Hands-on Kernel Lab

Based on the Yocto Project 1.4 release (dylan)

April 2013

Tom Zanussi <tom.zanussi@linux.intel.com>
Darren Hart <dvhart@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**)

- Modifying the locally cloned linux-yocto kernel source and verifying the changes in the new image (**lab4**)

- Using a 'bare' local clone of a linux-yocto- kernel recipe to demonstrate a typical development workflow (**lab4**)

- Modifying the locally cloned 'bare' linux-yocto kernel source and verifying the changes in the new image (**lab4**)

- Adding and using an external kernel module via a module recipe (**lab4**)

- Using the 'Yocto BSP Tools' yocto-bsp tool generate a new Yocto BSP (**lab5**)

- Using the 'Yocto BSP Tools' yocto-kernel tool to add kernel patches and config fragments (**lab5**)

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 12.04 "Precise Pangolin" 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 17 system.

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

http://www.yoctoproject.org/docs/1.4/yocto-project-qs/yocto-project-qs.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 download the Yocto 1.4 'dylan' sources into your home directory. From your terminal shell, type:

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.4/poky-dylan-
9.0.0.tar.bz2
$ bunzip2 -c poky-dylan-9.0.0.tar.bz2 | tar xvf -
```

Once you've gotten the Yocto 1.4 sources, you should cd into the the poky-dylan-9.0.0 directory that was created::

```
$ cd poky-dylan-9.0.0/
```

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

```
$ ls
bitbake LICENSE  meta-hob  meta-yocto  oe-init-build-env  README.hardware
documentation  meta  meta-skeleton  meta-yocto-bsp  README  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-
1.4-layers.tar.bz2
$ bunzip2 -c kernel-lab-1.4-layers.tar.bz2 | tar xvf -
```

Listing the files in the current directory, which should still be poky-dylan-9.0.0, should now show the following files and subdirectories:

```
$ ls
bitbake                         meta-hob            meta-lab5-qemuarm.finished
README
documentation                   meta-lab1-qemux86  meta-skeleton
README.hardware
kernel-lab-1.4-layers.tar.bz2   meta-lab2-qemux86  meta-yocto
scripts
LICENSE                         meta-lab3-qemux86  meta-yocto-bsp
meta                            meta-lab4-qemux86  oe-init-build-env
```

Lab 1: Traditional Kernel Recipe

In this lab you will modify a stock 3.0.18 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.

Set up the Environment

```
$ cd ~/poky-dylan-9.0.0/
$ source oe-init-build-env
```

Open local.conf:

```
$ gedit conf/local.conf
```

and uncomment the following lines:

```
BB_NUMBER_THREADS = "4"
PARALLEL_MAKE = "-j 4"
```

If you have a different number of processors, change those values to match the number of processors you have in your machine.

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

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

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit 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-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab1-qemux86 \
  "
```

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

The meta-lab1-qemux86 layer provides a very simple Linux 3.0.18 recipe. Open it in gedit for review:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab1-qemux86/recipes-
kernel/linux/linux_3.0.18.bb
```

This is a bare-bones simple Linux kernel recipe. It inherits all of the logic for configuring and building the kernel from the **kernel.bbclass** (the 'inherit kernel' line). It specifies the Linux kernel sources in SRC_URI as a recent tarball release from kernel.org. It also specifies a **defconfig**, this file is used as the **.config** to build the kernel. It is empty for now, so the Linux kernel configuration system will use

www.yoctoproject.org

yocto · PROJECT   THE LINUX FOUNDATION

defaults. 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 gedit for review:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab1-qemux86/conf/machine/lab1-qemux86.conf
```

Without going in to 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 a specific implementation and/or version if it knows it doesn't want to use the defaults. It may also want to specify specific 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"
```

The reason it does this is that if it didn't, it would pick up the default linux-yocto_3.8 kernel (which is the version specified in the qemu.inc file included following that line). Also, because there's only a single linux_* 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.

yocto · PROJECT

THE LINUX FOUNDATION

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).
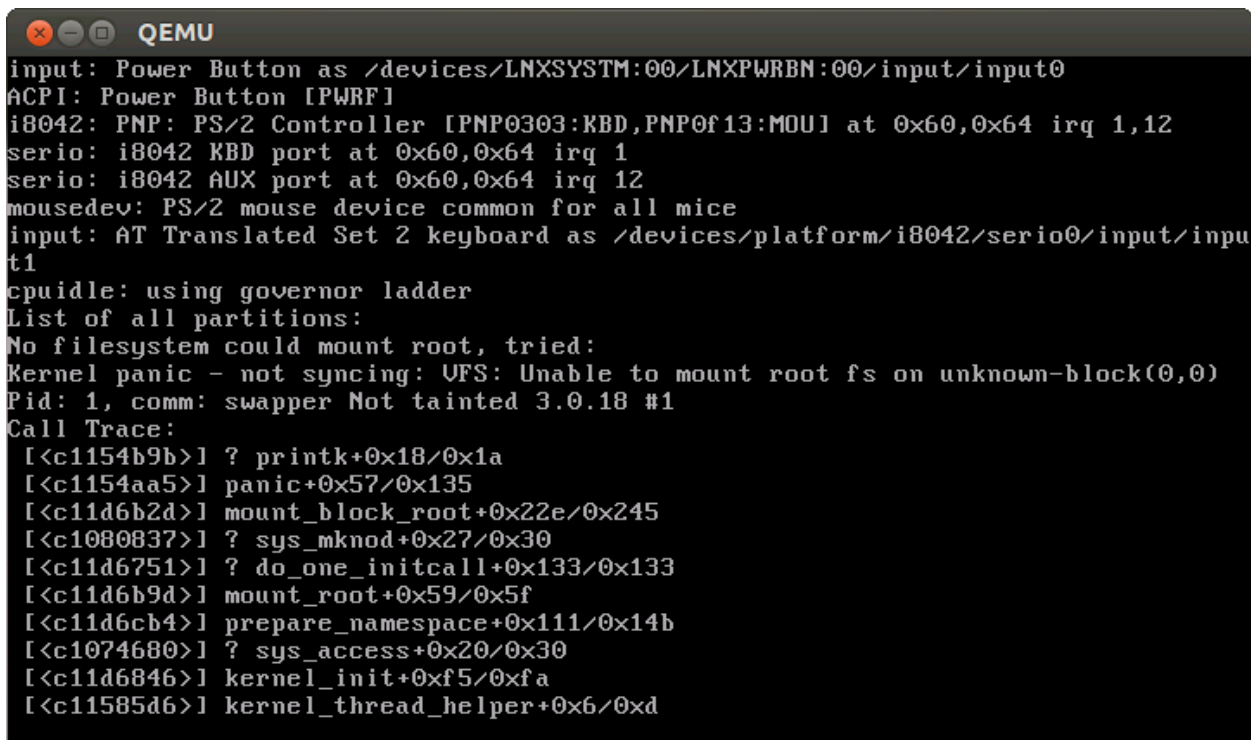
```
$ bitbake core-image-minimal
```

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

Now boot the image with QEMU:

```
$ runqemu tmp/deploy/images/bzImage-lab1-qemux86.bin tmp/deploy/images/core-
image-minimal-lab1-qemux86.ext3
```

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

```
😵⊖⊡  QEMU
input: Power Button as /devices/LNXSYSTM:00/LNXPWRBN:00/input/input0
ACPI: Power Button [PWRF]
i8042: PNP: PS/2 Controller [PNP0303:KBD,PNP0f13:MOU] at 0x60,0x64 irq 1,12
serio: i8042 KBD port at 0x60,0x64 irq 1
serio: i8042 AUX port at 0x60,0x64 irq 12
mousedev: PS/2 mouse device common for all mice
input: AT Translated Set 2 keyboard as /devices/platform/i8042/serio0/input/inpu
t1
cpuidle: using governor ladder
List of all partitions:
No filesystem could mount root, tried:
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
Pid: 1, comm: swapper Not tainted 3.0.18 #1
Call Trace:
 [<c1154b9b>] ? printk+0x18/0x1a
 [<c1154aa5>] panic+0x57/0x135
 [<c11d6b2d>] mount_block_root+0x22e/0x245
 [<c1080837>] ? sys_mknod+0x27/0x30
 [<c11d6751>] ? do_one_initcall+0x133/0x133
 [<c11d6b9d>] mount_root+0x59/0x5f
 [<c11d6cb4>] prepare_namespace+0x111/0x14b
 [<c1074680>] ? sys_access+0x20/0x30
 [<c11d6846>] kernel_init+0xf5/0xfa
 [<c11585d6>] kernel_thread_helper+0x6/0xd
```

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.

Reconfigure the Linux Kernel

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

```
$ bitbake linux -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
  Generic Driver Options  --->
    [*] Maintain a devtmpfs filesystem to mount at /dev
    [*]   Automount devtmpfs at /dev, after the kernel mounted the rootfs
File systems  --->
  [*] Ext3 journaling file system support
```

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 linux -c deploy
$ runqemu tmp/deploy/images/bzImage-lab1-qemux86.bin tmp/deploy/images/core-image-minimal-lab1-qemux86.ext3
```

www.yoctoproject.org

yocto·
PROJECT

THE
LINUX
FOUNDATION

```
QEMU – Press Ctrl-Alt to exit mouse grab
```

```
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/i
nput2
Starting udev
udevd[36]: error getting socket: Function not implemented

error initializing udev control socketudevd[36]: error initializing udev control
 socket
error getting socket: Function not implemented
udevadm[37]: error getting socket: Function not implemented

error getting socket: Function not implemented
udevadm[39]: error getting socket: Function not implemented

Starting Bootlog daemon: bootlogd.
Configuring network interfaces... ifconfig: socket: Function not implemented
ifconfig: socket: Function not implemented
hwclock: can't open '/dev/misc/rtc': No such file or directory
INIT: Entering runlevel: 5
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Stopping Bootlog daemon: bootlogd.

Poky 9.0 (Yocto Project 1.4 Reference Distro) 1.4 lab1-qemux86 tty1

lab1-qemux86 login: _
```

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/3.0.18-r0/linux-3.0.18/.config
~/poky-dylan-9.0.0/meta-lab1-qemux86/recipes-kernel/linux/linux/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:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab1-qemux86/recipes-
kernel/linux/linux/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:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab1-qemux86/recipes-
kernel/linux/linux_3.0.18.bb
```

Uncomment the following line:

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

Save your changes and close gedit.

Now use **menuconfig** to enable the driver. Bitbake will detect the recipe file has changed and start by fetching the new sources and apply the patch.
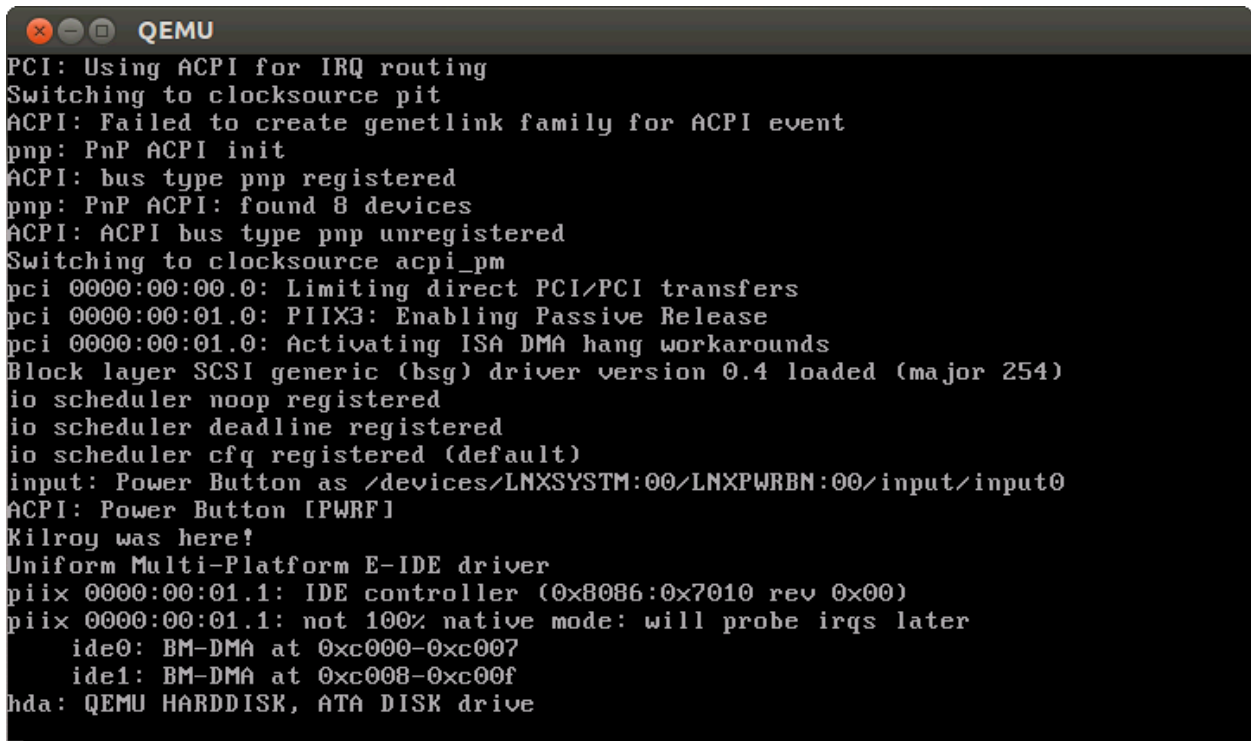
```
$ bitbake linux -c menuconfig

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

Exit and save your changes as before.

Hands-on Kernel Lab
Page 10 of 39

www.yoctoproject.org

yocto ·

THE
LINUX
FOUNDATION

Now rebuild the kernel and boot it in QEMU.

```
$ bitbake linux -c deploy
$ runqemu tmp/deploy/images/bzImage-lab1-qemux86.bin tmp/deploy/images/core-
image-minimal-lab1-qemux86.ext3
```

You can scroll back through the boot log using `Shift+PgUp`. You should find the Yocto test driver message in there.

```
PCI: Using ACPI for IRQ routing
Switching to clocksource pit
ACPI: Failed to create genetlink family for ACPI event
pnp: PnP ACPI init
ACPI: bus type pnp registered
pnp: PnP ACPI: found 8 devices
ACPI: ACPI bus type pnp unregistered
Switching to clocksource acpi_pm
pci 0000:00:00.0: Limiting direct PCI/PCI transfers
pci 0000:00:01.0: PIIX3: Enabling Passive Release
pci 0000:00:01.0: Activating ISA DMA hang workarounds
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
input: Power Button as /devices/LNXSYSTM:00/LNXPWRBN:00/input/input0
ACPI: Power Button [PWRF]
Kilroy was here!
Uniform Multi-Platform E-IDE driver
piix 0000:00:01.1: IDE controller (0x8086:0x7010 rev 0x00)
piix 0000:00:01.1: not 100% native mode: will probe irqs later
    ide0: BM-DMA at 0xc000-0xc007
    ide1: BM-DMA at 0xc008-0xc00f
hda: QEMU HARDDISK, ATA DISK drive
```

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-3.0.18-r0/linux-3.0.18/.config
~/poky-dylan-9.0.0/meta-lab1-qemux86/recipes-kernel/linux/linux/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 ~/poky-dylan-9.0.0/
$ source oe-init-build-env
```

Open local.conf:

```
$ gedit conf/local.conf
```

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

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

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit 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/poky-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab2-qemux86 \
  "
```

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

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_3.4.bbappend
    linux-yocto_3.8.bbappend
    files /
      pramfs.cfg
      lab2.cfg
      yocto-testmod.patch
```

Open the 3.8 kernel recipe:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_3.8.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/common-pc/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 essentially maps to the 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 to the meta-lab1-qemux86 in Lab 1. Open it in gedit for review:

```
$ gedit ~/poky-dylan-9.0.0/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 ?= "3.8%"
```

Because the Lab 2 layer has multiple kernel implementations available to it (linux-yocto_3.4, linux-yocto_3.8, and linux-yocto-dev), 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_3.8 version via the PREFERRED_VERSION setting (the trailing '%' serves as a wildcard, meaning in this case to essentially 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 3.8 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/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

The `linux-yocto` repository meta-data already provides the driver support needed to boot in QEMU – the missing configuration settings for PIIX/ICH, ext3, 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 uproductive 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:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-kernel/linux/linux-
yocto_3.8.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 gedit.

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:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-
kernel/linux/files/lab2.cfg
```

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

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

Close gedit.

> **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/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

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 LTSI Kernel Option

Note: This exercise shows how to enable the LTSI pramfs feature using the **pramfs.cfg** config option. This is very similar to the previous exercise in which you enabled the Yocto 'testmod' using the **lab2.cfg** fragment. The value in this case is to show that LTSI kernel options can be used in exactly the same way. As such, if you're not interested specifically in LTSI, you can safely skip the next few sections.

The linux-yocto-3.8 kernel is the most up-to-date version of the linux-yocto kernel available in Yocto 1.4. Because it hasn't yet had a corresponding LTSI version released for it yet, it doesn't yet support any LTSI features. However, the linux-yocto-3.4 kernel is also available, and it has had the LTSI kernel patches merged in. To take advantage of those LTSI features, you'll first need to switch kernels to the linux-yocto-3.4 kernel. To do that, you'll change the preferred kernel version to 3.4 in the lab2 machine configuration by editing the lab2-qemux86.conf file:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/conf/machine/lab2-qemux86.conf
```

Change the preferred version of the kernel in that file from 3.8 to 3.4. In other words, change:

```
PREFERRED_VERSION_linux-yocto ?= "3.8%"
#PREFERRED_VERSION_linux-yocto ?= "3.4%"
```

to:

```
#PREFERRED_VERSION_linux-yocto ?= "3.8%"
PREFERRED_VERSION_linux-yocto ?= "3.4%"
```

Because the linux-yocto-3.4 kernel has transparently merged in the 3.4 LTSI kernel, all of the capabilities of the LTSI kernel are now available and accessible via linux-yocto config fragments.

We'll demonstrate this by enabling a config option normally available only to the LTSI kernel, for the 'pramfs' file system, now also available in the linux-yocto kernel by virtue of the merger of the LTSI kernel into linux-yocto-3.4.

Edit the linux-yocto kernel recipe:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-kernel/linux/linux-
yocto_3.4.bbappend
```

and uncomment the line including the pramfs config fragment:

```
SRC_URI += "file://pramfs.cfg"
```

Save your changes and close gedit.

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 **pramfs.cfg** file.

Open the file:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-
kernel/linux/files/pramfs.cfg
```

and examine the following lines, which enables the pramfs filesystem module as a built-in kernel module:

```
# Enable LTSI's pramfs
CONFIG_PRAMFS=y
CONFIG_PRAMFS_WRITE_PROTECT=y
```
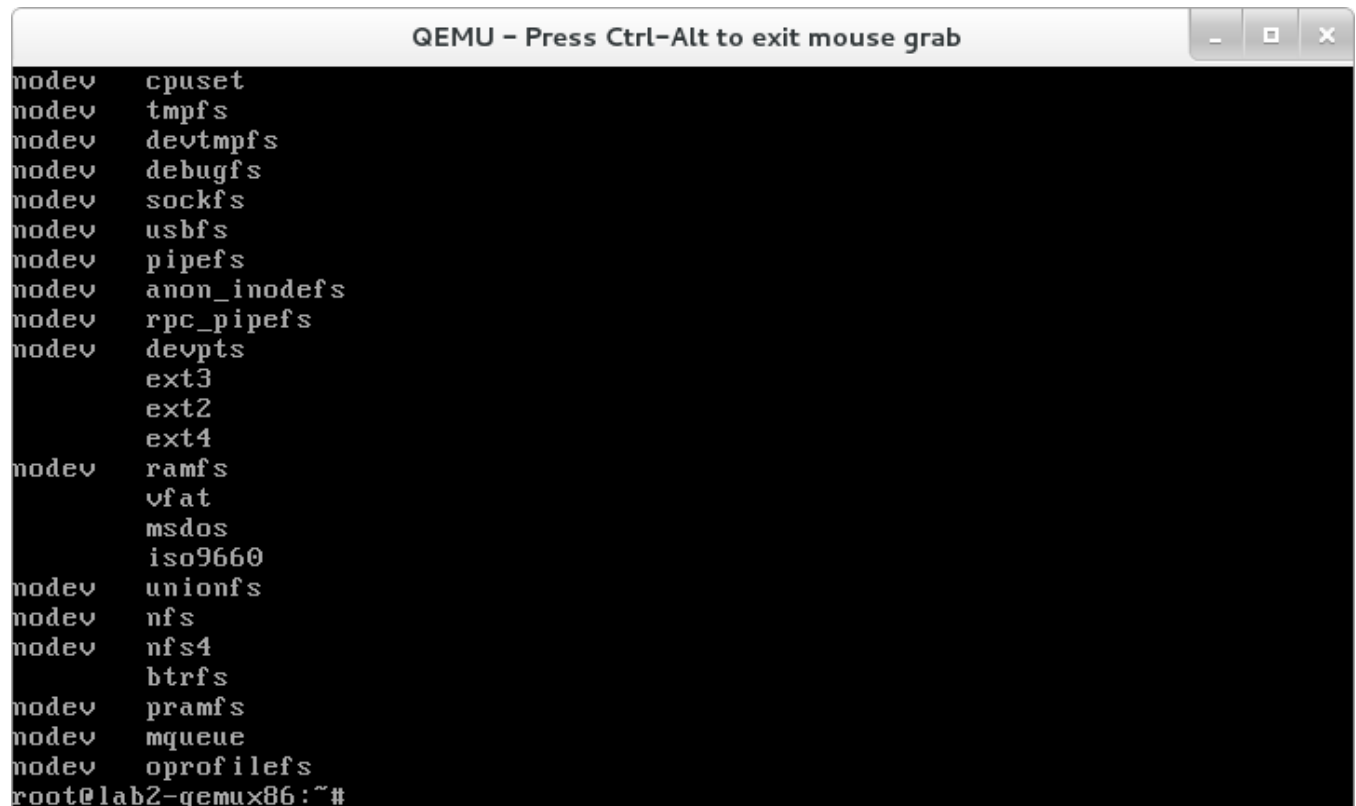
Close gedit.

Rebuild the Kernel

Now you can rebuild and boot the new kernel:

```
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

You can verify that the LTSI pramfs module has been added to the kernel by looking at the output of 'cat /proc/filesystems', which will now show an entry for the pramfs filesystem, which wasn't there before (see the third entry from the bottom):

www.yoctoproject.org

yocto · PROJECT

THE LINUX FOUNDATION

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 enabled an LTSI kernel option using a config fragment, which is possible because the 3.4 linux-yocto kernel has merged the 3.4 LTSI code. 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 the **WORKDIR**:

```
$ cd ~/poky-dylan-9.0.0/build/tmp/work/lab2_qemux86-poky-linux/linux-
yocto/3.8.4+gitAUTOINC+27b63fdbd25ad1a37bacc05f49a205c150d21779_d9a45e3325030
f7bd6f37947a7a0b12da7f602c3-r4.1.0/linux
```

> **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 ~/poky-dylan-9.0.0/build
$ bitbake linux-yocto -c compile -f
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

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**.

Hands-on Kernel Lab
Page 17 of 39

www.yoctoproject.org

yocto ·
PROJECT

THE
LINUX
FOUNDATION

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 ~/poky-dylan-9.0.0/
$ source oe-init-build-env
```

Open local.conf:

```
$ gedit conf/local.conf
```

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

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

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit 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/poky-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab3-qemux86 \
  "
```

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

Review the Lab 3 Layer

This layer differs from meta-lab3-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:

```
$ gedit ~/poky-dylan-9.0.0/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 poky-dylan-9.0.0/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 tarball-based 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/bzImage-lab3-qemux86.bin tmp/deploy/images/core-
image-minimal-lab3-qemux86.ext3
```

Modify the Kernel

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

Edit the linux-yocto-custom kernel recipe:

```
$ gedit ~/poky-dylan-9.0.0/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 gedit.

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:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab3-qemux86/recipes-kernel/linux/linux-
yocto-custom/lab3.cfg
```

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

www.yoctoproject.org

yocto · PROJECT

THE LINUX FOUNDATION

```
# 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 gedit.

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/bzImage-lab3-qemux86.bin tmp/deploy/images/core-image-minimal-lab3-qemux86.ext3
```

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 (ye should see a 'kernel/drivers/misc' directory in /lib/modules/3.8.11-yocto-standard, containing your yocto-testmod.ko kernel module binary):

```
                  QEMU – Press Ctrl–Alt to exit mouse grab                 _ □ ✕

INIT: version 2.88 booting

Please wait: booting...
Starting udev
udevd[64]: starting version 182
Starting Bootlog daemon: bootlogd.
Configuring network interfaces... udhcpc (v1.20.2) started
Sending discover...
Sending discover...
Sending discover...
No lease, failing
INIT: Entering runlevel: 5
Starting syslogd/klogd: done
Stopping Bootlog daemon: bootlogd.

Poky 9.0 (Yocto Project 1.4 Reference Distro) 1.4 lab3-qemux86 tty1

lab3-qemux86 login: root
root@lab3-qemux86:~# lsmod
    Not tainted
root@lab3-qemux86:~# ls /lib/modules/3.8.11-yocto-standard/
modules.alias        modules.dep          modules.softdep
modules.alias.bin    modules.dep.bin      modules.symbols
modules.builtin.bin  modules.devname      modules.symbols.bin
root@lab3-qemux86:~#
```

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 ~/poky-dylan-9.0.0/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-
3.8.11+git0+c1be5a5b1b355d40e6cf79cc979eb66dafa24ad1-r0.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:

```
$ gedit ~/poky-dylan-9.0.0/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. To do that, you'll uncomment the
following line to the linux-yocto-custom recipe:

```
module_autoload_yocto-testmod = "yocto-testmod"
```

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

---

www.yoctoproject.org   yocto · PROJECT   THE LINUX FOUNDATION

```
$ gedit ~/poky-dylan-9.0.0/meta-lab3-qemux86/recipes-kernel/linux/linux-
yocto-custom.bb
```

Because you've changed the output of the recipe, you also need to 'bump the PR' to tell the build system to rebuild the kernel. Find the line that says 'PR= "r1"' and change it to:
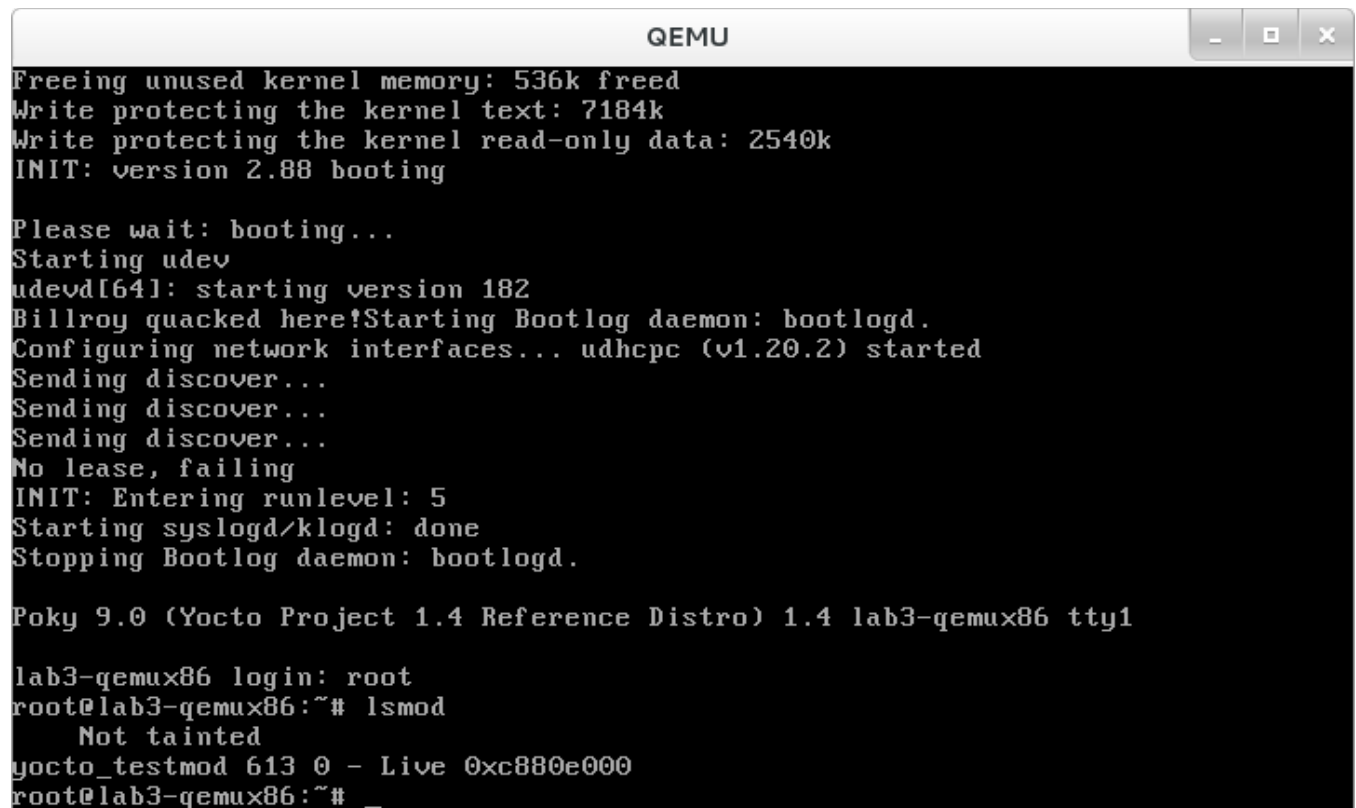
```
PR = "r2"
```

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/bzImage-lab3-qemux86.bin tmp/deploy/images/core-
image-minimal-lab3-qemux86.ext3
```

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 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 making changes to the kernel code, which this lab will also demonstrate. This lab will also demonstrate how to create and use a recipe used to build and install an external kernel module.

Set up the Environment

```
$ cd ~/poky-dylan-9.0.0/
$ source oe-init-build-env
```

Open local.conf:

```
$ gedit conf/local.conf
```

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

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

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit 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/poky-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab4-qemux86 \
  "
```

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

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:

```
$ gedit ~/poky-dylan-9.0.0/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 poky-dylan-9.0.0/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:

```
$ gedit ~/poky-dylan-9.0.0/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 poky-dylan-9.0.0 directory and create a local clone of the linux-stable kernel:

```
$ cd ~/poky-dylan-9.0.0
$ 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: 3017120, done.
remote: Compressing objects: 100% (469816/469816), done.
```

www.yoctoproject.org

yocto · PROJECT

THE LINUX FOUNDATION

```
remote: Total 3017120 (delta 2535629), reused 3000147 (delta 2519086)
Receiving objects: 100% (3017120/3017120), 625.23 MiB | 610 KiB/s, done.
Resolving deltas: 100% (2535629/2535629), done.
Checking out files: 100% (41520/41520), 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 ~/poky-dylan-9.0.0/linux-stable-work.git
$ git checkout -b work-branch remotes/origin/linux-3.8.y
```

You should see something like the following as output:

```
Branch work-branch set up to track remote branch linux-3.8.y from origin.
Switched to a new branch 'work-branch'
```

Edit the linux-yocto-custom kernel recipe:

```
$ gedit ~/poky-dylan-9.0.0/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/poky-dylan-9.0.0/linux-stable-
work.git;protocol=file;bareclone=1"
```

Save your changes and close gedit.

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.

Build the Image

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

```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/bzImage-lab4-qemux86.bin tmp/deploy/images/core-
image-minimal-lab4-qemux86.ext3
```

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:

```
$ gedit ~/poky-dylan-9.0.0/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-
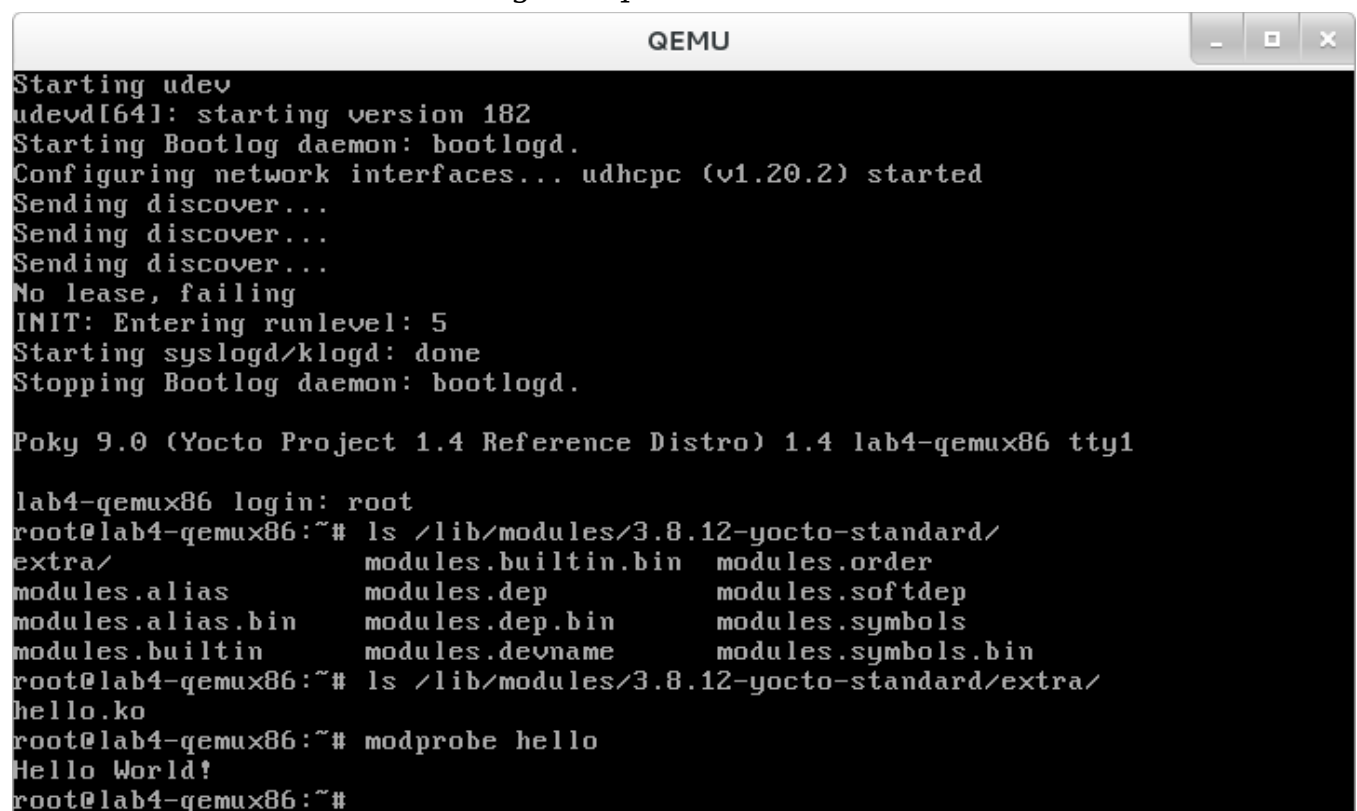
module-' as you did in lab3. That's because it has its own package created for it by virtue of the fact that it's a standalone recipe, rather than the synthesized package created by the kernel recipe in the case of lab3.

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/bzImage-lab4-qemux86.bin tmp/deploy/images/core-image-minimal-lab4-qemux86.ext3
```

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/3.8.23-yocto-standard/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 ~/poky-dylan-9.0.0/linux-stable-work.git
$ gedit fs/filesystems.c
```

Scroll down to the filesystems_proc_show(...) function (you can use the Search | Find... option in Gedit 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 96f2428..9712c33 100644
--- a/fs/filesystems.c
```

www.yoctoproject.org

yocto · PROJECT

THE LINUX FOUNDATION

```
+++ b/fs/filesystems.c
@@ -231,6 +231,9 @@ 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 c619044] fs/filesystems.c: add a message that will be logged to
the kernel log when you 'cat /proc/filesystems'.
 1 file changed, 2 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 8823416a46ecf40ae88baa10ecacdf670c2c4f48
Author: Tom Zanussi <tom.zanussi@linux.intel.com>
Date:   Thu May 9 11:13:49 2013 -0500

    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 forget to cd back into the build directory):

```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake -c cleanall virtual/kernel
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/bzImage-lab4-qemux86.bin tmp/deploy/images/core-
image-minimal-lab4-qemux86.ext3
```

Note: The 'cleanall' step isn't necessary when using a 'bare clone', which you can see an example of in the 'Using a local linux-yocto-based kernel as a bare clone' section below. You can use the 'bare clone' method described there in place of the 'straight clone' method here if desired – simply create linux-stable-work.git as a bare clone and push to it from a straight clone of that.

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 branches instead of one – the 'machine' and the 'meta' branches. The following steps can be used to use a local version of the linux-yocto kernel.

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 poky-dylan-9.0.0 directory and create a local clone of the linux-yocto-3.8 kernel:

```
$ cd ~/poky-dylan-9.0.0
$ git clone git://git.yoctoproject.org/linux-yocto-3.8 linux-yocto-3.8.git
```

You should see something like the following as output:

```
Cloning into 'linux-yocto-3.8.git'...
remote: Counting objects: 2851607, done.
remote: Compressing objects: 100% (456267/456267), done.
remote: Total 2851607 (delta 2389861), reused 2829536 (delta 2367803)
Receiving objects: 100% (2851607/2851607), 604.77 MiB | 727 KiB/s, done.
Resolving deltas: 100% (2389861/2389861), done.
Checking out files: 100% (41521/41521), done.
```

Note: Cloning the kernel can take a long time. You can speed up the clone if you

Hands-on Kernel Lab
Page 29 of 39

www.yoctoproject.org

yocto ·
PROJECT

THE
LINUX
FOUNDATION

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 create a working branch named 'standard/common-pc/base':

```
$ cd ~/poky-dylan-9.0.0/linux-yocto-3.8.git
$ git checkout -b standard/common-pc/base remotes/origin/standard/common-pc/base
```

You should see something like the following as output:

```
Branch standard/common-pc/base set up to track remote branch standard/common-pc/base from origin.
Switched to a new branch 'standard/common-pc/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 ~/poky-dylan-9.0.0/build
$ gedit conf/local.conf
```

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

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

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit 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/poky-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab2-qemux86 \
  "
```

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

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

Edit the linux-yocto kernel recipe:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_3.8.bbappend
```

You'll need to add a new SRC_URI to point to the local clone you just created, but to do that, you need to start with a SRC_URI that makes sense for the linux-yocto-3.8 kernel. In gedit, use 'File | Open' to navigate to and open the base linux-yocto-3.8 recipe: ~/poky-dylan-9.0.0/meta/recipes-kernel/linux/linux-yocto_3.8.bb. Copy the following SRC_URI line from the linux-yocto_3.8 bb file to the linux-yocto_3.8.bbappend in the lab2-qemux86 layer:

```
SRC_URI = "git://git.yoctoproject.org/linux-yocto-
3.8.git;protocol=git;nocheckout=1;branch=${KBRANCH},meta;name=machine,meta"
```
Replace the 'git.yoctoproject.org/linux-yocto-3.8.git' kernel with the path to the local linux-yocto-3.8 kernel that you just cloned (be careful to use the full path to the local repo in the SRC_URI i.e. the triple forward slash in the SRC_URI is not a mistake). Also, change the 'protocol' to 'file' and replace 'nocheckout=1' with 'bareclone=1'. The final SRC_URI should look something like this:
```
SRC_URI = "git:///home/trz/poky-dylan-9.0.0/linux-yocto-
3.8.git;protocol=file;bareclone=1;branch=${KBRANCH},meta;name=machine,meta"
```
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}"
```
Save your changes and close gedit.

Rebuild the Kernel

OK, you have done this before (don't forget to cd back into the build directory):
```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

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, build, and see the results of any kernel changes you make to the local linux-yocto-3.8 repository.

Using a local linux-yocto-based kernel as a bare clone

In this lab so far, you've simply used a straight clone of a git kernel as a working clone. There is another workflow based on a 'bare clone' that for some developers is easier to work with. A large bonus realized from using this method is that in the case of bare clones, the build system is able to automatically pick up source changes without the need for a 'cleanall' step.

The bare clone method does add an extra step to the modification step, but the time savings of not having to do a 'cleanall' and subsequent new fetch of the (albeit local) kernel adds up over time. The following steps can be used to use a bare clone local version of the linux-yocto kernel.

Because in this lab you're building the kernel from a bare local repository, you first need to create a bare local clone of the kernel you want to use and clone that to create a 'working' clone. During development, you'll push your changes from the working clone back into the bare clone. To set this up, cd into the poky-dylan-9.0.0 directory and create a bare local clone of the linux-yocto-3.8 kernel:
```
$ cd ~/poky-dylan-9.0.0
$ git clone --bare git://git.yoctoproject.org/linux-yocto-3.8 linux-yocto-
3.8-bare.git
```
You should see something like the following as output:

```
Cloning into bare repository 'linux-yocto-3.8-bare.git'...
remote: Counting objects: 2851607, done.
remote: Compressing objects: 100% (456267/456267), done.
remote: Total 2851607 (delta 2389855), reused 2829542 (delta 2367803)
Receiving objects: 100% (2851607/2851607), 604.78 MiB | 566 KiB/s, done.
Resolving deltas: 100% (2389855/2389855), done.
```

Now create a local working clone of the local bare clone:

```
$ git clone linux-yocto-3.8-bare.git linux-yocto-3.8-work
```

You should see something like the following as output:

```
Cloning into 'linux-yocto-3.8-work'...
done.
Checking out files: 100% (41521/41521), done.
```

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

```
$ cd linux-yocto-3.8-work/
$ git checkout -b standard/common-pc/base remotes/origin/standard/common-pc/base
```

You should see something like the following as output:

```
Branch standard/common-pc/base set up to track remote branch standard/common-pc/base from origin.
Switched to a new branch 'standard/common-pc/base'
```

Edit the linux-yocto kernel recipe and change the SRC_URI to point to the bare clone:

```
$ gedit ~/poky-dylan-9.0.0/meta-lab2-qemux86/recipes-kernel/linux/linux-yocto_3.8.bbappend
```

The final SRC_URI should look something like this:

```
SRC_URI = "git:///home/trz/poky-dylan-9.0.0/linux-yocto-3.8-bare.git;protocol=file;bareclone=1;branch=${KBRANCH},meta;name=machine,meta"
```

Now rebuild the kernel:

```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/bzImage-lab2-qemux86.bin tmp/deploy/images/core-image-minimal-lab2-qemux86.ext3
```

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. The difference in this case is that you'll be modifying the code in the linux-yocto-3.8-work clone of the bare clone, and once you've committed your change in the work clone:

```
$ cd ~/poky-dylan-9.0.0/linux-yocto-3.8-work
$ gedit fs/filesystems.c
$ git commit -a -m "fs/filesystems.c: Hey, this is the commit I pushed from a working clone into this bare clone!  Neat, eh?"
```

you need to push it to the bare clone:

```
$ git push origin standard/common-pc/base:standard/common-pc/base
```

You should see something like the following as output:

```
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
```

www.yoctoproject.org

yocto · PROJECT

THE LINUX FOUNDATION

```
Writing objects: 100% (4/4), 412 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To /home/trz/poky-dylan-9.0.0/linux-yocto-3.8-bare.git
   dd089cb..2d3d273  standard/common-pc/base -> standard/common-pc/base
```

Verify that the changes you made in the working clone made it to the bare clone:

```
$ cd ~/poky-dylan-9.0.0/linux-yocto-3.8-bare.git
$ git log standard/common-pc/base
```

You should see something like the following as output:

```
commit 2d3d2734768cd8bbbff4211379a0f4eb2d6d49d3
Author: Tom Zanussi <tom.zanussi@linux.intel.com>
Date:   Fri May 10 08:31:50 2013 -0500

    fs/filesystems.c: Hey, this is the commit I pushed from a working clone
into this bare clone!  Neat, eh?

    Signed-off-by: Tom Zanussi <tom.zanussi@linux.intel.com>
```

Now rebuild the kernel (no need to do a 'cleanall' this time) and you should see your change appear:

```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake -c deploy virtual/kernel
$ runqemu tmp/deploy/images/bzImage-lab2-qemux86.bin tmp/deploy/images/core-
image-minimal-lab2-qemux86.ext3
```

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 linux-yocto-based kernel as well as with a bare clone of the linux-yocto kernel. You also added and loaded an external kernel module. This concludes Lab 4.

Lab 5: The Yocto BSP Tool

Up until now, you have modified existing layers by manually editing individual files and directly manipulating the configuration of the kernel. This is a skill you need to have in order to understand and work with existing BSPs. However, there is an easier way to create an initial layer with the customizations you want.

The Yocto BSP Tool consists of a small set of scripts which generate a standardized Yocto BSP layer, including machine configuration, supporting recipes, and `README` files. It also allows the user to add (and remove) patches and kernel config fragments to a linux-yocto kernel without having to edit or learn the sordid details of the linux-yocto meta-data.

Create the Lab 5 Layer

You will use the `yocto-bsp` command to create a new layer for a machine called lab5-qemuarm. Create the layer alongside the other lab layers in `~/poky-dylan-9.0.0`:

```
$ cd ~/poky-dylan-9.0.0/
$ source oe-init-build-env
$ cd ~/poky-dylan-9.0.0/
$ yocto-bsp create lab5-qemuarm qemu
```

Select the following answers for the queries presented when prompted (by entering the given numbers or `(y/n)` values as appropriate, followed by the `Enter` key (if you don't explicitly select anything by just pressing Enter, the default value listed will be chosen for you. Only the non-default values are highlighted red):

```
Which qemu architecture would you like to use? [default: i386]
      1) i386     (32-bit)
      2) x86_64   (64-bit)
      3) ARM      (32-bit)
      4) PowerPC  (32-bit)
      5) MIPS     (32-bit)
3
Would you like to use the default (3.8) kernel? (y/n) [default: y]
Do you need a new machine branch for this BSP (the alternative is to re-use
an existing branch)? [y/n] [default: y]
Getting branches from remote repo git://git.yoctoproject.org/linux-yocto-
3.8.git...
Please choose a machine branch to base your new BSP branch on: [default:
standard/base]
      1) standard/arm-versatile-926ejs
      2) standard/base
      3) standard/beagleboard
      .
      .
1
Would you like SMP support? (y/n) [default: y]
Does your BSP have a touchscreen? (y/n) [default: n]
Does your BSP have a keyboard? (y/n) [default: y]
```

On success, the tool will report:

```
New qemu BSP created in meta-lab5-qemuarm
```

Set up the Environment

Change directories back into the build directory:

```
$ cd build
```

Open local.conf:

```
$ gedit conf/local.conf
```

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

```
MACHINE ?= "lab5-qemuarm"
```

Save your changes and close gedit.

Now open bblayers.conf:

```
$ gedit conf/bblayers.conf
```

and add the 'meta-lab5-qemuarm' 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-lab5-qemux86'):

```
BBLAYERS ?= " \
  /home/myacct/poky-dylan-9.0.0/meta \
  /home/myacct/poky-dylan-9.0.0/meta-yocto \
  /home/myacct/poky-dylan-9.0.0/meta-yocto-bsp \
  /home/myacct/poky-dylan-9.0.0/meta-lab5-qemuarm \
  "
```

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

Build the Image

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

```
$ cd ~/poky-dylan-9.0.0/build
$ bitbake core-image-minimal
$ runqemu tmp/deploy/images/zImage-lab5-qemuarm.bin tmp/deploy/images/core-image-minimal-lab5-qemuarm.ext3
```

Modify the Kernel Recipe

Rather than manually editing the recipe files as in previous labs, you are going to use the Yocto BSP Tool to add the driver patch and update the configuration.

Remember that the base **linux-yocto** recipe takes care of the standard hardware requirements for the qemu machine, but you will still need to tell it about the options to enable the test driver. For the kernel patch, you'll just re-use the yocto-testmod.patch from lab2:

Add the patch using the 'yocto-kernel patch add' command:

```
$ yocto-kernel patch add lab5-qemuarm ~/poky-dylan-9.0.0/meta-lab2-
qemux86/recipes-kernel/linux/files/yocto-testmod.patch
```

You should see the following output from the command:

```
Added patches:
        yocto-testmod.patch
```

You can verify that the patch is included in the lab5-qemuarm kernel by using the 'yocto-kernel patch list' command :

```
$ yocto-kernel patch list lab5-qemuarm
The current set of machine-specific patches for lab5-qemuarm is:
        1) yocto-testmod.patch
```

Similarly, you can add config options using the 'yocto-kernel config add' command:

```
$ yocto-kernel config add lab5-qemuarm CONFIG_YOCTO_TESTMOD=y
```

You should see the following output from the command:

```
Added items:
        CONFIG_YOCTO_TESTMOD=y
```

You can verify that the config item was added to the lab5-qemuarm kernel configuration by using the 'yocto-kernel config list' command :

```
$ yocto-kernel config list lab5-qemuarm
The current set of machine-specific kernel config items for lab5-qemuarm is:
        1) CONFIG_YOCTO_TESTMOD=y
```

Review the Layer

The meta-lab5-qemuarm layer contains the following set of files for the kernel:

```
meta-lab5-qemuarm/recipes-kernel/
  linux /
    files /
      lab5-qemuarm.cfg
      lab5-qemuarm-preempt-rt.scc
      lab5-qemuarm-tiny.scc
      lab5-qemuarm.scc
      lab5-qemuarm-standard.scc
      lab5-qemuarm-user-config.cfg
      lab5-qemuarm-user-patches.scc
      lab5-qemuarm-user-features.scc
      yocto-testmod.patch
    linux-yocto_3.8.bbappend
    linux-yocto_3.8.bbappend.prev
```

At the top level of the `linux/` directory, you can see the `.bbappend` file, which is an extension of the `linux-yocto-3.8` Linux kernel recipe.

The `linux-yocto_3.8.bbappend` is similar to lab2's Linux kernel recipe extension. If you open it up and look at it, you will see that the files in the `files` subdirectory are referenced in the `SRC_URI` variable append (the line that starts with `SRC_URI +=`). Notice that includes the `yocto-testmod.patch` you added using the `yocto-kernel patch add` command.

If you open the `lab5-qemuarm-user-config.cfg` file in the `files` subdirectory, you will see the configuration options you added for the Yocto test module using the `yocto-kernel config add` command. Finally, note the `linux-yocto_3.8.bbappend.prev` file, which is simply a backup copy of the corresponding `.bbappend` file, kept in case something goes wrong with the add (or delete) patch (or config) operation.

Build the Image

Now you can rebuild and boot the new kernel:

```
$ bitbake linux-yocto -c deploy
$ runqemu tmp/deploy/images/zImage-lab5-qemuarm.bin tmp/deploy/images/core-image-minimal-lab5-qemuarm.ext3
```

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.

Lab 5 Conclusion

In this lab you used the Yocto BSP tools to generate a complete BSP layer, including patches and config fragments, without writing any of it yourself. This concludes Lab 5.

Hands-on Kernel Lab
Page 37 of 39

www.yoctoproject.org

yocto
PROJECT

THE
LINUX
FOUNDATION

www.yoctoproject.org

www.yoctoproject.org