Building Images with Yocto Project- Lab

Tim Orling - Yocto Project, Intel Open Source Technology Center

March 9, 2018

These lab instructions are written for the *Building Images with Yocto Project* tutorial of the *Embedded Apprentice Linux Engineer* track. They are designed to work for the *PocketBeagle* hardware platform.

Initial build environment configuration

NOTE: This lab and the `yocto-intro` lab were written independently at first, so this section should be review from that session and you can skip ahead to the next section.

To get started with the Yocto Project, one of the easiest things to do is to clone the *poky* repository which will enable us to build the *poky* reference distribution.

The metadata layers at `https://github.com/e-ale/yocto-e-ale` provide a layers to work with *poky* to build a complete Linux distribution for the *PocketBeagle* platform. Let us go ahead and clone those now:

```
cd ..
git clone https://github.com/yocto-e-ale
```

However, for educational purposes, we are going to start straight from the official vanilla *poky*, and build our configuration from scratch.

Getting poky

Start by cloning poky from the official Yocto Project Git repository:

```
$ git clone https://git.yoctoproject.org/git/poky
$ cd poky/
```

(Note: if download speed is too slow, you can use the `poky.tar.xz` tarball provided by the instructor)

We'll base our work on "master", which is the latest development branch. This branch will be released soon as the *2.5* or *sumo* release of the Yocto Project. Once it is released, an official *yocto-2.5* tag will become available and a *sumo* branch will be created which will provide on-going support for future dot releases (e.g. 2.5.1). As of this writing the project is in milestone three, or *M3* of the 2.5 release.

Setting up our local configuration

We can now setup a build environment and begin by building an image for *qemuarm*:

```
$ . oe-init-build-env ../build-qemuarm
```

By default, the above command will build images for the *qemux86* machine, so we will need to alter the default configuration to enable the *qemuarm* machine, by uncommenting (removing the leading # on the following line in `conf/local.conf`:

```
#MACHINE ?= "qemuarm"
```

To save build and download time, we will use *shared state* and *downloads* already prepared for you by your instructor. Shared state, also known as *sstate* or *sstate-cache*, is a specially organized directory or cache of all the *tasks* that have been run to build a particular package or image. For teams, like ours, that are rebuilding the same set of packages, this is a way to dramatically speed up build time. The downloads directory can also be shared to save *fetch* time, or download time from the internet. Change the following lines in `conf/local.conf`:

```
#DL_DIR ?= "${TOPDIR}/downloads"
#SSTATE_DIR ?= "${TOPDIR}/sstate-cache"
```

to

```
DL_DIR ?= "${HOME}/DOWNLOADS"
SSTATE_DIR ?= "${HOME}/SSTATE"
```

We also might want to build *debian* packages, like the *BeagleBoard.org* stock images. To do so we need to change the following line in `conf/local.conf`:

```
PACKAGE_CLASSES ?= "package_rpm"
```

to

```
PACKAGE_CLASSES ?= "package_deb"
```

Also, to help speed up the build and save disk space, we can prevent temporary work being saved to disk by appending the following to `conf/local.conf`:

```
INHERIT += "rm_work"
```

Building a minimal image

To build a minimal image, with a very basic text console environment, we run the following command:

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

This will perform all the tasks necessary to fetch the source code, build the native host tools, compile and package the individual pieces of the image, and generate the root file system and the final image.

Running our image in emulation

We can test the result of our efforts by running the image in the *qemu* emulated environment, without graphics and using non-privileged network devices:

```
runqemu nographic slirp
```

Because the default setting is `EXTRA_IMAGE_FEATURES ?= "debug-tweaks"`, we can simply login as the *root* user with no password. This is only for development purposes and you should never ship a product with such an insecure setting.

Notice that the text before the login prompt represents the "branding" of the *poky* distribution:

```
Poky (Yocto Project Reference Distro) 2.4+snapshot qemuarm /dev/ttyAMA0

qemuarm login:
```

To change this default behavior, we will want to create our own custom Linux distribution, by creating a distro layer. We also want to add specific support for the *PocketBeagle*, so we will need to create a bsp layer. We will want to add some of our *recipes*, so we should create an application layer where we can put our work.

Create application layer

NOTE: This lab was originally written with the intent to create the distro and bsp layers first, but it makes more sense to follow up with your experience in the `yocto-intro` course. We will instead jump straight to the creation of the application layer. We will be using the distro and bsp layers, but we will go over those layers after this section (or leave those sections as follow up to do on your own).

Create application layer

NOTE: if time is short or you want to skip ahead, the content of this section can be cloned by running:

```
$ cd ../yocto-e-ale
$ git checkout 03_create_apps_layer
```

If you are running the lab "out of order", meaning starting with the application layer first, you should start with:

```
$ cd ../yocto-e-ale
$ git checkout 02_create_bsp_layer
```

As this will prepare you to run the instructions in this section.

First, we will use the `bitbake-layers` tool to create a simple, but valid, layer skeleton that we can build upon:

```
$ bitbake-layers create-layer ../yocto-e-ale/meta-e-ale-apps
```

Add `hello-recipe` to our image

In the earlier `yocto-intro` session, you created a "hello world" application. We will now add that to our new `meta-e-ale-apps layer`.

```
$ mv ${HOME}/e-ale/hello-recipe meta-e-ale-apps/recipes-example
```

We want our image to always build with this recipe, so we are going to create an image append recipe and add to it. Note that image recipes are different in that they do not have a version (`PV`).

```
$ mkdir -p recipes-image/images
$ vim recipes-image/images/core-image-minimal.bbappend
```

There are several ways we can add the "hello" recipe to the image, but the author's favorite is the `CORE_IMAGE_EXTRA_INSTALL` variable. Use the following as the contents for the bbappend file:

```
CORE_IMAGE_EXTRA_INSTALL += "hello"
```

We can now re-build `core-image-minimal` and our "hello" program will be included:

```
$ bitbake core-image-minimal
$ runqemu nographic slirp
...
root@qemuarm:~# hello
```

Using `devtool` in our workflow

One of the most important changes the Yocto Project has brought since the the Open Embedded Classic days is improved developer workflow tools. The poster-child for developer workflow tools is *devtool*.

Just to show the process, we are going to add another "hello" program, this time from GNU [1]. To keep this recipe distinct from our previous "hello" recipe, we will name this package "gnu-hello". Since both install to `/usr/bin/hello` they will not coexist, so this is mostly just an excercise to show you how to use `devtool add` with a known, simple package.

```
$ devtool add gnu-hello https://ftp.gnu.org/gnu/hello/\
hello-2.10.tar.gz
```

What did `devtool` create? All the work is in a temporary "sandbox" layer in your build directory. This layer is called "workspace" by default.

```
workspace
├─ appends
│  └─ gnu-hello_2.10.bbappend
├─ conf
│  └─ layer.conf
├─ README
├─ recipes
│  └─ gnu-hello
│     └─ gnu-hello_2.10.bb
└─ sources
   └─ gnu-hello
      └─ ...
```

We can build our "sandboxed" recipe with the following command:

```
$ devtool build gnu-hello
```

Note: there is an issue with this source that causes it to fail to build the first time. This is a known issue, but we have not fixed it yet. Simply run the above command again and it will build successfully the second time.

We can add this recipe to our image if we like, but note that the binary will probably overwrite

[1]https://www.gnu.org/software/hello/

the `hello-recipe` we added earlier:

```
$ sed -i -e 's:"hello":"gnu-hello hello":g' recipes-image/images/\
core-image-minimal.bbappend
```

Now, if we are satisfied with our recipe, we can add it to our layer with the `devtool finish` command. This command takes a recipe name and a destination (path to layer) as arguments. It will complain that our source is not clean, so we will also use the `-f` or "force" option:

```
$ devtool finish -f gnu-hello ../yocto-e-ale/meta-e-ale-apps/recipes-example/
```

A `NOTE` statement will let you know that the `gnu-hello` source directory has not been deleted automatically, but that you can delete it now. This is because `devtool` never wants to assume it knows what is safe to erase or not. Note, however, that if you were to try to work with the same recipe again with `devtool`, that it will refuse to do anything if the source directory of the same name already exists. It is good practice to go ahead and remove the source directory after you are finished:

```
$ rm -rf workspace/sources/gnu-hello
```

Upgrading recipes with `devtool`

Another common task in embedded development is upgrading an existing application to the latest upstream version. We are going to use the `nano` text editor as an example.

We are going to cheat and grab the existing 2.7.4 recipe from the `meta-openembedded/meta-oe` layer. We will use `devtool` to upgrade the `nano` editor to the latest version [2].

```
$ mkdir recipes-support/nano
$ pushd recipes-support/nano
$ wget http://git.openembedded.org/meta-openembedded/plain/ \
meta-oe/recipes-support/nano/nano_2.7.4.bb
$ wget http://git.openembedded.org/meta-openembedded/plain/ \
meta-oe/recipes-support/nano/nano.inc
$ popd
```

Let's go ahead and build this recipe:

```
$ bitbake nano
```

We can add it to our image (this syntax is assuming you did not add "gnu-hello" to your image):

```
$ sed -i -e 's:"hello":"hello nano":g' recipes-image/images/ \
core-image-minimal.bbappend
```

We can rebuild our image and then run the "nano" program.

```
$ bitbake core-image-minimal
$ runqemu nographic slirp
```

[2]https://www.nano-editor.org

```
...
root@qemuarm:~# nano
```

Now upgrade the version to the latest release (2.9.4 as of this writing):

```
$ devtool upgrade nano
```

Notice that we did not provide a version to which to upgrade. Because `devtool` knows about package data (both parsed when `bitbake` first starts and built package data, which helps it to better determine dependencies).

Behind the scenes, `devtool` is using a default value for `UPSTREAM_CHECK_REGEX`, a pattern to use to determine the latest version from the upstream source (the `SRC_URI`). The combination of this knowledge and the current version (`PV`) is all that is needed to know whether or not we need to upgrade at all and what version to which to upgrade. Note that some packages need help with the `UPSTREAM_CHECK_REGEX`, so the default values won't always work.

What did `devtool` create?

```
workspace
│
├─ appends
│  └─ nano_2.9.4.bbappend
│
├─ conf
│  └─ layer.conf
│
├─ README
│
├─ recipes
│  └─ nano
│     ├─ nano_2.9.4.bb
│     │
│     └─ nano.inc
│
└─ sources
   └─ nano
      └─ ...
```

Finally, let us finish with our work on `nano` and clean up (again forcing the finish even though our source tree is "not clean"):

```
$ devtool finish -f nano ../yocto-e-ale/meta-e-ale-apps/recipes-support/
$ rm -rf workspace/sources/nano
```

Create a distro layer

NOTE: if time is short or you want to skip ahead, the content of this section can be cloned by running:

```
$ cd ../yocto-e-ale
$ git checkout 01_create_distro_layer
```

Bitbake comes with a tool to create the directory structure of a generic metadata layer. We will use this tool now to create our distro layer skeleton.

```
bitbake-layers create-layer ../yocto-e-ale/meta-e-ale-distro
```

Let's look at what was created:

```
meta-e-ale-distro
├─ conf
│  └─ layer.conf
├─ COPYING.MIT
├─ README
└─ recipes-example
   └─ example
      └─ example.bb
```

This is a generic metadata layer, more like an application or functional layer, and does not have the pieces that make it a distro layer. We will now delete what we do not need and add what is missing.

```
$ cd meta-e-ale
$ rm -rf recipes-example
$ mkdir -p conf/distro/include
$ vim conf/distro/e-ale.conf
```

Add the following content:

```
DISTRO = "e-ale"
DISTRO_NAME = "e-ale Linux"

DISTRO_VERSION = "1.0+snapshot-${DATE}"

E_ALE_DEFAULT_DISTRO_FEATURES = "systemd"

LAYER_CONF_VERSION ?= "1"

# Add ssh server so we can connect to system running this image remotely,
# add development tools (gcc, make, etc), and -dev packages for installed #
E_ALE_DEFAULT_DISTRO_FEATURES += "ssh-server-openssh tools-sdk dev-pkgs"

DISTRO_FEATURES ?= "${DISTRO_FEATURES_DEFAULT} ${DISTRO_FEATURES_LIBC} ${E_

VIRTUAL-RUNTIME_init_manager = "systemd"

INHERIT += "uninative"

UNINATIVE_URL = "http://downloads.yoctoproject.org/releases/uninative/1.7/"
UNINATIVE_CHECKSUM[i686] ?= "de51bc9162b07694d3462352ab25f636a6b50235438c1b
UNINATIVE_CHECKSUM[x86_64] ?= "ed033c868b87852b07957a4400f3b744c00aef5d6470
```

In order to add our branding before the login prompt, we need to enable our local changes to override the default in the `base-files` recipe:

```
$ mkdir -p recipes-core/base-files
$ pushd recipes-core/base-files
$ echo '# look for files in this layer first' > base-files_%.bbappend
$ echo 'FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"' >> base-files_%.bbapp
```

This tells *Bitbake* to look at our `meta-e-ale-distro` layer before any other layer of lower priority (allows us to override default behavior). The := operator says to immediately expand the variables `${THISDIR}` and `${PN}`. The : at the end is important and easy to miss, with that your paths will be mangled in a non-functional way.

Next, we want to add our branding to the file which will be installed in our root file system at `/etc/issue`:

```
$ mkdir recipes-core/base-files/base-files
$ vim recipes-core/base-files/base-files/issue
```

We want our result to look like the following:

```
                    _        _     _
                   | |      | |   (_)
    ___        __ _| | ___  | |    _ _ __  _   ___  __
   / _ \______/ _` | |/ _ \ | |   | | ’_ \| | | \ \/ /
  |  __/_____| (_| | |  __/ | |___| | | | | |_| |>  <
   \___|      \__,_|_|\___| \_____/_|_| |_|\__,_/_/\_\

  Embedded Apprentice Linux Engineer http://e-ale.org
```

Unfortunately, we need to escape all the backslash characters, so the actual content is less readable in the file itself:

```
                    _        _     _
                   | |      | |   (_)
    ___        __ _| | ___  | |    _ _ __  _   ___  __
   / _ \\______/ _` | |/ _ \\ | |   | | ’_ \\| | | \\ \\/ /
  |  __/_____| (_| | |  __/ | |___| | | | | |_| |>  <
   \\___|      \\__,_|_|\\___| \\_____/_|_| |_|\\__,_/_/\\_\\

  Embedded Apprentice Linux Engineer http://e-ale.org
```

We also want branding for network access (like ssh):

```
$ pushd recipes-core/base-files/base-files
$ cp issue issue.net
$ popd
```

Configuration Templates

We would like to make it easy to start working with our distro layer, so we will provide templates for `local.conf` and `bblayers.conf`. These are created in the `meta-e-ale-disro/conf` directory as `local.conf.sample` and `bblayers.conf.sample`.

For `local.conf.sample`, we can simply copy our `local.conf` from our prior session and just change the distro from "poky" to "e-ale":

```
$ cp ../build-qemuarm/conf/local.conf \
   meta-e-ale-distro/conf/local.conf.sample
$ sed -i -e ’s:DISTRO = "poky":DISTRO = "e-ale":g’ \
   meta-e-ale-distro/conf/local.conf.sample
```

For `bblayers.conf.sample`, the template has content that is automatically replaced, so the syntax is a bit trickier and we use the following content:

```
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "1"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  ##OEROOT##/meta \
  ##OEROOT##/../yocto-e-ale/meta-e-ale-distro \
  "
```

The special tokens `##OEROOT##` are substituted (with the full path to openembedded-core) automatically when the template is used to create the `bblayers.conf` file. For now, we only include the distro layer we are currently working on. We will add more layers in the next sections. Be careful with the paths that you use, as you need to remember that this file will be sourced in the context of the environment variable `${OEROOT}`, which in our usage will be set to be the "poky" directory by our `init-build-env` script. We will populate that script now.

Rather than using the default *openembedded-core* init script contained in *poky*, we will use our own that will setup the build environment for our distro (and in the next section, board support for the *PocketBeagle*).

Create the build environment initialization script `init-build-env` in the `yocto-e-ale` directory with the following content:

```
!/bin/sh

# OE Build Environment Setup Script
#
# Copyright (C) 2006-2011 Linux Foundation
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

#
# Normally this is called as '. ./oe-init-build-env <builddir>'
#
# This works in most shells (not dash), but not all of them \
pass the arguments
# when being sourced.  To workaround the shell limitation \
use "set <builddir>"
# prior to sourcing this script.
#
# LAYER_CONF_VERSION is increased each time \
build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "7"
    THIS_SCRIPT_DIR=$(dirname "$THIS_SCRIPT")
    THIS_SCRIPT_DIR=$(readlink -f "$THIS_SCRIPT_DIR")

    if [ -d "$THIS_SCRIPT_DIR/meta-e-ale-distro/conf" ]; then
        TEMPLATECONF="$THIS_SCRIPT_DIR/meta-e-ale-distro/conf"
    elif [ -f "$THIS_SCRIPT_DIR/.templateconf" ]; then
        source $THIS_SCRIPT_DIR/.templateconf
    elif [ -d "$THIS_SCRIPT_DIR/.template" ]; then
        TEMPLATECONF="$THIS_SCRIPT_DIR/.template"
    fi
fi
```

continued:

```
if [ -z "$OEROOT" ]; then
    OEROOT=$(dirname "$THIS_SCRIPT")/../poky
    OEROOT=$(readlink -f "$OEROOT")
fi

export OEROOT

if [ -z "$BITBAKEDIR" ]; then
    BITBAKEDIR=$(dirname "$THIS_SCRIPT")/../poky/bitbake
    BITBAKEDIR=$(readlink -f "$BITBAKEDIR")
fi

export BITBAKEDIR

unset THIS_SCRIPT_DIR
unset THIS_SCRIPT

. $OEROOT/scripts/oe-buildenv-internal &&
    TEMPLATECONF="$TEMPLATECONF" \
    $OEROOT/scripts/oe-setup-builddir || {
    unset OEROOT
    return 1
}
unset OEROOT

[ -z "$BUILDDIR" ] || cd "$BUILDDIR"

# Shutdown any bitbake server if the BBSERVER variable is not set
if [ -z "$BBSERVER" ] && [ -f bitbake.lock ]; then
    grep ":" bitbake.lock > /dev/null && BBSERVER=$(cat \
    bitbake.lock) bitbake --status-only
    if [ $? = 0 ]; then
        echo "Shutting down bitbake memory resident server \
        with bitbake -m"
        BBSERVER=$(cat bitbake.lock) bitbake -m
    fi
fi
```

We can now create a new build environment with our new distro layer:

```
$ . ./init-build-env ../build-e-ale-1
$ bitbake core-image-minimal
$ runqemu nographic slirp
```

The text at the login will now be what we want for our branding:

```
                  _        _     _
                 | |      | |   (_)
  ___        __ _| | ___  | |    _ _ __  _   ___  __
 / _ \______/ _` | |/ _ \ | |   | | '_ \| | | \ \/ /
|  __/______| (_| | |  __/ | |___| | | | | |_| |>  <
 \___|      \__,_|_|\___| |_____|_|_| |_|\__,_/_/\_\

Embedded Apprentice Linux Engineer http://e-ale.org
e-ale Linux 1.0+snapshot qemuarm ttyAMA0

qemuarm login:
```

Just to quickly review, our directory structure should now look like the following:

```
yocto-e-ale
├─ init-build-env
└─ meta-e-ale-distro
   ├─ conf
   │  ├─ bblayers.conf.sample
   │  ├─ distro
   │  │  └─ e-ale.conf
   │  ├─ layer.conf
   │  └─ local.conf.sample
   ├─ COPYING.MIT
   ├─ README
   └─ recipes-core
      └─ base-files
         ├─ base-files
         │  ├─ issue
         │  └─ issue.net
         └─ base-files_%.bbappend
```

Create a bsp layer

NOTE: if time is short or you want to skip ahead, the content of this section can be cloned by running:

```
$ cd ../yocto-e-ale
$ git checkout 02_create_bsp_layer
```

Similar to what we did in the prior section, we will use the `bitbake-layers` tool to create our bsp layer skeleton.

```
bitbake-layers create-layer ../yocto-e-ale/meta-e-ale-bsp
```

This is a generic metadata layer, more like an application or functional layer, and does not have the pieces that make it a bsp layer. We will now delete what we do not need and add what is missing.

```
$ pushd meta-e-ale-bsp
$ rm -rf recipes-example
$ mkdir -p conf/machine/include
$ vim conf/machine/pocketbeagle.conf
```

We can borrow some hints from Koen Kooi's fork of the `meta-beagleboard` layer [3]. He recently updated his fork for a *PocketBeagle* demo for *FOSDEM 2018* [4].

Use the following content for `pocketbeagle.conf`:

```
include conf/machine/include/ti33x.inc

PREFERRED_PROVIDER_virtual/kernel = "linux-pocketbeagle"

SPL_BINARY = "MLO"
PREFERRED_PROVIDER_virtual/bootloader = "u-boot"
PREFERRED_PROVIDER_u-boot = "u-boot"
```

For simplicity, we are going to copy some dependent files from the `meta-ti` layer, although we could also clone and depend on that vendor layer. We plan to go into more detail of utilizing the `meta-ti` layer in a future advanced training session.

```
$ pushd conf/machine/include
$ wget http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/plain/ \
conf/machine/include/ti33x.inc
$ wget http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/plain/ \
conf/machine/include/ti-soc.inc
$ popd
```

[3] https://github.com/koenkooi/meta-beagleboard/blob/master/common-bsp/conf/machine/pocketbeagle.conf
[4] https://plus.google.com/+KoenKooi/posts/dVnZfMgZ9Ma

In our `pocketbeagle.conf`, we set the `PREFERRED_PROVIDER` for the `virtual/kernel` to be "linux-pocketbeagle". That configuration does not exist, so we will create it. First we need to create the directory structure for this recipe.

```
$ mkdir -p recipes-kernel/linux/linux-pocketbeagle
$ vim recipes-kernel/linux/linux-pocketbeagle_4.14.bb
```

We will base our recipe on `meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb`. Use the following content:

```
# linux-pocketbeagle.bb:
#
#   Based on meta-skeleteon/recipes-kernel/linux/linux-yocto-custom.bb
#
#   An example kernel recipe that uses the linux-yocto and oe-core
#   kernel classes to apply a subset of yocto kernel management to git
#   managed kernel repositories.
#
#   To use linux-yocto-custom in your layer, copy this recipe (optionally
#   rename it as well) and modify it appropriately for your machine. i.e.:
#
#     COMPATIBLE_MACHINE_yourmachine = "yourmachine"
#
#   You must also provide a Linux kernel configuration. The most direct
#   method is to copy your .config to files/defconfig in your layer,
#   in the same directory as the copy (and rename) of this recipe and
#   add file://defconfig to your SRC_URI.
#
#   To use the yocto kernel tooling to generate a BSP configuration
#   using modular configuration fragments, see the yocto-bsp and
#   yocto-kernel tools documentation.
#
# Warning:
#
#   Building this example without providing a defconfig or BSP
#   configuration will result in build or boot errors. This is not a
#   bug.
#
#
```

continued:

```
# Notes:
#
#   patches: patches can be merged into to the source git tree itself,
#            added via the SRC_URI, or controlled via a BSP
#            configuration.
#
#   defconfig: When a defconfig is provided, the linux-yocto configuration
#              uses the filename as a trigger to use a 'allnoconfig' baseli
#              before merging the defconfig into the build.
#
#              If the defconfig file was created with make_savedefconfig,
#              not all options are specified, and should be restored with t
#              defaults, not set to 'n'. To properly expand a defconfig lik
#              this, specify: KCONFIG_MODE="--alldefconfig" in the kernel
#              recipe.
#
#   example configuration addition:
#            SRC_URI += "file://smp.cfg"
#   example patch addition (for kernel v4.x only):
#            SRC_URI += "file://0001-linux-version-tweak.patch"
#   example feature addition (for kernel v4.x only):
#            SRC_URI += "file://feature.scc"
#

inherit kernel
require recipes-kernel/linux/linux-yocto.inc

# Override SRC_URI in a copy of this recipe to point at a different source
# tree if you do not want to build from Linus' tree.
SRC_URI = "\
    git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git;p
    file://defconfig \
    file://0001-Stripped-back-pocketbeagle-devicetree.patch \
"

LINUX_VERSION ?= "4.14.18"
LINUX_VERSION_EXTENSION_append = "-pocketbeagle"

# Modify SRCREV to a different commit hash in a copy of this recipe to
# build a different release of the Linux kernel.
# tag: v4.14.18 81d0cc85caabe062991ea45ddada814835d47fb0
SRCREV_stable="81d0cc85caabe062991ea45ddada814835d47fb0"

PV = "${LINUX_VERSION}+git${SRCPV}"
```

continued:

```
# Override COMPATIBLE_MACHINE to include your machine in \
a copy of this recipe file.
COMPATIBLE_MACHINE = "pocketbeagle"
```

We will re-use the `defconfig` and patch from the `buildroot-e-ale` Git repository:

```
$ pushd recipes-kernel/linux/linux-pocketbeagle
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
master/board/pocketbeagle/linux.config -O defconfig
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
master/board/pocketbeagle/patches/linux/ \
0001-Stripped-back-pocketbeagle-devicetree.patch
$ popd
```

Now, we need to change our distro layer configuration from the prior section to reflect hardware that is supported, namely the *PocketBeagle*. Change the `conf/local.conf.sample` "Machine Selection" section to be the following:

```
#
# Machine Selection
#
# You need to select a specific machine to target the build with.
# This sets the default machine to be pocketbeagle if no other
# machine is selected:
MACHINE ??= "pocketbeagle"
```

This is a major change to the layer, so we should bump the layer configuration version.

We also know that we need changes to the U-Boot bootloader. Let us create the directories to hold the u-boot recipes and patches.

```
$ mkdir -p recipes-bsp/u-boot/u-boot
```

Once again we will re-use the files from the `builroot-e-ale` repo:

```
$ pushd recipes-bsp/u-boot/u-boot
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
master/board/pocketbeagle/patches/u-boot/ \
0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
master/board/pocketbeagle/patches/u-boot/ \
0002-U-Boot-BeagleBone-Cape-Manager.patch
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
master/board/pocketbeagle/patches/u-boot/ \
0003-pocketbeagle-tweaks.patch
$ wget https://raw.githubusercontent.com/e-ale/buildroot-e-ale/ \
```

```
master/board/pocketbeagle/uEnv.txt
$ popd
```

Now we will append the stock `u-boot` recipe:

```
$ vim recipes-bsp/u-boot/u-boot_2018.01.bb
```

Use the following content:

```
# look for files in this layer first
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

SRC_URI += "\
    file://0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch \
    file://0002-U-Boot-BeagleBone-Cape-Manager.patch \
    file://0003-pocketbeagle-tweaks.patch \
    file://uEnv.txt \
"
```

Summary

We have learned a lot more details about the different kinds of layers and how to create our own custom layers. Many developers new to the Yocto Project and Open Embedded fail to create their own layers and instead just fork `poky` and do all their work "in-tree". This is a major mistake and creates future technical debt to upgrade to a newer Yocto Project release. It also clutters your git repository and completely ignores the intent of layers: flexibility and modularity. Using the skills you learned in this lab to build your own layers will really pay off in your future embedded Linux development efforts.