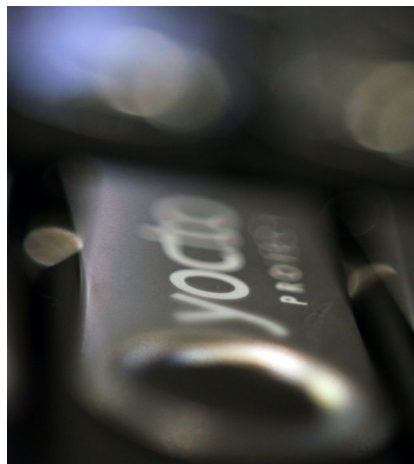


# Yocto Project Developer Day

## Intro to Yocto Project



# Creating a Custom Embedded Linux Distribution for Any Embedded Device Using the Yocto Project



Behan Webster  
Jan-Simon Möller  
The Linux Foundation  
Feb 24, 2017

# Yocto Project Dev Day Wifi information

If you want to connect to the Internet:

SSID: lfevents

Password: linux1991



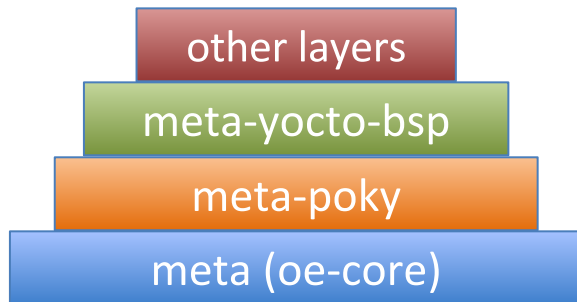
The URL for this presentation

<http://bit.ly/2mrr2jh>



# Yocto Project Overview

- **Collection of tools and methods enabling**
  - ◆ Rapid evaluation of embedded Linux on many popular off-the-shelf boards
  - ◆ Easy customization of distribution characteristics
- **Supports x86, ARM, MIPS, Power**
- **Based on technology from the [OpenEmbedded Project](#)**
- **Layer architecture allows for easy re-use of code**





# What is the Yocto Project?

- **Umbrella organization under Linux Foundation**
- **Backed by many companies interested in making Embedded Linux easier for the industry**
- **Co-maintains OpenEmbedded Core and other tools (including opkg)**

# Yocto Project Governance

- Organized under the Linux Foundation
- Split governance model
- Technical Leadership Team
- Advisory Board made up of participating organizations



# Yocto Project Member Organizations

Gold members



Silver members



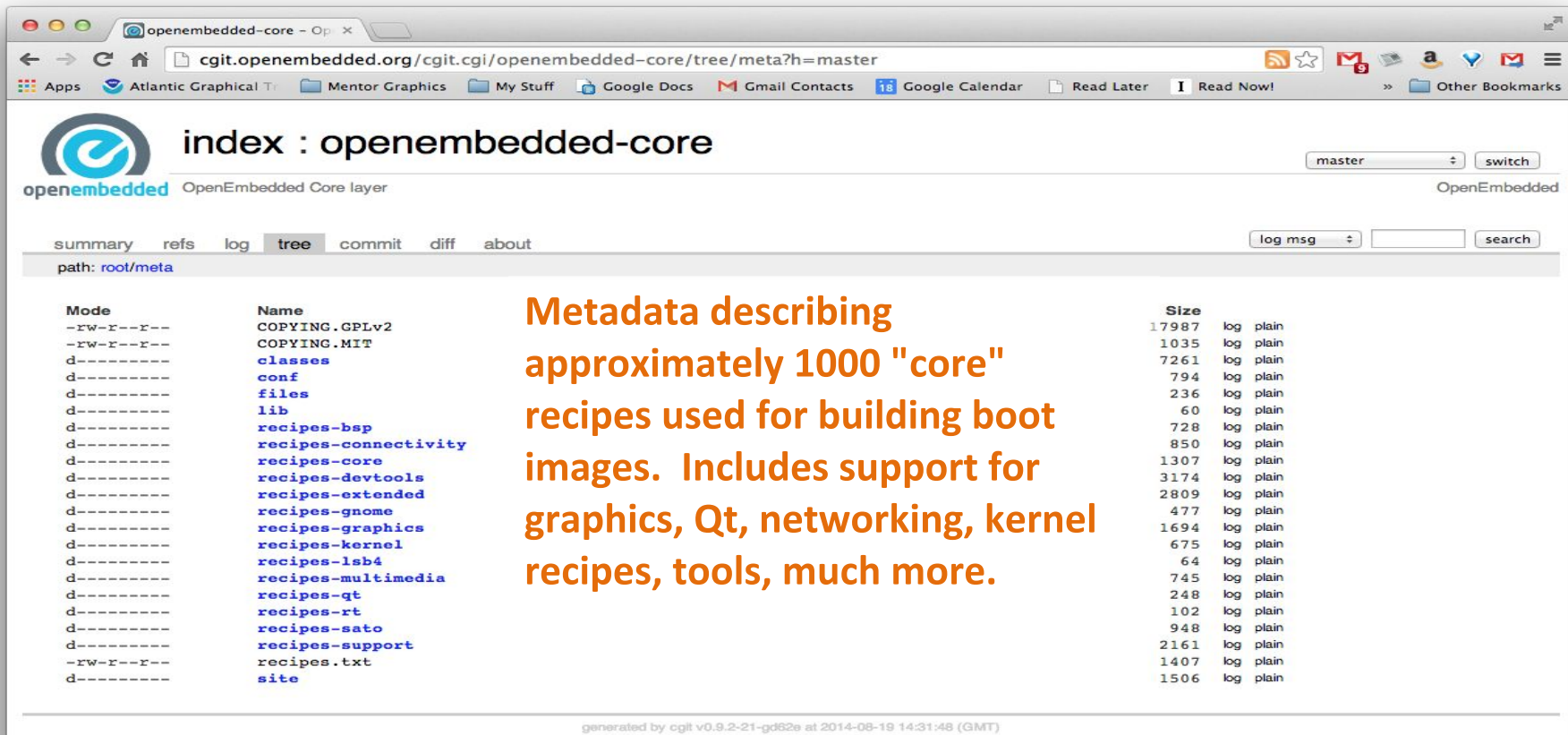
# Yocto Project Overview

- **YP builds packages - then uses these packages to build bootable images**
- **Supports use of popular package formats including:**
  - ◆ rpm, deb, ipk
- **Releases on a 6-month cadence**
- **Latest (stable) kernel, toolchain and packages, documentation**
- **App Development Tools including Eclipse plugin, SDK, toaster**

# Yocto Project Release Versions

Name	Revision	Poky	Release Date
Bernard	1.0	5.0	Apr 5, 2011
Edison	1.1	6.0	Oct 17, 2011
Denzil	1.2	7.0	Apr 30, 2012
Danny	1.3	8.0	Oct 24, 2012
Dylan	1.4	9.0	Apr 26, 2013
Dora	1.5	10.0	Oct 19, 2013
Daisy	1.6	11.0	Apr 24, 2014
Dizzy	1.7	12.0	Oct 31, 2014
Fido	1.8	13.0	April 22, 2015
Jethro	2.0	14.0	Oct 31, 2015
Krogoth	2.1	15.0	April 29, 2016
Morty	2.2	16.0	Oct 28, 2016
Pytro	2.3	17.0	April, 2017

# Yocto is based on OpenEmbedded-core



The screenshot shows the OpenEmbedded-core Git repository index page. The browser address bar displays `cggit.openembedded.org/cggit.cgi/openembedded-core/tree/meta?h=master`. The page title is "index : openembedded-core". The OpenEmbedded logo is visible on the left. The "tree" tab is selected in the navigation bar. The path shown is `path: root/meta`. The file tree structure is listed with columns for Mode, Name, Size, and file type. The files include `COPYING.GPLv2`, `COPYING.MIT`, `classes`, `conf`, `files`, `lib`, `recipes-bsp`, `recipes-connectivity`, `recipes-core`, `recipes-devtools`, `recipes-extended`, `recipes-gnome`, `recipes-graphics`, `recipes-kernel`, `recipes-lsb4`, `recipes-multimedia`, `recipes-qt`, `recipes-rt`, `recipes-sato`, `recipes-support`, `recipes.txt`, and `site`. The file types are mostly "log" and "plain".

Mode	Name	Size		
-rw-r--r--	COPYING.GPLv2	17987	log	plain
-rw-r--r--	COPYING.MIT	1035	log	plain
d-----	classes	7261	log	plain
d-----	conf	794	log	plain
d-----	files	236	log	plain
d-----	lib	60	log	plain
d-----	recipes-bsp	728	log	plain
d-----	recipes-connectivity	850	log	plain
d-----	recipes-core	1307	log	plain
d-----	recipes-devtools	3174	log	plain
d-----	recipes-extended	2809	log	plain
d-----	recipes-gnome	477	log	plain
d-----	recipes-graphics	1694	log	plain
d-----	recipes-kernel	675	log	plain
d-----	recipes-lsb4	64	log	plain
d-----	recipes-multimedia	745	log	plain
d-----	recipes-qt	248	log	plain
d-----	recipes-rt	102	log	plain
d-----	recipes-sato	948	log	plain
d-----	recipes-support	2161	log	plain
-rw-r--r--	recipes.txt	1407	log	plain
d-----	site	1506	log	plain

generated by cggit v0.9.2-21-gd62e at 2014-08-18 14:31:48 (GMT)

Metadata describing approximately 1000 "core" recipes used for building boot images. Includes support for graphics, Qt, networking, kernel recipes, tools, much more.

# Intro to OpenEmbedded

- **The OpenEmbedded Project co-maintains OE-core build system:**
  - ◆ bitbake build tool and scripts
  - ◆ Metadata and configuration
- **Provides a central point for new metadata**
  - ◆ (see the OE Layer index)

# What is Bitbake?



## Bitbake

- ◆ Powerful and flexible build engine (Python)
- ◆ Reads metadata
- ◆ Determines dependencies
- ◆ Schedules tasks



**Metadata** – a structured collection of "recipes" which tell BitBake what to build, organized in layers

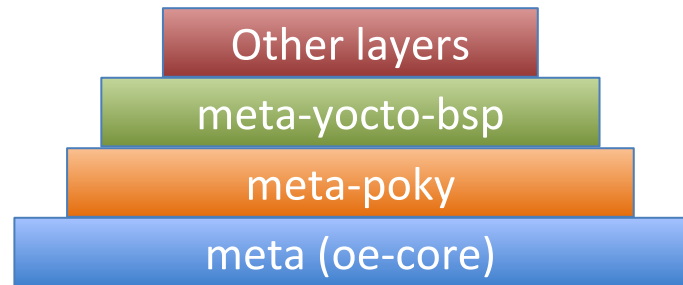


# OK, so what is Poky?

- Poky is a reference distribution
- Poky has its own git repo
  - ◆ git clone git://git.yoctoproject.org/poky

- **Primary Poky layers**

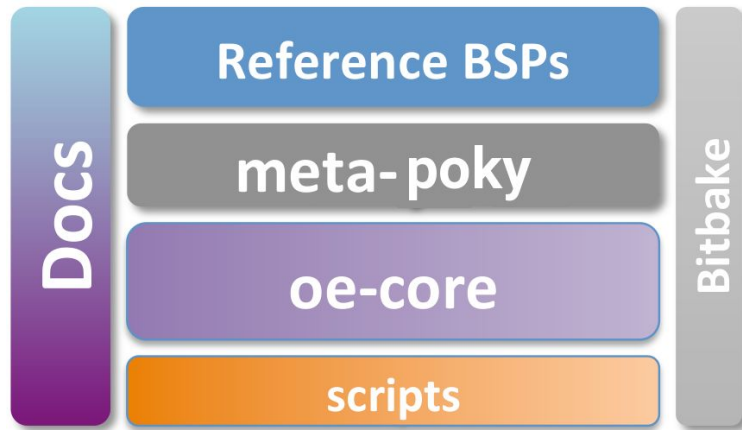
- ◆ oe-core (poky/meta)
- ◆ meta-poky (poky/meta-poky)
- ◆ meta-yocto-bsp



- Poky is the starting point for building things with the Yocto Project

# Poky in Detail

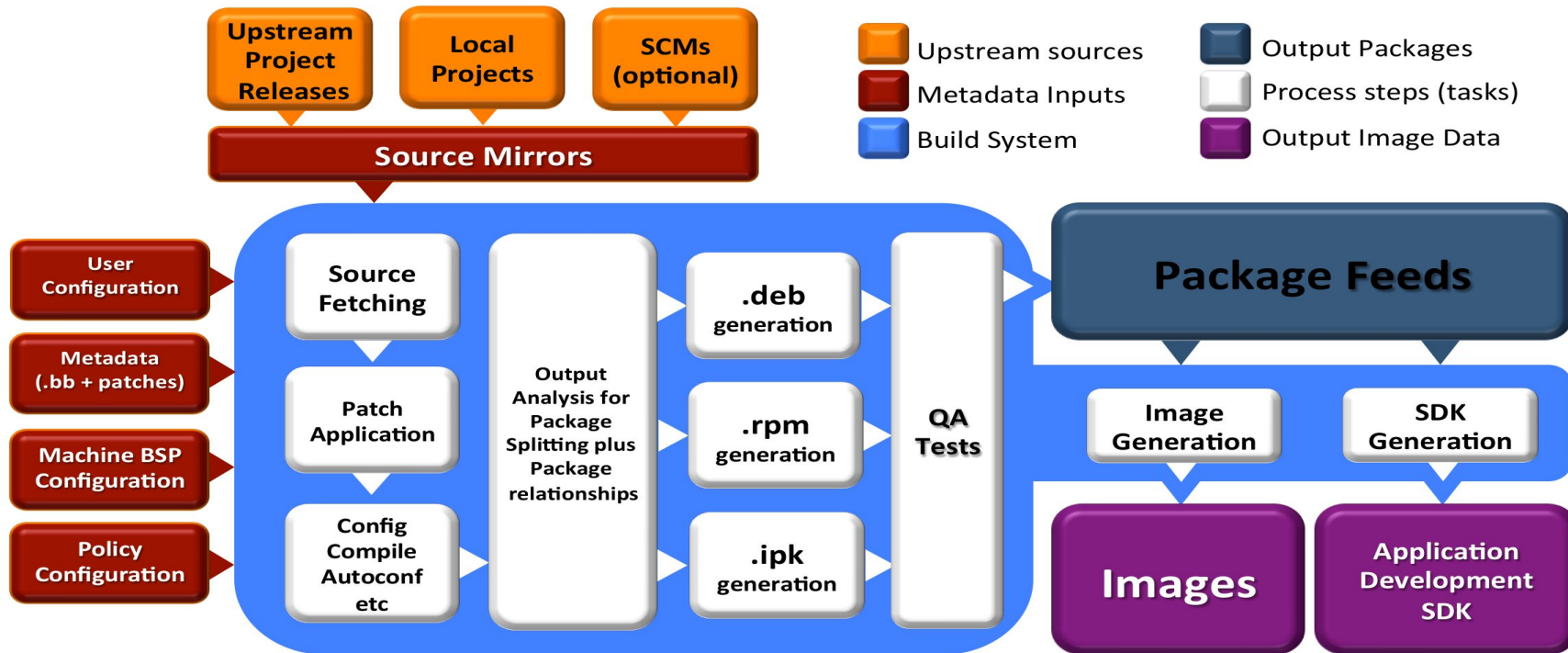
- **Contains core components**
- ◆ Bitbake tool: A python-based build engine
  - ◆ Build scripts (infrastructure)
  - ◆ Foundation package recipes (oe-core)
  - ◆ meta-poky (Contains distribution policy)
  - ◆ Reference BSPs
  - ◆ Yocto Project documentation



# Putting It All Together

- **Yocto Project** is a large collaboration
- **OpenEmbedded** is the build system
- **Bitbake** is the built tool
- **Poky** is the Yocto Project's reference distribution
  - Poky contains a version of bitbake and oe-core from which you can start your project

# Build System Workflow

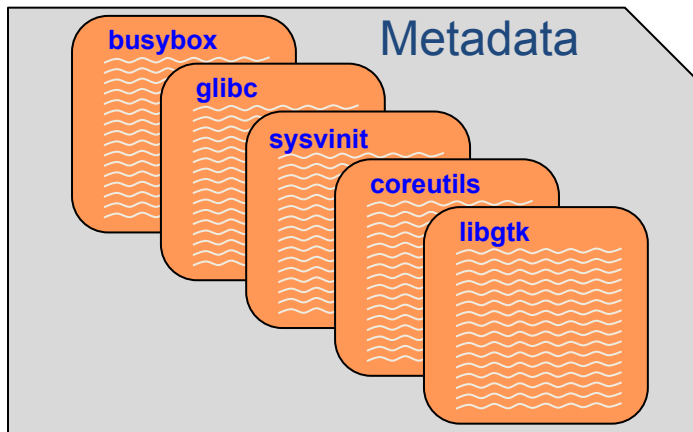


# BITBAKE

**This section will introduce the concept of the bitbake build tool and how it can be used to build recipes**

# Metadata and bitbake

- Most common form of metadata: **The Recipe**
- A *Recipe* provides a “list of ingredients” and “cooking instructions”
- Defines settings and a set of tasks used by bitbake to build binary packages



# What is Metadata?

- **Metadata exists in four general categories:**
- **Recipes (\*.bb)**
  - ◆ Usually describe build instructions for a single package
- **PackageGroups (special \*.bb)**
  - ◆ Often used to group packages together for a FS image
- **Classes (\*.bbclass)**
  - ◆ Inheritance mechanism for common functionality
- **Configuration (\*.conf)**
  - ◆ Drives the overall behavior of the build process

# Other Metadata

## ➤ Append files (\*.bbappend)

- ◆ Define additional metadata for a similarly named .bb file
- ◆ Can add or override previously set values

## ➤ Include files (\*.inc)

- ◆ Files which are used with the *include* directive
- ◆ Include files are typically found via the `BBPATH` variable



# OE-CORE Breakdown

openembedded-core - Op x

cgит.openembedded.org/cgit.cgi/openembedded-core/tree/meta?h=master

Apps Atlantic Graphical Tr Mentor Graphics My Stuff Google Docs Gmail Contacts 18 Google Calendar Read Later Read Now! Other Bookmarks

index : openembedded-core

OpenEmbedded Core layer

master switch

OpenEmbedded

summary refs log tree commit diff about

log msg search

path: root/meta

Mode	Name	Size
-rw-r--r--	COPYING.GPLv2	17987
-rw-r--r--	COPYING.MIT	1035
d-----	classes	7261
d-----	conf	794
d-----	files	236
d-----	lib	60
d-----	recipes-bsp	728
d-----	recipes-connectivity	850
d-----	recipes-core	1307
d-----	recipes-devtools	3174
d-----	recipes-extended	2809
d-----	recipes-gnome	477
d-----	recipes-graphics	1694
d-----	recipes-kernel	675
d-----	recipes-lsb4	64
d-----	recipes-multimedia	745
d-----	recipes-qt	248
d-----	recipes-rt	102
d-----	recipes-sato	948
d-----	recipes-support	2161
-rw-r--r--	recipes.txt	1407
d-----	site	1506

\*.bb: 839

Packagegroup\*: 23

\*.bbclass: 178

\*.conf: 65

\*.inc: 265

generated by cgit v0.9.2-21-gd62e at 2014-08-19 14:31:48 (GMT)

# Introduction to Bitbake

- ***Bitbake*** is a task executor and scheduler
- By default the ***build*** task for the specified recipe is executed  
\$ ***bitbake myrecipe***
- You can indicate which task you want run  
\$ ***bitbake -c clean myrecipe***
- You can get a list of tasks with  
\$ ***bitbake -c listtasks myrecipe***

# Building Recipes

- **By default the highest version of a recipe is built** (can be overridden with `DEFAULT_PREFERENCE` or `PREFERRED_VERSION` metadata)  
\$ **bitbake myrecipe**
- **You can specify the version of the package you want built (version of upstream source)**  
\$ **bitbake myrecipe-1.0**
- **You can also build a particular revision of the package metadata**  
\$ **bitbake myrecipe-1.0-r0**
- **Or you can provide a recipe file to build**  
\$ **bitbake -b mydir/myrecip.bb**

# Running bitbake for the First Time

- When you do a really big build, running with **--continue (-k)** means bitbake will proceed as far as possible after finding an error  
\$ **bitbake -k core-image-minimal**
- ◆ When running a long build (e.g. overnight) you want as much of the build done as possible before debugging issues
- Running bitbake normally will stop on the first error found  
\$ **bitbake core-image-minimal**
- *We'll look at debugging recipe issue later...*

# Bitbake is a Task Scheduler

- Bitbake builds recipes by scheduling build tasks in parallel  
**\$ bitbake recipe**
- This looks for `recipe.bb` in `BBFILES`
- Each recipe defines build tasks, each which can depend on other tasks
- Recipes can also depend on other recipes, meaning more than one recipe may be built
- Tasks from more than one recipe are often executed in parallel at once on multi-cpu build machines

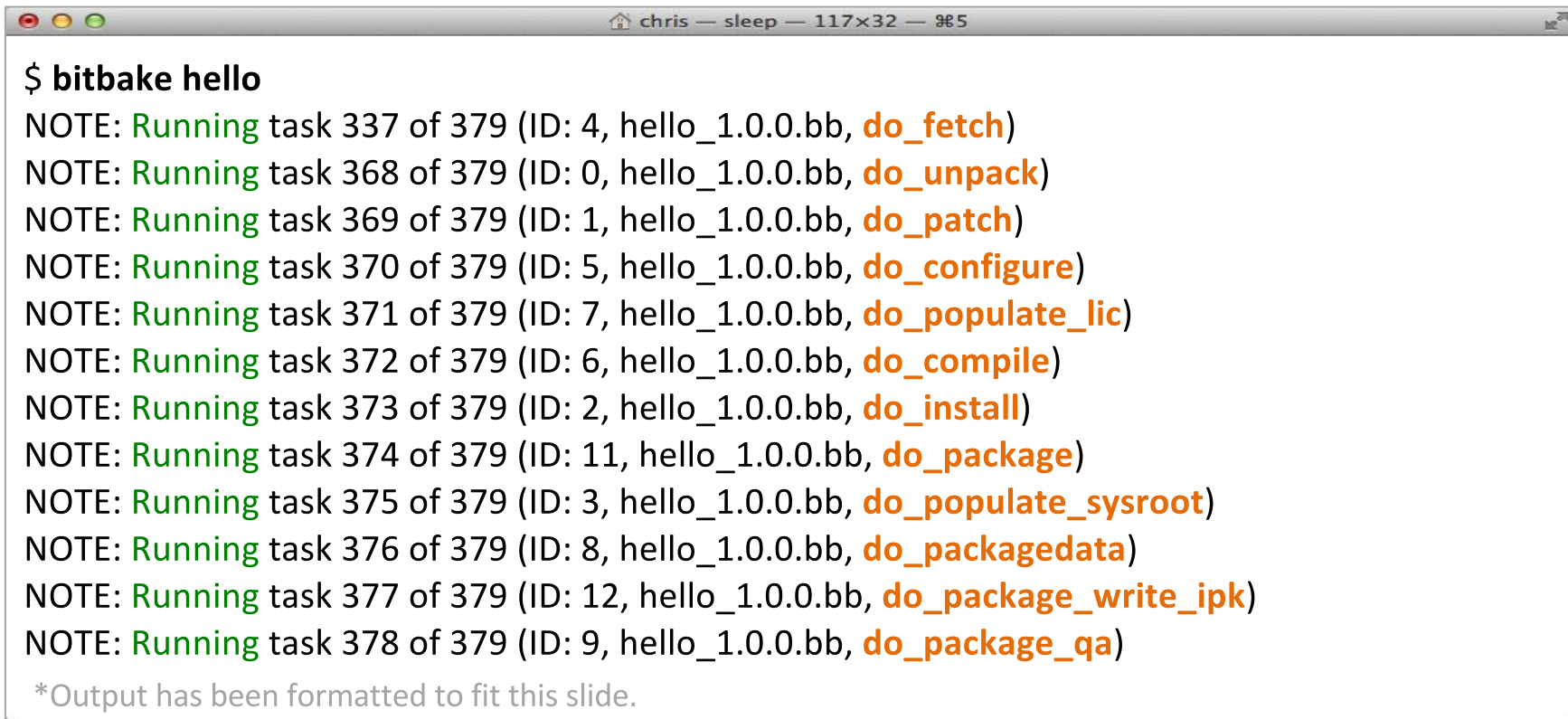
# Recipe Basics – Default Tasks\*

<code>do fetch</code>	Locate and download source code
<code>do unpack</code>	Unpack source into working directory
<code>do patch</code>	Apply any patches
<code>do configure</code>	Perform any necessary pre-build configuration
<code>do compile</code>	Compile the source code
<code>do install</code>	Installation of resulting build artifacts in WORKDIR
<code>do populate sysroot</code>	Copy artifacts to sysroot
<code>do package *</code>	Create binary package(s)

Note: to see the list of all possible tasks for a recipe, do this:  
`$ bitbake -c listtasks <recipe_name>`

\*Simplified for illustration

# Simple recipe task list\*

A terminal window with a title bar showing 'chris — sleep — 117x32 — 985'. The terminal displays the output of a 'bitbake hello' command, showing a sequence of tasks being executed. Each line starts with 'NOTE: Running' followed by task details and a task name in orange. The tasks are: task 337 (do\_fetch), task 368 (do\_unpack), task 369 (do\_patch), task 370 (do\_configure), task 371 (do\_populate\_lic), task 372 (do\_compile), task 373 (do\_install), task 374 (do\_package), task 375 (do\_populate\_sysroot), task 376 (do\_packagedata), task 377 (do\_package\_write\_ipk), and task 378 (do\_package\_qa).

```
$ bitbake hello
NOTE: Running task 337 of 379 (ID: 4, hello_1.0.0.bb, do_fetch)
NOTE: Running task 368 of 379 (ID: 0, hello_1.0.0.bb, do_unpack)
NOTE: Running task 369 of 379 (ID: 1, hello_1.0.0.bb, do_patch)
NOTE: Running task 370 of 379 (ID: 5, hello_1.0.0.bb, do_configure)
NOTE: Running task 371 of 379 (ID: 7, hello_1.0.0.bb, do_populate_lic)
NOTE: Running task 372 of 379 (ID: 6, hello_1.0.0.bb, do_compile)
NOTE: Running task 373 of 379 (ID: 2, hello_1.0.0.bb, do_install)
NOTE: Running task 374 of 379 (ID: 11, hello_1.0.0.bb, do_package)
NOTE: Running task 375 of 379 (ID: 3, hello_1.0.0.bb, do_populate_sysroot)
NOTE: Running task 376 of 379 (ID: 8, hello_1.0.0.bb, do_packagedata)
NOTE: Running task 377 of 379 (ID: 12, hello_1.0.0.bb, do_package_write_ipk)
NOTE: Running task 378 of 379 (ID: 9, hello_1.0.0.bb, do_package_qa)

*Output has been formatted to fit this slide.
```

\*Simplified for illustration

# SSTATE CACHE

- **Several bitbake tasks can use past versions of build artefacts if there have been no changes since the last time you built them**

do_packagedata	Creates package metadata used by the build system to generate the final packages
do_package	Analyzes the content of the holding area and splits it into subsets based on available packages and files
do_package_write_rpm	Creates the actual RPM packages and places them in the Package Feed area
do_populate_lic	Writes license information for the recipe that is collected later when the image is constructed
do_populate_sysroot	Copies a subset of files installed by do_install into the sysroot in order to make them available to other recipes



# Simple recipe build from sstate cache\*



A terminal window with a title bar showing 'chris — sleep — 117x32 — 95'. The terminal contains the following text:

```
$ bitbake -c clean hello
$ bitbake hello
NOTE: Running setscene task 69 of 74 (hello_1.0.0.bb, do_populate_sysroot_setscene)
NOTE: Running setscene task 70 of 74 (hello_1.0.0.bb, do_populate_lic_setscene)
NOTE: Running setscene task 71 of 74 (hello_1.0.0.bb, do_package_qa_setscene)
NOTE: Running setscene task 72 of 74 (hello_1.0.0.bb, do_package_write_ipk_setscene)
NOTE: Running setscene task 73 of 74 (hello_1.0.0.bb, do_packagedata_setscene)
```

\*Output has been formatted to fit this slide.

# RECIPES

**This section will introduce the concept of metadata and recipes and how they can be used to automate the building of packages**

# What is a Recipe?

- **A recipe is a set of instructions for building packages, including:**
  - ◆ Where to obtain the upstream sources and which patches to apply (this is called “*fetching*”)
    - `SRC_URI`
  - ◆ Dependencies (on libraries or other recipes)
    - `DEPENDS`, `RDEPENDS`
  - ◆ Configuration/compilation options
    - `EXTRA_OECONF`, `EXTRA_OEMAKE`
  - ◆ Define which files go into what output packages
    - `FILES_*`

# Example Recipe – ethtool\_3.15.bb

```
chris — ssh — 80x24

SUMMARY = "Display or change ethernet card settings"
DESCRIPTION = "A small utility for examining and tuning the settings of your ethernet-based network interfaces."
HOMEPAGE = "http://www.kernel.org/pub/software/network/ethtool/"
SECTION = "console/network"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://ethtool.c;beginline=4;endline=17;md5=c19b30548c582577fc6b443626fc1216"

SRC_URI = "${KERNELORG_MIRROR}/software/network/ethtool/ethtool-${PV}.tar.gz \
           file://run-ptest \
           file://avoid_parallel_tests.patch \
           file://ethtool-uint.patch \
           "

SRC_URI[md5sum] = "7e94dd958bcd639aad2e5a752e108b24"
SRC_URI[sha256sum] = "562e3cc675cf5b1ac655cd060f032943a2502d4d59e5f278f02aae92562ba261"

inherit autotools ptest
RDEPENDS_${PN}-ptest += "make"
```

# What can a Recipe Do?

- **Build one or more packages from source code**
  - ◆ Host tools, compiler, utilities
  - ◆ Bootloader, Kernel, etc
  - ◆ Libraries, interpreters, etc
  - ◆ Userspace applications
- **Package Groups**
- **Full System Images**

# Recipe Operators

**A = "foo"**

(late assignment)

**B ?= "0t"**

(default value)

**C ??= "abc"**

(late default)

**D := "xyz"**

(Immediate assignment)

**A .= "bar"**



**"foobar"**

(append)

**B =. "WO"**



**"W00t"**

(prepend)

**C += "def"**



**"abc def"**

(append)

**D =+ "uvw"**



**"uvw xyz"**

(prepend)

# More Recipe Operators

A = "foo"

A\_append = "bar" → "foobar"

B = "0t"

B\_prepend = "WO" → "W00t"

**OVERRIDES = "os:arch:machine"**

A = "abc"

A\_os = "ABC"

(Override)

A\_append\_arch = "def"

(Conditional append)

A\_prepend\_os = "XYZ"

(Conditional prepend)

# Bitbake Variables/Metadata

- **These are set automatically by bitbake**
  - ◆ **TOPDIR** – The build directory
  - ◆ **LAYERDIR** – Current layer directory
  - ◆ **FILE** – Path and filename of file being processed
- **Policy variables control the build**
  - ◆ **BUILD\_ARCH** – Host machine architecture
  - ◆ **TARGET\_ARCH** – Target architecture
  - ◆ And many others...



# Build Time Metadata

- **PN** – Package name (“myrecipe”)
- **PV** – Package version (1.0)
- **PR** – Package Release (r0)
- **P** = “\${PN}-\${PV}”
- **PF** = “\${PN}-\${PV}-\${PR}”
- **FILE\_DIRNAME** – Directory for FILE
- **FILESPATH** = “\${FILE\_DIRNAME}/\${PF}:\
- \${FILE\_DIRNAME}/\${P}:\
- \${FILE\_DIRNAME}/\${PN}:\
- \${FILE\_DIRNAME}/files:\${FILE\_DIRNAME}

# Build Time Metadata

- **TOPDIR** - The build directory
- **TMPDIR** = "\${TOPDIR}/tmp"
- **WORKDIR** = "\${TMPDIR}/work/\${PF}"
- **S** = "\${WORKDIR}/\${P}" (Source dir)
- **B** = "\${S}" (Build dir)
- **D** = "\${WORKDIR}/\${image}" (Destination dir)
- **DEPLOY\_DIR** = "\${TMPDIR}/deploy"
- **DEPLOY\_DIR\_IMAGE** = "\${DEPLOY\_DIR}/images"

# Dependency Metadata

## ➤ Build time package variables

- ◆ **DEPENDS** – Build time package dependencies
- ◆ **PROVIDES** = “`${P} ${PF} ${PN}`”

## ➤ Runtime package variables

- ◆ **RDEPENDS** – Runtime package dependencies
- ◆ **RRECOMMENDS** – Runtime recommended packages
- ◆ **RSUGGESTS** – Runtime suggested packages
- ◆ **RPROVIDES** – Runtime provides
- ◆ **RCONFLICTS** – Runtime package conflicts
- ◆ **RREPLACES** – Runtime package replaces

# Common Metadata

- **Variables you commonly set**
  - ◆ **SUMMARY** – Short description of package/recipe
  - ◆ **HOMEPAGE** – Upstream web page
  - ◆ **LICENSE** – Licenses of included source code
  - ◆ **LIC\_FILES\_CHKSUM** – Checksums of license files at time of packaging (checked for change by build)
  - ◆ **SRC\_URI** – URI of source code, patches and extra files to be used to build packages. Uses different fetchers based on the URI.
  - ◆ **FILES** – Files to be included in binary packages

# Examining Recipes: bc

➤ Look at 'bc' recipe:

➤ Found in

**poky/meta/recipes-extended/bc/bc\_1.06.bb**

- ◆ Uses **LIC\_FILES\_CHKSUM** and **SRC\_URI** checksums
- ◆ Note the **DEPENDS** build dependency declaration indicating that this package depends on `flex` to build

# Examining Recipes: bc.bb

SUMMARY = "Arbitrary precision calculator language"

Homepage = "http://www.gnu.org/software/bc/bc.html"

LICENSE = "GPLv2+ & LGPLv2.1"

LIC\_FILES\_CHKSUM = "file://COPYING;md5=94d55d512a9ba36caa9b7df079bae19f \  
file://COPYING.LIB;md5=d8045f3b8f929c1cb29a1e3fd737b499 \  
file://bc/bcdefs.h;endline=31;md5=46dffdaf10a99728dd8ce358e45d46d8 \  
file://dc/dc.h;endline=25;md5=2f9c558cdd80e31b4d904e48c2374328 \  
file://lib/number.c;endline=31;md5=99434a0898abca7784acfd36b8191199"

SECTION = "base"

DEPENDS = "flex"

PR = "r3"

SRC\_URI = "\${GNU\_MIRROR}/bc/bc-\${PV}.tar.gz \  
file://fix-segment-fault.patch "

SRC\_URI[md5sum] = "d44b5dddebd8a7a7309aea6c36fda117"

SRC\_URI[sha256sum] = "4ef6d9f17c3c0d92d8798e35666175ecd3d8efac4009d6457b5c99cea72c0e33"

inherit autotools texinfo update-alternatives

ALTERNATIVE\_\${PN} = "dc"

ALTERNATIVE\_PRIORITY = "100"

BBCLASSEXTEND = "native"

# Building upon bbclass

- Use inheritance for common design patterns
- Provide a class file (`.bbclass`) which is then inherited by other recipes (`.bb` files)

`inherit autotools`

- ◆ Bitbake will include the *autotools.bbclass* file
- ◆ Found in a *classes* directory via the `BBPATH`

# Examining Recipes: flac

➤ Look at 'flac' recipe

➤ Found in

`poky/meta/recipes-multimedia/flac/flac_1.3.1.bb`

- ◆ Inherits from both *autotools* and *gettext*
- ◆ Customizes autoconf configure options (`EXTRA_OECONF`) based on "TUNE" features
- ◆ Breaks up output into multiple binary packages
  - See `PACKAGES` var. This recipe produces additional packages with those names, while the `FILES_*` vars specify which files go into these additional packages



# Examining Recipes: flac.bb

SUMMARY = "Free Lossless Audio Codec"

DESCRIPTION = "FLAC stands for Free Lossless Audio Codec, a lossless audio compression format."

Homepage = "https://xiph.org/flac/"

BUGTRACKER = "http://sourceforge.net/p/flac/bugs/"

SECTION = "libs"

LICENSE = "GFDL-1.2 & GPLv2+ & LGPLv2.1+ & BSD"

LIC\_FILES\_CHKSUM = "file://COPYING.FDL;md5=ad1419ecc56e060eccf8184a87c4285f \  
file://src/Makefile.am;beginline=1;endline=17;md5=0a853b81d9d43d8aad3b53b05cfcc37e \  
file://COPYING.GPL;md5=b234ee4d69f5fce4486a80fdaf4a4263 \  
file://src/flac/main.c;beginline=1;endline=18;md5=d03a766558d233f9cc3ac5dfafd49deb \  
file://COPYING.LGPL;md5=fbc093901857fcd118f065f900982c24 \  
file://src/plugin\_common/all.h;beginline=1;endline=18;md5=7c8a3b9e1e66ed0aba765bc6f35da85d \  
file://COPYING.Xiph;md5=a2c4b71c0198682376d483eb5bcc9197 \  
file://include/FLAC/all.h;beginline=65;endline=70;md5=64474f2b22e9e77b28d8b8b25c983a48"

DEPENDS = "libogg"

SRC\_URI = "http://downloads.xiph.org/releases/flac/\${BP}.tar.xz"

SRC\_URI[md5sum] = "b9922c9a0378c88d3e901b234f852698"

SRC\_URI[sha256sum] = "4773c0099dba767d963fd92143263be338c48702172e8754b9bc5103efe1c56c"

(con't next page)

# Examining Recipes: flac.bb (con't)

(con't from previous page)

inherit autotools gettext

```
EXTRA_OECONF = "--disable-oggtest \  
               --with-ogg-libraries=${STAGING_LIBDIR} \  
               --with-ogg-includes=${STAGING_INCDIR} \  
               --disable-xmms-plugin \  
               --without-libiconv-prefix \  
               ac_cv_prog_NASM="" \  
"
```

```
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "altivec", " --enable-altivec", \  
                                       " --disable-altivec", d)}"
```

```
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "core2", " --enable-sse", "", d)}"
```

```
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "corei7", " --enable-sse", "", d)}"
```

```
PACKAGES += "libflac libflac++ liboggflac liboggflac++"
```

```
FILES_${PN} = "${bindir}/*"
```

```
FILES_libflac = "${libdir}/libFLAC.so.*"
```

```
FILES_libflac++ = "${libdir}/libFLAC++.so.*"
```

```
FILES_liboggflac = "${libdir}/libOggFLAC.so.*"
```

```
FILES_liboggflac++ = "${libdir}/libOggFLAC++.so.*"
```

# Grouping Local Metadata

- Sometimes sharing metadata between recipes is easier via an *include file*

**include file.inc**

- ◆ Will include `.inc` file if found via BBPATH
- ◆ Can also specify an absolute path
- ◆ If not found, will continue without an error

**require file.inc**

- ◆ Same as an include
- ◆ Fails with an error if not found

# Examining Recipes: ofono

➤ Look at 'ofono' recipe(s):

➤ Found in

`poky/meta/recipes-connectivity/ofono/ofono_1.16.bb`

- ◆ Splits recipe into common `.inc` file to share **common metadata** between multiple recipes
- ◆ Sets a conditional build configuration options through the `PACKAGECONFIG` var based on a `DISTRO_FEATURE` (in the `.inc` file)
- ◆ Sets up an init service via `do_install_append()`
- ◆ Has a `_git` version of the recipe (not shown)

# Examining Recipes: ofono.bb

```
require ofono.inc
```

```
SRC_URI = "\
    ${KERNELORG_MIRROR}/linux/network/${BPN}/${BP}.tar.xz \
    file://ofono \
    file://Revert-test-Convert-to-Python-3.patch \
    file://0001-backtrace-Disable-for-non-glibc-C-libraries.patch \
"
SRC_URI[md5sum] = "c31b5b55a1d68354bff771d3edf02829"
SRC_URI[sha256sum] = \
    "403b98dadece8bc804c0bd16b96d3db5a3bb0f84af64b3d67924da2d1a754b07"

CFLAGS_append_libc-uclibc = " -D_GNU_SOURCE"
```

# Examining Recipes: ofono.inc

```

HOMEPAGE = "http://www.ofono.org"
SUMMARY = "open source telephony"
DESCRIPTION = "oFono is a stack for mobile telephony devices on Linux. oFono supports speaking to telephony devices through specific drivers, or with generic AT commands."
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=eb723b61539feef013de476e68b5c50a \
    file://src/ofono.h;beginline=1;endline=20;md5=3ce17d5978ef3445def265b98899c2ee"

inherit autotools pkgconfig update-rc.d systemd bluetooth

DEPENDS = "dbus glib-2.0 udev mobile-broadband-provider-info"

INITSCRIPT_NAME = "ofono"
INITSCRIPT_PARAMS = "defaults 22"

PACKAGECONFIG ??= "\
    ${@bb.utils.contains('DISTRO_FEATURES', 'systemd', 'systemd', '', d)} \
    ${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth', 'bluez', '', d)} \
"

PACKAGECONFIG[systemd] = "--with-systemdunitdir=${systemd_unitdir}/system/, \
    --with-systemdunitdir="
PACKAGECONFIG[bluez] = "--enable-bluetooth, --disable-bluetooth, ${BLUEZ}"
```

(con't next page)

# Examining Recipes: ofono.inc

(con't from previous page)

```
EXTRA_OECONF += "--enable-test"
```

```
SYSTEMD_SERVICE_${PN} = "ofono.service"
```

```
do_install_append() {  
    install -d ${D}${sysconfdir}/init.d/  
    install -m 0755 ${WORKDIR}/ofono ${D}${sysconfdir}/init.d/ofono  
}
```

```
PACKAGES += "${PN}-tests"
```

```
RDEPENDS_${PN} += "dbus"
```

```
FILES_${PN} += "${base_libdir}/udev ${systemd_unitdir}"
```

```
FILES_${PN}-tests = "${libdir}/${BPN}/test"
```

```
RDEPENDS_${PN}-tests = "python python-pyobject python-dbus"
```

# WHEN THINGS GO WRONG

**Some useful tools to help guide you when something goes wrong**



# Bitbake Environment

- Each recipe has its own environment which contains all the variables and methods required to build that recipe
- You've seen some of the variables already
  - ◆ DESCRIPTION, SRC\_URI, LICENSE, S, LIC\_FILES\_CHKSUM, do\_compile(), do\_install()
- Example
  - ◆ S = "\${WORKDIR}"
  - ◆ What does this mean?

# Examine a Recipe's Environment

- **To view a recipe's environment**

```
$ bitbake -e myrecipe
```

- **Where is the source code for this recipe"**

```
$ bitbake -e virtual/kernel | grep "^S="
```

```
S="${HOME}/yocto/build/tmp/work-shared/qemuarm/kernel-source"
```

- **What file was used in building this recipe?**

```
$ bitbake -e netbase | grep "^FILE="
```

```
FILE="${HOME}/yocto/poky/meta/recipes-core/netbase/netbase_5.3.bb"
```

# Examine a Recipe's Environment (cont'd)

## ➤ What is this recipe's full version string?

```
$ bitbake -e netbase | grep "^PF="
PF="netbase-1_5.3-r0"
```

## ➤ Where is this recipe's BUILD directory?

```
$ bitbake -e virtual/kernel | grep "^B="
B="${HOME}/yocto/build/tmp/work/qemuarm-poky-linux-\
gnueabi/linux-yocto/3.19.2+gitAUTOINC+9e70b482d3\
_473e2f3788-r0/linux-qemuarm-standard-build"
```

## ➤ What packages were produced by this recipe?

```
$ bitbake -e virtual/kernel | grep "^PACKAGES="
PACKAGES="kernel kernel-base kernel-vmlinux kernel-image \
kernel-dev kernel-modules kernel-devicetree"
```

# BitBake Log Files

➤ **Every build produces lots of log output for diagnostics and error chasing**

◆ **Verbose log of bitbake console output:**

○ Look in `.../tmp/log/cooker/<machine>`

```
$ cat tmp/log/cooker/qemuarm/20160119073325.log | grep 'NOTE:.*task.*Started'
```

```
NOTE: recipe hello-1.0.0-r0: task do_fetch: Started  
NOTE: recipe hello-1.0.0-r0: task do_unpack: Started  
NOTE: recipe hello-1.0.0-r0: task do_patch: Started  
NOTE: recipe hello-1.0.0-r0: task do_configure: Started  
NOTE: recipe hello-1.0.0-r0: task do_populate_lic: Started  
NOTE: recipe hello-1.0.0-r0: task do_compile: Started  
NOTE: recipe hello-1.0.0-r0: task do_install: Started  
NOTE: recipe hello-1.0.0-r0: task do_populate_sysroot: Started  
NOTE: recipe hello-1.0.0-r0: task do_package: Started  
NOTE: recipe hello-1.0.0-r0: task do_packagedata: Started  
NOTE: recipe hello-1.0.0-r0: task do_package_write_rpm: Started  
NOTE: recipe hello-1.0.0-r0: task do_package_qa: Started  
NOTE: recipe ypdd-image-1.0.0-r0: task do_rootfs: Started
```

# BitBake Per-Recipe Log Files

- Every **recipe** produces lots of log output for diagnostics and debugging
- Use the Environment to find the log files for a given recipe:

```
$ bitbake -e hello | grep "^T="
```

```
T="${HOME}yocto/build/tmp/work/armv5e-poky-linux-gnueabi/hello/1.0.0-r0/temp"
```

- Each task that runs for a recipe produces "log" and "run" files in **`${WORKDIR}/temp`**

# BitBake Per-Recipe Log Files

**\$ cd \${T}** (See definition of T in previous slide)

**\$ find . -type l -name 'log.\*'**

**./log.do\_package\_qa**

**./log.do\_package\_write\_rpm**

**./log.do\_package**

**./log.do\_fetch**

**./log.do\_populate\_lic**

**./log.do\_install**

**./log.do\_configure**

**./log.do\_unpack**

**./log.do\_populate\_sysroot**

**./log.do\_compile**

**./log.do\_packagedata**

**./log.do\_patch**



These files contain the  
output of the respective  
tasks for each recipe

# BitBake Per-Recipe Log Files

**\$ cd \${T}** (See definition of T in previous slide)

**\$ find . -type l -name 'run.\*'**

**./run.do\_fetch**

**./run.do\_patch**

**./run.do\_configure**

**./run.do\_populate\_sysroot**

**./run.do\_package\_qa**

**./run.do\_unpack**

**./run.do\_compile**

**./run.do\_install**

**./run.do\_packagedata**

**./run.do\_populate\_lic**

**./run.do\_package**

**./run.do\_package\_write\_rpm**



These files contain the  
commands executed which  
produce the build results

# BUILDING A FULL EMBEDDED IMAGE WITH YOCTO

This section will introduce the concept of building an initial system image



# Quick Start Guide in one Slide

## 1. Download Yocto Project sources:

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-2.2/poky-morty-16.0.0.tar.bz2
```

```
$ tar xf poky-morty-16.0.0.tar.bz2
```

◆ Can also use git and checkout a known branch e.g. morty

```
$ git clone -b morty git://git.yoctoproject.org/poky.git
```

## 2. Build one of the reference Linux distributions:

```
$ source poky-morty-16.0.0/oe-init-build-env
```

◆ Check/Edit local.conf for sanity (e.g. modify MACHINE = “qemux86” or MACHINE = “qemuarm”)

```
build$ bitbake -k core-image-minimal
```

## 3. Run the image under emulation:

```
$ runqemu qemux86
```

## 4. Profit!!! (well... actually there is more work to do...)

# Host System Layout

`$HOME/yocto/`

`|---build` (or whatever name you choose)

Project build directory

`|---downloads` (`DL_DIR`)

Downloaded source cache

`|---poky` (**(Do Not Modify anything in here\*)**)

Poky, bitbake, scripts, oe-core, metadata

`|---sstate-cache` (`SSTATE_DIR`)

Binary build cache

**\* We will cover how to use layers to make changes later**

# Poky Layout

\$HOME/yocto/poky/

| --- LICENSE

| --- README

| --- README.hardware

| --- bitbake/

| --- documentation/

| --- meta/

| --- meta-poky/

| --- meta-yocto-bsp/

| --- oe-init-build-env

| --- scripts/

(The build tool)

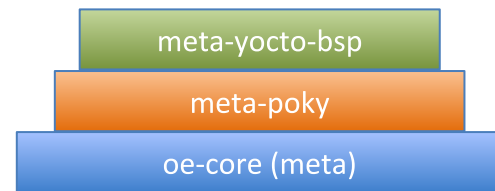
(oe-core)

(Yocto distro metadata)

(Yocto Reference BSPs)

(Project setup script)

(Scripts and utilities)



*Note: A few files have been items omitted to facility the presentation on this slide*

# Setting up a Build Directory

## ➤ Start by setting up a build directory

- ◆ Local configuration
- ◆ Temporary build artifacts

```
$ cd $HOME/yocto/
```

```
$ source ./poky/oe-init-build-env build
```

## ➤ Replace *build* with whatever directory name you want to use for your project

## ➤ You need to re-run this script in any new terminal you start (and don't forget the project directory)

# Build directory Layout

\$HOME/yocto/build/

| - - - bitbake.lock

| - - - cache/

(bitbake cache files)

| - - - conf/

|     | - - bblayers.conf

(bitbake layers)

|     | - - local.conf

(local configuration)

|     ` - - site.conf

(optional site conf)

` - - - tmp/

(Build artifacts)

*Note: A few files have been items omitted to facility the presentation on this slide*

# Building a Linux Image

## ➤ General Procedure:

- ◆ Create a project directory using
  - `source oe-init-build-env [prj-dir]`
- ◆ Configure build by editing `local.conf`
- ◆ `$HOME/yocto/build/conf/local.conf`
  - Select appropriate `MACHINE` type
  - Set shared downloads directory (`DL_DIR`)
  - Set shared state directory (`SSTATE_DIR`)
- ◆ Build your selected Image
- ◆ `$ bitbake -k core-image-minimal`
- ◆ (Detailed steps follow...)

# Update Build Configuration

- Configure build by editing local.conf

`$HOME/yocto/build/conf/local.conf`

- ◆ Set appropriate MACHINE, DL\_DIR and SSTATE\_DIR
- ◆ Add the following to the bottom of local.conf

```
MACHINE = "qemuarm"
```

```
DL_DIR = "${TOPDIR}/../downloads"
```

```
SSTATE_DIR = "${TOPDIR}/../sstate-cache/${MACHINE}"
```

- Notice how you can use variables in setting these values

# Building an Embedded Image

- This builds an entire embedded Linux distribution
- Choose from one of the available Images
- The following builds a minimal embedded target  
`$ bitbake -k core-image-minimal`
- On a fast computer the first build may take the better part of an hour on a slow machine multiple ...
- The next time you build it (with no changes) it may take as little as 5 mins (due to the shared state cache)



# Booting Your Image with QEMU

- The **runqemu** script is used to boot the image with QEMU
- It auto-detects settings as much as possible, enabling the following command to boot our reference images:  
\$ **runqemu qemuarm [nographic]**
  - ◆ Use nographic if using a non-graphical session (ssh), do not type the square brackets
- Replace **qemuarm** with your value of **MACHINE**
- Your QEMU instance should boot
- Quit by closing the qemu window
- If using “nographic”, kill it from another terminal:  
\$ **killall qemu-system-arm**

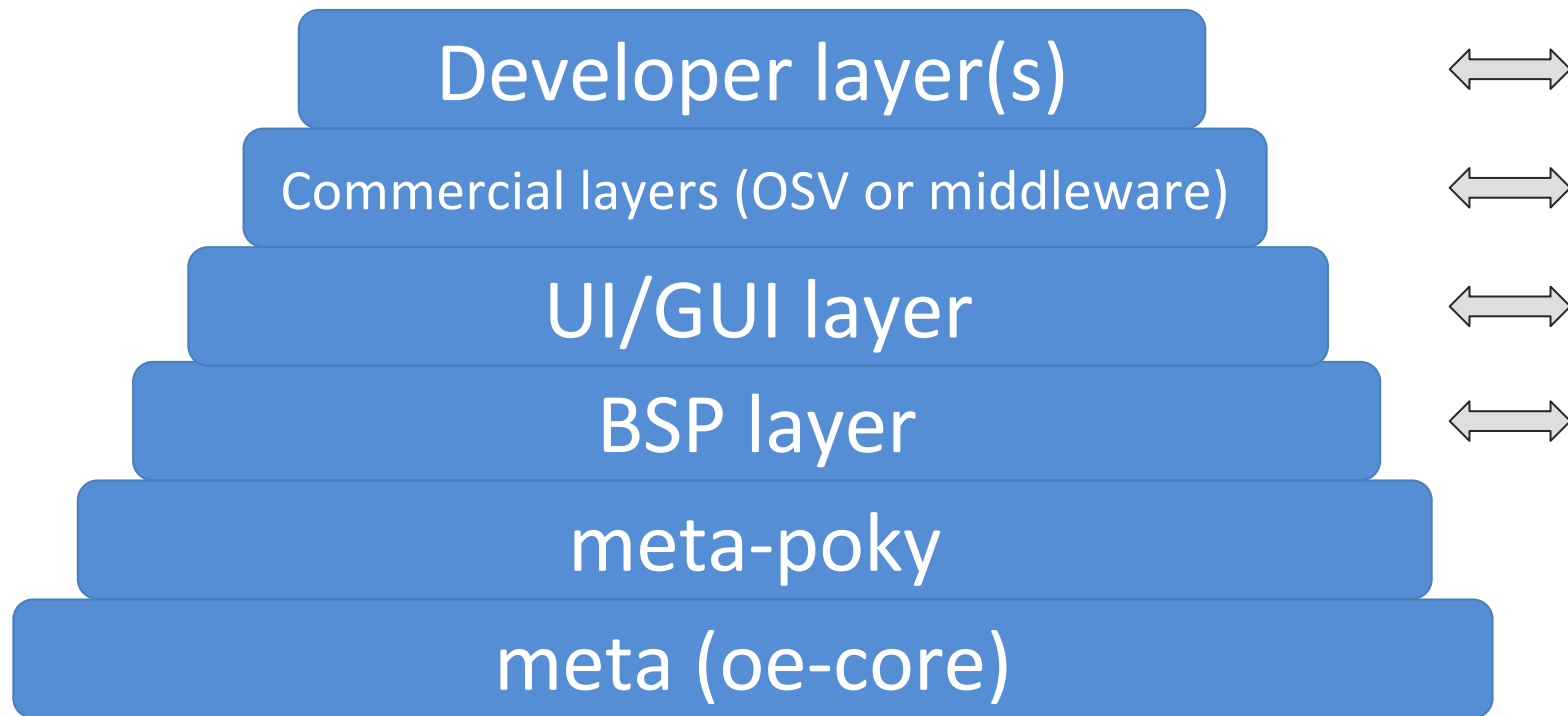
# LAYERS

**This section will introduce the concept of layers and how important they are in the overall build architecture**

# Layers

- Metadata is provided in a series of layers which allow you to override any value without editing the originally provided files
- A layer is a logical collection of metadata in the form of recipes
- A layer is used to represent oe-core, a Board Support Package (BSP), an application stack, and your new code
- All layers have a priority and can override policy, metadata and config settings of layers with a lesser priority

# Layer Hierarchy



# Using Layers

- Layers are added to your build by inserting them into the BBLAYERS variable within your bblayers file

`$HOME/yocto/build/conf/bblayers.conf`

```
BBLAYERS ?= "  
    ${HOME}/yocto/poky/meta  
    ${HOME}/yocto/poky/meta-poky  
    ${HOME}/yocto/poky/meta-yocto-bsp  
"
```

# Board Support Packages

- **BSPs are layers to enable support for specific hardware platforms**
- **Defines machine configuration variables for the board (`MACHINE`)**
- **Adds machine-specific recipes and customizations**
  - ◆ Boot loader
  - ◆ Kernel config
  - ◆ Graphics drivers (e.g, Xorg)
  - ◆ Additional recipes to support hardware features

# Notes on using Layers

- When doing development with Yocto, **do not edit files within the Poky source tree**
- Use a new custom layer for modularity and maintainability
- Layers also allow you to easily port from one version of Yocto/Poky to the next version

# Creating a Custom Layer

- Layers can be created manually
- They all start with “*meta-*” by convention
- However using the *yocto-layer* tool is easier
  - \$ *yocto-layer create ypdd*
  - ◆ This will create *meta-ypdd* in the current dir
- For Board Support Package Layers there is the *yocto-bsp* tool
  - \$ *yocto-bsp create mybsp arm*
  - ◆ This will create *meta-mybsp* in the current dir



# Create a Custom Layer

```
$ cd yocto
```

```
yocto$ source poky/oe-init-build-env build
```

```
yocto/build$ yocto-layer create ypdd
```

```
Please enter the layer priority you'd like to use for the layer: [default: 6] 6
```

```
Would you like to have an example recipe created? (y/n) [default: n] y
```

```
Please enter the name you'd like to use for your example recipe: [default:  
example] example
```

```
Would you like to have an example bbappend file created? (y/n) [default: n] n
```

New layer created in meta-ypdd.

Don't forget to add it to your BBLAYERS (for details see meta-ypdd\README).  
yocto/build\$

# The new Custom Layer

```
yocto/build$ tree meta-ypdd
```

```
meta-ypdd/
```

--COPYING.MIT	(The license file)
--README	(Starting point for README)
--conf	
`--layer.conf	(Layer configuration file)
`--recipes-example	(A grouping of recipies)
--example	(The example package)
--example-0.1	(files for v0.1 of example)
--example.patch	
`--helloworld.c	
`--example_0.1.bb	(The example recipe)

# Layer.conf

# We have a conf and classes directory, add to BBPATH

```
BBPATH += ":{LAYERDIR}"
```

# We have recipes-\* directories, add to BBFILES

```
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \  
            ${LAYERDIR}/recipes-*/*/*.bbappend"
```

```
BBFILE_COLLECTIONS += "ypdd"
```

```
BBFILE_PATTERN_ypdd = "^${LAYERDIR}/"
```

```
BBFILE_PRIORITY_ypdd = "6"
```

# Adding Layers to Your Build

- Add your layer to *bblayers.conf*
- `$HOME/yocto/build/conf/bblayers.conf`

```
BBLAYERS  ?=  "  
    ${HOME}/yocto/poky/meta  
    ${HOME}/yocto/poky/meta-poky  
    ${HOME}/yocto/poky/meta-yocto-bsp  
➔  ${HOME}/yocto/build/meta-ypdd  
    "
```

# Build Your New Recipe

- You can now build the new recipe  
\$ **bitbake example**
- This will now build the ***example\_0.1.bb*** recipe  
which is found in  
**meta-ypdd/recipes-example/example/example\_0.1.bb**

Note: Build fails w/o \${CFLAGS} and \${LDFLAGS} meanwhile (QA-error) in the recipe.

# IMAGES

**This section will introduce the concept of images; recipes which build embedded system images**

# What is an Image?

- **Building an image creates an entire Linux distribution from source**
  - ◆ **Compiler, tools, libraries**
  - ◆ **BSP: Bootloader, Kernel**
  - ◆ **Root filesystem:**
    - **Base OS**
    - **services**
    - **Applications**
    - **etc**

# Extending an Image

- You often need to create your own Image recipe in order to add new packages or functionality
- With Yocto/OpenEmbedded it is always preferable to extend an existing recipe or inherit a class
- The simplest way is to inherit the core-image bbclass
- You add packages to the image by adding them to **IMAGE\_INSTALL**



# A Simple Image Recipe

## ➤ Create an **images** directory

```
$ mkdir -p ${HOME}/yocto/build/meta-ypdd/recipes-core/images
```

## ➤ Create the image recipe

```
$ vi ${HOME}/yocto/build/meta-ypdd/recipes-core/images/ypdd-image.bb
```

```
DESCRIPTION = "A core image for YPDD"
```

```
LICENSE = "MIT"
```

```
# Core files for basic console boot
```

```
IMAGE_INSTALL = "packagegroup-core-boot"
```

```
# Add our desired packages
```

```
IMAGE_INSTALL += "psplash dropbear"
```

```
inherit core-image
```

```
IMAGE_ROOTFS_SIZE ?= "8192"
```

# Build and Boot Your Custom Image

- Enable the `meta-ypdd` layer in your build
- Edit `conf/bblayers.conf` and add the path to `meta-ypdd` to the `BBLAYERS` variable declaration

(example in the next slide)

# Add Your Layer

- Make sure your layer is added to BBLAYERS in *bblayers.conf*

```
$HOME/yocto/build/conf/bblayers.conf
```

```
BBLAYERS  ?= "  
    ${HOME}/yocto/poky/meta  
    ${HOME}/yocto/poky/meta-poky  
    ${HOME}/yocto/poky/meta-yocto-bsp  
    ➔ ${HOME}/yocto/build/meta-ypdd  
    "
```

- (We already did this step in a previous section)

# Build and Boot Your Custom Image

## ➤ Build your custom image:

```
$ bitbake ypdd-image
```

(If your SSTATE\_DIR is configured correctly from a previous build this should take less than 5 minutes)

## ➤ Boot the image with QEMU:

```
$ runqemu qemuarm \  
tmp/deploy/images/qemuarm/ypdd-image-qemuarm.ext4 \  
[nographic]
```

Use nographic if using ssh environment

# Build and Boot Your Custom Image

- Verify that dropbear ssh server is present  
\$ **which dropbear**
- If you used the graphical invocation of QEMU using VNC viewer, you will see the splash screen on boot.

# Toaster

The following section introduces toaster

## Recent builds

core-image-sato (+3) qemux86	<div><div></div></div>	ETA: 16:34
core-image-minimal qemuarm	<div><div></div></div>	ETA: 15:52
✔ core-image-sato atom-pc (15:22)	⚠ 4 warnings	Build time: 00:36:55
✖ core-image-x11 qemux86 (12:01)	✖ 3 errors ⚠ 10 warnings	Build time: 00:27:45
✔ core-image-sato atom-pc (11:54)	⚠ 4 warnings	Build time: 00:36:55

## All builds

Search builds		Search		Edit columns ▾		Show rows: 10 ▾	
Outcome ▾	Target	Machine	Completed on ▾	Failed tasks ▾	Errors ▾	Warnings ▾	Output
✔	core-image-sato	atom-pc	11/06/13 at 15:22			4 warnings	ext3, hddimg, iso, tar.bz2
✖	core-image-x11	qemux86	11/06/13 at 12:01	acl_2.2.51-r3 do_configure	3 errors	10 warnings	
✔	core-image-sato	atom-pc	11/06/13 at 11:54			4 warnings	ext3, hddimg, iso

# About Toaster

- **Toaster is a web interface to OpenEmbedded and BitBake, the Yocto Project build system.**
- **Toaster allows you configure and run your builds, and provides information and statistics about the build process.**



# toaster in one slide

## 1. Download clone poky git repo:

```
$ mkdir -p ${HOME}/toaster ; cd ${HOME}/toaster  
$ git clone -b krogoth git://git.yoctoproject.org/poky  
# Note: git checkout is required by toaster - do not use the release tarball
```

## 2. Download/install dependencies:

```
$ sudo apt-get install python-virtualenv  
$ virtualenv venv  
$ source venv/bin/activate          # you'll have to activate it every time to enter the environment  
(venv)$ pip install -r poky/bitbake/toaster-requirements.txt
```

## 3. Start toaster:

```
(venv)$ source poky/oe-init-build-env toasterprjdir  
(venv)$ ../poky/bitbake/bin/toaster start      # wait 2 minutes ...
```

## 4. Profit!!! (well... actually there is more work to do...)

```
$ firefox http://localhost:8000
```

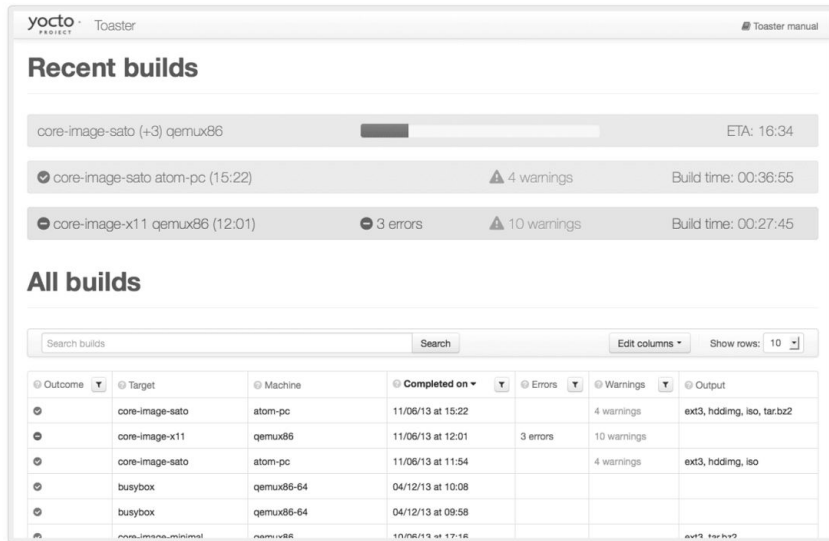
# This is Toaster

A web interface to [OpenEmbedded](#) and [BitBake](#), the [Yocto Project](#) build system.

To start building, create your first Toaster project

[Read the Toaster manual](#)

[Contribute to Toaster](#)



The screenshot displays the Yocto Project Toaster web interface. At the top, there's a header with the 'yocto PROJECT' logo, the word 'Toaster', and a 'Toaster manual' link. Below the header, the 'Recent builds' section shows three build entries with progress bars and status indicators (ETA, warnings, errors, build time). The 'All builds' section features a search bar, a table with columns for Outcome, Target, Machine, Completed on, Errors, Warnings, and Output, and a table with five rows of build data.

### Recent builds

core-image-sato (+3) qemux86	ETA: 16:34
✓ core-image-sato atom-pc (15:22)	4 warnings Build time: 00:36:55
✗ core-image-x11 qemux86 (12:01)	3 errors 10 warnings Build time: 00:27:45

### All builds

Outcome	Target	Machine	Completed on	Errors	Warnings	Output
✓	core-image-sato	atom-pc	11/06/13 at 15:22		4 warnings	ext3, hddimg, iso, tar.bz2
✗	core-image-x11	qemux86	11/06/13 at 12:01	3 errors	10 warnings	
✓	core-image-sato	atom-pc	11/06/13 at 11:54		4 warnings	ext3, hddimg, iso
✓	busybox	qemux86-64	04/12/13 at 10:08			
✓	busybox	qemux86-64	04/12/13 at 09:58			
✗	core-image-minimal	qemux86	10/06/13 at 17:18			ext3, tar.bz2

# Create new build ...

## Create a new project 1)

Project name (required)

Release ?

Toaster will run your builds with the tip of the [Yocto Project 2.0 "Jethro"](#) branch.

Create project

2)

### Machine

Machine changes have a big impact on build configuration. Changing the machine with the previous ones.

Save

Cancel

[View compatible machines](#)

3)

?

Build

**core-image-minimal** [ openembedded-core ]

**core-image-minimal-dev** [ openembedded-core ]

**core-image-minimal-initramfs** [ openembedded-core ]

**core-image-minimal-mtdutils** [ openembedded-core ]

# Toaster demo/walkthrough

- **Main page**
- **Create new project**
- **Select Machine**
- **Add custom layers or recipes**
- **Build an image**
- **Image**
- **Image manifest**

# BUILD AN APPLICATION

**Adding a "hello world" application to our custom image**

# Building an Application

- **General procedure:**
  - ◆ Write hello world application (**hello.c**)
  - ◆ Create recipe for hello world application
  - ◆ Modify image recipe to add hello world application to your image
- **What follows is the example of a simple one C file application**
- (Building a more complicated recipe from a tarball would specify how to find the upstream source with the `SRC_URI`)

# Add Application Code

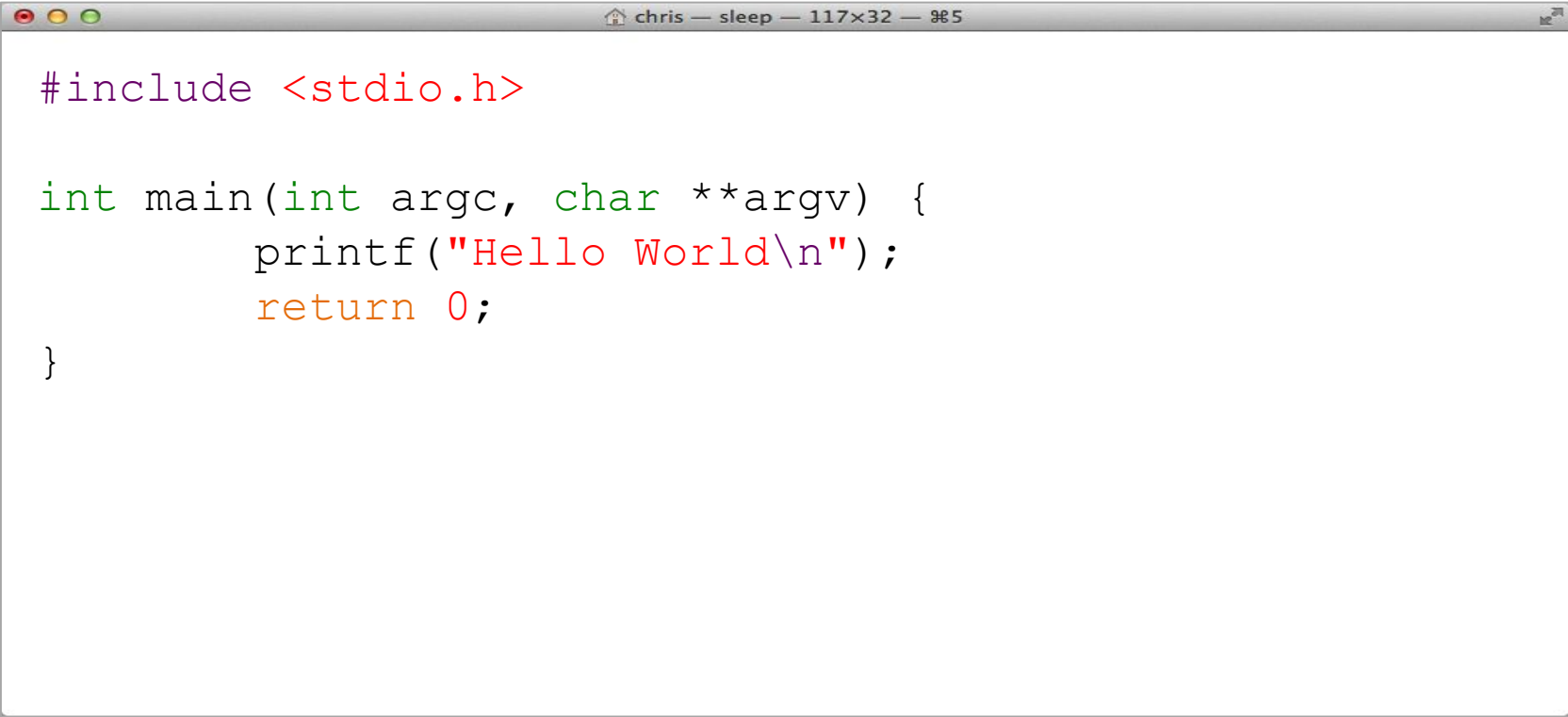
- For a simple one C file package, you can add the hello application source to a directory called *files* in the *hello* package directory

```
$ mkdir -p ${HOME}/yocto/build/meta-ypdd/  
recipes-core/hello/files
```

```
$ vi /scratch/sandbox/meta-ypdd/recipes-core/  
hello/files/hello.c
```

# Application Code

\$ vi /scratch/sandbox/meta-ypdd/recipes-core/hello/files/hello.c



```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello World\n");
    return 0;
}
```



# Add Application Recipe

- Write hello world recipe
- Create directory to hold the recipe and associated files

```
$ mkdir -p ${HOME}/yocto/build/meta-ypdd/\
recipes-core/hello
```

– (We actually did this already in the previous step)

- Create hello\_1.0.bb (next slide)

```
$ vi ${HOME}/yocto/build/meta-ypdd/\
recipes-core/hello/hello_1.0.bb
```

# Application Recipe

\$ vi \${HOME}/yocto/build/meta-ypdd/recipes-core/hello/hello\_1.0.bb

A screenshot of a terminal window with a title bar that reads "chris — sleep — 117x32 — 985". The terminal displays the contents of a Yocto recipe file, which is a BB file. The code is color-coded: blue for variable names, red for string literals, and purple for function names. The code defines a recipe for a "Hello World" example, including a description, license, source URI, and build/install functions.

```
DESCRIPTION = "Hello World example"
LICENSE = "MIT"

LIC_FILES_CHKSUM =
"file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de2
0420"

S = "${WORKDIR}"

SRC_URI = "file://hello.c"

do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} hello.c -o hello
}

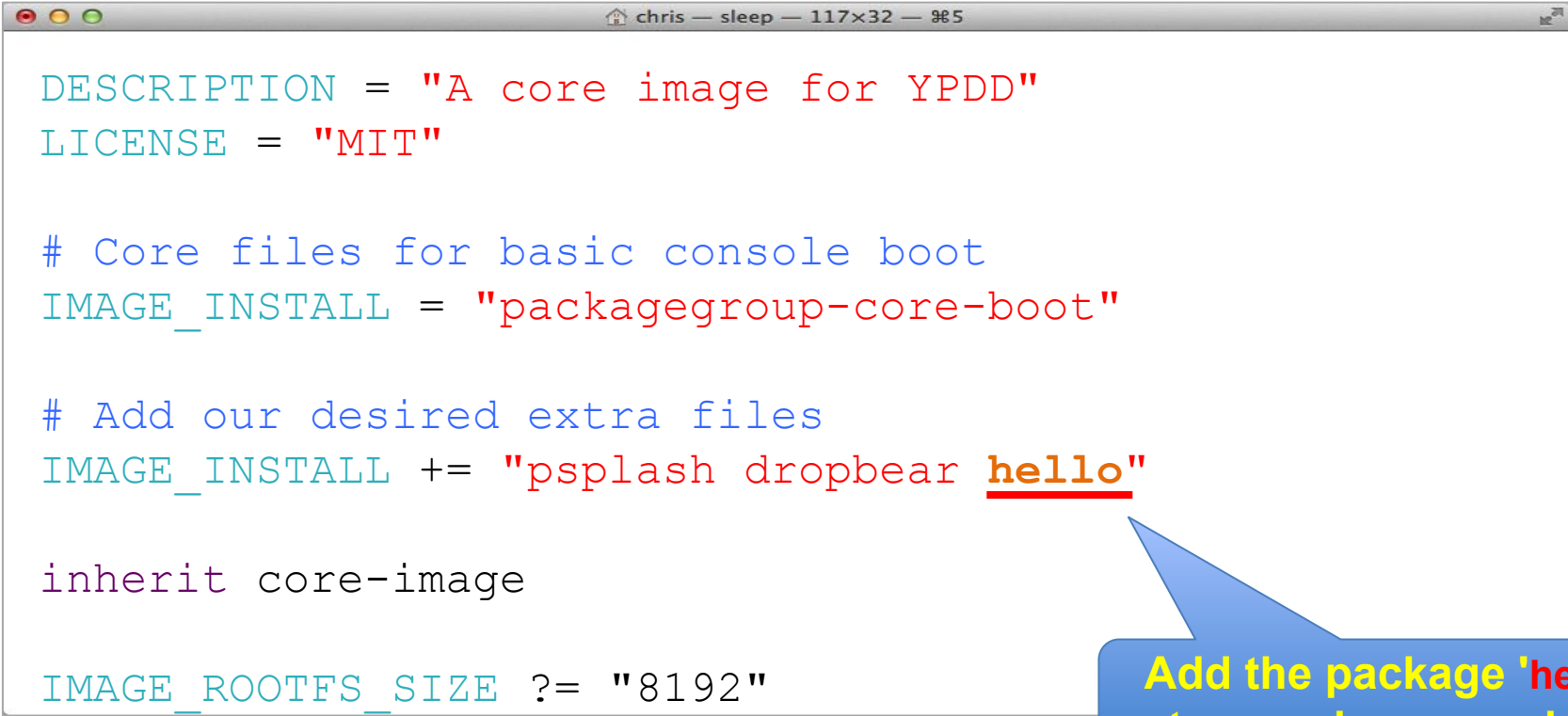
do_install() {
    install -d -m 0755 ${D}/${bindir}
    install -m 0755 hello ${D}/${bindir}/hello
}
```

# Add Application to the Image

- **Modify image recipe to add hello world application to your image**
- **See example on next slide**

# Add hello to Image

```
$ vi ${HOME}/yocto/build/meta-ypdd/recipes-core/images/ypdd-image.bb
```



```
DESCRIPTION = "A core image for YPDD"
LICENSE = "MIT"

# Core files for basic console boot
IMAGE_INSTALL = "packagegroup-core-boot"

# Add our desired extra files
IMAGE_INSTALL += "psplash dropbear hello"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
```

Add the package 'hello'  
to your image recipe

# Build and Test Application

- Now (re)build your image recipe

\$ **bitbake ypdd-image**

◆ `hello_1.0.bb` will be processed because it is in your custom layer, and referenced in your image recipe.

- Boot your image using **runqemu**, as before:

\$ **runqemu qemuarm tmp/deploy/images/\  
qemuarm/ypdd-image-qemuarm.ext4 nographic**

- You should be able to type "**hello**" at the command line and see "Hello World"

*It's **not** an Embedded  
Linux Distribution*

*It Creates a      ●  
Custom One For You*



# Embedded Linux Development with Yocto Project

## Training from The Linux Foundation

Want to learn how to use Yocto Project like a Pro?

<https://training.linuxfoundation.org/>

Embedded Linux Platform Development with Yocto Project

<http://bit.ly/elgyocto>

# TIPS HINTS AND OTHER RESOURCES

The following slides contain reference material that will help you climb the Yocto Project learning curve



# Common Gotchas When Getting Started

- Working behind a network proxy? Please follow this guide:
  - [https://wiki.yoctoproject.org/wiki/Working\\_Behind\\_a\\_Network\\_Proxy](https://wiki.yoctoproject.org/wiki/Working_Behind_a_Network_Proxy)
- Do not try to re-use the same shell environment when moving between copies of the build system
- `oe-init-build-env` script appends to your `$PATH`, it's results are cumulative and can cause unpredictable build errors
- Do not try to share `sstate-cache` between hosts running different Linux distros even if they say it works

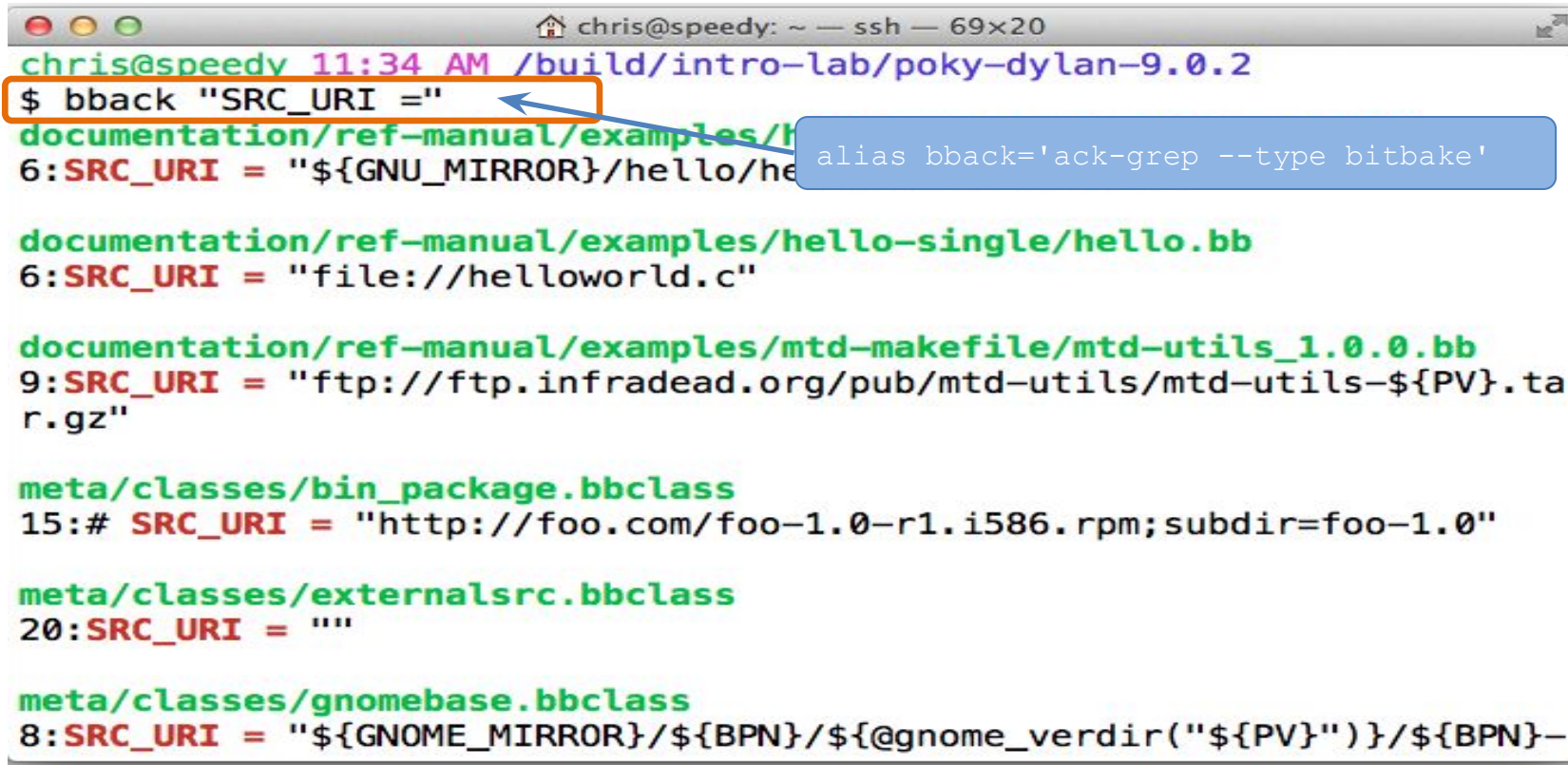
# Project Resources

- The Yocto Project is an open source project, and aims to deliver an open standard for the embedded Linux community and industry
- Development is done in the open through public mailing lists: `openembedded-core@lists.openembedded.org`, `poky@yoctoproject.org`, and [yocto@yoctoproject.org](mailto:yocto@yoctoproject.org)
- And public code repositories:
- <http://git.yoctoproject.org> and
- <http://git.openembedded.org>
- Bug reports and feature requests
- <http://bugzilla.yoctoproject.org>

# Tip: ack-grep

- **Much faster than grep for the relevant use cases**
- **Designed for code search**
- **Searches only relevant files**
  - ◆ Knows about many types: C, asm, perl
  - ◆ By default, skips .git, .svn, etc.
  - ◆ Can be taught arbitrary types
- **Perfect for searching metadata**

# Tip: ack-grep



```
chris@speedy 11:34 AM /build/intro-lab/poky-dylan-9.0.2
$ bback "SRC_URI ="
documentation/ref-manual/examples/hello-single/hello.bb
6:SRC_URI = "${GNU_MIRROR}/hello/hello.c"

documentation/ref-manual/examples/mtd-makefile/mtd-utils_1.0.0.bb
9:SRC_URI = "ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-${PV}.tar.gz"

meta/classes/bin_package.bbclass
15:# SRC_URI = "http://foo.com/foo-1.0-r1.i586.rpm;subdir=foo-1.0"

meta/classes/externalsrc.bbclass
20:SRC_URI = ""

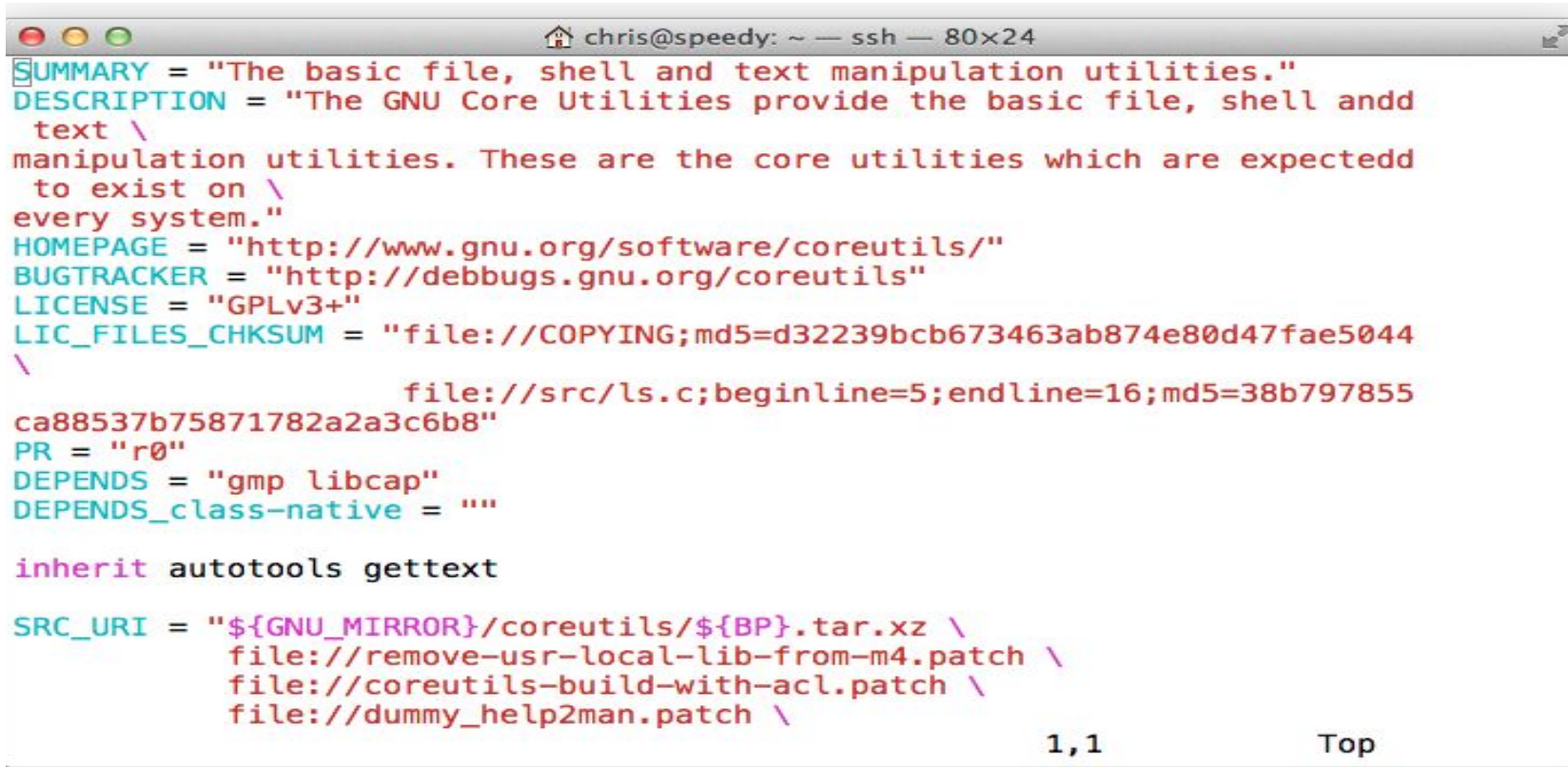
meta/classes/gnomebase.bbclass
8:SRC_URI = "${GNOME_MIRROR}/${BPN}/${@gnome_verdir("${PV}")}/${BPN}-
```

# TIP: VIM Syntax Highlighting

- <https://github.com/openembedded/bitbake/tree/master/contrib/vim>
- Install files from the above repo in ~/.vim/
- Add "syntax on" in ~/.vimrc

```
$ tree ~/.vim/  
/Users/chris/.vim/  
├── ftdetect  
│   └── bitbake.vim  
├── ftplugin  
│   └── bitbake.vim  
├── plugin  
│   └── newbb.vim  
└── syntax  
    └── bitbake.vim
```

# TIP: VIM Syntax Highlighting



```
SUMMARY = "The basic file, shell and text manipulation utilities."
DESCRIPTION = "The GNU Core Utilities provide the basic file, shell and
text \
manipulation utilities. These are the core utilities which are expectedd
to exist on \
every system."
HOMEPAGE = "http://www.gnu.org/software/coreutils/"
BUGTRACKER = "http://debbugs.gnu.org/coreutils"
LICENSE = "GPLv3+"
LIC_FILES_CHKSUM = "file://COPYING;md5=d32239bcb673463ab874e80d47fae5044
\
file://src/ls.c;beginline=5;endline=16;md5=38b797855
ca88537b75871782a2a3c6b8"
PR = "r0"
DEPENDS = "gmp libcap"
DEPENDS_class-native = ""

inherit autotools gettext

SRC_URI = "${GNU_MIRROR}/coreutils/${BP}.tar.xz \
file://remove-usr-local-lib-from-m4.patch \
file://coreutils-build-with-acl.patch \
file://dummy_help2man.patch \
```

1,1 Top

# Lab:   yocto for the          minnowboard

The following section introduces the minnowboard as example hardware. Other boards like the Beaglebone are supported in a similar manner.

# minnowboard in one Slide

## 1. Download poky tool:

```
$ mkdir -p ${HOME}/myproject  
$ cd ${HOME}/myproject  
$ wget -nd -c "http://downloads.yoctoproject.org/releases/yocto/yocto-2.2/poky-morty-16.0.0.tar.bz2"  
$ tar -xf poky-morty-16.0.0.tar.bz2
```

## 2. Download meta-intel layer:

```
$ wget -nd -c \  
"http://downloads.yoctoproject.org/releases/yocto/yocto-2.2/machines/intel-corei7-64/intel-corei7-64-6.0-morty-2.2.tar.bz2"  
$ tar -xf intel-corei7-64-6.0-morty-2.2.tar.bz2
```

## 3. Configure:\$

```
$ source poky-morty-16.0.0/oe-init-build-env mybuild  
$ echo "BBLAYERS += \"$(pwd)/../intel-corei7-64-6.0-morty-2.2 \" " >> conf/bblayers.conf  
$ echo 'MACHINE = "intel-corei7-64" ' >> conf/local.conf
```

## 4. Build the image:

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

## 5. Profit!!! (well... actually there is more work to do...)



# the minnowboard

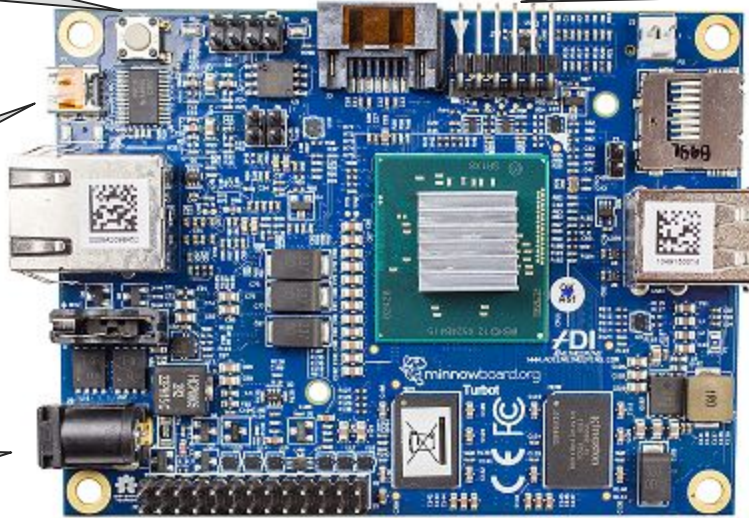
reset switch

Serial port header

HDMI

SD-Card slot

Power connector



# SD-card installation

- `$ cd tmp/deploy/images/intel-corei7-64/`
- `$ dd if=core-image-minimal-intel-corei7-64.hddimg of=/dev/<YourSDCard> bs=1M`  
or
- `$ dd if=core-image-minimal-intel-corei7-64.wic of=/dev/<YourSDCard> bs=1M`
- Connect to the serial and boot.

# Hands-on Lab



Wire-up your minnow



Use the ready-made SD-Cards



Connect to the serial and boot



Extra labs:

- Check-out the board with a hello-world.c (compiled on the board itself)
- Homework: to it right and build hello-world.c in a recipe, include it in your image and boot that.

# End

**HAVE FUN** and thank you for joining !

