Embedded Linux Systems with the Yocto Project

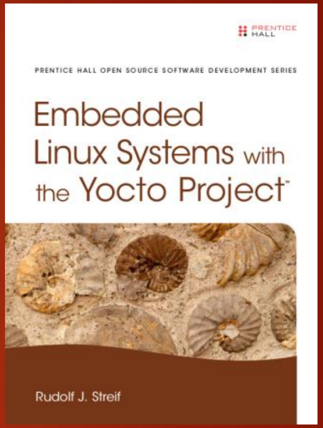
COURSE MATERIAL

Audience

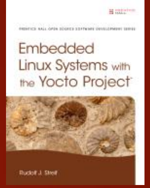
- You will have the best experience and will benefit the most from this course if
  - You have a fairly good understanding of Linux from using it. However, you do not need to have any prior background in Linux system architecture, Linux kernel development, or embedded Linux.
  - You have experience using the C programming language, how programs are compiled and built.
  - You have a good understanding of system programming in a UNIX or Linux environment on the application or user-space level.
  - You know how to use text editors, either command line (vi, emacs, etc.) or with a graphical UI.
  - You have basic knowledge in UNIX shell scripting and Python programming.

References

- Embedded Linux Systems with the Yocto Project, Rudolf J. Streif, Prentice Hall, https://www.pearsonhighered.com/program/Streif-Embedded-Linux-Systems-with-the-Yocto-Project/PGM275649.html
- Yocto Project Quick Start Manual, http://www.yoctoproject.org/docs/2.1/yocto-project-qs/yocto-project-qs.html
- Yocto Project Reference Manual, http://www.yoctoproject.org/docs/2.1/ref-manual/ref-manual.html
- Yocto Project Linux Kernel Development Manual, http://www.yoctoproject.org/docs/2.1/kernel-dev/kernel-dev.html
- OpenEmbedded Wiki, http://openembedded.org/wiki/Main_Page


PRENTICE HALL
PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES
Embedded Linux Systems with the Yocto Project
Rudolf J. Streif

Conventions



4

Page

Attention – something can go wrong here.

Tip – this can make your work easier.

Further reading and more details.

Content

The Yocto Project Ecosystem

WHAT IT IS, WHO THE PARTICIPANTS ARE, AND WHY YOU SHOULD CARE…

yocto PROJECT

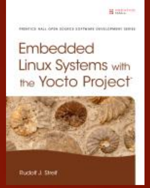
The Yocto Project is not an Embedded Linux Distribution.

It creates a custom one for You!

The Yocto Project is not a Single Open Source Project.

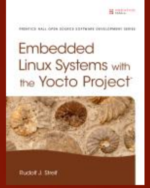
It is an Ecosystem.

The Yocto Project combines the convenience of a ready-to-run Linux distribution with the flexibility of a custom Linux operating system stack.



Why Linux for Embedded Systems?

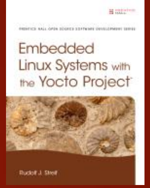
- The Case for Linux
  - Royalty-free
  - Hardware Support
  - Networking Support and Protocols
  - Modularity
  - Scalability
  - Source Code
  - Developer Support
  - Commercial Support
  - Tooling
- The Case against Linux
  - General Purpose OS
  - File System
  - Memory Management Unit
  - Not a real-time OS


Embedded Linux Systems with the Yocto Project

3, 4

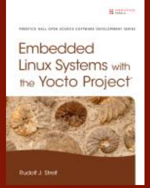
Embedded Linux Distributions

- Android - http://source.android.com
  - Excellent for systems with ARM-based SoCs and touch screens
  - Includes build system and development tools
- Angstrom Distribution - www.angstrom-distribution.org
  - Community distribution with a growing list of supported development boards
  - Yocto Project build system
- OpenWrt - www.openwrt.org
  - Debuted as open source OS for embedded devices routing network traffic
  - Originally created from the Linksys GPL sources for their WRT54G residential gateway
  - Buildroot-based build system
  - Headless operation with web UI



Embedded Versions of Mainstream Linux Distributions

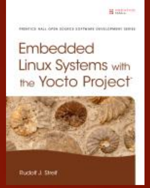
- Debian - www.emdebian.org - inactive
- Fedora - https://fedoraproject.org/wiki/Embedded - inactive
- Gentoo - https://wiki.gentoo.org/wiki/Project:Embedded - inactive
- SuSE - https://tr.opensuse.org/MicroSUSE - inactive
- Ubuntu – www.ubuntu.com/Internet-of-things
  - Ubuntu Core maintained by Canonical
  - Snappy application development platform


Embedded Linux Systems with the Yocto Project

5, 6

Embedded Linux Development Tools

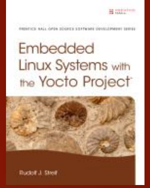
- Baserock - http://wiki.baserock.org
  - Git server at the core to manage build instructions, source code and build artifacts
  - Native compliation – no cross-build support
  - YAML-based build instructions
- Buildroot - https://buildroot.org
  - Make as the build engine – build instructions are makefiles
  - uClibc target library
- OpenEmbedded - www.openembedded.org
- The Yocto Project - www.yoctoproject.org



8, 9

Embedded Linux – Why is it Challenging?

- Dependency Management – Software packages depend on each other and on libraries requiring compatibility of APIs and dynamically linked libraries.
- Conflicting Software Packages – Multiple software packages provide the same functionality often with the same APIs but cannot be installed on a system at the same time.
- Toolchain – Bootstrapping a toolchain with C/C++ compiler, assembler, linker, and many other tools, eventually for cross building, is difficult.
- Kernel and Device Drivers – Linux kernel configuration has about 7,000 parameters to enable functionality and drivers.



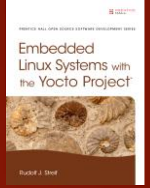
8,9

Top-down or Bottom-up?

- Top-down
  - Start with a maintained and tested Linux distribution
  - Modify it by installing additional packages, removing unneeded packages, configuring system settings
  - Install proprietary software
  - Create an image and install it on the target hardware
- Bottom-up
  - Build the entire system from source code
  - Select toolchain, bootloader, kernel, C library and other packages
  - Install only the required software packages

Jump Start

Flexibility / Control



8,9

Top-down or Bottom-up?

- Top-down
  - Start with a maintained and tested Linux distribution
  - Modify it by installing additional packages, removing unneeded packages, configuring system settings
  - Install proprietary software
  - Create an image and install it on the target hardware
- Bottom-up
  - Build the entire system from source code
  - Select toolchain, bootloader, kernel, C library and other packages
  - Install only the required software packages

Jump Start

Flexibility / Control

Can we have both?

What is the Yocto Project?

- An open source project providing
  - A build system for building custom Linux distributions from source to deployable image,
  - Blueprints for distributions and root file systems to jump-start development while maintaining flexibility for full customization,
  - Software Development Toolkits (SDK) and emulators (QEMU) that can be integrated with common Integrated Development Environments such as Eclipse and Qt Creator for roundrip application development,
  - Comprehensive documentation,
  - Graphical UI tools for managing dependencies and customizing root file systems,
  - Frameworks for continuous integration, automated test and building.

Who is the Yocto Project

Member Organizations

- Texas Instruments
- ENEA group
- WIND – An Intel Company
- Mentor Graphics
- Huawei
- Linaro
- Xilinx
- MontaVista
- openembedded
- Broadcom
- AMD
- Renesas
- LG Electronics
- intel
- O.S. Systems
- Juniper Networks
- freescale semiconductor
- Dell
- timesys – Embedded Linux from a Trusted Source

Administrative Project Leadership

Supporting Organizations

- Qt
- MiS
- fluentd
- gumstix – dream, design, deliver
- NetLogic Microsystems
- Panasonic
- Silica – An Avnet Company
- Ångström
- Sidebranch
- Tilera
- Mindspeed – Build it first
- Toradex – Embedded. Computing.
- CableLabs
- SDG Systems
- KOAN – embedded software engineering
- GENIVI
- DOULOS Embedded
- chargestorm
- GL research
- Free Electrons – Embedded Linux Developers
- Q technology
- Eukrea électromatique
- EXPEMB 360 – Embedded Experts
- TechNexion
- Cavium
- Secret Lab
- RidgeRun
- SanDisk
- HTP – monkey-project.com
- move innovation
- BEC Systems
- ALICO Systems Incorporated
- dynamicdevices
- net Module
- denx
- wolfSSL
- Linutronix
- minnowboard.org

Project Contributions

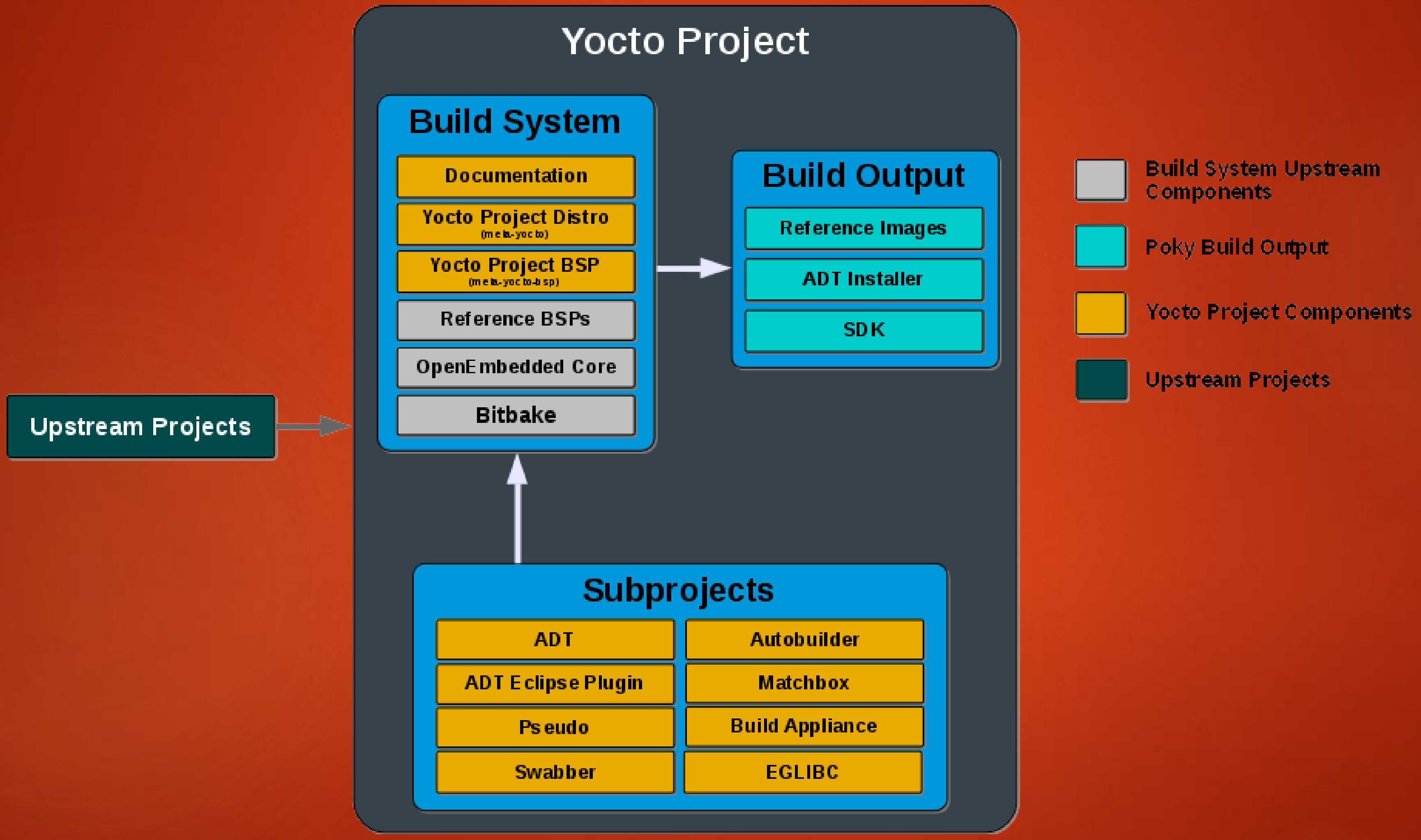
For Developers

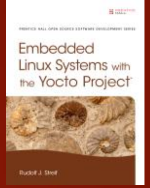
- Build a complete Linux system from source in about 60 Minutes (90 Minutes with X Window support)
- Start with a validated collection of software – toolchain, bootloader, kernel, user space packages
- Root file system image blueprints
  - Get you started quickly
  - Can be customized any way you need it
- Support for *System Developers* and *Application Developers*
  - Tools to create your own layers, application recipes, kernel recipes and BSP
  - Kernel configuration and development tools
  - SDKs for use on the command line or to integrate with Eclipse or Qt Creator, remote on target debugging and more
- Support for all major architectures – x86, x86-64, ARM, MIPS, PPC
- Active and friendly support community

For Organizations and Companies

- Industry ecosystem
- Backed and managed by the Linux Foundation
- High-availability continuous integration and automated test infrastructure
- Continuous maintenance with two major releases per year (April and October)
- Support from consulting companies
- Board Support Packages (BSP) for CPUs and SoCs from all major semiconductor vendors

Yocto Project Components

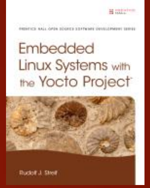




Meet the Yocto Project Family of Projects

- OpenEmbedded Build System – Reference Build System
- BitBake – Build Engine
- OpenEmbedded Core – Base Meta-data Layer (meta)
- Poky – Yocto Project Reference Distribution (meta-poky)
- Yocto Project BSP – Reference BSP (meta-yocto-bsp)
- Application Development Toolkit (ADT) – Development environment (SDK) for user-space application to run on OS stacks built by the build system.
- Eclipse IDE Plugin – Integrates ADT/SDK with Eclipse IDE
- Autobuilder – Build automation and continuous integration based on Buildbot
- Build Appliance – Virtual machine images for trials
- Pseudo – System administration simulation environment
- Swabber – Host contamination detection
- Toaster – Web-bases graphical user interface

Yocto Project & OpenEmbedded



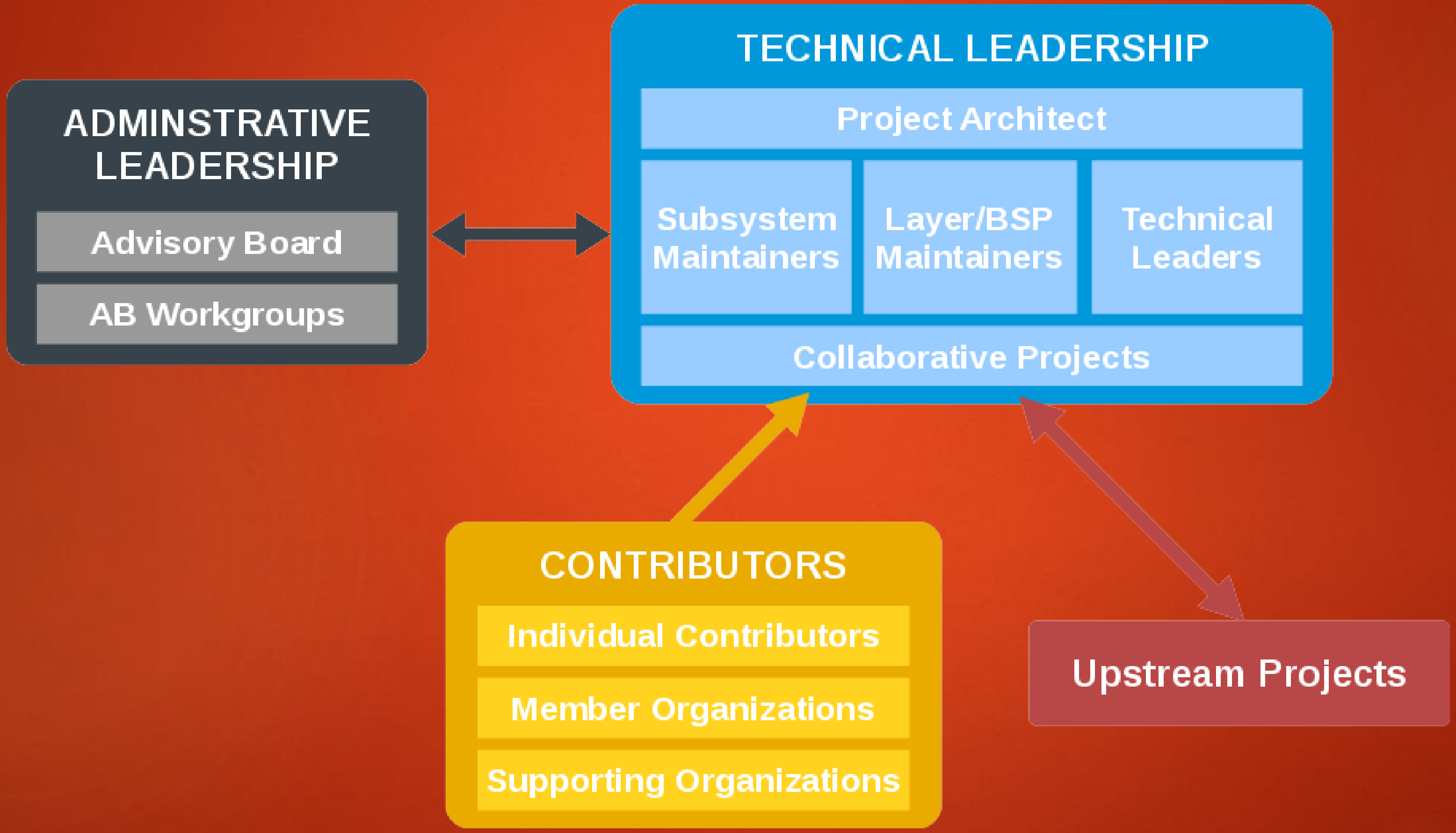
18-30

- OpenEmbedded
  - Created by merging the work of the OpenZaurus project with contributions from other projects including Familiar Linux, OpenSIMpad, etc. into a common code base.
  - Community project focused on cutting edge technology, latest hardware and broad architecture support.
  - Large library of recipes for thousands of open source software packages.
- Yocto Project
  - Self-contained build system providing tools and blueprints for Linux OS stacks.
  - Tested integration of OpenEmbedded build system with tools, reference distribution and best practices.
  - Regular release cadence (twice a year).
  - Dedicated staff (from member companies) for development, maintenance, continuous integration and quality assurance.
  - Standardized components with compliance program.

Why not just use OpenEmbedded?

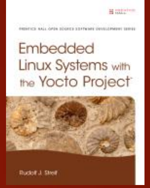
- OpenEmbedded is an open source project providing a build framework for Linux systems:
  - Designed as a foundation,
  - Cutting-edge technologies and software packages,
  - Rapid development cycle.
- The Yocto Project is focused on enabling commercial product development:
  - Provides reference distribution policies and root file system blueprints,
  - Co-maintains OpenEmbedded components with dedicated staff and improves their quality,
  - Adds tooling such as Autobuilder and QA tests,
  - Provides tools for system and application development such as devtool, ADT/SDK, Eclipse plugin, etc.

Yocto Project Developer Community



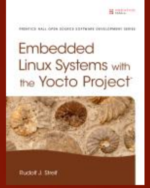
Getting Started

ALL YOU NEED TO KNOW TO GET YOUR FEET WET AND A LITTLE BEYOND…



Build Host Requirements

- Hardware
  - CPU - x86-64, multiple cores/hyperthreading, build system automatically parallelizes, hence the more cores the faster the build
  - Memory - at least 4 GB (16 GB or more recommended)
  - Storage - at least 100 GB available storage, SSD preferred for performance, RAID levels supporting striping increase performance
- Software
  - Mainstream Linux distribution – these are regularly tested by the Yocto Project development team and officially supported: CentOS, Debian, Fedora, OpenSuSE, Ubuntu
  - Git version 1.8.3.1 or greater, tar version 1.24 or greater, Python version 2.7.3 or greater (excluding Python 3.x)
  - Additional packages dependent on the chosen distribution (details follow)
- Infrastructure
  - High-speed Internet Access



Build Host Software Setup

- CentOS

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch
  diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath socat perl-
  Data-Dumper perl-Text-ParseWords perl-Thread-Queue SDL-devel xterm
```

- Debian / Ubuntu

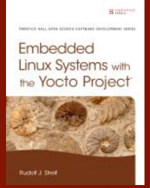
```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
  build-essential chrpath socat libsdl1.2-dev xterm
```

- Fedora

```
$ sudo dnf install gawk make wget tar bzip2 gzip python unzip perl patch diffutils
  diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath ccache perl-Data-Dumper
  perl-Text-ParseWords perl-Thread-Queue perl-bignum socat findutils which SDL-
  devel xterm
```

- OpenSuSE

```
$ sudo zypper install python gcc gcc-c++ git chrpath make wget python-xml diffstat
  makeinfo python-curses patch socat libSDL-devel xterm
```



Obtaining the Yocto Project Tools

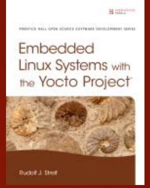
- Download the current release (or previous release) tarball from the Yocto Project Website:
  - https://www.yoctoproject.org/downloads
- Clone from the Yocto Project Git repository:
  - Preferred as it gives you the bleeding edge master development branch as well as the release branches.
  - Master branch:

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

  - Release branch:

```
$ git clone -b <branch> git://git.yoctoproject.org/poky.git
```

Initializing the Build Environment

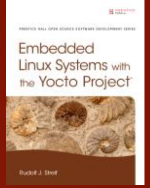


- Create and initialize a build environment:
  - `source <pokypath>/oe-init-build-env <builddir>`
  - `<pokypath>` is the directory where you installed the build tools
  - `<builddir>` is the name of directory where your build environment will be set up in, default is `build`
- Every time you want to use a build environment you have to initialize it with the above command.

The script `oe-init-build-env` must be “sourced” to export the variable settings to the current shell.

Add <pokypath> to your .bashrc file:

```
# Yocto Project Setup
YOCTODIR="${HOME}/yocto"
POKYDIR="${YOCTODIR}/poky"
export YOCTODIR POKYDIR
PATH="${POKYDIR}:${PATH}"
```

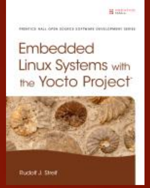


Configuring the Build Environment

- The file con/local.conf configures your build environment by setting various variables.
- The file is automatically created and populated with default settings.
- The most commonly adjusted settings are:
  - `MACHINE ?= "qemux86" # configure the target platform (machine)`
  - `DL_DIR ?= "${TOPDIR}/downloads" # where to place the source downloads`
  -
    ```
    SSTATE_DIR ?= "${TOPDIR}/sstate-cache" # where to place the shared
                                           # cache files
    ```

The variable `TOPDIR` references the top-level directory if the build environment.

Launching the Build



- From top-level directory of your build environment run:

```
$ bitbake <build-target>
```

- To build a default image target:

```
$ bitbake core-image-sato
```

- To download all the source files without building:

```
$ bitbake -c fetchall core-image-sato
```

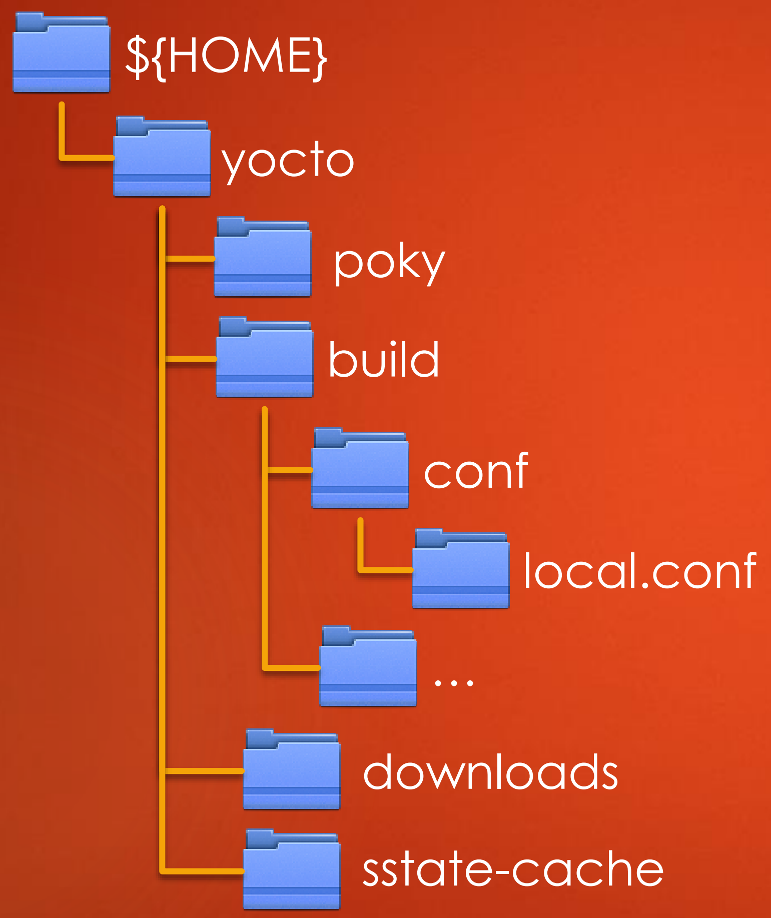
The build process stops at the first error encountered. To continue building tasks that are not impeded by the error, use the –k option:

```
$ bitbake -k core-image-sato
```

Lab Exercise

- Prepare your build host by installing the required packages according to the Linux distribution.
- Setup the `.bashrc` file with the variables `YOCTODIR` and `POKYDIR`. Source the file for the changes to become effective.
- Create the `yocto` directory in your home directory. Change into that directory.
- Clone the `krogoth` branch of the build system from the Yocto Project repository.
- Create a build environment.
- Edit the `conf/local.conf` file:
  - Build for the machine `qemux86-64`.
  - Place the downloads into `${TOPDIR}/../downloads`.
  - Place the shared state cache files into `${TOPDIR}/../sstate-cache`.
- Build the target image `core-image-minimal`.

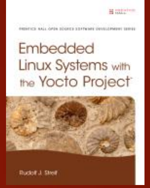
Directory Structure



Inside the Build System

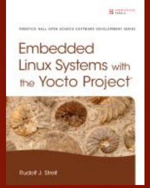
A CLOSE LOOK AT HOW THE BUILD SYSTEM WORKS

Build System Terms (1/6)



31-33

- Append File - An append file extends an existing recipe. BitBake verbatim appends the contents of an append file to the corresponding recipe, creating a single file before parsing it. Variables in an append file can override the same variables defined in the corresponding recipe. Append files use the bbappend extension.
- BitBake - The build engine in the OpenEmbedded build system, BitBake is a task executor and scheduler. Its input are metadata files such as configuration files and recipes through which BitBake processing is controlled.
- Board Support Package (BSP) - Documentation, binaries, code, and other implementation-specific support data in the BSP enable a given operating system to run on a particular target hardware platform. Sometimes a BSP also contains complete root file systems and a cross-development environment to create application programs running on the target hardware platform.

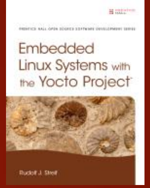


31-33

Build System Terms (2/6)

- Class - In BitBake terminology, a class is a metadata file providing logic encapsulation and a basic inheritance mechanism allowing commonly used patterns to be defined once and used with many recipes. BitBake class files use the bbclass extension.
- Configuration File - Configuration files are BitBake metadata files providing global definition and settings for variables that affect the build process.
- Cross-development Toolchain - A cross-development toolchain is a collection of software development tools allowing software development for target systems employing a different architecture than the development host. Architecture in this context refers to different CPU instruction sets (for instance, ARM, MIPS, PowerPC, x86) as well as different bit sizes (for instance 8, 16, 32, and 64 bit). Typically, a cross-development toolchain includes one or more language compilers, assembler, linker and often debuggers, emulators, and other tools that are specific to the target architecture.

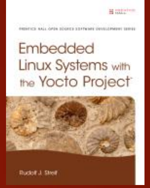
Build System Terms (3/6)



31-33

- Image – A binary file, often compressed, an image contains a boot loader, an operating system kernel, and a root file system to be copied to a storage media from which the target system can boot and run. The term image is also used to mean just an operating system kernel (kernel image) or the root file system (root file system image).
- Layer – In BitBake terminology, a layer is a collection of metadata (configuration files, recipes, etc.) organized into a file and directory structure. BitBake can include layers to extend its functionality. Yocto Project BSPs are provided as layers.
- Metadata – In BitBake terminology, metadata includes all files that instruct BitBake how to execute build processes. BitBake metadata includes classes, recipes (with append files), and configuration files.

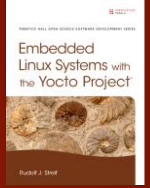
Build System Terms (4/6)



31-33

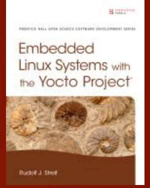
- OpenEmbedded Core (OE Core) – A core set of metadata in the OpenEmbedded build system that is shared between OpenEmbedded and the Yocto Project, OE Core is a BitBake layer co-maintained by the OpenEmbedded Project and the Yocto Project.
- Package –A package is a software bundle containing executable binaries, libraries, documentation, configuration information, and other files following a specific format that an operating system's package management system can install or uninstall. Packages commonly also include information on dependencies on and incompatibilities with other software packages that the package management system can use to automatically resolve and/or inform the user about.
  The Yocto Project also uses the term package to mean the recipes and other metadata used to build the respective software bundle. Dependent on the context, the term then refers either to the actual software bundle or to the metadata that builds the software bundle.

Build System Terms (5/6)



31-33

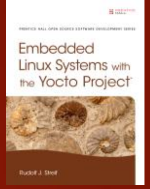
- Package Management System – A package management system is a collection of software tools automating the process of installing, upgrading, configuring, and removing software packages for a computer's operating system. It typically maintains a database of the installed software on the computer, including version information, dependencies, incompatibilities, and more, to prevent system faults through software mismatches and missing prerequisites.
- Poky – Poky is the Yocto Project's reference distribution.
- Recipe – A recipe is a metadata file containing directives for BitBake on how to build a particular software package. Through its directives, a recipe describes from where to obtain the source code, what patches to apply and how to apply them, how to build the binaries and associated files, how to install the build results on a target system, how to create the packaged software bundle, and much more. Recipes also describe dependencies during build and runtime on other software packages, hence creating a logical hierarchy of the pieces required for the build process. Recipes use the bb file extension.



31-33

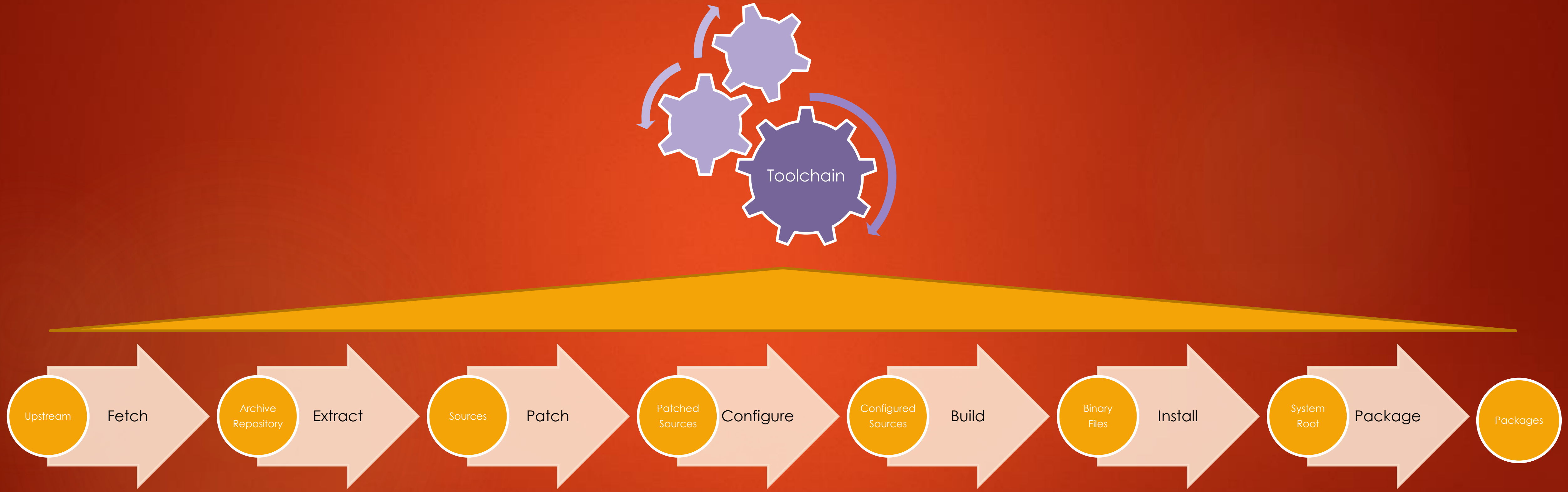
Build System Terms (6/6)

- Task – BitBake recipes may contain executable metadata, or code, that BitBake executes during the build process. Execution steps can be grouped into metadata functions. A metadata function can be declared as a task by inserting it into the BitBake task list.
- Upstream – In software development, particularly in open source, upstream references the direction to the originators, that is, the original authors or maintainers, of the software. Commonly, the term is used as a qualifier, such as upstream repository and upstream patch.
- YP Core – The build system including BitBake, OpenEmbedded Core, reference distribution and BSP layers as well as integration scripts.

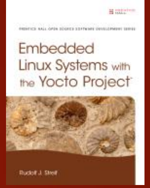


35-39

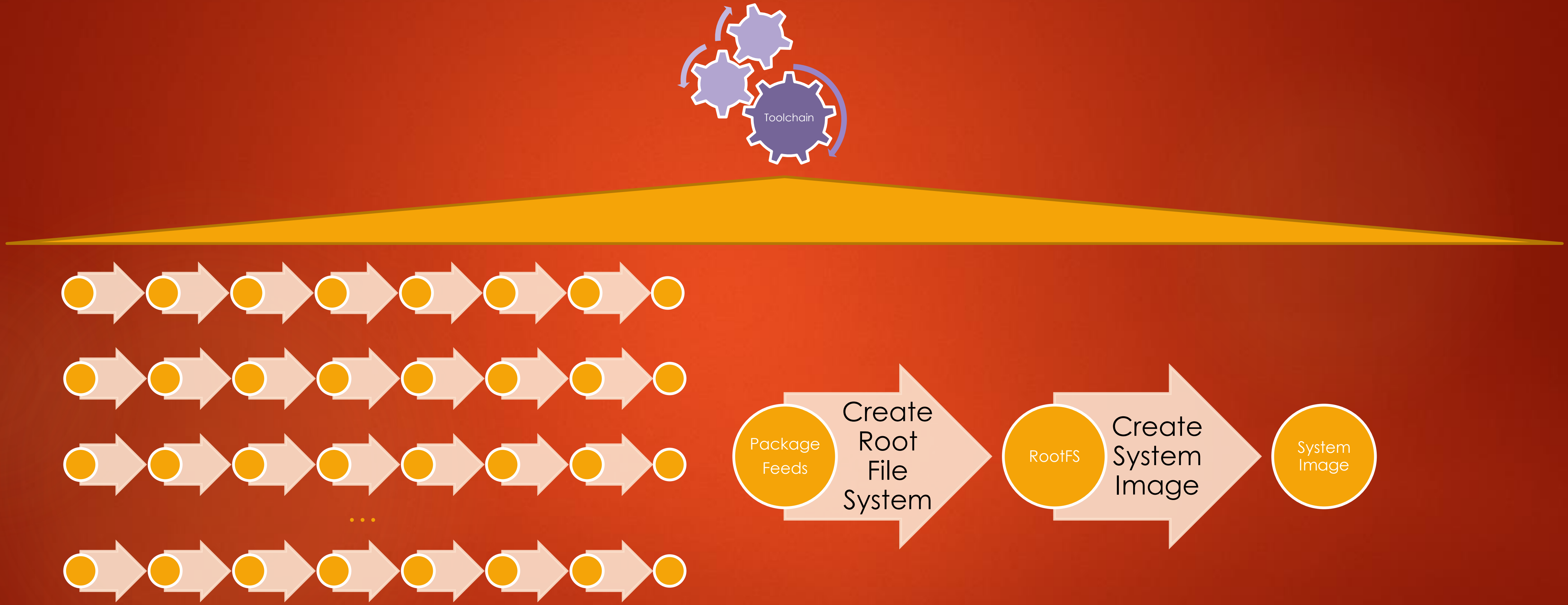
Building Software Packages



From Source to System Image



35-39



Package Builds

Image Assembly

Toolchain

- Tools used by the build system during the build process such as compilers (C, C++, Java, assembler, linker, archiver, compressor/decompressor, SCM tools, etc.
- The OpenEmbedded build system boostraps the entire toolchain from source:
  - Build host toolchain is only used for bootstrapping.
  - Avoid dependency on host toolchain.
  - Ensure that toolchain is compatible with software packages in particular the Linux kernel and the C library.
- Cross Toolchain
  - Tools that run on one architecture but create output for another, for example gcc compiler running on x86-64 creating ARM assembly code.

System Root

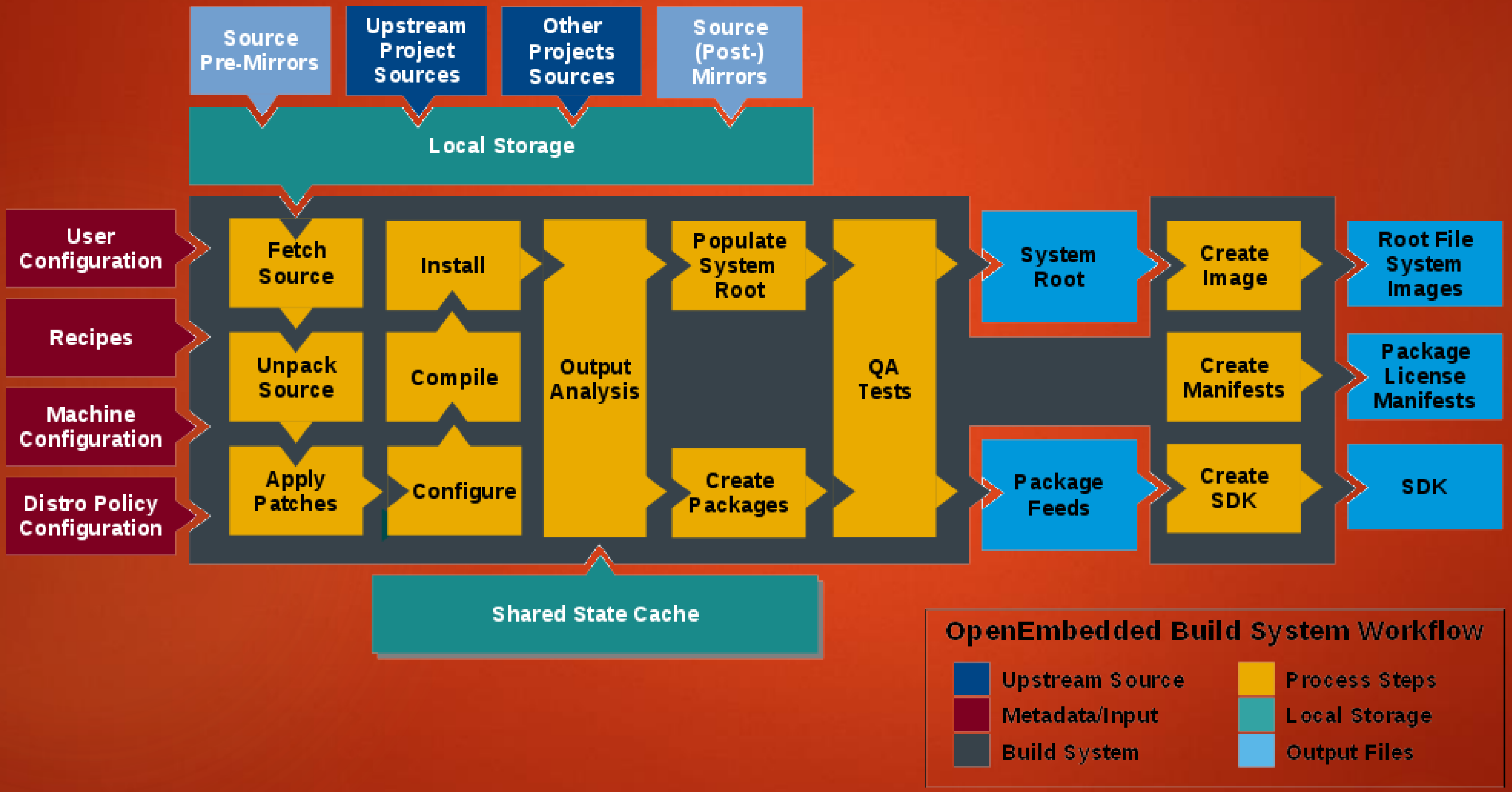
- System root on UNIX/Linux systems is /.
- Paths in makefiles and other build scripts reference the system root of the build host with absolute paths:

```
includedir = /usr/include
libdir = /usr/lib
bindir = /usr/bin
```

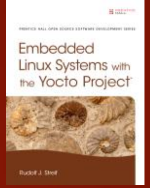
- Building a system for a different target than the build host requires setting the system root to a different directory on the build host.
- Tools such as gcc and other support setting an alternative system root using the `--sysroot` flag.

⚠ (Build) Host Contamination:
Makefiles and other scripts referencing paths must be written correctly to support alternative system roots. Otherwise, files from the build host will be used. Best case this will produce a build error, worst case it will cause hard to track runtime errors.

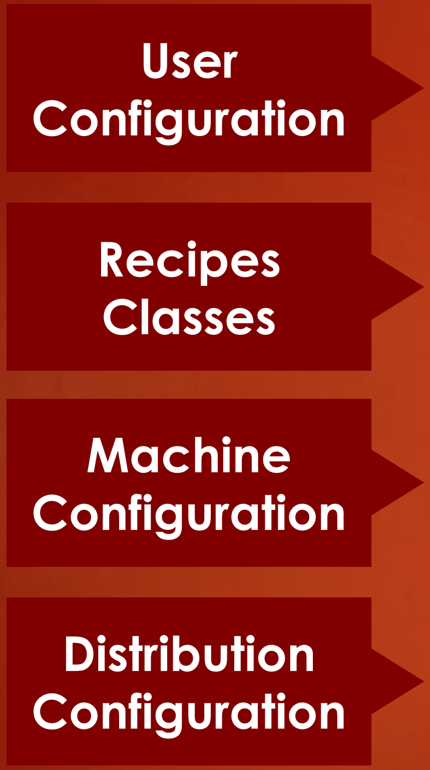
OpenEmbedded Build System Workflow



Meta-data Files

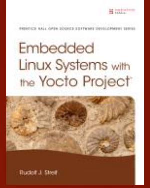


41-43



- Configuration Files
  - Global build system configuration through variable assignments
  - Maintained by BitBake in a data dictionary that is accessible by other meta-data files
- BitBake Master Configuration File (`meta/conf/bitbake.conf`)
- Layer Configuration File (`${LAYERDIR}/conf/layer.conf`)
- Build Environment Layer Configuration (`${TOPDIR}/conf/bblayers.conf`)
- Build Environment Configuration (`${TOPDIR}/conf/local.conf`)
- Distribution Configuration (`<distribution-name>.conf`)
- Machine Configuration (`<machine-name>.conf`)
- Build Instructions (Recipes and Classes)

User Configuration



41-43

User Configuration

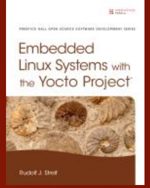
Recipes Classes

Machine Configuration

Distribution Configuration

- Build Environment Configuration `(${TOPDIR}/conf/local.conf)` – define what you are building:
  - Target `MACHINE ?= "qemux86-64"`
  - Target `DISTRO ?= "poky"`
  - Download Directory `DL_DIR ?= "${TOPDIR}/../downloads"`
  - Shared Cache Directory `SSTATE_DIR ?= "${TOPDIR}/../sstate-cache"`
  - License Configuration `INCOMPATIBLE_LICENSE ?= "GPLv3"`
  - Package Management Systems `PACKAGE_CLASSES ?= "package_rpm"`
  - Image Features `EXTRA_IMAGE_FEATURES ?= "debug-tweaks dev-pkgs"`
- Build Environment Layer Configuration `(${TOPDIR}/conf/bblayers.conf)`
  - Include additional layers into your build environment

Recipes and Classes



41-43

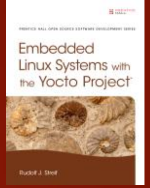
User Configuration

Recipes Classes

Machine Configuration

Distribution Configuration

- Build Instructions
  - Variable assignments and tasks scripts
  - Recipes build individual software packages: meta/recipes-core/busybox_1.24.1.bb
  - Classes contain build instructions that are shared between recipes: meta/classes/kernel.bbclass
- Recipes inherit the system configuration and adjust it to describe how to build and package the software.
- Recipes can be extended and enhanced through append-files from other layers.
- When necessary, recipes can apply local patches and configuration files for the software package such as kernel .config.



41-43

Machine Configuration

User Configuration

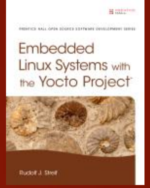
Recipes Classes

Machine Configuration

Distribution Configuration

- Machine Descriptions
  - Define target-specific kernel and bootloader configuration
  - Describe formfactor (display, mouse, touch, keyboard) configuration
  - Parameters for architecture-specific CPU/SoC tuning
- Machine configuration files are part of BSP layers. BSP layers provide any hardware-specific recipes and/or extensions to existing recipes.
- Machine configuration determines hardware-specific parameters for the kernel, bootloaders, graphics environment.
- BSP layers can be included with the build environment allowing to build the same OS stack for different machines.
- Selected from the build environment configuration by setting the `MACHINE` variable.

Distribution Configuration



User Configuration

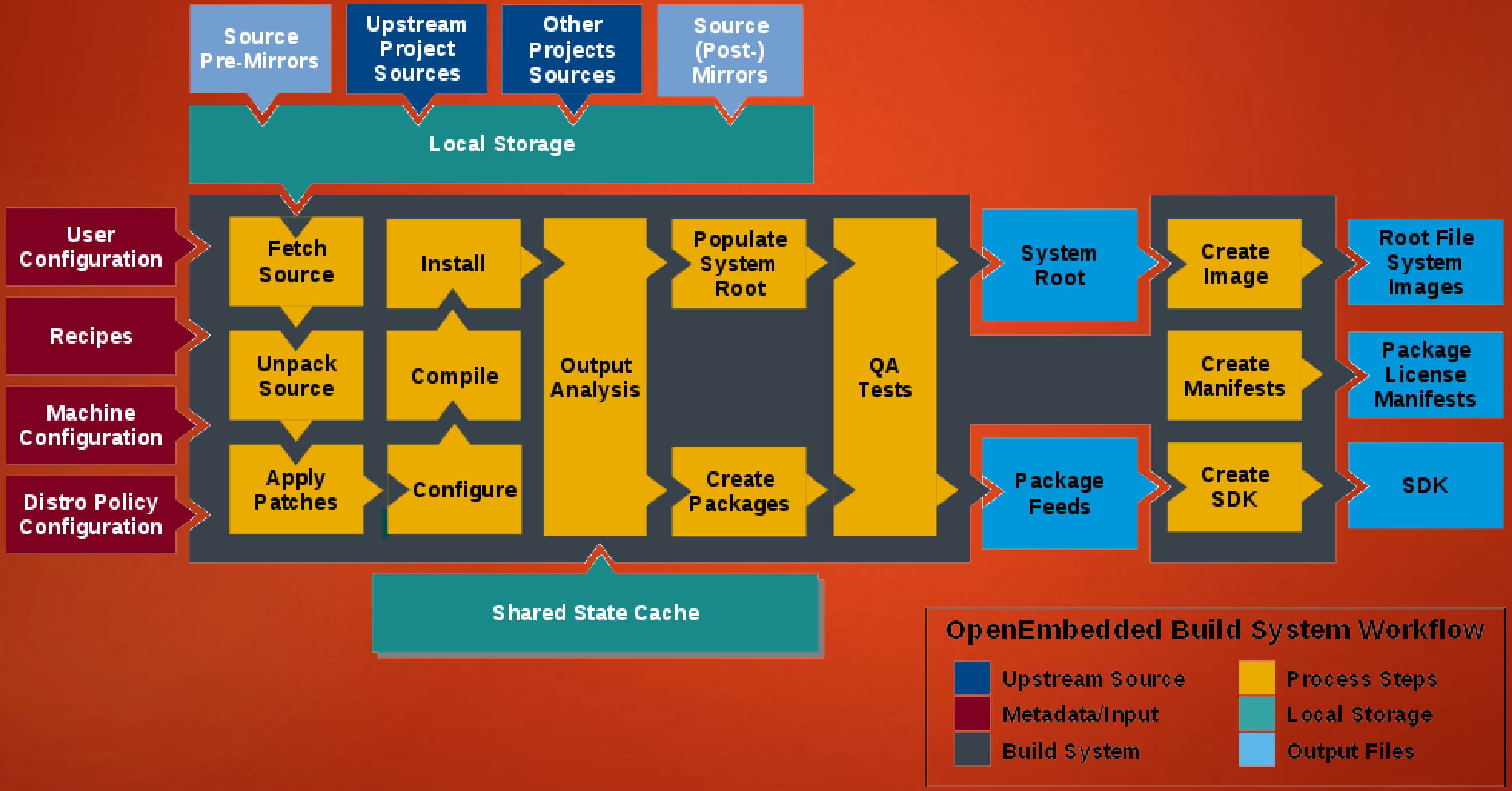
Recipes Classes

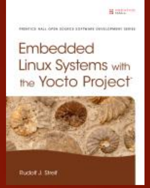
Machine Configuration

Distribution Configuration

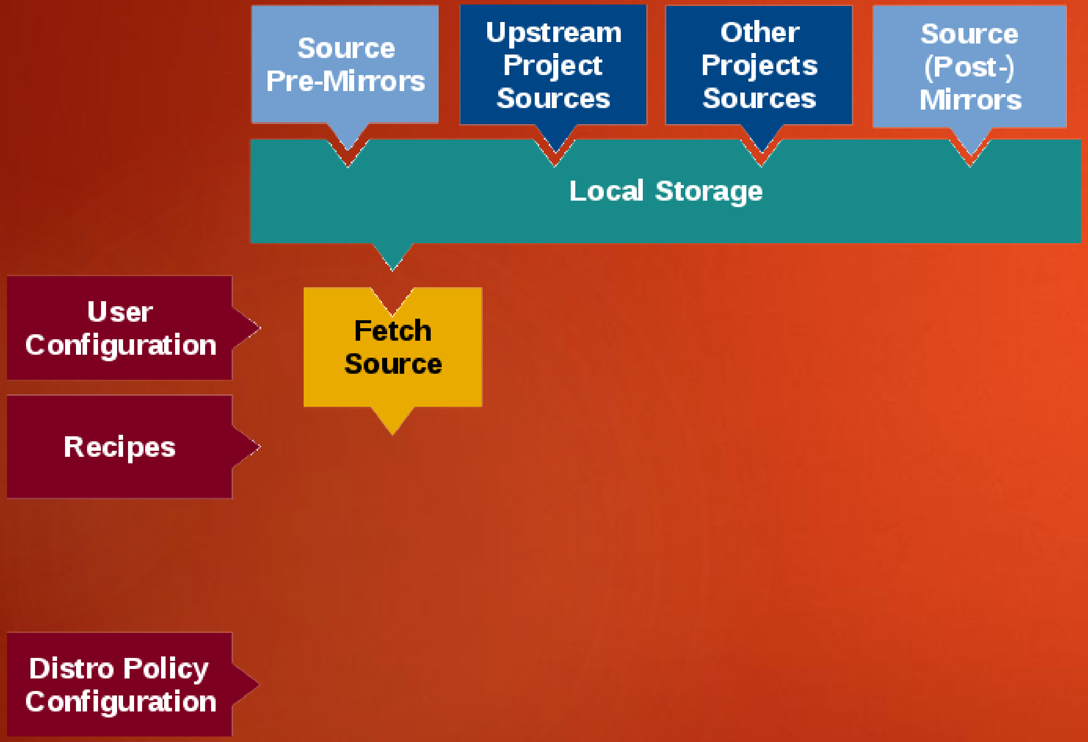
- Defines build system wide policies that affect the way how the OS stack is built:
  - Set preferred providers for functionality (i.e. OpenSSH vs DropBear SSH server)
  - Set alternative preferred versions of software packages
  - Enable/disable LibC functionality (i.e. i18n)
  - Enable/disable features (i.e. pam, selinux)
  - Configure specific packaging rules
  - Configure source mirrors
  - Set distribution information such as name, login prompt, version number, etc.
- Selected from the build environment configuration by setting the `DISTRO` variable.

How does it work?



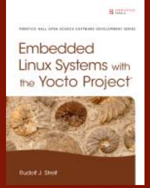


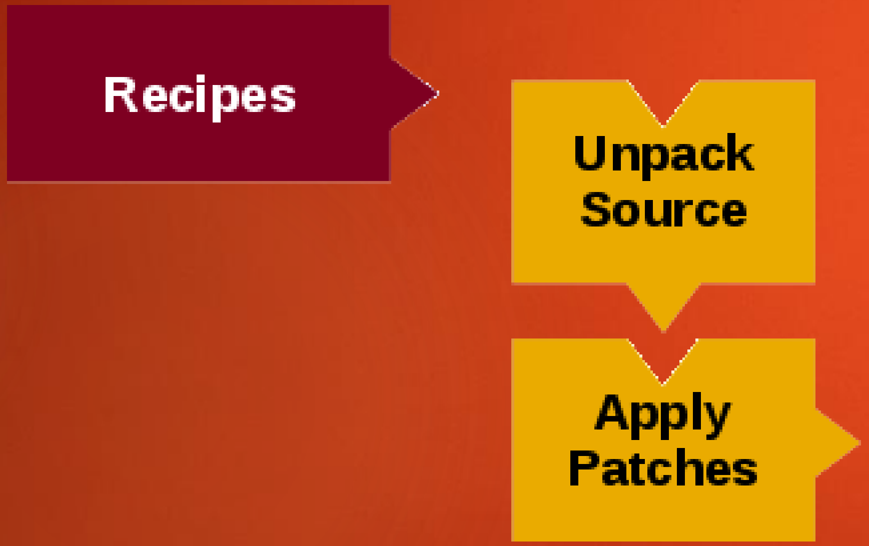
Source Fetching



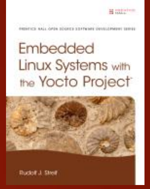
- Recipes call out the location of the sources such as source packages, patches and auxiliary files using the `SRC_URI` variable.
- BitBake can retrieve sources from local and remote locations (git, svn, cvs, p4, hg, bzr, osc, repo, ssh, ftp, http, https, file) in raw or compressed formats (tar, zip, rar, xz, gz, bz2).
- Source download locations are tried in the following order:
  - Local Download Directory (`DL_DIR`)
  - Source Pre-Mirrors (`PREMIRRORS`)
  - Upstream Project Sources (`SRC_URI`)
  - Source (Post-) Mirrors (`MIRRORS`)

Source Extracting and Patching

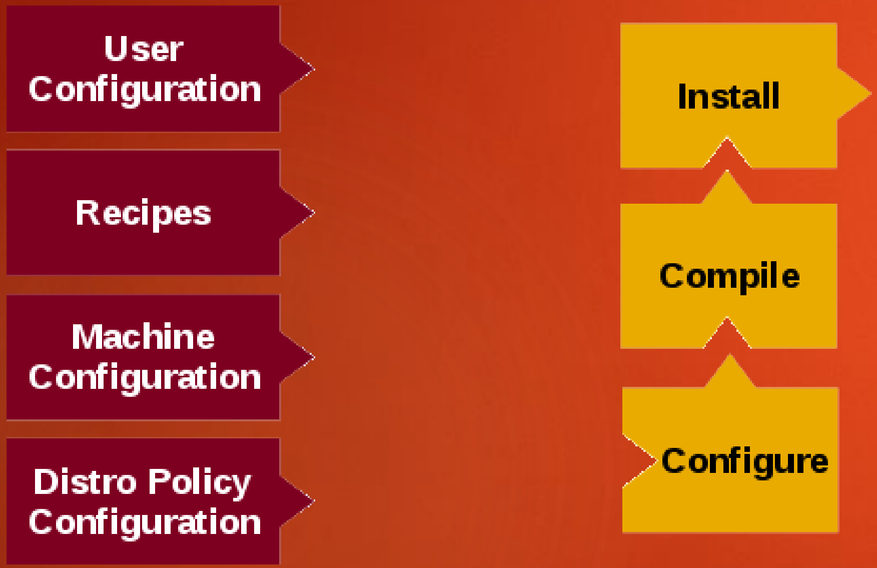




- Source Extraction
  - Unpack tarballs or other archives
  - Check out source from SCM branch, tag, version
- Patch Application
  - Local integration patches, if any, are applied in the order they appear in SRC_URI using quilt.
  - Integration patches are sometimes necessary to adjust the sources for building in a system root environment.
  - Cross-building for other architectures may also require patching the sources.



Build and Install



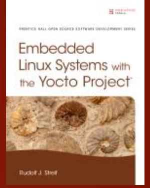
- Recipe specifies build rules
  - Configure – Source configuration using GNU Autotools or other mechanisms
  - Compile – Build the software using the specified build tools such as compilers, linkers and other for any language
  - Install – Copy the build artifacts into a private system root for packaging
- Standard build rules in form of classes for common builders (make, GNU Autotools, NodeJS npm, cmake)

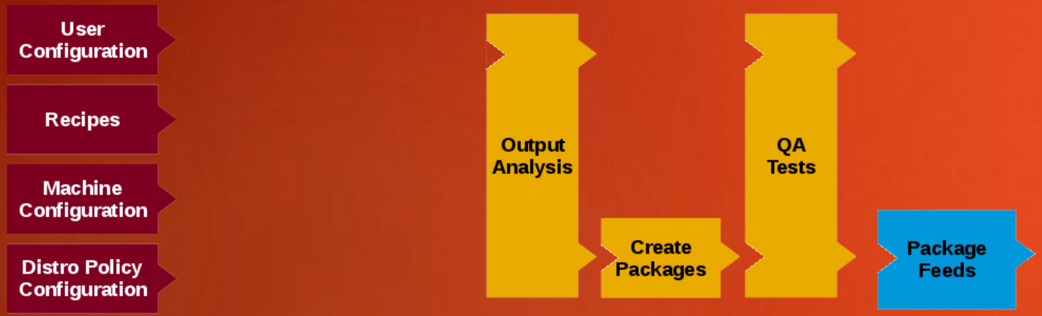
GNU Autotools Recipe Example

```
SUMMARY = "GNU Helloworld Application"
SECTION = "examples"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = " \
    file://COPYING;md5=751419260aa954499f7abaabaa882bbe"
SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
inherit autotools gettext
```

- Short Description
- Section information for package management system
- License used by the software
- File containing the license with checksum to track changes to the license
- Upstream source location
- Classes for build instructions and GNU Gettext i18n, l10n

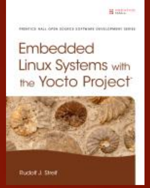
Output Analysis and Packaging



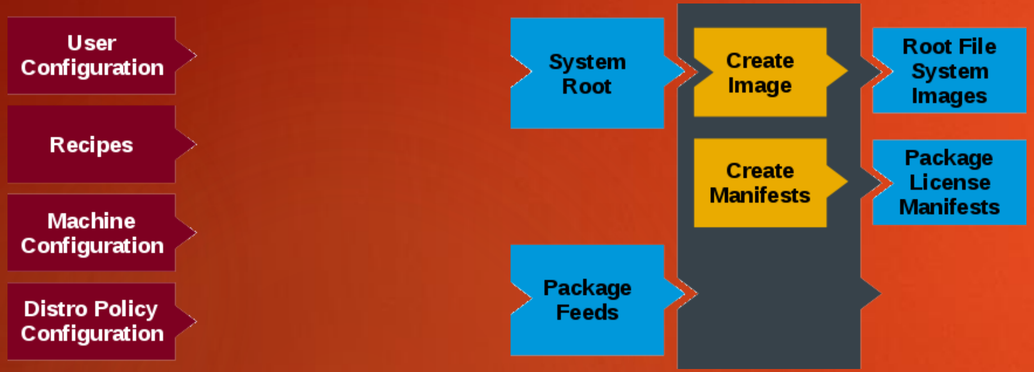


- Output Analysis
  - Categorize created artifacts (runtime, debug, development, documentation, locales)
  - Split into different packages according the standard Linux packaging rules
- QA Checks
  - Verify if all generated artifacts are packaged
  - Check symbolic links for shared libraries
- Package Generation
  - Create package formats rpm, deb, ipk, tar dependent on `PACKAGE_CLASSES`

Image Creation

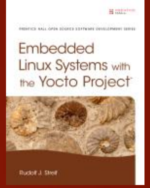


43-45

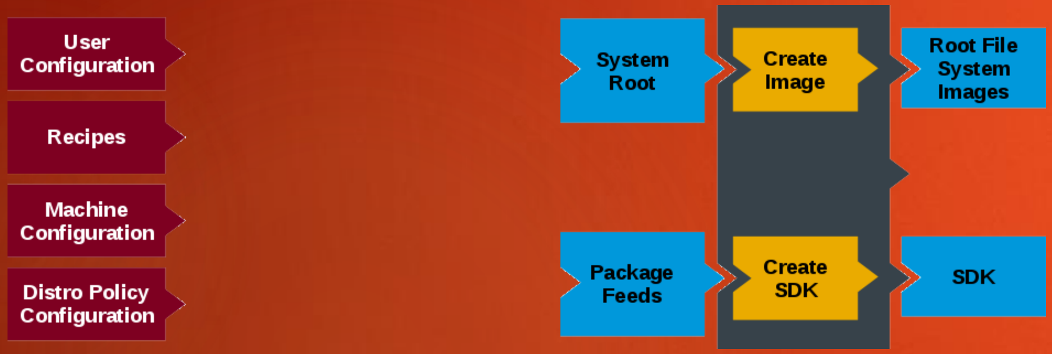


- Images are constructed using the package feeds built by the software package recipes.
- Image creation is based on the required set of components specified by the IMAGE_INSTALL variable of an image recipe.
- The build system automatically expands the specified set of components to include all of their runtime dependencies.
- Images contain the root file system for the OS stack.
- Root file system images can be created in different formats:
  - Compressed tar – extract into a formatted partition on media
  - File system images (ext2, ext3, ext4, jffs, btrfs, …) – bytecopy to raw media
- Root file system and license manifests are generated.

SDK Generation

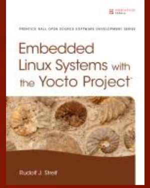


43-45

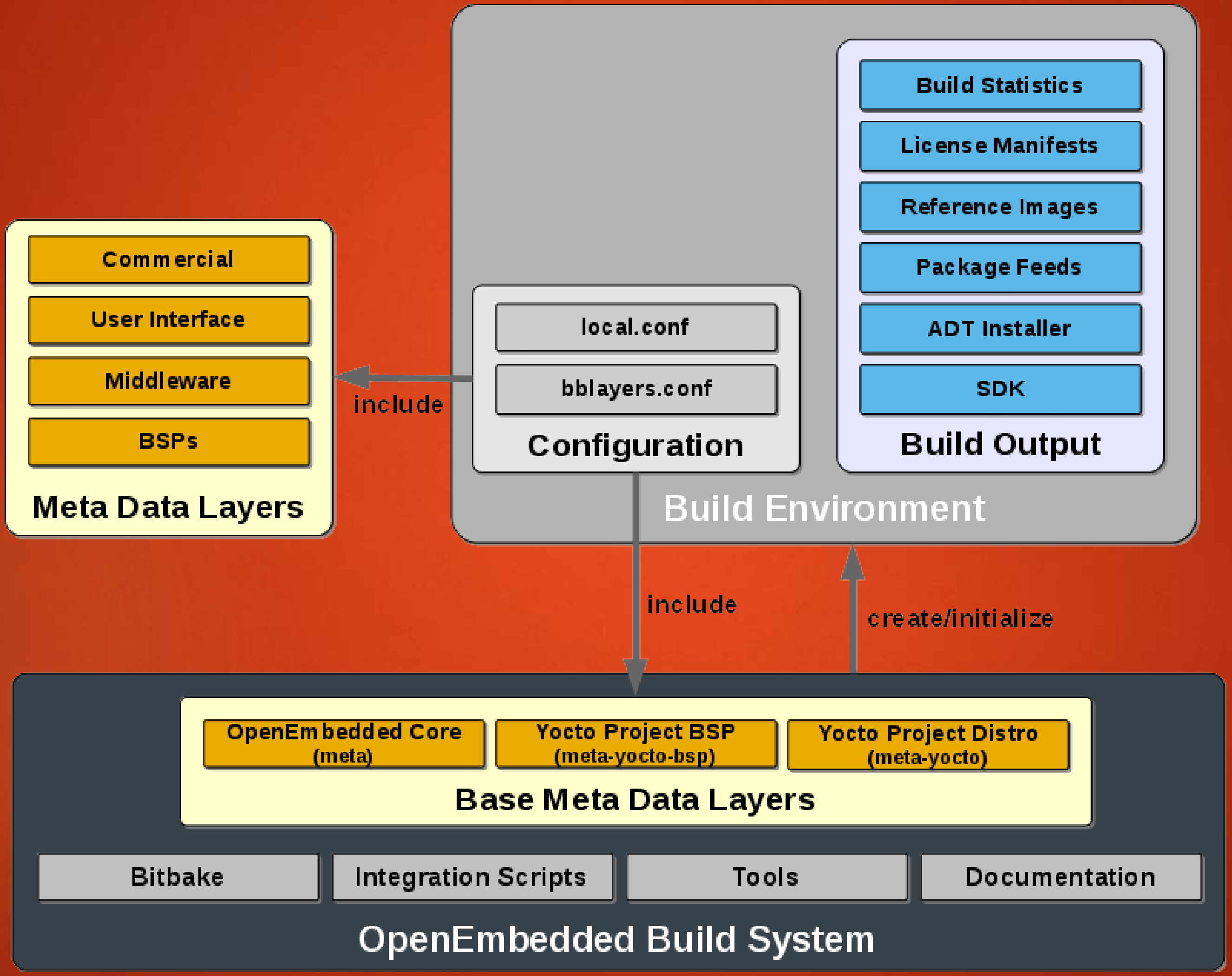


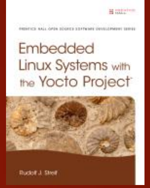
- An SDK consists of
  - Cross-toolchain
  - Native tools and scripts such as QEMU for target emulation
  - Root file system based on the image recipe containing development and debug packages for all installed components
- An SDK can be used from the command line invoking make, gcc, etc. or can be integrated with the Eclipse IDE and Qt Creator IDE.
- An SDK is packaged as a self-installing archive simplifying distribution to application developers.
- An SDK may also contain remote on-target debugging and target profiling tools.

Build System Architecture



45-47



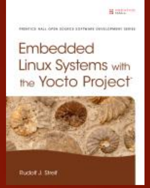

Embedded Linux Systems with the Yocto Project

45-47

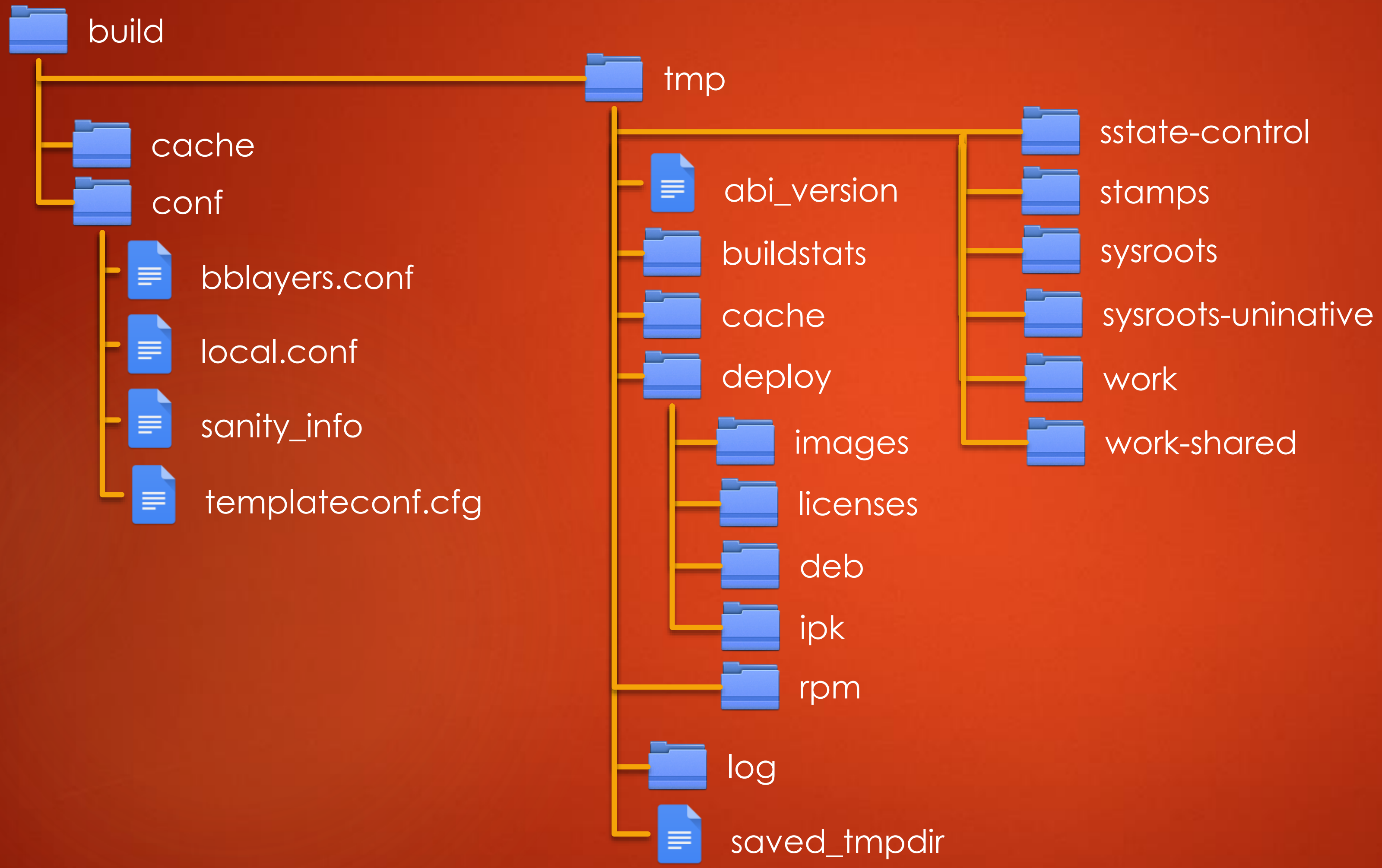
Build System Structure

- poky
  - bitbake
  - documentation
  - LICENSE
  - meta
  - meta-poky
  - meta-selftest
  - meta-skeleton
  - meta-yocto
  - meta-yocto-bsp
  - oe-init-build-env
  - oe-init-build-env-memres
  - README
  - README.hardware
  - scripts

- Build System Components
  - bitbake – BitBake task execution engine
  - documentation – complete documentation set in DocBook format
  - LICENSE – build system license
  - meta – OpenEmbedded Core layer
  - meta-poky – Poky reference distribution layer
  - meta-selftest – test layer used by the Yocto Project continuous integration
  - meta-skeleton – template layer
  - meta-yocto – backwards compatibility layer
  - meta-yocto-base – reference BSP for hardware supported by the Yocto Project
  - oe-init-build-env – initialization script for build environments
  - oe-init-build-env-memres – initialization script for build environments with memory-resident BitBake
  - README – basic info and contribution instructions
  - README.hardware – instructions for the reference BSP
  - scripts – setup and support scripts

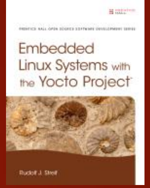


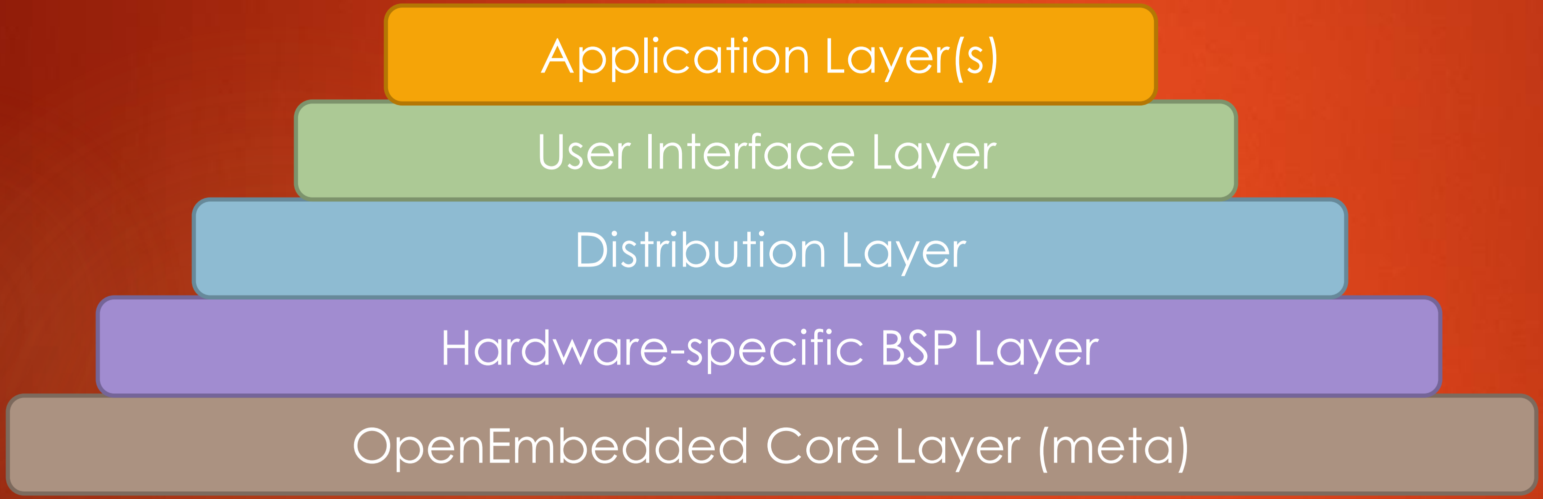
Build Environment Structure



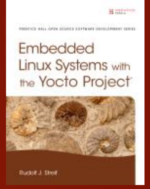
- /build/cache – recipe cache
- /build/conf – build environment configuration
- /build/tmp – build output
  - buildstats – build staticstics
  - cache – cache for specific build artefacts
  - deploy – target build output
  - log – BitBake logging
  - sstate-control – shared state cache manifests
  - stamps – task completion tags and signatures
  - sysroots – root file systems organized by architecture
  - sysroots-uninative – unified build host root file system
  - work – build directories
  - work-shared – build directories for shared packages

Layers

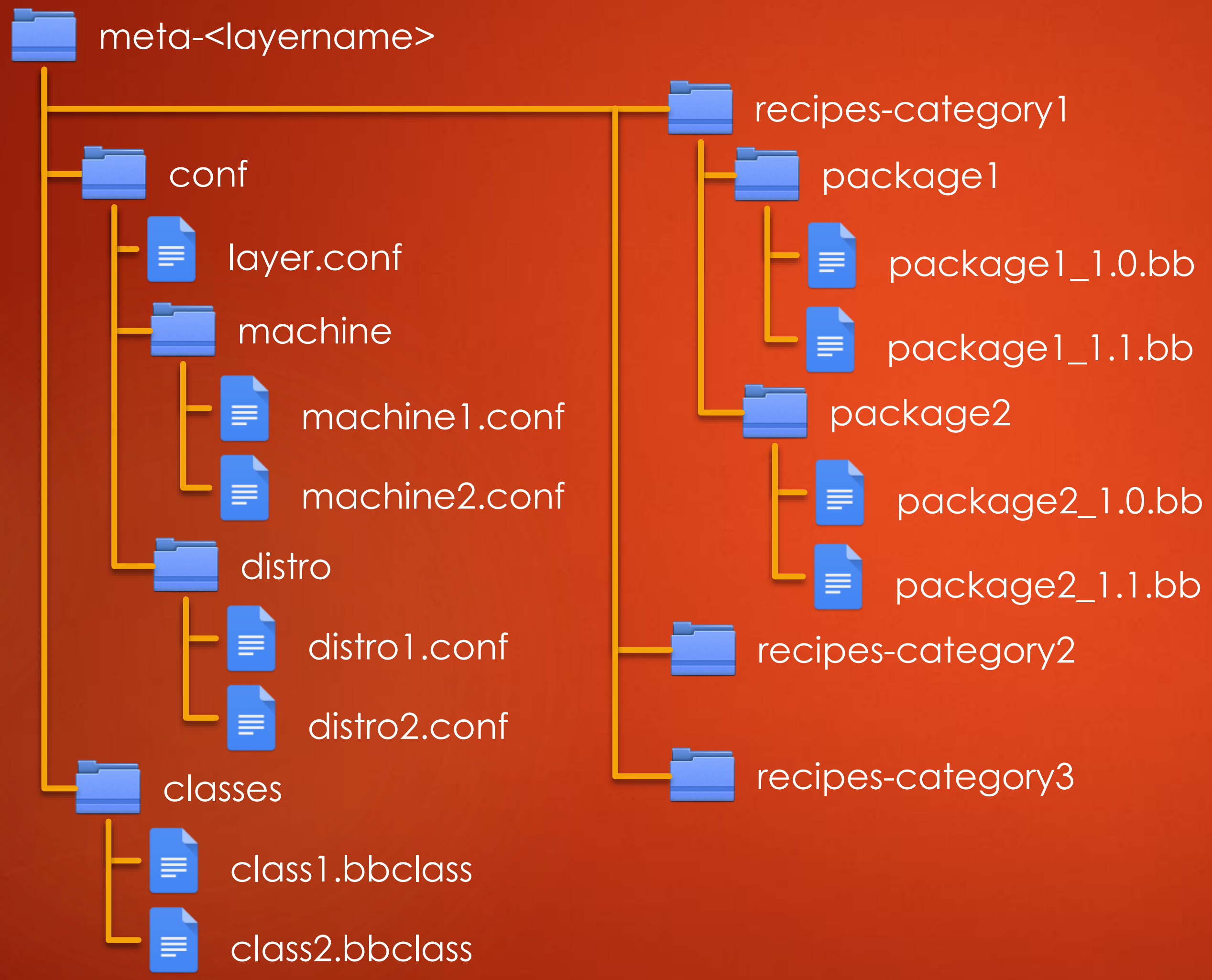




- Layers are containers that organize meta-data into logical entities.
- Layers are like building blocks supporting reusability of components.
- Layers commonly build upon and extend each other.

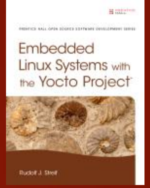


Layer Layout



- A layer is simply a directory layout following certain conventions.
- Every layer must have a `conf/layer.conf` file defining the layer setup.
- All other directories and files depend on the layer purpose:
  - BSP layers define machines.
  - Distribution layers define distros.
- A layer may define classes shared between recipes and for reuse by dependent layers.
- Layers organize recipes into categories i.e. `recipes-networking`, `recipes-graphics` etc.

Layer Configuration



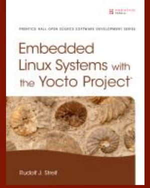
```
# Add the layer's directory to BBPATH
BBPATH =. "${LAYERDIR}:"

# Add the layer's recipe files to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

# Add the name of the layer to the layer collections
BBFILE_COLLECTIONS += "layername"

# Set the recipe file search pattern
BBFILE_PATTERN_layername = "^${LAYERDIR}/"

# Set the priority of this layer
BBFILE_PRIORITY_layername = "5"

# Set version of this layer
# (should only be incremented if changes break compatibility)
LAYERVERSION_layername = "2"

# Specify other layers this layer depends on. This is a white
# space-delimited list of layer names. If this layer depends on a
# particular version of another layer, it can be specified by
# adding the version with a colon to the layer name: e.g.,
# anotherlayer:3
LAYERDEPENDS_layername = "core"
```

- A layer is identified as a layer and configured by its `conf/layer.conf` file.
- Replace `layername` with the actual name of your layer from `meta-layername`.

Creating Layers

Using the `yocto-layer` script simplifies creating layers by setting up the base structure and the `conf/layer.conf` file:

```
yocto-layer create layername
```



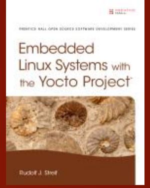
Layer Best Practices

- Use layers for your projects:
  - Layers help separating your own recipes from the standard recipes.
  - The small overhead of creating a layer at the beginning pays off during maintenance.
- Group your recipes:
  - Organize your recipes into subdirectories according to logical grouping.
  - For example: recipes-apps for applications, recipes-network for networking
- Append don't overlay:
  - Reuse recipes from other layers by appending them with bbappend files rather than copying the recipe into your own layer.
- Include don't duplicate:
  - Reuse include files from other layers rather than copying them or their content.
  - For example: `require recipes-core/udev/udev.inc`

Troubleshooting

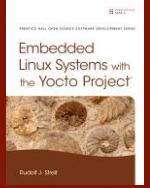
IT IS NEVER A QUESTION OF IF BUT WHEN THINGS GO WRONG

Logging


Embedded Linux Systems with the Yocto Project

110-115

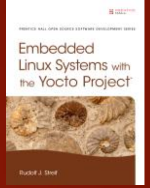
- BitBake logs various events:
  - Debug statements inserted into executable metadata.
  - Output from any command executed by tasks and other code.
  - Error messages emitted by any command executed by tasks and other code.
- Log Files
  - General log files of the BitBake cooker process:
    - LOGDIR = "${TMPDIR}/log/cooker"
    - Log files are organized in subdirectories by target system with timestamps as names.
  - Task log files are stored in the work directory of the recipe:
    - T = ${WORKDIR}/temp"
    - Task log files are named log.do_*taskname*.*pid*; the symbolic link log.do_*taskname* always points to the most recent task log file.



Cooker Log File

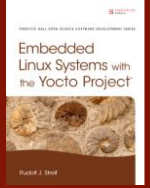
```
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION        = "1.21.1"
BUILD_SYS         = "x86_64-linux"
NATIVELSBSTRING   = "Fedora-18"
TARGET_SYS        = "i586-poky-linux"
MACHINE           = "qemux86"
DISTRO            = "poky"
DISTRO_VERSION    = "1.5+snapshot-20140210"
TUNE_FEATURES     = "m32 i586"
TARGET_FPU        = ""
meta
meta-yocto
meta-yocto-bsp    =
"master:095bb006c3dbbfbdfa05f13d8d7b50e2a5ab2af0"

NOTE: Preparing runqueue
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks

<task execution order>
```

- Build Configuration
  - **BB_VERSION**: The BitBake version number.
  - **BUILD_SYS**: Type of the build system. The variable is defined in bitbake.conf as BUILD_SYS = "${BUILD_ARCH}${BUILD_VENDOR}-${BUILD_OS}". BUILD_ARCH contains the output of uname -m, BUILD_OS contains the output of uname -s, and BUILD_VENDOR is a custom string that is commonly empty.
  - **NATIVELSBSTRING**: Distributor ID and release number concatenated with a dash as obtained by the lsb_release command.
  - **TARGET_SYS**: Type of the target system. This variable is defined in bitbake.conf as TARGET_SYS = "${TARGET_ARCH}${TARGET_VENDOR}${@['-' + d.getVar('TARGET_OS', True), ''][d.getVar('TARGET_OS', True) == ('' or 'custom')]}".
  - **MACHINE**: The target machine BitBake is building for.
  - **DISTRO**: The name of the target distribution.
  - **DISTRO_VERSION**: The version of the target distribution.
  - **TUNE_FEATURES**: Tuning parameters for the target CPU architecture.
  - **TARGET_FPU**: Identification for the floating-point unit of the target architecture.
  - **meta[-xxxx]**: Branch and commit ID for the metadata layers if they were checked out from a Git repository.



Task Logging

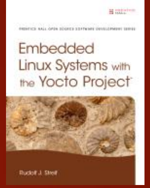
- Tasks in recipes can log messages as well as raise warnings and errors:
  - **Plain**: Logs the message exactly as passed without any additional information.
  - **Debug**: Logs the message prefixed with DEBUG:. Debug message have a log level between 1 and 3.
  - **Note**: Logs the message prefixed with NOTE:. It is used to inform the user about a condition or information to be aware of.
  - **Warn**: Logs the message prefixed with WARNING:. Warnings indicate problems that eventually should be taken care of by the user; however, they do not cause a build failure.
  - **Error**: Logs the message prefixed with ERROR:. Errors indicate problems that need to be resolved to complete the build.
  - **Fatal**: Logs the message prefixed with FATAL:. Fatal conditions cause BitBake to halt the build process right after the message has been logged.
- All messages levels are always logged to the respective log files.
- Note, warning, error and fatal messages are also outputted to the console.
- Debug messages are only sent to the console if if BitBake's debug level is equal or higher to the message level:
  - bitbake –D <target> - level 1
  - bitbake –DD <target> - level 2
  - bitbake –DDD <target> - level 3


Embedded Linux Systems with the Yocto Project

115-119

Task Execution

- Sometimes build failures are due to the task execution order of a recipe. BitBake shows the task execution order for a recipe with:
  - `bitbake <target> -c listtasks`
- Running individual tasks provides for tracking down issues:
  - `bitbake <target> -c <task>`
- Force task execution:
  - `bitbake <target> -C <task>`
- BitBake creates script files for each task:
  - Located in `T = "${WORKDIR}/temp"` and named `run.do_<taskname>.pid`
  - Task script files contain the environment settings (variables) and commands. They can be run directly from your command line.
  - Script files are organized by process id so that they can be compared.
- Clean the recipe build environment:
  - `bitbake <target> -cleanall`

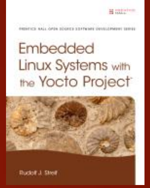


Metadata Analysis

- BitBake outputs its global metadata store with its default settings, before substitutions for specific recipes have occurred with:
  - `bitbake -e`
- BitBake outputs metadata for a specific recipe by providing the recipe name:
  - `bitbake -e <target>`

Metadata output includes all metadata, variables as well as functions. However, most of time you only need to examine the variables. A simple task added to a class can solve this problem:

```
addtask showvars
do_showvars[nostamp] = "1"
python do_showvars() {
        # emit only the metadata that are variables and not functions
        isfunc = lambda key: bool(d.getVarFlag(key, 'func'))
        vars = sorted((key for key in bb.data.keys(d) \
               if not key.startswith('__')))
        for var in vars:
            if not isfunc(var):
                try:
                    val = d.getVar(var, True)
                except Exception as exc:
                    bb.plain('Expansion of %s threw %s: %s' % \
                              (var, exc.__class__.__name__, str(exc)))
                bb.plain('%s="%s"' % (var, val))
}
```

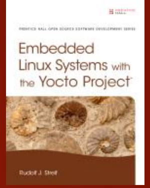


110-121

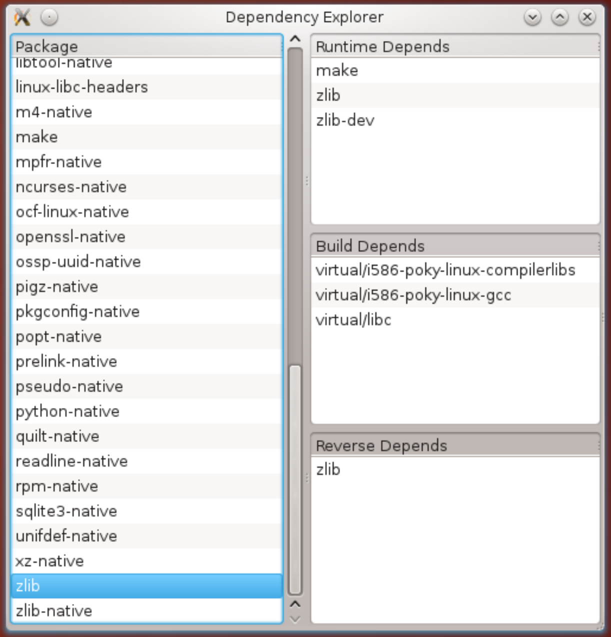
Development Shell

- For debugging build failures it is helpful to be able to directly run configure, make, etc. with the exact same configuration and environment of the build system:
  - `bitbake <target> -c devshell`
- The command opens a shell window and sets all environment variables to point to the build system toolchain and system root.
- You can make changes to the source code and run make to build the package.
- BitBake tries to open a suitable shell window automatically according to your Linux desktop configuration. You can override it by setting the `OE_TERMINAL` variable in `conf/local.conf` of your build environment i.e.:
  - `OE_TERMINAL = "gnome"`

Dependency Graphs



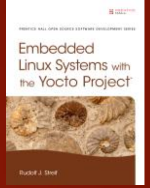
- Creating Dependency Graphs
  - DOT Graphs: `bitbake -g <target>`
  - View a DOT Graph: `dot -Tpng -o pn-depends.png pn-depends.dot`
  - Dependency Explorer: `bitbake -g -u depexp <target>`
- Dependency files:
  - **pn-buildlist**: This file is not a DOT file but contains the list of packages in reverse build order starting with the target.
  - **pn-depends.dot**: Contains the package dependencies in a directed graph declaring the nodes first and then the edges.
  - **package-depends.dot**: Essentially the same as pn-depends.dot but declares the edges for a node right after the node. This file may be easier to read by humans because it groups the edges ending on a node with the node.
  - **task-depends.dot**: Declares the dependencies on the task level.



Dependency Explorer

```
Package              Runtime Depends
libtool-native       make
linux-libc-headers   zlib
m4-native            zlib-dev
make
mpfr-native
ncurses-native
ocf-linux-native
openssl-native       Build Depends
ossp-uuid-native     virtual/i586-poky-linux-compilerlibs
pigz-native          virtual/i586-poky-linux-gcc
pkgconfig-native     virtual/libc
popt-native
prelink-native
pseudo-native
python-native
quilt-native         Reverse Depends
readline-native      zlib
rpm-native
sqlite3-native
unifdef-native
xz-native
zlib
zlib-native
```

Layer Debugging



122-123

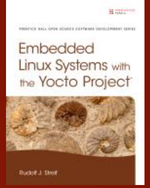
- The bitbake-layers command can assist with debugging layers such as layer priority, recipe list with version and layers that provide them and more:
  - `bitbake-layers help` – Usage information
  - `bitbake-layers show-layers` – Display a list of the layers used by the build environment with their paths and priority.
  - `bitbake-layers show-recipes` – Display a list of recipes in alphabetical order including the layer providing it.
  - `bitbake-layers show-overlayed` – Displays a list of overlaid recipes. A recipe is overlaid if another recipe with the same name exists in a different layer.
  - `bitbake-layers show-appends` – Displays a list of recipes with the files appending them. The appending files are shown in the order they are applied.

Lab Exercise

- Create a dependency graph for busybox with the task dependencies.
- Use the development shell to make changes to the configuration file `.config` and rebuild busybox.
- Clean the bash build environment, then execute just the compile task and find the log file.
- Run the compile task file for bash manually.

Building Custom Linux Systems

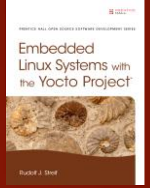
RECIPES FOR CREATING YOUR OWN LINUX DISTRIBUTION FOR ANYTHING FROM EMBEDDED DEVICES TO CLOUD SERVERS



146-149

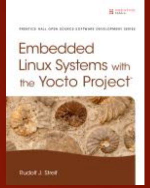
Core Images – Linux Distribution Blueprints

- The OE Core metadata layer provides a set of sample images called core images:
  - Core images range from simple command-line systems to systems with graphical UI support.
  - Core images can be used as a base for your own custom system image or as examples on how to construct system images from scratch.
- Find images from the metadata layers with
  - `find ./meta*/recipes*/images -name "*.bb" -print`



Sample Core Images (1/3)

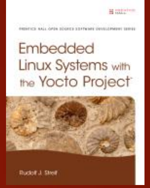
- **core-image-minimal**: This is the most basic image allowing a device to boot to a Linux command-line login. Login and command-line interpreter are provided by BusyBox.
- **core-image-minimal-initramfs**: This image is essentially the same as core-image-minimal but with a Linux kernel that includes a RAM-based initial root filesystem (initramfs).
- **core-image-minimal-mtdutils**: Based on core-image-minimal, this image also includes user space tools to interact with the memory technology device (MTD) subsystem in the Linux kernel to perform operations on flash memory devices.
- **core-image-minimal-dev**: Based on core-image-minimal, this image also includes all the development packages (header files, etc.) for all the packages installed in the root filesystem. If deployed on the target together with a native target toolchain, it allows software development on the target. Together with a cross-toolchain, it can be used for software development on the development host.
- **core-image-rt**: Based on core-image-minimal, this image target builds the Yocto Project real-time kernel and includes a test suite and tools for real-time applications.



146-149

Sample Core Images (2/3)

- **core-image-rt-sdk**: In addition to core-image-rt, this image includes the system development kit (SDK) consisting of the development packages for all packages installed; development tools such as compilers, assemblers, and linkers; as well as performance test tools and Linux kernel development packages. This image allows for software development on the target.
- **core-image-base**: Essentially a core-image-minimal, this image also includes middleware and application packages to support a variety of hardware such as WiFi, Bluetooth, sound, and serial ports. The target device must include the necessary hardware components, and the Linux kernel must provide the device drivers for them.
- **core-image-full-cmdline**: This minimal image adds typical Linux command-line tools—bash, acl, attr, grep, sed, tar, and many more—to the root filesystem.
- **core-image-lsb**: This image contains packages required for conformance with the Linux Standard Base (LSB) specification.
- **core-image-lsb-dev**: This image is the same as the core-image-lsb but also includes the development packages for all packages installed in the root filesystem.



146-149

Sample Core Images (3/3)

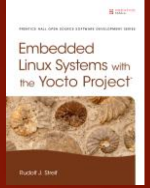
- **core-image-lsb-sdk**: In addition to core-image-lsb-dev, this image includes development tools such as compilers, assemblers, and linkers as well as performance test tools and Linux kernel development packages.
- **core-image-x11**: This basic graphical image includes the X11 server and an X11 terminal application.
- **core-image-directfb**: An image that uses DirectFB for graphics and input device management, DirectFB may include graphics acceleration and a windowing system. Because of its much smaller footprint compared to X11, DirectFB is the preferred choice for lower-end embedded systems that need graphics support but not the entire functionality of X11.
- **core-image-clutter**: This is an X11-based image that also includes the Clutter toolkit. Clutter is based on OpenGL and provides functionality for animated graphical user interfaces.
- **core-image-weston**: This image uses Weston instead of X11. Weston is a compositor that uses the Wayland protocol and implementation to exchange data with its clients. This image also includes a Wayland-capable terminal program.

Image Recipes

```
SUMMARY = "A small image just capable of allowing a device to boot."

IMAGE_INSTALL = "packagegroup-core-boot ${ROOTFS_PKGMANAGE_BOOTSTRAP} \
                 ${CORE_IMAGE_EXTRA_INSTALL}"

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
IMAGE_ROOTFS_EXTRA_SPACE_append = \
   "${@bb.utils.contains("DISTRO_FEATURES", "systemd", " + 4096", "" ,d)}"
```

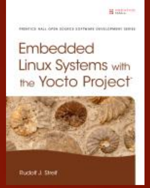
- Images are built by recipes like any software package.
- Image creation logic is provided by the imaging classes:
  - `image` – Base imaging class that does not add any packages by default.
  - `core-image` – Class used to create the core images. Adds packagegroup-core-boot and packagegroup-base-extended packages by default and provides image features.
- The variable IMAGE_INSTALL contains a list of packages and package groups to be added to the image.
- Other variables control various aspects of the image creation process (we discuss them later).

Extending Images



- You can easily extend images by adding packages and package groups to the `IMAGE_INSTALL` variable in the `conf/local.conf` file of your build environment:
  - `IMAGE_INSTALL_append = " <package> <package group> …"`
- Directly adding to `IMAGE_INSTALL` every image. If you only want to add packages to a particular image, specify the name of the image:
  - `IMAGE_INSTALL_append_pn-<image name> = " <package> …"`
- Core images, that is images whose recipe inherits the `core-image` class, can also be extended by using:
  - `CORE_IMAGE_EXTRA_INSTALL = "<package> <package group> …"`

Don't forget to add a space in front of the first entry as the `_append` operator does not do it automatically.


Embedded Linux Systems with the Yocto Project

150-151

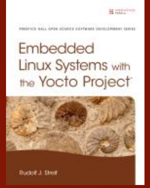
Testing Your Image with QEMU

- QEMU is an open source
  - Machine Emulator running on a host with one architecture emulating the instruction set of another.
  - Virtualizer running guest code for the same architecture as the host directly on the host CPU.
- The build system builds QEMU for your build host according to the target machine selection.
- You can run QEMU directly from within a build environment using the `runqemu` script:
  - `runqemu <target arch>` - Run the latest image built for the target architecture i.e. `runqemu qemux86-64`.
  - `runqemu <target arch> <image>` - Run `<image>` for `<target arch>` i.e. `runqemu qemux86-64 core-image-sato`.

Lab Exercise

- Extend images with `IMAGE_INSTALL_append` in the `conf/local.conf` file of your build environment adding the minicom package.
- Build `core-image-minimal`.
- Launch your image in QEMU and verify that minicom is installed.

Extending an Image with a Recipe

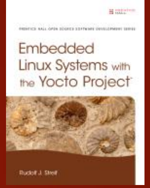


```
DESCRIPTION = "A console image with hardware\
               support for our IoT device"

require recipes-core/images/core-image-base.bb

IMAGE_INSTALL += "sqlite3 mtd-utils coreutils"
IMAGE_FEATURES = "dev-pkgs"
```

- Unlike classes, you need to provide the path relative to the layer for BitBake to find the recipe file to include, and you need to add the `.bb` file extension.
- While you can use either `include` or `require` to include the recipe you are extending, we recommend the use of `require`, since it causes BitBake to exit with an explicit error message if it cannot locate the included recipe file.
- Remember to use the += operator to add to `IMAGE_INSTALL`. Do not use = or := because they overwrite the content of the variable defined by the included recipe.

- Adding package and package groups to `IMAGE_INSTALL` and `CORE_IMAGE_EXTRA_INSTALL` is quick, easy and a good solution for testing but lacks portability and reusability.
- Using a recipe that includes another image recipe solves this problem.
- You can directly add packages and package groups to `IMAGE_INSTALL`.
- Use `IMAGE_FEATURES` to easily include functionality without being concerned about the packages providing the functionality.

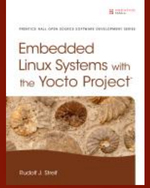
Image Features



153-154

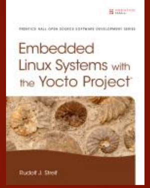
- Image Features are convenient way of adding functionality to your target.
- Image Features are defined by the `image`,the `core-image` and the `populate_sdk_base` classes.
- To use an image in an image recipe add it to the `IMAGE_FEATURES` variable.
- To use an image feature in the `conf/local.conf` configuration file of your build environment, add it to the `EXTRA_IMAGE_FEATURES` variable.
- The build system concatenates `EXTRA_IMAGE_FEATURES` to `IMAGE_FEATURES` to combine the two. Duplicate inclusion of Image Features are take care of and not an issue.

Image Class Image Features



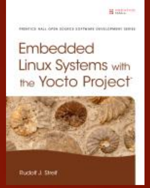
153-154

- **debug-tweaks**: Prepares an image for development purposes. In particular, it sets empty root passwords for console and Secure Shell (SSH) login.
- **package-management**: Installs the package management system according to the package management class defined by `PACKAGE_CLASSES` for the root filesystem.
- **read-only-rootfs**: Creates a read-only root filesystem. This image feature works only if System V Init (SysVinit) system is used rather than sytemd.
- **Splash**: Enables showing a splash screen instead of the boot messages during boot. By default, the splash screen is provided by the psplash package, which can be customized. You can also define an alternative splash screen package by setting the `SPLASH` variable to a different package name.



Core-Image Class Image Features (1/2)

- **eclipse-debug**: Installs remote debugging tools for integration with the Eclipse IDE, namely the GDB debugging server, the Eclipse Target Communication Framework (TCF) agent, and the OpenSSH SFTP server.
- **Hwcodecs**: Installs the hardware decoders and encoders for audio, images, and video if the hardware platform provides them.
- **nfs-server**: Installs Network File System (NFS) server, utilities, and client.
- **qt4-pkgs**: Installs the Qt4 framework and demo applications.
- **ssh-server-dropbear**: Installs the lightweight SSH server Dropbear, which is popular for embedded systems. This image feature is incompatible with ssh-server-openssh. Either one of the two, but not both, can be used.
- **ssh-server-openssh**: Installs the OpenSSH server. This image feature is incompatible with ssh-server-dropbear. Either one of the two, but not both, can be used.

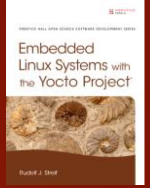


153-154

Core-Image Class Image Features (2/2)

- **tools-debug**: Installs debugging tools, namely the GDB debugger, the GDB remote debugging server, the system call tracing tool strace, and the memory tracing tool mtrace for the GLIBC library if it is the target library.
- **tools-profile**: Installs common profiling tools such as oprofile, powertop, latencytop, lttng-ust, and valgrind.
- **tools-sdk**: Installs software development tools such as the GCC compiler, Make, autoconf, automake, libtool, and many more.
- **tools-testapps**: Installs test applications such as tests for X11 and middleware packages like the telephony manager oFono and the connection manager ConnMan.
- **x11**: Installs the X11 server.
- **x11-base**: Installs the X11 server with windowing system.
- **x11-sato**: Installs the OpenedHand Sato user experience for mobile devices.

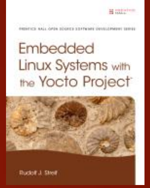
SDK Image Features



153-154

- **dbg-pkgs**: Installs the debug packages containing symbols for all packages installed in the root filesystem.
- **dev-pgks**: Installs the development packages containing headers and other development files for all packages installed in the root filesystem.
- **doc-pkgs**: Installs the documentation packages for all packages installed in the root filesystem.
- **staticdev-pkgs**: Installs the static development packages such as static library files ending in *.a for all packages installed in the root filesystem.
- **ptest-pkgs**: Installs the package test (ptest) packages for all packages installed in the root filesystem.

Creating Layers

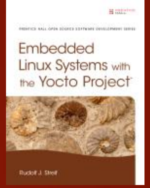


56

- Recipes must always reside in a layer and be included into the build environment by adding the layer path to the `BBLAYERS` variable in the `conf/bblayers.conf` configuration file of the build environment.
- While it might be quick to simply add a recipe to an existing layer such as one of the core metadata layers, it makes good sense that you create your own layers for your recipes.
- The yocto-layer tool makes creating layers easy, eleviating you from setting up the layer structure and configuration files yourself by interactively prompting you for the parameters:
  - `yocto-layer create <layername>`
  - Specify `layername` without the leading meta- as the tool adds it automatically.

Lab Exercise

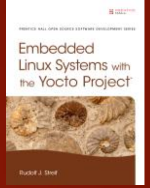
- Create a layer using yocto-layer inside your build environment.
- Add the layer to your build environment.
- Create a directory `recipes-core` for your image recipes inside your layer.
- Create an image recipe that builds on top of core-image-minimal that adds minicom to IMAGE_INSTALL and uses the package management image feature.
- Build your image recipe and verify that the components have correctly been installed in your image.



Package Groups

- Adding individual packages to `IMAGE_INSTALL` of image recipes can be tedious for package sets that are commonly installed and used together.
- For that purpose the build system provides Package Groups that allow grouping packages under a symbolic name.
- The symbolic name can then be used in `IMAGE_INSTALL` and `IMAGE_EXTRA_INSTALL` to add all of the packages of a package group at once.
- Package groups are defined in recipes that inherit from the `packagegroup` class.

```
SUMMARY = "Custom package group for our IoT devices"
DESCRIPTION = "This package group adds standard \
               functionality required by \
               our IoT devices."

LICENSE = "MIT"

inherit packagegroup
PACKAGES = "\
   packagegroup-databases \
   packagegroup-python \
   packagegroup-servers"

RDEPENDS_packagegroup-databases = "\
   db \
   sqlite3"

RDEPENDS_packagegroup-python = "\
   python \
   python-sqlite3"

RDEPENDS_packagegroup-servers = "\
   openssh \
   openssh-sftp-server"

RRECOMMENDS_packagegroup-python = "\
   ncurses \
   readline \
   zip"
```

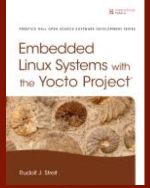


155-159

Predefined Package Groups (1/6)

- **packagegroup-core-ssh-dropbear**: Provides packages for the Dropbear SSH server popular for embedded systems because of its smaller footprint compared to the OpenSSH server. This package group conflicts with packagegroup-core-ssh-openssh. You can include only one of the two in your image. The ssh-server-dropbear image feature installs this package group.
- **packagegroup-core-ssh-openssh**: Provides packages for the standard OpenSSH server. This package group conflicts with packagegroup-core-ssh-dropbear. You can include only one of the two in your image. The ssh-server-openssh image feature installs this package group.
- **packagegroup-core-buildessential**: Provides the essential development tools, namely the GNU Autotools utilitis autoconf, automake and libtool, the GNU binary tool set binutils which includes the linker ld, assembler as and other tools, the compiler collection cpp, gcc, g++, the GNU internationalization and localization tool gettext, make, libstc++ with development packages and pkgconfig.
- **packagegroup-core-tools-debug**: Provides the essential debugging tools, namely the GDB debugger, the GDB remote debugging server, the system call tracing tool strace and for the GLIBC target library the memory tracing tool mtrace.

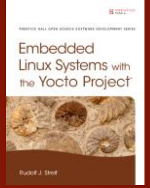
Predefined Package Groups (2/6)



155-159

- **packagegroup-core-sdk**: This package group combines the packagegroup-core-buildessential package group with additional tools for development such as GNU Core Utilities coreutils with shell, file, and text manipulation utilities; dynamic linker ldd; and others. Together with packagegroup-core-standalone-sdk-target, this package group forms the tools-sdk image feature.
- **packagegroup-core-standalone-sdk-target**: Provides the GCC and standard C++ libraries. Together with packagegroup-core-sdk, this package group forms the tools-sdk image feature.
- **packagegroup-core-eclipse-debug**: Provides the GDB debugging server, the Eclipse TCF agent, and the OpenSSH SFTP server for integration with the Eclipse IDE for remote deployment and debugging. The image feature eclipse-debug installs this package group.
- **packagegroup-core-tools-testapps**: Provides test applications such as tests for X11 and middleware packages like the telephony manager oFono and the connection manager ConnMan. The tools-testapps image feature installs this package group.

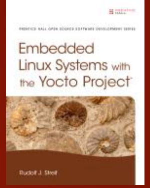
Predefined Package Groups (3/6)



155-159

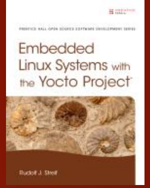
- **packagegroup-self-hosted**: Provides all necessary packages for a self-hosted build system. The build-appliance image target uses this package group.
- **packagegroup-core-boot**: Provides the minimum set of packages necessary to create a bootable image with console. All core-image targets install this package group. The core-image-minimal installs just this package group and the postinstallation scripts.
- **packagegroup-core-nfs**: Provides NFS server, utilities, and client. The nfs-server image feature installs this package group.
- **packagegroup-base**: This recipe provides multiple package groups that depend on each other as well as on machine and distribution configuration. The purpose of these package groups is to add hardware, networking protocol, USB, filesystem, and other support to the images dependent on the machine and distribution configuration.

Predefined Package Groups (4/6)


Embedded Linux Systems with the Yocto Project

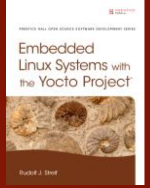
- **packagegroup-cross-canadian**: Provides SDK packages for creating a toolchain using the Canadian Cross technique, which is building a toolchain on system A that executes on system B to create binaries for system C. A use case for this package group is to build a toolchain with the Yocto Project on your build host that runs on your image target but produces output for a third system with a different architecture than your image target.
- **packagegroup-core-tools-profile**: Provides common profiling tools such as oProfile, PowerTOP, LatencyTOP, LTTng-UST, and Valgrind. The tools-profile image feature uses this package group.
- **packagegroup-core-device-devel**: Provides distcc support for an image. Distcc allows distribution of compilation across several machines on a network. The distcc must be installed, configured, and running on your build host. On the target you must define the cross-compiler variable to use distcc instead of the local compiler (e.g., export CC="distcc").
- **packagegroup-qt-toolchain-target**: Provides the package to build applications for the X11-based version of the Qt development toolkit on the target system.

Predefined Package Groups (5/6)



155-159

- **packagegroup-qte-toolchain-target**: Provides the package to build applications for the embedded version of the Qt development toolkit on the target system.
- **packagegroup-core-qt**: Provides all necessary packages for a target system using the X11-based version of the Qt development toolkit.
- **packagegroup-core-qt4e**: Provides all necessary packages for a target system using the embedded Qt toolkit. The qt4e-demo-image installs this package group.
- **packagegroup-core-x11-xserver**: Provides the X.Org X11 server only.
- **packagegroup-core-x11**: Provides packagegroup-core-x11-xserver plus basic utilities such as xhost, xauth, xset, xrandr, and initialization on startup. The x11 image feature installs this package group.
- **packagegroup-core-x11-base**: Provides packagegroup-core-x11 plus middleware and application clients for a working X11 environment that includes the Matchbox Window Manager, Matchbox Terminal, and a fonts package. The x11-base image feature installs this package group.



155-159

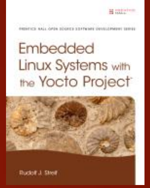
Predefined Package Groups (6/6)

- **packagegroup-core-x11-sato**: Provides the OpenedHand Sato user experience for mobile devices, which includes the Matchbox Window Manager, Matchbox Desktop, and a variety of applications. The x11-sato image feature installs this package group. To utilize this package group for your target image, you also have to install packagegroup-core-x11-base.
- **packagegroup-core-clutter-core**: Provides packages for the Clutter graphical toolkit. To use the toolkit for your target image, you also have to install packagegroup-core-x11-base.
- **packagegroup-core-directfb**: Provides packages for the DirectFB support without X11. Among others, the package group includes the directfb package and the directfb-example package, and it adds touchscreen support if provided by the machine configuration.
- **packagegroup-core-lsb**: Provides all packages required for LSB support.
- **packagegroup-core-full-cmdline**: Provides packages for a more traditional Linux system by installing the full command-line utilities rather than the more compact BusyBox variant.

Lab Exercise

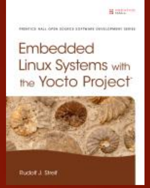
- To the `recipes-core` directory of your layer add a `packagegroups` subdirectory.
- Create a package group recipe in that subdirectory that defines the package groups with their respective content:
  - Package group apps to contain sqlite3, python, python-sqlite3
  - Package group tools to contain sudo, gzip, tar
- Add the package group to your image recipe.
- Build the image and verify that the components of your package groups have been installed.

Core Image from Scratch



- Inheriting from the image class in your image recipe directly gives you the most control over the content of your root file system images.
- The image class does not install packages and package groups by default. If used with an empty `IMAGE_INSTALL` variable it produces an empty root file system image.
- The `core-image` class builds on top of the image class and adds `packagegroup-core-boot` and `packagegroup-base-extended` to the image by default to produce a minimal bootable image.

```
SUMMARY = "Custom image recipe that does not get \
          any simpler"
DESCRIPTION = "Well yes, you could remove SUMMARY, \
              DESCRIPTION, LICENSE."

LICENSE = "MIT"

inherit core-image
```



Base Core Image with the Image Class

```
SUMMARY = "Custom image recipe from scratch"
DESCRIPTION = "Directly assign IMAGE_INSTALL and IMAGE_FEATURES for direct control over \
               image content."

LICENSE = "MIT"

# We are using the assignment operator (=) below to purposely overwrite
# the default from the core-image class.
IMAGE_INSTALL = "packagegroup-core-boot packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL}"
IMAGE_FEATURES = "${EXTRA_IMAGE_FEATURES} splash"
CORE_IMAGE_EXTRA_INSTALL ?= ""

inherit image
```

- The example emulates the `core-image-class` in an image recipe using the `image` class.
- Of course you will not get the image features defined by the `core-image-class`.

Lab Exercise

- Create an image recipe that inherits from `core-image` and adds the mtd-utils package and the splash image feature.
- Build your image and test it.

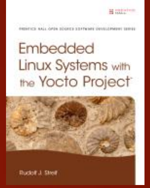
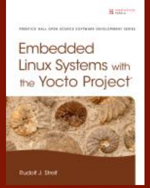


Image Options

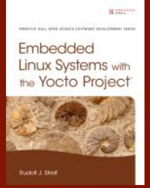
- The creation of images and their content can further be tuned by specifying certain options:
  - Languages and Locales – `IMGE_LINGUAS`
  - Package Management – `PACKAGE_CLASSES`
  - Image Size – `IMAGE_ROOTFS_SIZE`, `IMAGE_ROOTFS_ALIGNMENT`, `IMAGE_ROOTFS_EXTRA_SPACE`, `IMAGE_OVERHEAD_FACTOR`
  - Root File System Types – `IMAGE_FSTYPES`
  - Users, Groups, Passwords
  - Root File System Postprocessing

Languages and Locales



- Software packages may provide native language support (NLS) with internationalization (i18n) and localization (l10n) through locale packages.
- Many such programs use the GNU gettext package but other schemes are supported too as long as the locale packages are provided.
- By default the build system installs the en-us locale package.
- Other locales can be installed by adding them to the `IMAGE_LINGUAS` variable:
  - `IMAGE_LINGUAS = “en-gb pt-br”`
- The image class adds
  - `IMAGE_LINGUAS ?= “de-de, fr-fr and en-gb”`

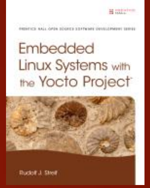
Package Management



- The build system can package software packages using the formats:
  - Debian Package Management (dpkg)
  - Open Package Management (opkg)
  - Red Hat Package Management (rpm)
  - Tape Archving (tar)
- The package management systems are selected by adding the packaging classes to the PACKAGE_CLASSES variable:
  - PACKAGE_CLASSES = "package_rpm package_ipk package_tar"
- More than one package class can be specified causing the build system to create packages using all specified formats.
- The first package class is used to create the root file system.
- Tar cannot be the only of the first package class in the list, as root file systems cannot be constructed from tar packages.

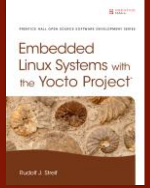
⚠ The build system does not automatically install the package manager in the target's root file system. You can install it by adding `package_management` to `IMAGE_FEATURES`.

Image Size

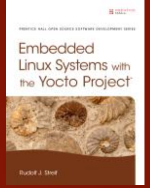


- The final size of a root file system is dependent on multiple factors but and computed dynamically by the build system.
- Several variables influence the sizing:
  - **IMAGE_ROOTFS_SIZE**: Defines the size in kilobytes of the created root filesystem image. The build system uses this value as a request or recommendation. The final root filesystem image size may be larger depending on the actual space required. The default value is 65536.
  - **IMAGE_ROOTFS_ALIGNMENT**: Defines the alignment of the root filesystem image in kilobytes. If the final size of the root filesystem image is not a multiple of this value, it is rounded up to the nearest multiple of it. The default value is 1.
  - **IMAGE_ROOTFS_EXTRA_SPACE**: Adds extra free space to the root filesystem image. The variable specifies the value in kilobytes. For example, to add an additional 4 GB of space, set the variable to IMAGE_ROOTFS_EXTRA_SPACE = "4194304". The default value is 0.
  - **IMAGE_OVERHEAD_FACTOR**: This variable specifies a multiplicator for the root filesystem image. The factor is applied after the actual space required by the root filesystem has been determined. The default value is 1.3.

Image Size Computation



```
_get_rootfs_size():
   ROOTFS_SIZE =`du -ks ${IMAGE_ROOTFS}`
   BASE_SIZE = ROOTFS_SIZE * IMAGE_OVERHEAD_FACTOR

   if (BASE_SIZE < IMAGE_ROOTFS_SIZE):
      IMG_SIZE = IMAGE_ROOTFS_SIZE +
IMAGE_ROOTFS_EXTRA_SPACE
   else:
      IMG_SIZE = BASE_SIZE + IMAGE_ROOTFS_EXTRA_SPACE

   IMG_SIZE = IMG_SIZE + IMAGE_ROOTFS_ALIGNMENT - 1
   IMG_SIZE = IMG_SIZE % IMAGE_ROOTFS_ALIGNMENT

   return IMG_SIZE
```

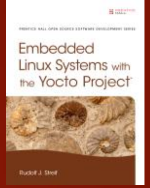


161-169

Root File System Creation

- The build system can create root file system in various formats as files.
- Some formats such as tar, tar.gz, tar.bz2 etc. are intended to be extracted onto a formatted partition of a storage media.
- Others such as ext3, btrfs etc. are created with partition and file system information and just need to be extracted directly to the media.
- File system creation is controlled by the `image_types` class and the `IMAGE_FSTYPES` variable:
  - IMAGE_FSTYPES = “ext3 tar.bz2 iso hddimg”
  - Multiple format can be created at the same time.

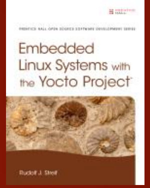
Root File System Types (1/2)



161-169

- **tar**, **tar.gz**, **tar.bz2**, **tar.xz**, **tar.lz3**: Create uncompressed and compressed root filesystem images in the form of tar archives.
- **ext2**, **ext2.gz**, **ext2.bz2**, **ext2.lzma**: Root filesystem images using the ext2 filesystem without or with compression.
- **ext3**, **ext3.gz**: Root filesystem images using the ext3 filesystem without or with compression.
- **Btrfs**: Root filesystem image with B-tree filesystem.
- **jffs2**, **jffs2.sum**: Uncompressed or compressed root filesystems based on the second generation of the Journaling Flash File System (JFFS2). Since JFFS2 directly supports NAND flash devices, it is a popular choice for embedded systems. It also provides journaling and wear-leveling.
- **cramfs**: Root filesystem image using the compressed ROM filesystem (cramfs). The Linux kernel can mount this filesystem without prior decompression. The compression uses the zlib algorithm that compresses files one page at a time to allow random access. This filesystem is read-only to simplify its design, as random write access with compression is difficult to implement.
- **iso**: Root filesystem image type using the ISO 9660 standard for bootable CD-ROM. This filesystem type is not a standalone format. It uses ext3 as the underlying filesystem type.

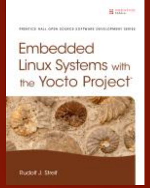
Root File System Types (2/2)



161-169

- **hddimg**: Root filesystem image for bootable hard drives. It uses ext3 as the actual filesystem type.
- **squashfs**, **squashfs-xz**: Compressed read-only root filesystem type specifically for Linux, similar to cramfs but with better compression and support for larger files and filesystems. Squashfs also has a variable block size from 0.5 kB to 64 kB over the fixed 4 kB block size of cramfs, which allows for larger file and filesystem sizes. Squashfs uses gzip compression, while squashfs-xz uses Lempel–Ziv–Markov (LZMA) compression for even smaller images.
- **ubi**, **ubifs**: Root filesystem images using the unsorted block image (UBI) format for raw flash devices. UBI File System (UBIFS) is essentially a successor to JFFS2. The main differences between the two is that UBIFS supports write caching. Using ubifs in IMAGE_FSTYPES just creates the ubifs root filesystem image. Using ubi creates the ubifs root filesystem image and also runs the ubinize utility to create an image that can be written directly to a flash device.
- **cpio**, **cpio.gz**, **cpio.xz**, **cpio.lzma**: Root filesystem images using uncompressed or compressed copy in and out (CPIO) streams.
- **Vmdk**: Root filesystem image using the VMware virtual machine disk format. It uses the ext3 as the underlying filesystem format.
- **elf**: Bootable root filesystem image created with the mkelfImage utility from the Coreboot project (www.coreboot.org).

Users, Groups, Passwords



```
SUMMARY = "Custom image using the extrausers class"
DESCRIPTION = "Create users, groups and set passwords"

LICENSE = "MIT"

IMAGE_INSTALL = "packagegroup-core-boot \
                 packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL}"

inherit core-image
inherit extrausers

# set image root password
ROOT_PASSWORD = "secret"
DEV_PASSWORD = "hackme"

EXTRA_USERS_PARAMS = "\
   groupadd developers; \
   useradd -p `openssl passwd ${DEV_PASSWORD}` developer; \
   useradd -g developers developer; \
   usermod -p `openssl passwd ${ROOT_PASSWORD}` root; \
   "
```

- The extrausers class provides a mechanism for a managing users, groups, and passwords.
- Commands:
  - useradd
  - usermod
  - userdel
  - groupadd
  - groupmod
  - groupdel
- Passwords must be provided in encrypted form.

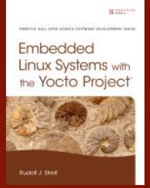


Image Postprocessing

- Sometimes it is necessary to do processing such as adding, modifying files and more after the root file system has been created but before it is packaged into the different formats.
- Using the variable `ROOTFS_POSTPROCESS_COMMANDS` specifies a list of shell functions to be executed.
- The variable and the functions are added to the image recipe.
- The functions are executed in the order they appear in the variable.
- The search path for shell commands includes the native system root of the build environment and the build host `PATH` from the user environment.

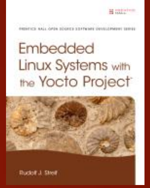


Image Postprocessing – Setting Login Shells

```
SUMMARY = "Image Postprocessing"
DESCRIPTION = "Modify login shells."

LICENSE = "MIT"

# We are using the assignment operator (=) below to purposely overwrite
# the default from the core-image class.
IMAGE_INSTALL = "packagegroup-core-boot packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL}"

inherit core-image

# Additional root filesystem processing
modify_shells() {
   printf "# /etc/shells: valid login shells\n/bin/sh\n/bin/bash\n" \
           > ${IMAGE_ROOTFS}/etc/shells
}
ROOTFS_POSTPROCESS_COMMAND += "modify_shells;"
```

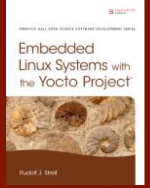


Image Postprocessing – Sudo Configuration

```
modify_sudoers() {
   sed 's/# %sudo/%sudo/' < ${IMAGE_ROOTFS}/etc/sudoers > \
        ${IMAGE_ROOTFS}/etc/sudoers.tmp
   mv ${IMAGE_ROOTFS}/etc/sudoers.tmp ${IMAGE_ROOTFS}/etc/sudoers
}
ROOTFS_POSTPROCESS_COMMAND += "modify_sudoers;"
```

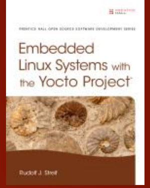


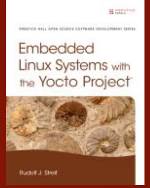
Image Postprocessing – SSH Server Configuration

```
configure_sshd() {
   # disallow password authentication
   echo "PasswordAuthentication no" >> ${IMAGE_ROOTFS}/etc/ssh/sshd_config
   # create keys in tmp/deploy/keys
   mkdir -p ${DEPLOY_DIR}/keys
   if [ ! -f ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot ]; then
      ssh-keygen -t rsa -N '' \
         -f ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot
   fi
   # add public key to authorized_keys for root
   mkdir -p ${IMAGE_ROOTFS}/home/root/.ssh
   cat ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot.pub \
       >> ${IMAGE_ROOTFS}/home/root/.ssh/authorized_keys
}
ROOTFS_POSTPROCESS_COMMAND += "configure_sshd;"
```

Lab Exercise

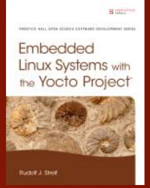
- Create an image that adds user and a group of your choice and makes the user a member of that group. Also assign a password to that user and to the root user. Build and test the image.
- Create a post-process command to add that user to the sudoers list. Build and test the image.
- Add the openssh server to your image and configure it:
  - Disable password login
  - Provision a key for the above user
  - Build and test the image by logging into the system remotely via ssh.

Distribution Configuration



169-180

- Distribution configuration are settings that globally apply to all images. The settings are provided by distribution configuration files.
- Any layer can contain distribution configuration files. They are typically located in the `conf/distro` subdirectory of a layer.
- A build environment selects the distribution configuration by setting the `DISTRO` variable in `conf/local.conf`:
  - `DISTRO ?= “poky”`
- The meta-poky layer provides the Poky distribution variants:
  - **poky**: Poky is the default policy for the Yocto Project’s reference distribution Poky.
  - **poky-bleeding**: This distribution configuration is based on poky but sets the versions for all packages to the latest revision.
  - **poky-lsb**: This distribution configuration is for a stack that complies with LSB.
  - **poky-tiny**: This distribution configuration tailors the settings to yield a very compact Linux OS stack for embedded devices.
- Distribution configuration files can set any variable but there is a set of settings commonly used.

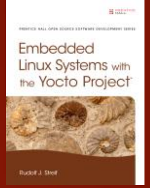


169-180

Distribution Configuration – Distribution Information

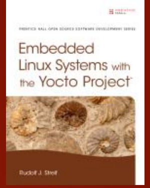
- **DISTRO**: Short name of the distribution. The value must match the base name of the distribution configuration file.
- **DISTRO_NAME**: The long name of the distribution. Various recipes reference this variable. Its contents is shown on the console boot prompt.
- **DISTRO_VERSION**: Distribution version string. It is referenced by various recipes and used in file names' distribution artifacts. Shown on the console boot prompt.
- **DISTRO_CODENAME**: A code name for the distribution. It is currently used only by the LSB recipes and copied into the lsb-release system configuration file.
- **MAINTAINER**: Name and e-mail address of the distribution maintainer.
- **TARGET_VENDOR**: Target vendor string that is concatenated with various variables, most notably target system (TARGET_SYS). TARGET_SYS is a concatenation of target architecture (TARGET_ARCH), target vendor (TARGET_VENDOR), and target operating system (TARGET_OS), such as i586-poky-linux. The three parts are delimited by hyphens. The TARGET_VENDOR string must be prefixed with the hyphen, and TARGET_OS must not. This is one of the many unfortunate inconsistencies of the OpenEmbedded build system. You may want to set this variable to your or your company's name.

Distribution Configuration – SDK Information



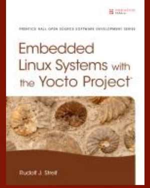
- **SDK_NAME**: The base name that the build system uses for SDK output files. It is derived by concatenating the DISTRO, TCLIBC, SDK_ARCH, IMAGE_BASENAME, and TUNE_PKGARCH variables with hyphens. There is not much reason for you to change that string from its default setting, as it provides all the information needed to distinguish different SDKs.
- **SDK_VERSION**: SDK version string, which is commonly set to DISTRO_VERSION.
- **SDK_VENDOR**: SDK vendor string, which serves a similar purpose as TARGET_VENDOR. Like TARGET_VENDOR, the string must be prefixed with a hyphen.
- **SDKPATH**: Default installation path for the SDK. The SDK installer offers this path to the user during installation of an SDK. The user can accept it or enter an alternative path. The default value /opt/${DISTRO}/${SDK_VERSION} installs the SDK into the /opt system directory, which requires root privileges. A viable alternative would be to install the SDK into the user's home directory by setting SDKPATH = "${HOME}/${DISTRO}/${SDK_VERSION}".

Distribution Configuration – Features, Preferences, Dependencies



169-180

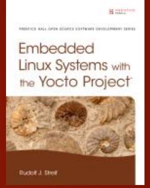
- **DISTRO_FEATURES**: A list of distribution features that enable support for certain functionality within software packages. The assignment in the poky.conf distribution policy file includes DISTRO_FEATURES_DEFAULT and DISTRO_FEATURES_LIBC. Both contain default distribution feature settings. We discuss distribution features and how they work and the default configuration in the next two sections.
- **PREFERRED_VERSION**: Using PREFERRED_VERSION allows setting particular versions for software packages if you do not want to use the latest version, as it is the default. Commonly, that is done for the Linux kernel but also for software packages on which your application software has strong version dependencies.
- **DISTRO_EXTRA_RDEPENDS**: Sets runtime dependencies for the distribution. Dependencies declared with this variable are required for the distribution. If these dependencies are not met, building the distributions fails.
- **DISTRO_EXTRA_RRECOMMENDS**: Packages that are recommended for the distribution to provide additional useful functionality. These dependencies are added if available but building the distribution does not fail if they are not met.



169-180

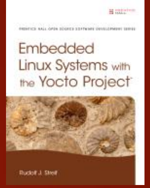
Distribution Configuration – Toolchain Configuration

- **TCMODE**: This variable selects the toolchain that the build system uses. The default value is default, which selects the internal toolchain built by the build system (gcc, binutils, etc.). The setting of the variable corresponds to a configuration file tcmode-${TCMODE}.inc, which the build system locates in the path conf/distro/include. This allows including an external toolchain with the build system by including a toolchain layer that provides the necessary tools as well as the configuration file. If you are using an external toolchain, you must ensure that it is compatible with the Poky build system.
- **TCLIBC**: Specifies the C library to be used. The build system currently supports EGLIBC, uClibc, and musl. The setting of the variable corresponds to a configuration file tclibc-${TCLIBC).inc that the build system locates in the path conf/distro/include. These configuration files set preferred providers for libraries and more.
- **TCLIBCAPPEND**: The build systems appends this string to other variables to distinguish build artifacts by C library. If you are experimenting with different C libraries, you may want to use the settings TCLIBCAPPEND = "-${TCLIBC}“ and TMPDIR .= "${TCLIBCAPPEND}“ in your distribution configuration, which creates a separate build output directory structure for each C library.



Distribution Configuration – Build System Configuration

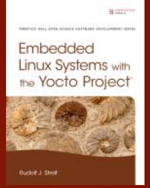
- **LOCALCONF_VERSION**: Sets the expected or required version for the build environment configuration file local.conf. The build system compares this value to the value of the variable CONF_VERSION in local.conf. If LOCALCONF_VERSION is a later version than CONF_VERSION, the build system may be able to automatically upgrade local.conf to the newer version. Otherwise, the build system exits with an error message.
- **LAYER_CONF_VERSION**: Sets the expected or required version for the bblayers.conf configuration file of a build environment. The build system compares this version to the value of LCONF_VERSION set by bblayers.conf. If LAYER_CONF_VERSION is a later version than LCONF_VERSION, the build system may be able to automatically upgrade bblayers.conf to the newer version. Otherwise, the build system exits with an error message.
- **OELAYOUT_ABI**: Sets the expected or required version for the layout of the output directory TMPDIR. The build system stores the actual layout version in the file abi_version inside of TMPDIR. If the two are incompatible, the build system exits with an error message. This typically happens only if you are using a newer version of the build system with a build environment that was created by a previous version and the layout changed incompatibly. Deleting TMPDIR resolves the issue by re-creating the directory.
- **BB_SIGNATURE_HANDLER**: The signature handler used for signing shared state cache entries and creating stamp files. Using the default value of OEBasicHash is typically sufficient for most applications.



169-180

Distribution Configuration – Build System Checks, QA Checks

- **INHERIT += "poky-sanity"**: Inherits the class poky-sanity, which is required to perform the build system checks. It is recommended that you include this directive in your own distribution configuration files.
- **CONNECTIVITY_CHECK_URIS**: A list of URIs that the build system tries to verify network connectivity. In the case of Poky, these point to files on the Yocto Project's high-availability infrastructure. If you intend to use your own mirrors for downloading source packages, you could use URIs pointing to files on you mirror servers to verify proper connectivity.
- **SANITY_TESTED_DISTROS**: A list of Linux distributions the Poky build system has been tested on. The build system verifies the Linux distribution it is running on against this list. If that distribution is not in the list, Poky displays a warning message and starts the build process regardless. Poky runs on most current Linux distributions, and in most cases, building works just fine even if the distribution is not officially supported.
- **WARN_QA**: A list of QA checks that create warning messages, but the build continues.
- **ERROR_QA**: A list of QA checks that create error messages and the build terminates.

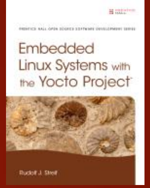


169-180

Distribution Configuration – Mirror Configuration

- **PREMIRRORS** and **MIRRORS**: The Poky distribution adds these variables to set its mirror configuration to use the Yocto Project repositories as a source for downloads.

Set your own mirrors in your distribution configuration so that all of your build environments use the same source file downloads.



System Manager

- The System Manager is the first user-space process the Linux kernel starts after booting.
- The System Manager is responsible for launching the user-space daemons and other processes up to the login prompt.
- Supported System Managers:
  - SysVinit – Common UNIX script-bases system management.
  - systemd – Enhanced system service management with parallel execution and prioritization.
- SysVinit is the default System Manager. To enable systemd add to your distribution configuration:
  - `DISTRO_FEATURES_append = “ systemd”`
  - `VIRTUAL-RUNTIME_init_manager = “systemd”`

You can switch the System Manger simply by assigning `VIRTUAL-RUNTIME_init_manager` to `systemd` or to `sysvinit`.

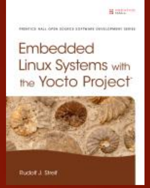
Lab Exercise

- Create a distribution configuration in your layer based on the default poky.conf file (just copy the file to your layer using a name of your choice).
- Change the distribution information to personalize it and add your info as a maintainer.
- Build an image and verify the results.

Software Package Recipes

INTEGRATING AND BUILDING SOFTWARE PACAKGES

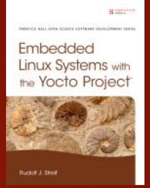
Recipe Layout and Conventions



185-196

- The OpenEmbedded community and the Yocto Project developers have established best practices and conventions on how to write recipes. It's like coding standards for the build system.
- Recipe Filename
  - Convention: `<packagename>_<version>-<revision>.bb`
  - If package sources are retrieved from an SCM: `<packagename>_<scm>.bb` i.e. `libxext_git.bb`
  - In this case PV must be set explicitly: `PV = "<version>+git${SRCREV}"`
- Recipe Layout
  - Recipe layout is not strictly defined but follows some core conventions to make them easier to understand.
  - For this course we break them up into sections which we explain.
  - Open the gettext recipe in an editor to follow along: `${POKYDIR}/meta/recipes-core/gettext/gettext_<version>.bb`

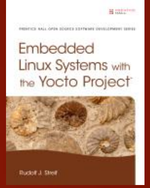
Recipe Metadata (1/11)



185-196

- Descriptive Metadata
  - **SUMMARY**: A one-line (up to 80 characters long), short description of the package.
  - **DESCRIPTION**: An extended (possibly multiple lines long), detailed description of the package and what it provides.
  - **AUTHOR**: Name and e-mail address of the author of the software package (not the recipe) in the form of AUTHOR = "Santa Claus <santa@northpole.com>". This can be a list of multiple authors.
  - **HOMEPAGE**: The URL, starting with http://, where the software package is hosted.
  - **BUGTRACKER**: The URL, starting with http://, to the project's bug tracking system.

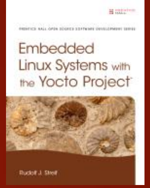
Recipe Metadata (2/11)



185-196

- Package Manager Metadata
  - **SECTION**: The category the software package belongs to.
  - **PRIORITY**: Priorities are used to tell the package management tools whether a software package is required for a system to operate, is optional, or eventually conflicts with other packages. Priorities are utilized only by the Debian package manager dpkg and the Open Package Manager opkg. The priorities are
    - **standard**—Packages that are standard for any Linux distribution, including a reasonably small but not too limited console-mode system.
    - **required**—Packages that are necessary for the proper function of the system.
    - **optional**—Packages that are not necessary for a functional system but for a reasonably usable system.
    - **extra**—Packages that may conflict with other packages from higher priorities or that have specialized requirements.

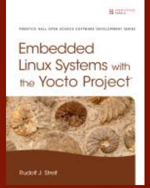
Recipe Metadata (3/11)



185-196

- Licensing Metadata
  - **LICENSE**: The name of the license (or licenses) used for this software package. In most cases, only a single license applies, but some open source software packages employ multiple licenses. These can be dual licenses allowing the user of a package to choose one of several licenses or multiple licenses where parts of the software package are licensed differently. Dual licenses are specified by concatenating the license names with the pipe symbol (|). Multiple licenses are specified by concatenating the license names with the ampersand (&) symbol. The build system also supports complex logical license "arithmetic," such as GLv2 & (LGPLv2.1 | MPL-1.1 | BSD).
  - **LIC_FILES_CHECKSUM**: This variable allows tracking changes to the license files itself. The variable contains a space-delimited list of license files with their respective checksums. After fetching and unpacking a software package's source files, the build system verifies the license by calculating a checksum over the license file, or portions thereof, and comparing it with the checksum provided.
- Inheritance Directives and Includes

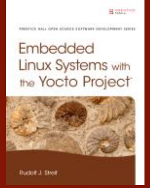
Recipe Metadata (4/11)



185-196

- Build Metadata
  - **PROVIDES**: Space-delimited list of one or more additional package names typically used for abstract provisioning.
  - **DEPENDS**: Space-delimited list of names of packages that must be built before this package can be built.
  - **PN**: The package name. The value of this variable is derived by BitBake from the base name of the recipe file. For most packages, this is correct and sufficient. Some packages may need to adjust this value. For example, the cross-toolchain applications for instance gcc-cross have the target architecture appended to their names.
  - **PV**: The package version, which is derived by BitBake from the base name of the recipe file. For all but packages that directly build from source repositories, this value is correct and sufficient. For those that build from SCM, Section 8.1.1 explains how to set PV correctly.

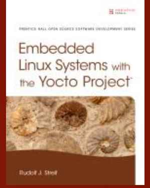
Recipe Metadata (5/11)



185-196

- Build Metadata (continued)
  - **PR**: The package revision. The default revision is r0. In the past, BitBake required you to increase the revision every time the recipe itself has changed to trigger a rebuild. However, the new signature handlers now calculate the signature of recipe metadata including functions. The build system now entirely relies on the signatures for rebuilding.
  - **SRC_URI**: Space-delimited list of URIs to download source code, patches, and other files from.
  - **SRCDATE**: The source code date. This variable applies only when sources are retrieved from SCM systems.
  - **S**: The directory location in the build environment where the build system places the unpacked source code. The default location depends on the recipe name and version: ${WORKDIR}/${PN}-${PV}. The default location is appropriate for virtually all packages built from archives. For packages directly built from SCM, you need to set this variable explicitly, such as ${WORKDIR}/git for GIT repositories.

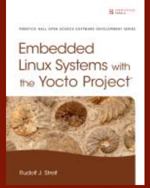
Recipe Metadata (6/11)



185-196

- Build Metadata (continued)
  - **B**: The directory location in the build environment where the build system places the object created during the build. The default is the same as S: ${WORKDIR}/${PN}-${PV}. Many software packages are built *in tree* or *in location*, placing the objects inside the source tree. Recipes building packages with GNU Autotools, the Linux kernel, and cross-toolchain applications separate source and build directories.
  - **FILESEXTRAPATHS**: Extends the build system’s search path for additional local files defined by FILESPATH. This variable is most commonly used for append files in the form FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}", which causes the build system to first look for additional files in a subdirectory with the name of the package of the directory where the append file is located before looking in the other directories specified by FILESEXTRAPATHS.

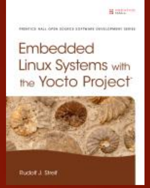
Recipe Metadata (7/11)



185-196

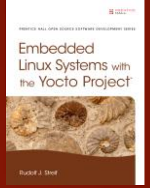
- Build Metadata (continued)
  - **PACKAGECONFIG**: This variable allows enabling and disabling features of a software package at build time.
  - **EXTRA_OECONF**: Additional configure script options.
  - **EXTRA_OEMAKE**: Additional options for GNU Make.
  - **EXTRA_OECMAKE**: Additional options for CMake.
  - **LDFLAGS**: Options passed to the linker. The default setting depends on what the build system is building: TARGET_LDFLAGS when building for the target, BUILD_LDFLAGS when building for the build host, BUILDSDK_LDFLAGS when building an SDK for the host. You typically won’t overwrite this variable entirely but instead will add options to it.
  - **PACKAGE_ARCH**: Defines the architecture of the software package.

Recipe Metadata (8/11)



- Packaging Metadata
  - **PACKAGES**: This variable is a space-delimited list of packages that are created during the packaging process. The default value of this variable is `"${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN} ${PN}"`.
  - **FILES**: The FILES variable defines lists of directories and files that are placed into a particular package. The build system defines default file and directory lists for the default packages, such as `FILES_${PN}-dbg = "<files>"`, where files is a space-delimited list of directories and files that can contain wildcards.
  - **PACKAGE_BEFORE_PN**: The variable lets you easily add packages before the final package name is created.

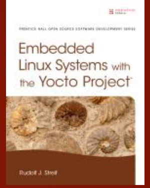
Recipe Metadata (9/11)



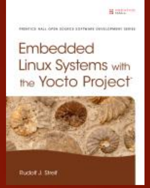
185-196

- Packaging Metadata (continued)
  - **PACKAGE_DEBUG_SPLIT_STYLE**: This variable determines how to split binary and debug objects when the ${PN}-dgb package is created. There are three variants:
    - **".debug"**: The files containing the debug symbols are placed in a .debug directory inside the directory where the binaries are installed on the target. For example, if the binaries are installed in /usr/bin, the debug symbol files are placed in /usr/bin/.debug. This option also installs the source files in .debug, which is the default behavior.
    - **"debug-file-directory"**: Debug files are placed under /usr/lib/debug on the target, separating them from the binaries.
    - **"debug-without-src"**: This variant is the same as .debug, but the source files are not installed.
- **PACKAGESPLITFUNCS**: This variable defines a list of functions that perform the package splitting. The default, defined by package.bbclass, is PACKAGESPLITFUNCS ?= "package_do_split_locales populate_packages". Recipes can prepend to this variable to run their own package-splitting functions before the default ones are run.

Recipe Metadata (10/11)



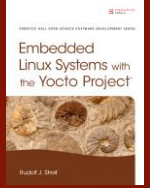
- Task Overrides, Prepends, and Appends
  - Replacements and/or modifications of tasks.
- Variants
  - **BBCLASSEXTENDS**: Define variant builds of the recipe.



185-196

Recipe Metadata (11/11)

- Runtime Metadata
  - **RDEPENDS**: A list of packages that this package depends on at runtime and that must be installed for this package to function correctly.
  - **RRECOMMENDS**: Similar to RDEPENDS but indicates a weak dependency, as these packages are not essential for the package to run.
  - **RSUGGESTS**: Similar to RRECOMMENDS but even weaker in the sense that package managers do not install these packages if they are available.
  - **RPROVIDES**: Package name alias list for runtime provisioning.
  - **RCONFLICTS**: List of names of conflicting packages.
  - **RREPLACES**: List of names of packages this package replaces.

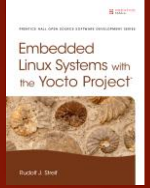


185-196

Mandatory Recipe Metadata

- Recipes are required to set these variables:
  - SUMMARY
  - LICENSE
  - LICENSE_FILES_CHKSUM (unless LICENSE = “closed”)
  - SRC_URI
  - SRC_URI[md5sum], SRC_URI[sha256sum] unless SRC_URI fetches from an SCM
- Other variables are optional, however, setting the above variables does not mean the recipe is functional.
- A package using make with proper variable definitions in the makefile i.e. CC, LD, etc. can be built without any class inheritance.
- For other build system classes are provided:
  - autotools
  - cmake

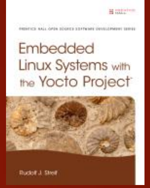
Recipe Formatting Guidelines (1/2)



185-196

- **Assignments**
  - Use a single space on each side of the assignment operator.
  - Use quotes only on the right hand side of the assignment. VARIABLE = "VALUE"
- **Continuation**
  - Continuation is used to split long variable lists, such as SRC_URI, for better readability.
  - Use the line continuation symbol (\).
  - Do not place any spaces after the line continuation symbol.
  - Indent successive lines up to the level of the start of the value.
  - Use spaces instead of tabs for indentation, since developers tend to set their tab sizes differently.
  - Place the closing quote on its own line.

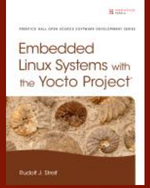
Recipe Formatting Guidelines (2/2)



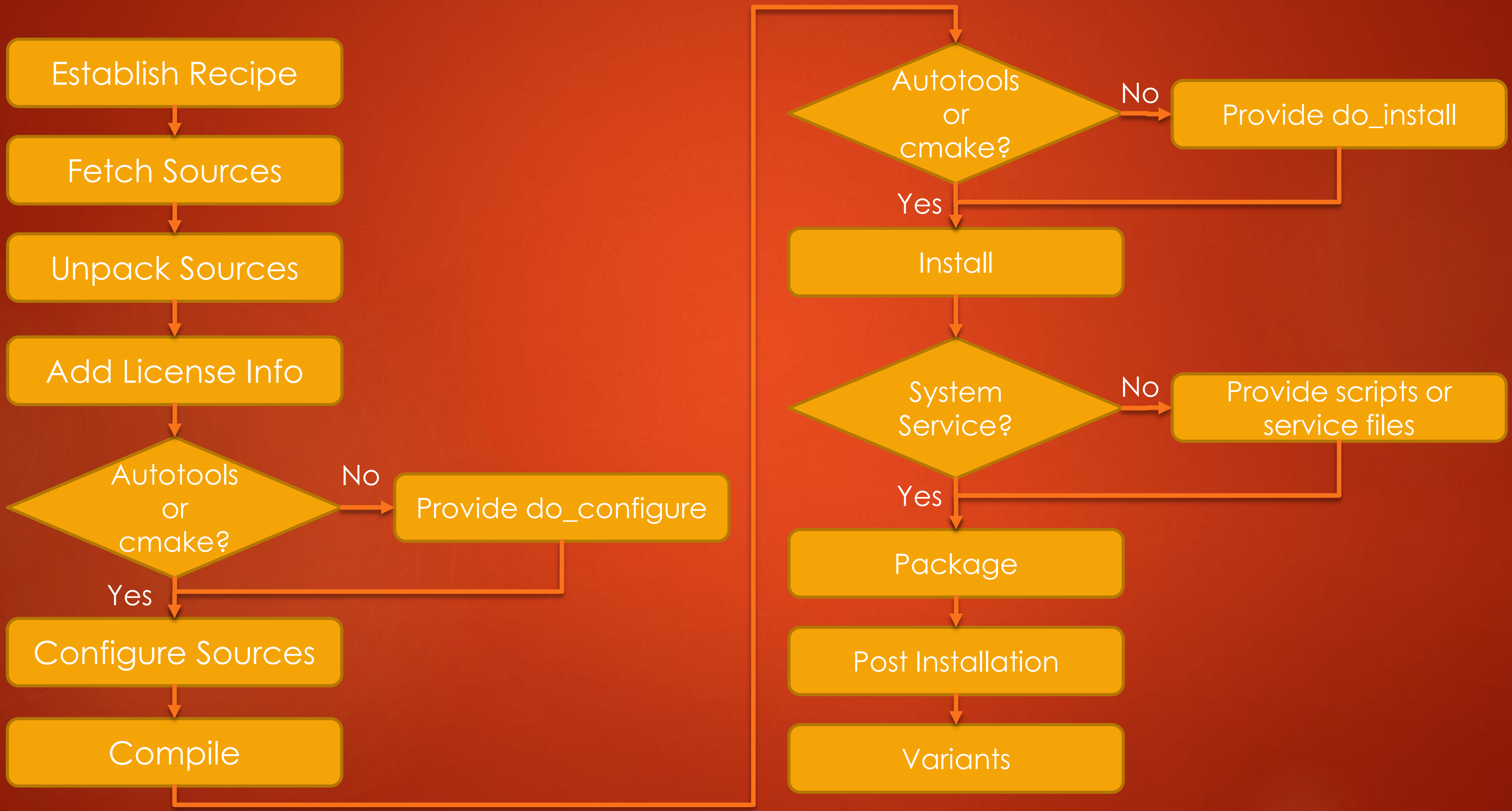
- **Python Functions**
  - Use four spaces per indent; do not use tabs.
  - Python is rather finicky about indentation. Never mix spaces and tabs.
- **Shell Functions**
  - Use four spaces per indent; do not use tabs.
  - Some layers, such as OECore, use tabs for indentation for shell functions. However, it is recommended that you use four spaces for new layers to stay consistent with Python functions.
- **Comments**
  - Comments are allowed and encouraged in recipes, classes, and configuration files.
  - Comments must start at the beginning of the line using the # character.
  - Comments cannot be used inside of a continuation.

Following the guidelines and best practices is required if you are looking to contribute layers and recipes to OpenEmbedded and the Yocot Project.

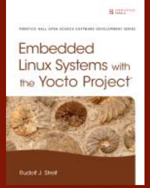
But even if you do not intend to make any contributions, following these simple guidelines makes it much simpler for you and your organization to share and maintain layers, recipes, configuration files and classes.



Writing a New Recipe - Workflow

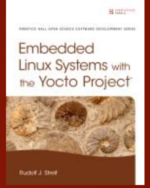


Writing a New Recipe - Setup



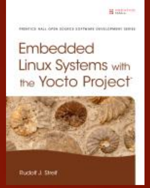
196-211

- Establish the Recipe
  - Create a layer if you do not already have one: yocto-layer create mylayer
  - Setup a skeleton recipe (yocto-layer can create one for you)
  - Add the layer to your build environment
- Fetch the Source Code
  - Set SRC_URI to point to your sources
- Unpack the Source Code
  - Adjust S if necessary e.g. for git repos: S = “${WORKDIR}/git”
- Patch the Source Code
  - If patches are required place them in subdirectory next to the recipe.
  - Add the patches to SRC_URI.



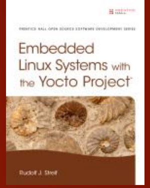
Writing a New Recipe – License Information

- Add Licensing Information
  - Set LICENSE:
    - For proprietary source code use `LICENSE = "closed"`.
    - Common open source licenses can be found in `${POKYDIR}/meta/files/common-licenses`.
  - Point LIC_FILES_CHKSUM to the license file: `LIC_FILES_CHKSUM = "file://COPYING;md5=<md5sum>"`
    - You can leave the checksum open and have the build system compute it for you.



Writing a New Recipe - Configuration

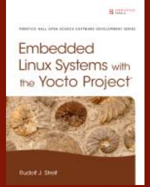
- Configure the Source Code
  - GNU Autotools
    - Inherit the `autotools` class.
    - For most source code adhering to the Autotools standards the `autotools` class will configure it correctly based on the `configure.ac` file.
  - CMake
    - Inherit the `cmake` class.
    - The `cmake` class will correctly configure the source code based on the `CMakeLists.txt` file.
  - Other
    - Whether a configuration step is necessary or not depends on the source code.
    - Provide your own configure task.



Writing a New Recipe - Building

- Compile
  - Run the compile task to see if your source code builds correctly.
  - Common issues:
    - Missing header files and/or libraries: Add the packages providing them to the DEPENDS variable.
    - Host leakage: The source code's build system references build host paths and files. The QA tasks typically detects them and issues an error message.
    - Parallel build issues: These are hard to track. Setting `PARALLEL_MAKE = ""` in `conf/local.conf` turns parallel building off for testing.

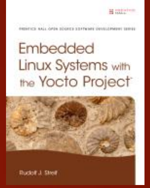
Writing a New Recipe - Installation



196-211

- Install the Build Output
  - GNU Autotools or CMake
    - The autotools and cmake classes respectively take care of the installation.
    - You just need to verify correct installation.
  - Make
    - The default install task runs the install target of the makefile.
    - You may need to make adjustments dependent on the setup of the install target.
  - Manual Installation
    - If the makefile does not contain an install target you need to write an installing the build output.
    - Always use the `install` command rather than `cp`.

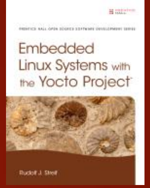
```
do_install() {
    install -d ${B}/bin/hello ${D}${bindir}
    install -d ${B}/lib/hello.lib ${D}${libdir}
}
```



Writing a New Recipe – System Services

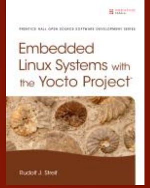
- Setup System Services
  - If your software package is a system service that eventually needs to be started when the system boots you need to add the scripts and service files.
  - SysVInit
    - Inherit `update-rc.d` class.
    - INITSCRIPT_PACKAGES: List of packages that contain the init scripts for this software package. This variable is optional and defaults to INITSCRIPT_PACKAGES = "${PN}".
    - INITSCRIPT_NAME: The name of the init script.
    - INITSCRIPT_PARAMS: The parameters passed to update-rc.d. This can be a string such as "defaults 80 20" to start the service when entering run levels 2, 3, 4, and 5 and stop it from entering run levels 0, 1, and 6.
  - systemd
    - Inherit `systemd` class.
    - SYSTEMD_PACKAGES: List of packages that contain the systemd service files for the software package. This variable is optional and defaults to SYSTEMD_PACKAGES = "${PN}".
    - SYSTEMD_SERVICE: The name of the service file.

Writing a New Recipe – Packaging



196-211

- Package Splitting – Packaging of the installed build artifacts into different packages:
  - **PACKAGES**: This variable is a space-delimited list of package names.
    - Default: `PACKAGES = "${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"`
    - The do_package task processes the list from the left to the right. The order is important, since a package consumes the files that are associated with it.
  - **FILES**: The FILES variable defines lists of directories and files that are placed into a particular package
    - FILES_${PN}-dbg = "<files>"
  - If there are unpackaged but installed build artifacts after the last package has been created, the build system issues an error message.



Writing a New Recipe – Custom Installation Scripts

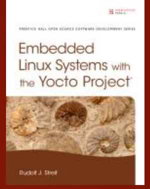
- Package management systems have the ability to run scripts before and after a package is installed, upgraded, or removed.
- These are typically shell scripts and they can be provided by the recipe using these variables:
  - `pkg_preinst_<packagename>`: Preinstallation script that is run *before the package is installed*.
  - `pkg_postinst_<packagename>`: Postinstallation script that is run *after the package is installed*.
  - `pkg_prerm_<packagename>`: Pre-uninstallation script that is run *before the package is uninstalled*.
  - `pkg_postrm_<packagename>`: Post-uninstallation script that is run *after the package is uninstalled*.

```
pkg_postinst_${PN}() {
#!/bin/sh
# shell commands go here
}
```

Script Skeleton

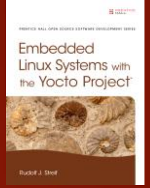
```
pkg_postinst_${PN}() {
#!/bin/sh
if [ x"$D" = "x" ]; then
   # target execution
else
   # build system execution
fi
}
```

Conditional Execution



Writing a New Recipe – Variants

- Add required variants to `BBCLASSEXTEND`:
  - native: Build for the build host
  - native-sdk: Build for the SDK

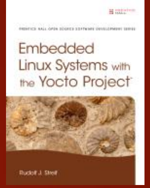


196-211

Lab Exercise

- Directly Building an Application
  - Create a recipe for this simple Hello World application by directly using the compile task to compile the three source files. Use ${CC} to invoke the compiler.
  - Build and test the application.
- Makefile-based Application
  - Create a makefile for the application.
  - Write a recipe to build the application with the makefile.
  - Build and test the application.

```
helloprint.h:
void printHello(void);

helloprint.c:
#include <stdio.h>
#include "helloprint.h"
void printHello(void) {
    printf("Hello, World! My first
Yocto Project recipe.\n");
    return;
}

hello.c:
#include "helloprint.h"
int main() {
    printHello();
    return(0);
}
```

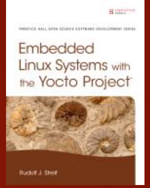


196-211

Devtool

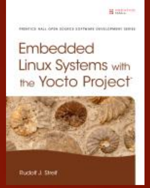
- Roundtrip development of recipes is greatly simplified using devtool directly from within your build environment.
- Devtool creates workspace layers, integrates them with your current build environment, lets you add new recipes and modify existing ones, launch a build and more.
- Devtool provides workflows for new and existing recipes.
- Devtool downloads and extracts sources into the workspace allowing you to modify them, create patches and add them to the recipe.
- Deploy a software package directly to a running target which can be QEMU or an actual hardware target.

Devtool – New Recipe Workflow



196-211

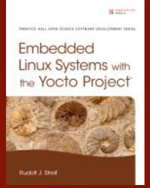
- Create workspace layer:
  - `devtool create-workspace [layerpath]`
- Add a new recipe to the workspace layer:
  - `devtool add <recipe-name> <source-path>`
  - `devtool add <recipt-name> <source-path> -f <source-uri>`
  - `<source-path>` points to a local source directory containing the sources. If you provide `-f <source-uri>`, devtool fetches the sources from the provided location and extracts them into `<source-path>`.
- Build the recipe:
  - `devtool build <recipe-name>`
- Deploy the software package to a running target:
  - `devtool deploy-target <recipe-name> [user@]target-host[:destdir]`
- Build an image:
  - `devtool build-image <image-name>`



Devtool – Existing Recipe Workflow

- Add an existing recipe from any layer to your workspace:
  - `devtool modify -x <recipe-name> <source-path>`
  - For example: `modify -x sqlite3 src/sqlite3`
- Make changes to devtoolthe sources and build the package:
  - `devtool build sqlite3`
- Create a patch and update the recipe:
  - `git add .`
  - `git commit -s`
  - `devtool update-recipe <recipe-name>` (update the original layer)
  - `devtool update-recipe <recipe-name> -a <layer-dir>` (create append file in <layer-dir)

Lab Exercise



- Create a workspace layer.
- Create a new recipe with devtool:
  - For the nano editor: https://nano-editor.org/dist/v2.6/nano-2.6.1.tar.gz
  - Or use sources for your own project
  - Build the recipe and deploy the package to QEMU.
- Modify an existing software package:
  - Sqlite3 or any package of your choice
  - Make changes to the source code (i.e. change the prompt of it).
  - Build the recipe and deploy the package to QEMU.
  - Create a patch.