Building a Custom Embedded Linux Distribution with the Yocto Project

Sean Hudson
Embedded Linux Architect
Embedded Software Division
Mentor Graphics, Inc

mentor.com/embedded
The challenge for this presentation
1. Building an embedded Linux distribution from source is a complex task
2. The Yocto Project is a collection of powerful tools with lots of complexity of their own

A word of thanks
This presentation is built on the work of several others including: Chris Hallinan, Chris Larson, Robert P.J. Day, Khem Raj and everyone that’s contributed to the documents in OE and YP.

Setting expectations
There is too much material to cover in a 50 minute presentation. So, in this presentation, I am going to make a quick survey of the basics and highlight the “big pieces”. Next, I will highlight some useful tools and lastly, I will provide a set of quick “how to” steps for some common tasks.

Important to highlight the excellent documentation that can be found on the project site: https://www.yoctoproject.org/documentation/current
It’s not an embedded Linux distribution – it creates a custom one for you

The Yocto Project is an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture.

Put another way:
The Yocto Project tries to provide the basic pieces for building an embedded Linux distribution. These pieces add little value for companies, but are necessary to build an end product. In short, the project should allow developers to focus on the features that matter to *their* customers.

Other points to make:
Customers can be internal and external.
Using an existing distro, e.g. Debian or Fedora is certainly a valid choice, but limits flexibility. There are reference distributions, e.g. Angstrom & Poky, however, Angstrom provides a binary feed and is primarily geared at enabling hobbyist boards. Poky is mainly geared towards testing the YP build system and is therefore not as stable.

Regardless of the choice, building a distro from source enhances:

Security
- Since source code is visible, can tell what code is doing and prevent malicious code
- Also enables ability to patch source quickly when exploits are discovered (see timeliness)

Licensing
- Provides discreet control over the source going into the image, so licensing can be known/controlled

Tweaking/Image size
- Provides complete control over the image and therefore allows for optimizations and tweaking that may not otherwise be possible
Timeliness

– Reduced dependency on outside packagers, e.g. Ubuntu, to fix a problem. Instead, can incorporate new features and fixes on personal schedule
Let’s get a (quick!) start

- The Yocto Project has some great documentation available on the [project website](http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html)

- In particular, the [quickstart](http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html) gives us a great way to take a look at an example build structure

Following the recipe of the quick start is a fun way to get started and will give us a good initial tree to break down further.
Highlight each of the directories and what they do.
Bitbake – is the build engine
Documentation – duh!
Meta-* - metadata – the meat! - critical information that drives the whole process
Scripts – “glue” pieces that makes things work, including wrappers, e.g. runqemu

Point out the layers, but save a deeper discussion for later.
Point out the script and explain what it does, leading to the next slide.
Initial script run:
Point out the bitbake targets, briefly describing each and point out the runqemu command
What did the script do?

Created the initial build tree and seeded it with a starting local.conf
Show the initial build tree, including the local.conf.
Highlight the bblayers.conf and defer it again.
Pretty boring, as yet.
Sets the path and a few ENV variables. Non-destructive to re-run, in fact, needed to set up the environment. Observe that the last bit is the same and explain the env variables quickly, e.g. BB_EXTRA_WHITE
No change

Show the initial build tree, including the local.conf. Highlight the bblayers.conf and defer it again. Pretty boring, as yet.
Let's run a build
Seriously, this may take a while depending on your machine and network speed. Typical times is 1-2 hours and don’t forget to have enough disk space!
So, what the heck is all this stuff?

1. downloads – source downloads (mention sharing the cache)
2. sstate-cache – package caching
3. tmp – the build output
So, what the heck is all this stuff?

Important bits
1. Buildstats – useful build information
2. deploy – this is where final images go
3. pkgdata – information used to describe the packages
4. work – this is where the “action” is. Source archive, logs, and scripts for each build go here
So, what’s in the work dir?
How far down do I need to go?!
Dumping a bitbake environment

- Dump the environment that bitbake uses for a target
  - `bitbake -e <target>`
- Especially useful for finding some things quickly, e.g.
- Where is the source for a recipe?
  - `$ Bitbake -e <target> | grep ^S=`
- Where is the working directory for a recipe?
  - `$ bitbake -e <target> | grep ^WORKDIR=`
- What image types are being built?
  - `$ bitbake -e <image-target> | grep ^IMAGE_FSTYPES=`

Note: bitbake –g also provides useful information by dumping the dependency graphs. Be careful, lots of information in this.
Bitbake layers provide a way to collect related recipes together. This modularity, when used correctly, enables better reuse and easier maintenance.

**Points to make:**
1. Types of layers vary, call out bsp layers (refer to docs)
2. Other examples, networking, ivi, gui, etc
How to explore layers efficiently

- It quickly becomes non-trivial to look at layers
- Luckily, there’s a tool to help out
- bitbake-layers
  - flatten
  - help
  - show-appends
  - show-cross-depends
  - show-layers
  - show-overlayed
  - show-recipes

$ bitbake-layers
usage: bitbake-layers <command> [arguments]

Available commands:
flatten
  flattens layer configuration into a separate output directory.
help
  display general help or help on a specified command
show-appends
  list bbappend files and recipe files they apply to
show-cross-depends
  figure out the dependency between recipes that crosses a layer boundary.
show-layers
  show current configured layers
show-overlayed
  list overlayed recipes (where the same recipe exists in another layer)
show-recipes
  list available recipes, showing the layer they are provided by
help
  display general help or help on a specified command
Analyze an example recipe.

Recipes contain the instructions on how to take a set of source files and generate one or more output packages.

Name of the recipe is automatically inherited by many variable in total, or in part.

NOTE: Name-spacing is used a lot to separate items

Important pieces include:
SRC_URI
License fields
Checksum fields
...
Wait, so what are packages then?

- Packages
  - Output built from the instructions in a recipe
  - Names are not (necessarily) the same as the recipe name, but they are usually related
  - Multiple packages can come from a single recipe
  - The specific files included in each package is controlled by the FILES_* variables

- NOTE: Images include packages, not recipes!

Refer to Chris Hallinan’s blog post:
http://blogs.mentor.com/chrishallinan/blog/2012/04/27/more-on-yocto-terminology-recipes-and-packages/

Default four packages, basic binary, -dev, -dbg, & -doc. Determined by the FILES_* variables

Example of multiple packages for a single recipe from Chris’ post:

```
$ bitbake -e python | grep ^PACKAGES=
```
When BitBake completes “baking” the python recipe, a package is created for each of the named elements show above in the ‘PACKAGES=’ listing.
So, what are .bbappend files?

- Used to add customizations without completely over-riding a recipe.
- Used with layers, this allows for a customization to track against an “upstream” recipe without a lot of overhead.
- At the core of how to use layers well.
- Note: As the name implies, these are generally additive.
  - Subtractive operations are difficult technically, so overriding the recipe is the best recourse
Chris Larson recently created a new tool, called ‘bb’.

Very useful for inspection of dependencies

$ bb
usage: bb [-h]

{whatdepends,help,showdepends,dependinfo,show,whatprovides,whatrprovides,showprovides}
...}

Prototype subcommand-based bitbake UI

positional arguments:

{whatdepends,help,showdepends,dependinfo,show,whatprovides,whatrprovides,showprovides}
  dependinfo
  help        Show overall help, or help for a specific subcommand
  show        Show bitbake metadata (global or recipe)
  showdepends Show what the specified target depends upon
  showprovides Show what the specified target provides
whatdepends    Show what depends on the specified target
whatprovides   Show what recipes provide the specified target
whatprovides   Show what recipes provide the specified target

optional arguments:
-h, --help      show this help message and exit
Putting it all together
Tracking down busybox

```
shudson@ronin:/data/projects/poky-danny-8.0/builds bitbake-layers show-recipes busy
WARNING: Host distribution "ubuntu 12.04.2 LTS" has not been validated with this version of the build system; you may possibly experience unexpected failures. It is recommended that you use a tested distribution.
Parsing recipes...done.
=== Available recipes matching busy: ===
busybox:
  1.28.2
shudson@ronin:/data/projects/poky-danny-8.0/builds bitbake -e busybox |grep "^1.28.2"
shudson@ronin:/data/projects/poky-danny-8.0/builds bitbake -e busybox |grep "^1.28.2"
shudson@ronin:/data/projects/poky-danny-8.0/builds bitbake -e busybox |grep "^1.28.2"
```

How do I add my application to an image?

- First, develop your application
  - Different workflows are possible
  - Chicken and egg problem?
- Create a recipe
  - Build on examples that are already in the layers
  - Ask for help, if you get stuck!
    - Mailing list or irc are great places for this
    - Have a look at the recipe skeleton script
- Add the recipe to a layer
- Add the desired package to the image
  - Inherit from a image
  - Add to the IMAGE_INSTALL

In almost all case, there are multiple ways to do things. Check out the docs for more alternatives.
How do I add packages to my recipe?

- Assuming you have a working recipe
- Add new packages using the PACKAGES variable
  - `PACKAGES += "foo bar"
- Add files to each package
  - `FILES_foo = "${bindir}/foo_files"
  - `FILES_bar = "${incdir}/bar_files"
- Don’t forget to bump your PR value!
My final thoughts are mostly common sense.

1. Building a distribution from scratch is a daunting task, the YP gives you a great running start
2. Use an existing BSP as your starting point
3. Get comfortable with the process and understand the different roles/workflows that your organization will need to support
4. Play around with it and explore
5. The tools aren’t perfect and there are significant gaps, but that’s where we can use help!
Q&A