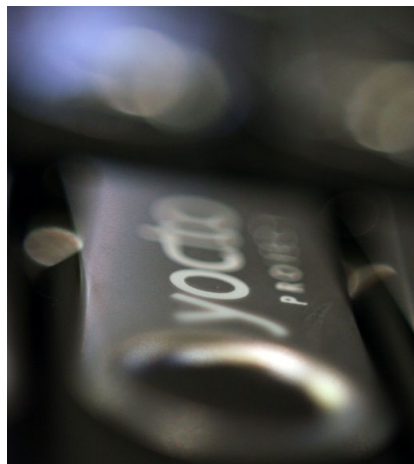


# E-ALE SCaLE 16X



## Learn Bitbake with Yocto Project



Tom King  
John Bonesio  
The Linux Foundation  
Mar 15, 2018  
(CC BY-SA 4.0)

The URL for this presentation

<https://goo.gl/d5SXS8>

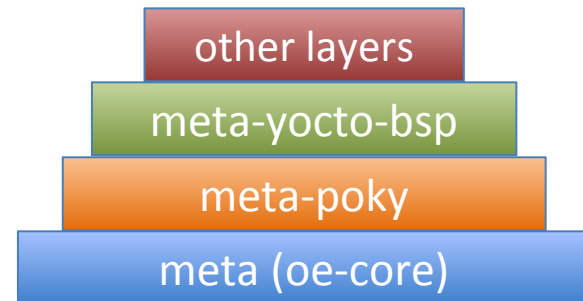


# Yocto/Openembedded Terms

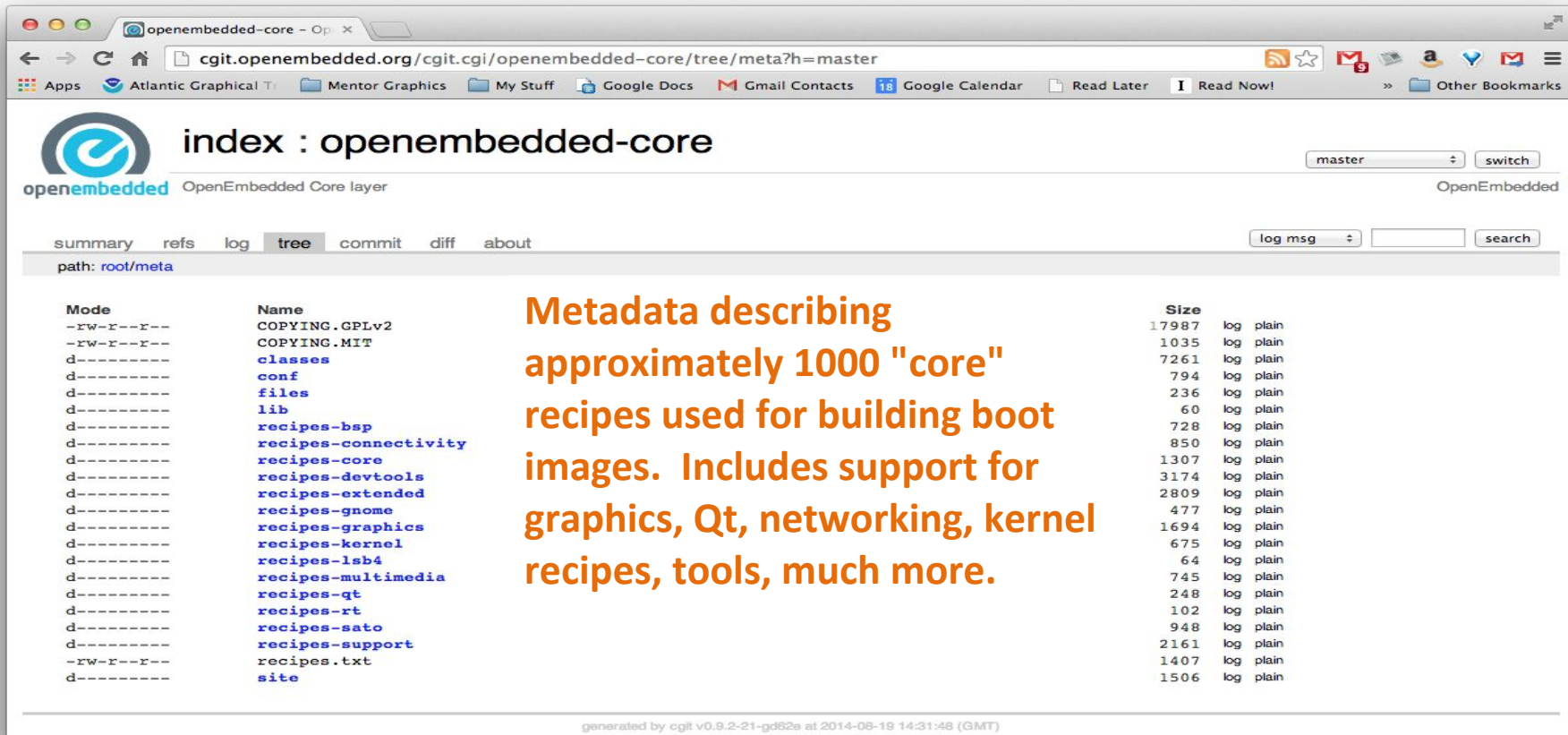
- **Yocto Project** is a large collaboration project
- **OpenEmbedded** is providing most metadata
- **Bitbake** is the build tool
- **Poky** is the Yocto Project's reference distribution
  - Poky contains a version of bitbake and oe-core from which you can start your project

# 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**



# Yocto is based on OpenEmbedded-core



The screenshot shows the OpenEmbedded-core Git repository tree view. The browser address bar displays `cgит.openembedded.org/cgit.cgi/openembedded-core/tree/meta?h=master`. The page title is "index : openembedded-core" and the sub-page is "OpenEmbedded Core layer". The "tree" tab is selected, showing a list of files and directories under the path `root/meta`. The list includes files like `COPYING.GPLv2`, `COPYING.MIT`, `classes`, `conf`, `files`, `lib`, and various recipe directories such as `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`. A table on the right shows the size of each file in bytes and its type (log or plain).

Mode	Name	Size	Type
-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 cgit v0.9.2-21-gd82e 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.

# What is Bitbake?

## ➤ Bitbake

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

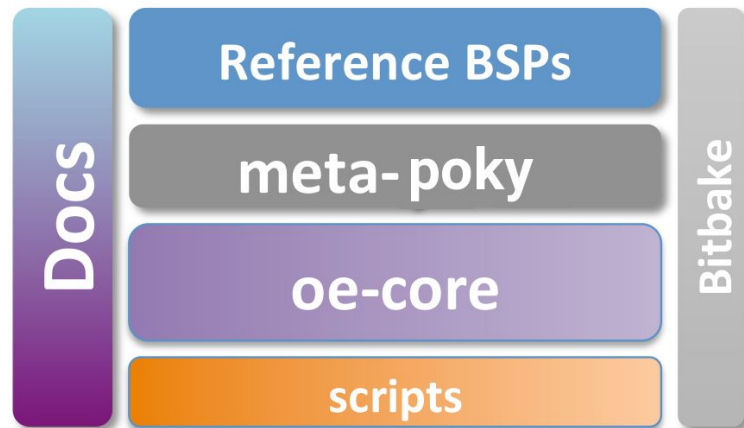


**Metadata** – a structured collection of "recipes" which tell BitBake what to build, organized in layers

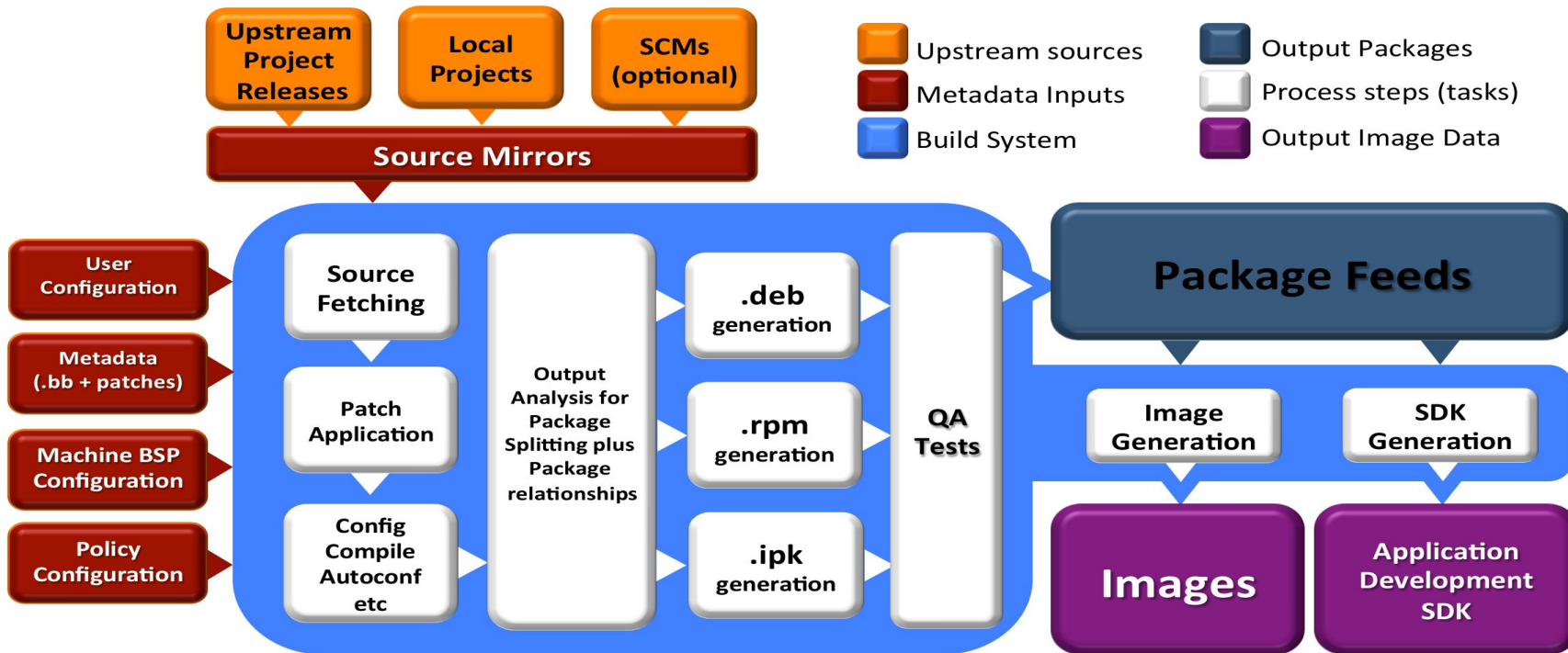
# Poky Reference Distro

## ➤ 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



# Build System Workflow



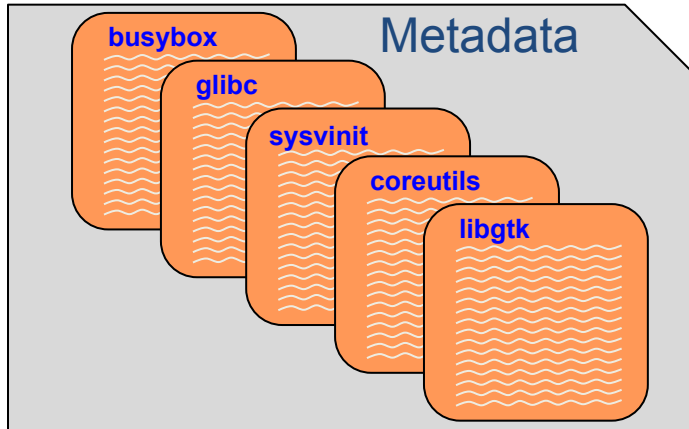


# BITBAKE

**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(\*.bb files)**
- 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

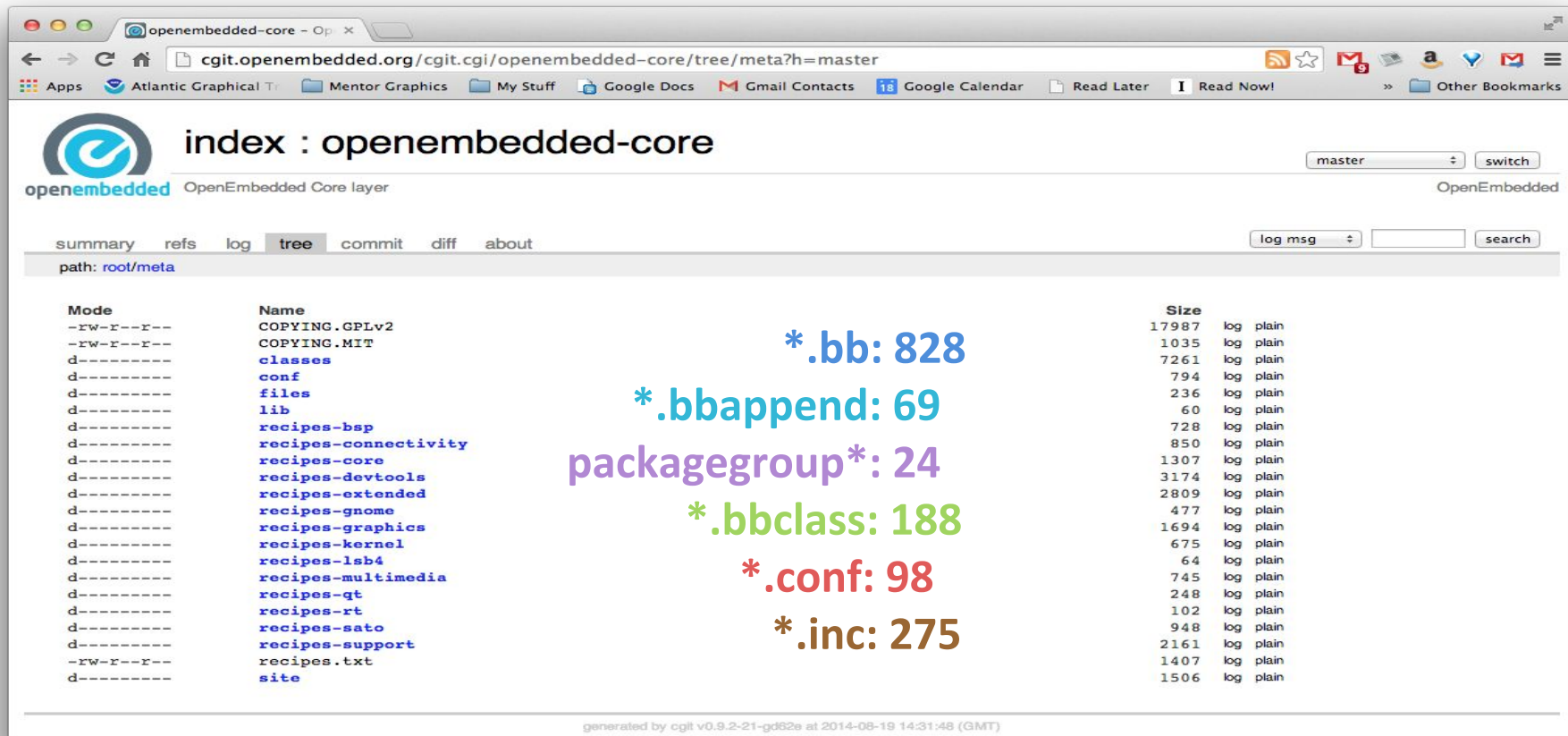
## ➤ **Append files (\*.bbappend)**

- ◆ 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 typical found via the `BBPATH` variable

# OE-CORE Breakdown



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

index : openembedded-core

OpenEmbedded Core layer

summary refs log **tree** commit diff about

path: root/meta

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

**\*.bb: 828**

**\*.bbappend: 69**

**packagegroup\*: 24**

**\*.bbclass: 188**

**\*.conf: 98**

**\*.inc: 275**

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**
  - \$ **bitbake -c cleanall myrecipe**
- You can get a list of tasks with
  - \$ **bitbake -c listtasks myrecipe**

# 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

# 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
```



# Building Images

- 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...*

# Running Bitbake: 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

# Running Bitbake: Task Log

```
chris — sleep — 117x32 — 965
$ 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\*

```
chris — sleep — 117x32 — 5
$ 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.
```

\*Simplified for illustration

# 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=c19b30548c582577
                    fc6b443626fc1216"

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] = "562e3cc675cf5b1ac655cd060f032943a2502d4d59e5f278f02aae9256
2ba261"

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



# What can a Recipe Do?

- **Build software from source code:**
  - ◆ Host tools, compiler, utilities
  - ◆ Target Bootloader, Kernel, etc
  - ◆ Target Libraries, interpreters, etc
  - ◆ Target Userspace applications
- **Package Groups**
- **Full System Images**

# 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"

HOME PAGE = "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"

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_1.3.2.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=09501c864f89dfc7ead65553129817ca \  
file://COPYING.GPL;md5=b234ee4d69f5fce4486a80fdaf4a4263 \  
file://src/flac/main.c;beginline=1;endline=18;md5=09777e2934947a36f13568d0beb81199 \  
file://COPYING.LGPL;md5=fbc093901857fcd118f065f900982c24 \  
file://src/plugin\_common/all.h;beginline=1;endline=18;md5=f56cb4ba9a3bc9ec6102e8df03215271 \  
file://COPYING.Xiph;md5=b59c1b6d7fc0fb7965f821a3d36505e3 \  
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] = "454f1bfa3f93cc708098d7890d0499bd"

SRC\_URI[sha256sum] = "91cfc3ed61dc40f47f050a109b08610667d73477af6ef36dcad31c31a4a8d53f"

(con't next page)

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

(con't from previous page)

```
CVE_PRODUCT = "libflac"
```

```
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_1.19.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 \
"
```

```
SRC_URI[md5sum] = "a5f8803ace110511b6ff5a2b39782e8b"
```

```
SRC_URI[sha256sum] =
```

```
"a0e09bdd8b53b8d2e4b54f1863ecd9aebe4786477a6cbf8f655496e8edb31c81"
```

```
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.filter('DISTRO_FEATURES', '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
```

```
    # Ofono still has one test tool that refers to Python 2 in the shebang
```

```
    sed -i -e '1s,#!.*python.*,#!${bindir}/python3,' ${D}${libdir}/ofono/test/set-ddr
```

```
}
```

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

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

```
RRECOMMENDS_${PN} += "kernel-module-tun mobile-broadband-provider-info"
```

```
FILES_${PN} += "${systemd_unitdir}"
```

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

```
RDEPENDS_${PN}-tests = "python3 python3-pyobject python3-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_package_data: 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-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

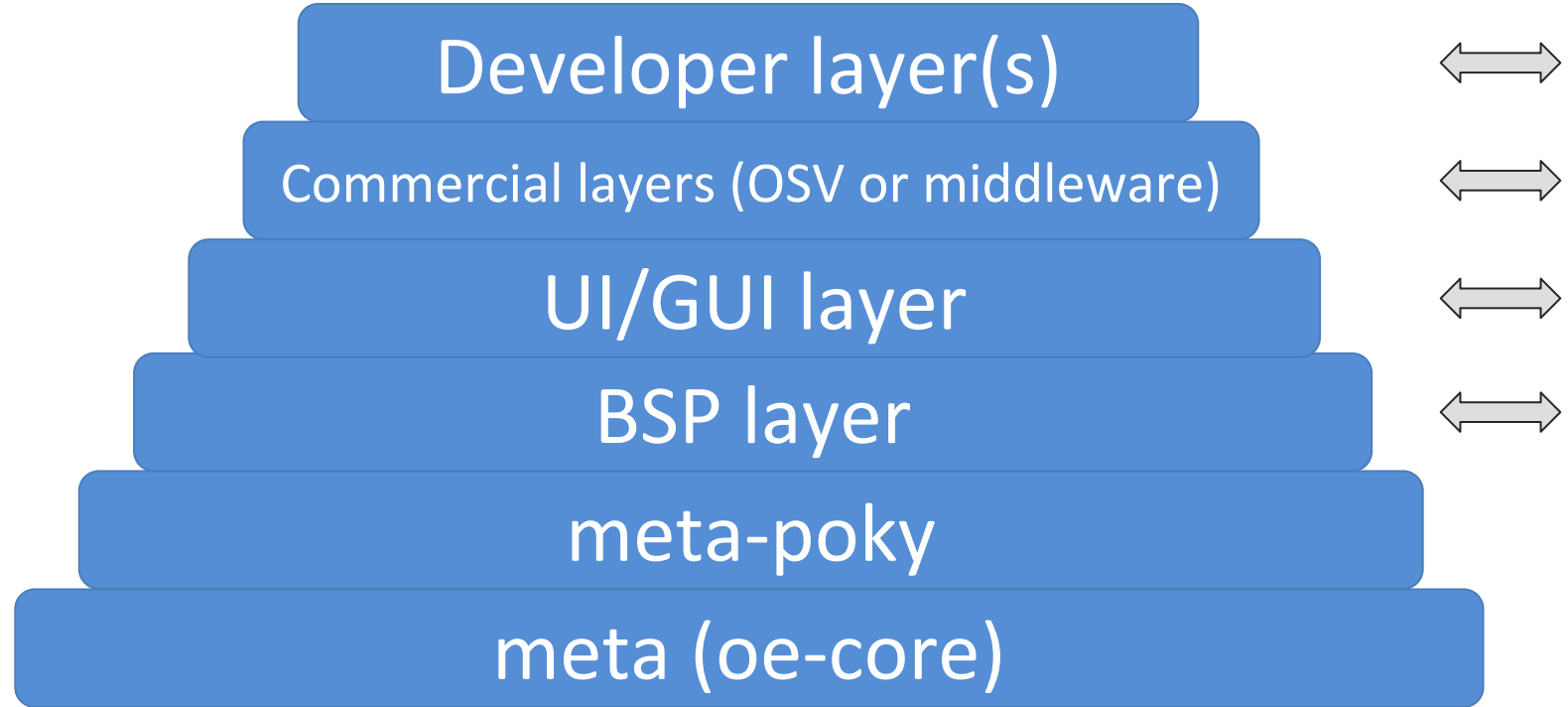
# LAYERS

**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



# 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
- <http://layers.openembedded.org> <<---- This site is your friend



# 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

# 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"
```

LAB

# **BUILDING A FULL EMBEDDED IMAGE WITH Bitbake**

# Workflow

## 1. Download Yocto Project sources:

```
$ mkdir e-ale ; cd e-ale
```

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-2.3.2/poky-pyro-17.0.2.tar.bz2
```

```
$ tar xf poky-pyro-17.0.2.tar.bz2
```

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

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

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

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

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

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

## 3. Run the image under emulation:

```
$ runqemu qemuarm
```

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



# Host System Layout

`$HOME/e-ale/`

`|---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(reference) Layout

\$HOME/e-ale/poky/

| --- LICENSE

| --- README

| --- README.hardware

| --- [bitbake/](#)

(The build tool)

| --- [documentation/](#)

| --- [meta/](#)

(oe-core)

| --- [meta-poky/](#)

(Yocto distro metadata)

| --- [meta-yocto-bsp/](#)

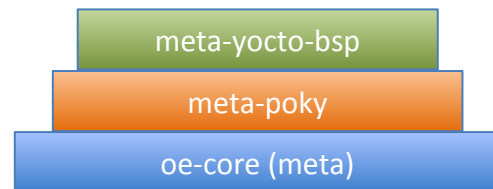
(Yocto Reference BSPs)

| --- [oe-init-build-env](#)

(Project setup script)

| --- [scripts/](#)

(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/e-ale/
```

```
$ 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/e-ale/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*

# Configuring Layers

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

`$HOME/e-ale/build/conf/bblayers.conf`

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

# Configure local.conf

## ➤ local.conf setup:

- ◆ Configure build by editing local.conf
- ◆ `$HOME/e-ale/build/conf/local.conf`
  - Select appropriate MACHINE type (target)
  - Set shared downloads directory (DL\_DIR)
  - Set shared state directory (SSTATE\_DIR)

```
MACHINE = "qemuarm"
```

```
DL_DIR ?= "$HOME/e-ale/DOWNLOADS"
```

```
SSTATE_DIR ?= "$HOME/e-ale/SSTATE"
```

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

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

# 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**



# 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.

# 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

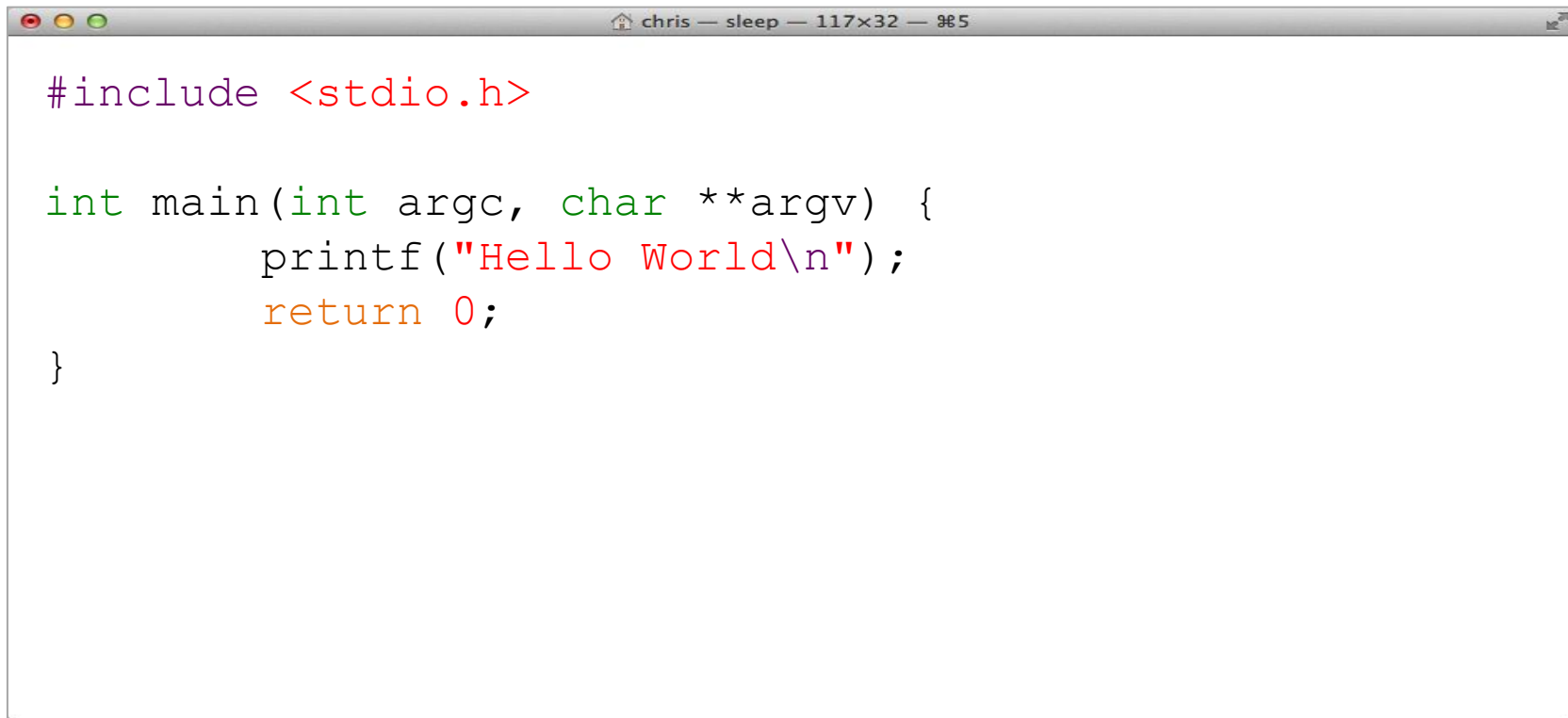
- For a simple one C file package, you can add the hello application source to a directory called *files* in the *hello* package directory

```
$ mkdir -p ${HOME}/e-ale/hello-recipe/files
```

```
$ vi ${HOME}/e-ale/hello-recipe/files/hello.c
```

# Application Code

\$ vi hello-recipe/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}/e-ale/hello-recipe
```

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

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

```
$ vi  
${HOME}/e-ale/hello-recipe/hello_1.0.bb
```

# Application Recipe

```
$ vi ${HOME}/e-ale/hello-recipe/hello_1.0.bb
```

A screenshot of a terminal window with a title bar that reads "chris — sleep — 117x32 — 965". The terminal displays the content of a BitBake recipe file named "hello\_1.0.bb". The code is color-coded: blue for variable names, red for string values, and purple for function names. The code defines variables for DESCRIPTION, LICENSE, LIC\_FILES\_CHKSUM, S, and SRC\_URI, and includes two functions: do\_compile and do\_install.

```
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
}
```

# 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-single/hello.bb

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}-
```

alias bback='ack-grep --type bitbake'

# 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

```
chris@speedy: ~ — ssh — 80x24
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
```

# The other boards ...



# dragonboard in one Slide

The dragonboard uses a 3rd-party repository.

## 1. Download repo tool:

```
$ mkdir -p ${HOME}/bin  
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ${HOME}/bin/repo  
$ chmod a+x ${HOME}/bin/repo  
$ export PATH=${HOME}/bin:${PATH}
```

## 2. Download the repositories with repo:

```
$ mkdir oe-qcom && cd oe-qcom  
$ repo init -u https://github.com/96boards/oe-rpb-manifest.git -b pyro  
$ repo sync  
$ source setup-environment      # SELECT the dragonboard
```

## 3. Build the image:

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

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

# YP for the beaglebone

The following section introduces the beaglebone as example hardware.

# PocketBeagle in one Slide

## 1. Download poky tool:

```
$ mkdir -p ${HOME}/myproject  
$ cd ${HOME}/myproject  
$ wget -nd -c "http://downloads.yoctoproject.org/releases/yocto/yocto-2.3.2/poky-pyro-17.0.2.tar.bz2"  
$ tar -xf poky-pyro-17.0.2.tar.bz2
```

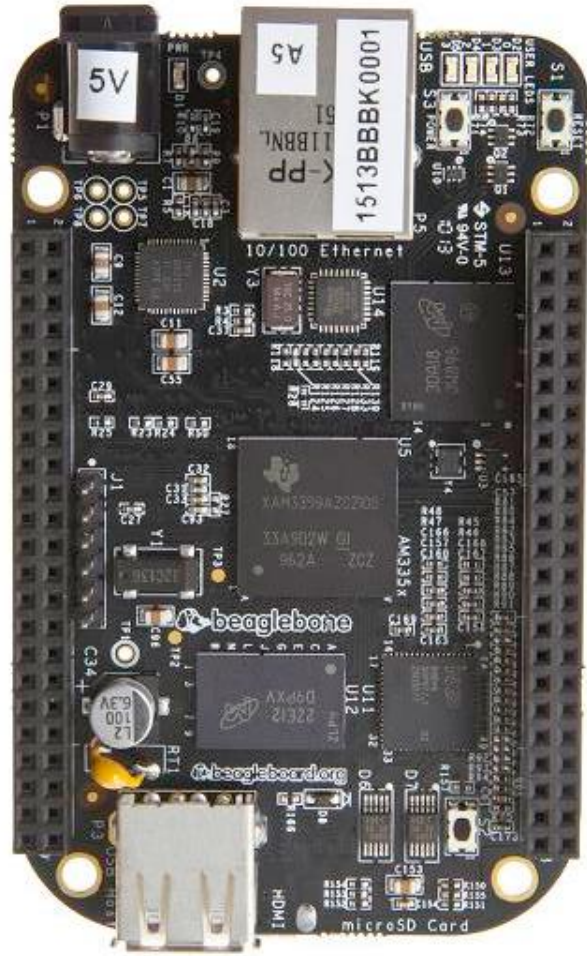
## 2. Configure:\$

```
$ source poky-morty-17.0.2/oe-init-build-env mybuild  
$ echo 'MACHINE = "beaglebone" ' >> conf/local.conf
```

## 3. Build the image:

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

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



# End

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

# BITBAKE REFERENCE

# Recipe Operators

<b>A = "foo"</b>	<b>(late assignment)</b>
<b>B ?= "0t"</b>	<b>(default value)</b>
<b>C ??= "abc"</b>	<b>(late default)</b>
<b>D := "xyz"</b>	<b>(Immediate assignment)</b>

<b>A .= "bar"</b>	<b>→</b>	<b>"foobar"</b>	<b>(append)</b>
<b>B =. "WO"</b>	<b>→</b>	<b>"W00t"</b>	<b>(prepend)</b>
<b>C += "def"</b>	<b>→</b>	<b>"abc def"</b>	<b>(append)</b>
<b>D =+ "uvw"</b>	<b>→</b>	<b>"uvw xyz"</b>	<b>(prepend)</b>

# 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
  - ◆ **HOME PAGE** – 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