```

Logging into the machine and looking around, you can see that the new module was indeed added to the image, in this case the /lib/modules/4.18.20-custom/extra directory, which you see contains your hello.ko module. You can load it and see the results using 'modprobe hello':



Modify the local kernel

The main reason to use a local kernel is to be able to easily modify and rebuild it, and test the changes.

To demonstrate that, you'll make a simple modification to the kernel code and see the results in the booted system.

Change directories into the local kernel repository and open the fs/filesystems.c source file:

```
$ cd ~/thud/linux-stable-work.git
$ vi fs/filesystems.c
```

Scroll down to the filesystems_proc_show(...) function (you can use the Search | Find... option in vi to more quickly locate it):

```
static int filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
           seq_printf(m, "%s¥t%s¥n",
                  (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
                  tmp->name);
           tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);
    return 0;
}
```

Add a simple printk() to that function, so that when you 'cat /proc/filesystems' in the booted image you'll see a message in the kernel logs.

```
printk("Kilfoy was here!¥n");
```

After adding the printk(), filesystems_proc_show(...) should look like this:

```
static int filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
           seq_printf(m, "%s¥t%s¥n",
                  (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
                  tmp->name);
           tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);

   printk("Kilfoy was here!¥n");

    return 0;
}
```

Verify that the code was changed using 'git diff':

```
$ git diff -p HEAD
```

You should see something like the following as output:

```
diff --git a/fs/filesystems.c b/fs/filesystems.c
index 5797d45..e954512 100644
--- a/fs/filesystems.c
+++ b/fs/filesystems.c
@@ -233,6 +233,9 @@ static int filesystems_proc_show(struct seq_file *m, void *v)
                tmp = tmp->next;
        }
        read_unlock(&file_systems_lock);
+
+       printk("Kilfoy was here!¥n");
+
        return 0;
 }
```

In order for the build to pick up the change, you need to commit the changes:

```
$ git commit -a -m "fs/filesystems.c: add a message that will be logged to the kernel log
when you 'cat /proc/filesystems'."
```

You should see the following output if your commit was successful:

```
[work-branch ca05d6b] fs/filesystems.c: add a message that will be logged to the kernel log
when you 'cat /proc/filesystems'.
 1 file changed, 3 insertions(+)
```

You can also verify that the change was indeed added to the current branch via 'git log':

```
$ git log
```

You should see something like this in the output of 'git log':

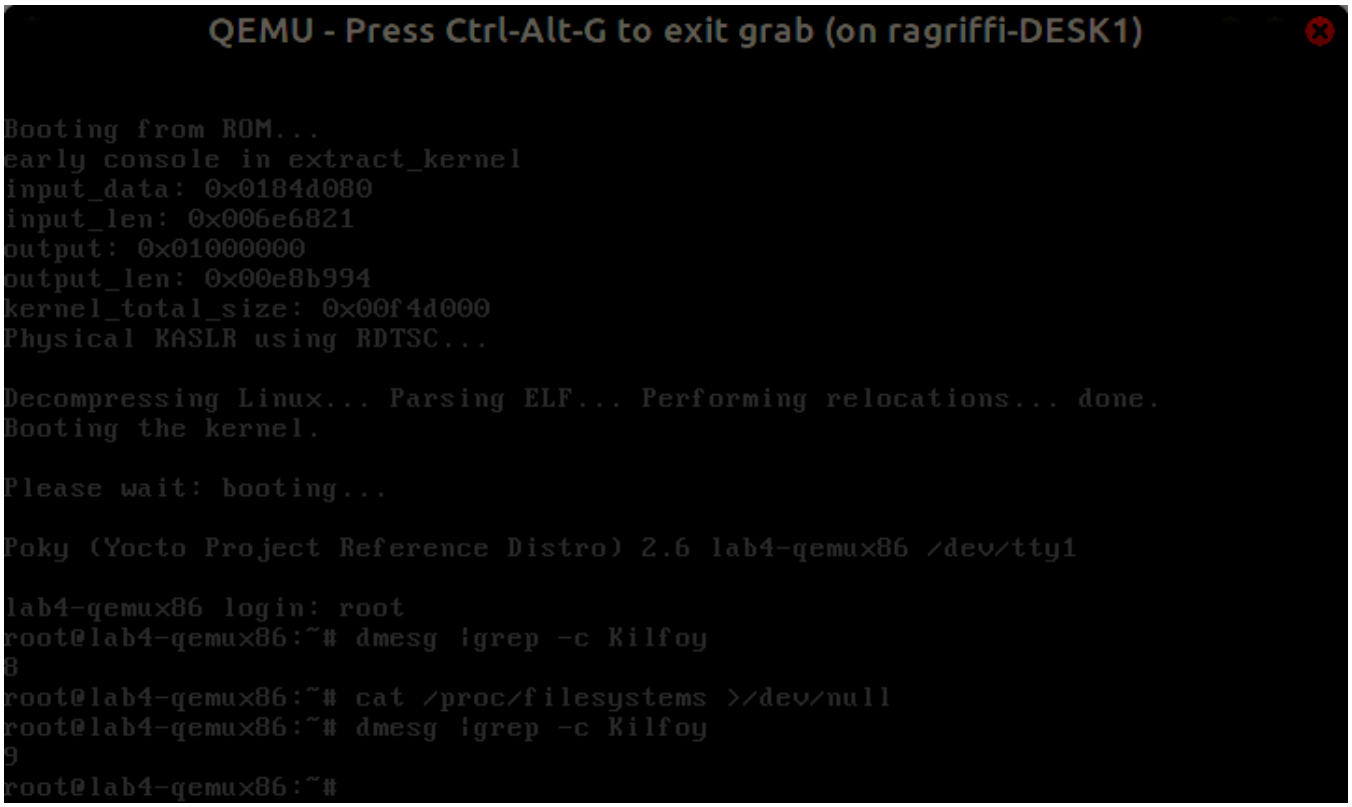
```
commit ca05d6b5a9f3d77ee07ebf3c7382ad9f244a62ea
Author: your name <your_email_address>
Date:   Mon Dec 7 13:19:54 2015 -0800
```

```
    fs/filesystems.c: add a message that will be logged to the kernel log when you 'cat
/proc/filesystems'.
```

You should now be able to rebuild the kernel and see the changes. There is one difference in this case however – when using a local clone, you need to do a 'cleanall' of the kernel recipe. The reason for that is that the build system caches the kernel (as a hidden file in the downloads/git2 in case you're interested) that it last downloaded and will use that cached copy if present and won't fetch the modified copy, even if built from a completely clean state. Forcing a 'cleanall' on the recipe clears out that cached copy as well and allows the build system to see your kernel changes (don't worry too much about the cost of having to fetch the kernel again – since the 'upstream' kernel is local the fetch is also local and significantly faster than a normal kernel fetch over the network):

```
$ cd ~/thud/build
$ bitbake -c cleanall virtual/kernel
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/lab4-qemux86/bzImage-lab4-qemux86.bin tmp/deploy/images/lab4-
qemux86/core-image-minimal-lab4-qemux86.ext4
```

The boot process output shows that /proc/filesystems is read by other processes, which produces multiple messages in the boot output. You can however show the new code in action by cat'ing that file yourself and seeing that the number of printk lines increases in the kernel log:



Using a local linux-yocto-based kernel

For this lab, you used the linux-yocto-custom recipe with a local repository, but it should be noted that you can do the same thing with the standard linux-yocto kernel, which is actually the more common use-case.

To do that you essentially repeat the previous set of steps but with the linux-yocto kernel instead. The main difference is that you need a slightly different SRC_URI, which needs to track two projects instead of one – the kernel source and kernel cache. The following steps can be used to use a local version of the linux-yocto kernel.

```
$ cd ~/thud
$ git clone -b v4.18 git://git.yoctoproject.org/linux-yocto linux-yocto-4.18.git
```

You should see something like the following as output:

```
Cloning into bare repository 'linux-yocto-4.18.git'...
remote: Counting objects: 4165216, done.
remote: Compressing objects: 100% (620692/620692), done.
Receiving objects: 100% (4165216/4165216), 860.73 MiB | 10.85 MiB/s, done.
remote: Total 4165216 (delta 3515097), reused 4163458 (delta 3513339)
Resolving deltas: 100% (3515097/3515097), done.
Checking connectivity... done.
```

Now cd into the working clone and create a working branch named 'standard/base':

```
$ cd linux-yocto-4.18.git/
$ git checkout v4.18/standard/base
```

You should see something like the following as output:

```
Previous HEAD position was 94710cac0ef4... Linux 4.18
Branch v4.18/standard/base set up to track remote branch v4.18/standard/base from origin.
Switched to a new branch 'v4.18/standard/base'
```

Switch to the lab2 layer

For this, you'll be reusing lab2, which uses the linux-yocto kernel already:

Open local.conf (don't forget to cd back into the build directory):

```
$ cd ~/thud/build
$ vi conf/local.conf
```

Add the following line just above the line that says 'MACHINE ??= “qemux86”:

```
MACHINE ?= "lab2-qemux86"
```

Save your changes and close vi.

Now open bblayers.conf:

```
$ vi conf/bblayers.conf
```

and add the 'meta-lab2-qemux86' layer to the BBLAYERS variable. The final result should look like this, assuming your account is called 'myacct' (simply copy the line containing 'meta-yocto-bsp' and replace 'meta-yocto-bsp' with 'meta-lab2-qemux86'):

```
BBLAYERS ?= " ¥
  /home/myacct/thud/meta ¥
  /home/myacct/thud/meta-yocto ¥
  /home/myacct/thud/meta-yocto-bsp ¥
  /home/myacct/thud/meta-lab2-qemux86 ¥
  "
```

You should not need to make any further changes. Save your changes and close vi.

Modify the lab2 kernel to use the local linux-yocto repo

Edit the linux-yocto kernel recipe:

```
$ vi ~/thud/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_4.18.bbappend
```

You'll need to enable the new SRC_URI to point to the local linux-yocto clone you just created. If you've done it as instructed, you should only need to change home/myacct to your home directory and uncomment the following lines:

```
SRC_URI = "git:///home/myacct/thud/linux-yocto-4.18.git;name=machine;branch=${KBRANCH}; ¥
          git://git.yoctoproject.org/yocto-kernel-cache;type=kmeta;name=meta;branch=yocto-
4.18;destsuffix=${KMETA}"
KERNEL_VERSION_SANITY_SKIP="1"
```

Also, comment out the current SRCREV lines and uncomment the following SRCREV lines:

```
SRCREV_machine_pn-linux-yocto_lab2-qemux86 ?= "${AUTOREV}"
SRCREV_meta_pn-linux-yocto_lab2-qemux86 ?= "${AUTOREV}"
```

This ensures that the kernel build will see the latest commits on the referenced git branches, which is what you

typically want during development. Save your changes and close vi.
Also, make sure that the PREFERRED_VERSION of linux-yocto is set to 4.18 (remember that for the second part of lab2, we switched it to 4.14, so need to switch it back to 4.18 now). Open the machine configuration file for lab2 in vi:

```
$ vi ~/thud/meta-lab2-qemux86/conf/machine/lab2-qemux86.conf
```

Change the preferred version of the linux-yocto kernel back to 4.18 by commenting out the 4.14 line and uncommenting the 4.18 line as such:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.18%"
#PREFERRED_VERSION_linux-yocto ?= "4.14%"
```

Rebuild the Kernel

OK, you have done this before (don't forget to cd back into the build directory):

```
$ cd ~/thud/build
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-qemux86/core-image-minimal-lab2-qemux86.ext4
```

Modify the local linux-yocto-based kernel

At this point, you have the same setup with the linux-yocto-based kernel as you did with the linux-yocto-custom-based kernel, so you should be able to follow the same sequence of steps outlined in the previous section titled '**Modify the local kernel**' to modify the kernel. i.e. add the following statement to filesystems_proc_show():

```
printk("Kilfoy was here!¥n");
```

using these steps:

```
$ cd ~/thud/linux-yocto-4.18.git
$ vi fs/filesystems.c
$ git commit -a -m "fs/filesystems.c: add a message that will be logged to the kernel log when you 'cat /proc/filesystems'."
```

After adding the printk(), filesystems_proc_show(...) should look like this:

```
static int filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
            seq_printf(m, "%s¥t%s¥n",
                    (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
                    tmp->name);
            tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);

   printk("Kilfoy was here!¥n");

    return 0;
}
```

Again, You can also verify that the change was indeed added to the current branch via 'git log':

```
$ git log v4.18/standard/base
```

You should see something like the following as output:

```
commit da690df77b440987db18936ec2ece2b6cf2097d6
Author: your name <your_email_address>
```

```
Date:   Mon Dec 7 13:19:54 2015 -0800

    fs/filesystems.c: add a message that will be logged to the kernel log when you 'cat
/proc/filesystems'.
```

Now rebuild the kernel and you should see your change appear:

```
$ cd ~/thud/build
$ bitbake -c cleanall virtual/kernel
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/lab2-qemux86/bzImage-lab2-qemux86.bin tmp/deploy/images/lab2-
qemux86/core-image-minimal-lab2-qemux86.ext4
```

Lab 4 Conclusion

In this lab you built and booted an arbitrary git-based non-linux-yocto Linux kernel as a local repository, which you then modified, and you immediately saw the results of your changes after rebuilding the kernel. In addition, you were also able to do the same workflow using a local clone of the linux-yocto kernel. You also added and loaded an external kernel module. This concludes Lab 4.

NOTES

NOTES