Scott Rifenbark, Scotty's Documentation Services, INC **<srifenbark@gmail.com>**

by Scott Rifenbark

Manual Notes

- This version of the Yocto Project Overview and Concepts Manual is for the 3.0.3 release of the Yocto Project. To be sure you have the latest version of the manual for this release, go to the Yocto Project documentation page [http://www.yoctoproject.org/documentation] and select the manual from that site. Manuals from the site are more up-to-date than manuals derived from the Yocto Project released TAR files.

- If you located this manual through a web search, the version of the manual might not be the one you want (e.g. the search might have returned a manual much older than the Yocto Project version with which you are working). You can see all Yocto Project major releases by visiting the Releases [https://wiki.yoctoproject.org/wiki/Releases] page. If you need a version of this manual for a different Yocto Project release, visit the Yocto Project documentation page [http://www.yoctoproject.org/documentation] and select the manual set by using the "ACTIVE RELEASES DOCUMENTATION" or "DOCUMENTS ARCHIVE" pull-down menus.

- To report any inaccuracies or problems with this manual, send an email to the Yocto Project discussion group at `yocto@yoctoproject.com` or log into the freenode `#yocto` channel.

Table of Contents

Chapter 1. The Yocto Project Overview and Concepts Manual

1.1. Welcome

Welcome to the Yocto Project Overview and Concepts Manual! This manual introduces the Yocto Project by providing concepts, software overviews, best-known-methods (BKMs), and any other high-level introductory information suitable for a new Yocto Project user.

The following list describes what you can get from this manual:

- Introducing the Yocto Project: This chapter provides an introduction to the Yocto Project. You will learn about features and challenges of the Yocto Project, the layer model, components and tools, development methods, the Poky [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#poky] reference distribution, the OpenEmbedded build system workflow, and some basic Yocto terms.

- The Yocto Project Development Environment: This chapter helps you get started understanding the Yocto Project development environment. You will learn about open source, development hosts, Yocto Project source repositories, workflows using Git and the Yocto Project, a Git primer, and information about licensing.

- Yocto Project Concepts: This chapter presents various concepts regarding the Yocto Project. You can find conceptual information about components, development, cross-toolchains, and so forth.

This manual does not give you the following:

- Step-by-step Instructions for Development Tasks: Instructional procedures reside in other manuals within the Yocto Project documentation set. For example, the Yocto Project Development Tasks Manual [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html] provides examples on how to perform various development tasks. As another example, the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html] manual contains detailed instructions on how to install an SDK, which is used to develop applications for target hardware.

- Reference Material: This type of material resides in an appropriate reference manual. For example, system variables are documented in the Yocto Project Reference Manual [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html]. As another example, the Yocto Project Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html] contains reference information on BSPs.

- Detailed Public Information Not Specific to the Yocto Project: For example, exhaustive information on how to use the Source Control Manager Git is better covered with Internet searches and official Git Documentation than through the Yocto Project documentation.

1.2. Other Information

Because this manual presents information for many different topics, supplemental information is recommended for full comprehension. For additional introductory information on the Yocto Project, see the Yocto Project Website [http://www.yoctoproject.org]. If you want to build an image with no knowledge of Yocto Project as a way of quickly testing it out, see the Yocto Project Quick Build [http://www.yoctoproject.org/docs/3.0.3/brief-yoctoprojectqs/brief-yoctoprojectqs.html] document. For a comprehensive list of links and other documentation, see the "Links and Related Documentation [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#resources-links-and-related-documentation]" section in the Yocto Project Reference Manual.

Chapter 2. Introducing the Yocto Project

2.1. What is the Yocto Project?

The Yocto Project is an open source collaboration project that helps developers create custom Linux-based systems that are designed for embedded products regardless of the product's hardware architecture. Yocto Project provides a flexible toolset and a development environment that allows embedded device developers across the world to collaborate through shared technologies, software stacks, configurations, and best practices used to create these tailored Linux images.

Thousands of developers worldwide have discovered that Yocto Project provides advantages in both systems and applications development, archival and management benefits, and customizations used for speed, footprint, and memory utilization. The project is a standard when it comes to delivering embedded software stacks. The project allows software customizations and build interchange for multiple hardware platforms as well as software stacks that can be maintained and scaled.

For further introductory information on the Yocto Project, you might be interested in this article [https://www.embedded.com/electronics-blogs/say-what-/4458600/Why-the-Yocto-Project-for-my-IoT-Project-] by Drew Moseley and in this short introductory video [https://www.youtube.com/watch?v=utZpKM7i5Z4].

The remainder of this section overviews advantages and challenges tied to the Yocto Project.

2.1.1. Features

The following list describes features and advantages of the Yocto Project:

- Widely Adopted Across the Industry: Semiconductor, operating system, software, and service vendors exist whose products and services adopt and support the Yocto Project. For a look at the Yocto Project community and the companies involved with the Yocto Project, see the "COMMUNITY" and "ECOSYSTEM" tabs on the Yocto Project [http://www.yoctoproject.org] home page.

- Architecture Agnostic: Yocto Project supports Intel, ARM, MIPS, AMD, PPC and other architectures. Most ODMs, OSVs, and chip vendors create and supply BSPs that support their hardware. If you have custom silicon, you can create a BSP that supports that architecture.

  Aside from lots of architecture support, the Yocto Project fully supports a wide range of device emulation through the Quick EMUlator (QEMU).

- Images and Code Transfer Easily: Yocto Project output can easily move between architectures without moving to new development environments. Additionally, if you have used the Yocto Project to create an image or application and you find yourself not able to support it, commercial Linux vendors such as Wind River, Mentor Graphics, Timesys, and ENEA could take it and provide ongoing support. These vendors have offerings that are built using the Yocto Project.

- Flexibility: Corporations use the Yocto Project many different ways. One example is to create an internal Linux distribution as a code base the corporation can use across multiple product groups. Through customization and layering, a project group can leverage the base Linux distribution to create a distribution that works for their product needs.

- Ideal for Constrained Embedded and IoT devices: Unlike a full Linux distribution, you can use the Yocto Project to create exactly what you need for embedded devices. You only add the feature support or packages that you absolutely need for the device. For devices that have display hardware, you can use available system components such as X11, GTK+, Qt, Clutter, and SDL (among others) to create a rich user experience. For devices that do not have a display or where you want to use alternative UI frameworks, you can choose to not install these components.

- Comprehensive Toolchain Capabilities: Toolchains for supported architectures satisfy most use cases. However, if your hardware supports features that are not part of a standard toolchain, you

can easily customize that toolchain through specification of platform-specific tuning parameters. And, should you need to use a third-party toolchain, mechanisms built into the Yocto Project allow for that.

- Mechanism Rules Over Policy: Focusing on mechanism rather than policy ensures that you are free to set policies based on the needs of your design instead of adopting decisions enforced by some system software provider.
- Uses a Layer Model: The Yocto Project layer infrastructure groups related functionality into separate bundles. You can incrementally add these grouped functionalities to your project as needed. Using layers to isolate and group functionality reduces project complexity and redundancy, allows you to easily extend the system, make customizations, and keep functionality organized.
- Supports Partial Builds: You can build and rebuild individual packages as needed. Yocto Project accomplishes this through its shared-state cache (sstate) scheme. Being able to build and debug components individually eases project development.
- Releases According to a Strict Schedule: Major releases occur on a six-month cycle [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-release-process] predictably in October and April. The most recent two releases support point releases to address common vulnerabilities and exposures. This predictability is crucial for projects based on the Yocto Project and allows development teams to plan activities.
- Rich Ecosystem of Individuals and Organizations: For open source projects, the value of community is very important. Support forums, expertise, and active developers who continue to push the Yocto Project forward are readily available.
- Binary Reproducibility: The Yocto Project allows you to be very specific about dependencies and achieves very high percentages of binary reproducibility (e.g. 99.8% for `core-image-minimal`). When distributions are not specific about which packages are pulled in and in what order to support dependencies, other build systems can arbitrarily include packages.
- License Manifest: The Yocto Project provides a license manifest [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#maintaining-open-source-license-compliance-during-your-products-lifecycle] for review by people who need to track the use of open source licenses (e.g.legal teams).

2.1.2. Challenges

The following list presents challenges you might encounter when developing using the Yocto Project:

- Steep Learning Curve: The Yocto Project has a steep learning curve and has many different ways to accomplish similar tasks. It can be difficult to choose how to proceed when varying methods exist by which to accomplish a given task.
- Understanding What Changes You Need to Make For Your Design Requires Some Research: Beyond the simple tutorial stage, understanding what changes need to be made for your particular design can require a significant amount of research and investigation. For information that helps you transition from trying out the Yocto Project to using it for your project, see the "What I wish I'd Known [http://www.yoctoproject.org/docs/what-i-wish-id-known/]" and "Transitioning to a Custom Environment for Systems Development [http://www.yoctoproject.org/docs/transitioning-to-a-custom-environment/]" documents on the Yocto Project website.
- Project Workflow Could Be Confusing: The Yocto Project workflow could be confusing if you are used to traditional desktop and server software development. In a desktop development environment, mechanisms exist to easily pull and install new packages, which are typically pre-compiled binaries from servers accessible over the Internet. Using the Yocto Project, you must modify your configuration and rebuild to add additional packages.
- Working in a Cross-Build Environment Can Feel Unfamiliar: When developing code to run on a target, compilation, execution, and testing done on the actual target can be faster than running a BitBake build on a development host and then deploying binaries to the target for test. While the Yocto Project does support development tools on the target, the additional step of integrating your changes back into the Yocto Project build environment would be required. Yocto Project supports an intermediate approach that involves making changes on the development system within the BitBake environment and then deploying only the updated packages to the target.

The Yocto Project OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] produces packages in standard formats (i.e. RPM, DEB, IPK, and TAR). You can deploy these packages into the running system on the target by using utilities on the target such as `rpm` or `ipk`.

- Initial Build Times Can be Significant: Long initial build times are unfortunately unavoidable due to the large number of packages initially built from scratch for a fully functioning Linux system. Once that initial build is completed, however, the shared-state (sstate) cache mechanism Yocto Project uses keeps the system from rebuilding packages that have not been "touched" since the last build. The sstate mechanism significantly reduces times for successive builds.

2.2. The Yocto Project Layer Model

The Yocto Project's "Layer Model" is a development model for embedded and IoT Linux creation that distinguishes the Yocto Project from other simple build systems. The Layer Model simultaneously supports collaboration and customization. Layers are repositories that contain related sets of instructions that tell the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] what to do. You can collaborate, share, and reuse layers.

Layers can contain changes to previous instructions or settings at any time. This powerful override capability is what allows you to customize previously supplied collaborative or community layers to suit your product requirements.

You use different layers to logically separate information in your build. As an example, you could have BSP, GUI, distro configuration, middleware, or application layers. Putting your entire build into one layer limits and complicates future customization and reuse. Isolating information into layers, on the other hand, helps simplify future customizations and reuse. You might find it tempting to keep everything in one layer when working on a single project. However, the more modular your Metadata, the easier it is to cope with future changes.

Notes

- Use Board Support Package (BSP) layers from silicon vendors when possible.
- Familiarize yourself with the Yocto Project curated layer index [https://caffelli-staging.yoctoproject.org/software-overview/layers/] or the OpenEmbedded layer index [http://layers.openembedded.org/layerindex/branch/master/layers/]. The latter contains more layers but they are less universally validated.
- Layers support the inclusion of technologies, hardware components, and software components. The Yocto Project Compatible [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#making-sure-your-layer-is-compatible-with-yocto-project] designation provides a minimum level of standardization that contributes to a strong ecosystem. "YP Compatible" is applied to appropriate products and software components such as BSPs, other OE-compatible layers, and related open-source projects, allowing the producer to use Yocto Project badges and branding assets.

To illustrate how layers are used to keep things modular, consider machine customizations. These types of customizations typically reside in a special layer, rather than a general layer, called a BSP Layer. Furthermore, the machine customizations should be isolated from recipes and Metadata that support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations, and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (`.bbappend`) file, which is described later in this section.

Note

For general information on BSP layer structure, see the Yocto Project Board Support Packages (BSP) Developer's Guide [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html].

The Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory] contains both general layers and BSP layers right out of the box. You can easily identify

layers that ship with a Yocto Project release in the Source Directory by their names. Layers typically have names that begin with the string `meta-`.

> **Note**
>
> It is not a requirement that a layer name begin with the prefix `meta-`, but it is a commonly accepted standard in the Yocto Project community.

For example, if you were to examine the tree view [https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/] of the poky repository, you will see several layers: `meta`, `meta-skeleton`, `meta-selftest`, `meta-poky`, and `meta-yocto-bsp`. Each of these repositories represents a distinct layer.

For procedures on how to create layers, see the "Understanding and Creating Layers [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#understanding-and-creating-layers]" section in the Yocto Project Development Tasks Manual.

2.3. Components and Tools

The Yocto Project employs a collection of components and tools used by the project itself, by project developers, and by those using the Yocto Project. These components and tools are open source projects and metadata that are separate from the reference distribution (Poky [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#poky]) and the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term]. Most of the components and tools are downloaded separately.

This section provides brief overviews of the components and tools associated with the Yocto Project.

2.3.1. Development Tools

The following list consists of tools that help you develop images and applications using the Yocto Project:

- CROPS: CROPS [https://git.yoctoproject.org/cgit/cgit.cgi/crops/about/] is an open source, cross-platform development framework that leverages Docker Containers [https://www.docker.com/]. CROPS provides an easily managed, extensible environment that allows you to build binaries for a variety of architectures on Windows, Linux and Mac OS X hosts.

- `devtool`: This command-line tool is available as part of the extensible SDK (eSDK) and is its cornerstone. You can use `devtool` to help build, test, and package software within the eSDK. You can use the tool to optionally integrate what you build into an image built by the OpenEmbedded build system.

  The `devtool` command employs a number of sub-commands that allow you to add, modify, and upgrade recipes. As with the OpenEmbedded build system, “recipes” represent software packages within `devtool`. When you use `devtool add`, a recipe is automatically created. When you use `devtool modify`, the specified existing recipe is used in order to determine where to get the source code and how to patch it. In both cases, an environment is set up so that when you build the recipe a source tree that is under your control is used in order to allow you to make changes to the source as desired. By default, both new recipes and the source go into a “workspace” directory under the eSDK. The `devtool upgrade` command updates an existing recipe so that you can build it for an updated set of source files.

  You can read about the `devtool` workflow in the Yocto Project Application Development and Extensible Software Development Kit (eSDK) Manual in the "Using `devtool` in Your SDK Workflow' [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html#using-devtool-in-your-sdk-workflow]" section.

- Extensible Software Development Kit (eSDK): The eSDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. The eSDK makes it easy to add new applications and libraries to an image, modify the source for an existing component, test changes on the target hardware, and integrate into the rest of the OpenEmbedded build system. The eSDK gives you a toolchain experience supplemented with the powerful set of `devtool` commands tailored for the Yocto Project environment.

  For information on the eSDK, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html] Manual.

- Toaster: Toaster is a web interface to the Yocto Project OpenEmbedded build system. Toaster allows you to configure, run, and view information about builds. For information on Toaster, see the Toaster User Manual [http://www.yoctoproject.org/docs/3.0.3/toaster-manual/toaster-manual.html].

2.3.2. Production Tools

The following list consists of tools that help production related activities using the Yocto Project:

- Auto Upgrade Helper: This utility when used in conjunction with the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] (BitBake and OE-Core) automatically generates upgrades for recipes that are based on new versions of the recipes published upstream.

- Recipe Reporting System: The Recipe Reporting System tracks recipe versions available for Yocto Project. The main purpose of the system is to help you manage the recipes you maintain and to offer a dynamic overview of the project. The Recipe Reporting System is built on top of the OpenEmbedded Layer Index [http://layers.openembedded.org/layerindex/layers/], which is a website that indexes OpenEmbedded-Core layers.

- Patchwork: Patchwork [http://jk.ozlabs.org/projects/patchwork/] is a fork of a project originally started by OzLabs [http://ozlabs.org/]. The project is a web-based tracking system designed to streamline the process of bringing contributions into a project. The Yocto Project uses Patchwork as an organizational tool to handle patches, which number in the thousands for every release.

- AutoBuilder: AutoBuilder is a project that automates build tests and quality assurance (QA). By using the public AutoBuilder, anyone can determine the status of the current "master" branch of Poky.

  > **Note**
  > AutoBuilder is based on buildbot [https://buildbot.net/].

  A goal of the Yocto Project is to lead the open source industry with a project that automates testing and QA procedures. In doing so, the project encourages a development community that publishes QA and test plans, publicly demonstrates QA and test plans, and encourages development of tools that automate and test and QA procedures for the benefit of the development community.

  You can learn more about the AutoBuilder used by the Yocto Project here [http://autobuilder.yoctoproject.org].

- Cross-Prelink: Prelinking is the process of pre-computing the load addresses and link tables generated by the dynamic linker as compared to doing this at runtime. Doing this ahead of time results in performance improvements when the application is launched and reduced memory usage for libraries shared by many applications.

  Historically, cross-prelink is a variant of prelink, which was conceived by Jakub Jelínek [http://people.redhat.com/jakub/prelink.pdf] a number of years ago. Both prelink and cross-prelink are maintained in the same repository albeit on separate branches. By providing an emulated runtime dynamic linker (i.e. `glibc`-derived `ld.so` emulation), the cross-prelink project extends the prelink software's ability to prelink a sysroot environment. Additionally, the cross-prelink software enables the ability to work in sysroot style environments.

  The dynamic linker determines standard load address calculations based on a variety of factors such as mapping addresses, library usage, and library function conflicts. The prelink tool uses this information, from the dynamic linker, to determine unique load addresses for executable and linkable format (ELF) binaries that are shared libraries and dynamically linked. The prelink tool modifies these ELF binaries with the pre-computed information. The result is faster loading and often lower memory consumption because more of the library code can be re-used from shared Copy-On-Write (COW) pages.

  The original upstream prelink project only supports running prelink on the end target device due to the reliance on the target device's dynamic linker. This restriction causes issues when developing a cross-compiled system. The cross-prelink adds a synthesized dynamic loader that runs on the host, thus permitting cross-prelinking without ever having to run on a read-write target filesystem.

- Pseudo: Pseudo is the Yocto Project implementation of fakeroot [http://man.he.net/man1/fakeroot], which is used to run commands in an environment that seemingly has root privileges.

During a build, it can be necessary to perform operations that require system administrator privileges. For example, file ownership or permissions might need definition. Pseudo is a tool that you can either use directly or through the environment variable `LD_PRELOAD`. Either method allows these operations to succeed as if system administrator privileges exist even when they do not.

You can read more about Pseudo in the "Fakeroot and Pseudo" section.

2.3.3. Open-Embedded Build System Components

The following list consists of components associated with the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term]:

- BitBake: BitBake is a core component of the Yocto Project and is used by the OpenEmbedded build system to build images. While BitBake is key to the build system, BitBake is maintained separately from the Yocto Project.

  BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. In short, BitBake is a build engine that works through recipes written in a specific format in order to perform sets of tasks.

  You can learn more about BitBake in the BitBake User Manual [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html].

- OpenEmbedded-Core: OpenEmbedded-Core (OE-Core) is a common layer of metadata (i.e. recipes, classes, and associated files) used by OpenEmbedded-derived systems, which includes the Yocto Project. The Yocto Project and the OpenEmbedded Project both maintain the OpenEmbedded-Core. You can find the OE-Core metadata in the Yocto Project Source Repositories [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta].

  Historically, the Yocto Project integrated the OE-Core metadata throughout the Yocto Project source repository reference system (Poky). After Yocto Project Version 1.0, the Yocto Project and OpenEmbedded agreed to work together and share a common core set of metadata (OE-Core), which contained much of the functionality previously found in Poky. This collaboration achieved a long-standing OpenEmbedded objective for having a more tightly controlled and quality-assured core. The results also fit well with the Yocto Project objective of achieving a smaller number of fully featured tools as compared to many different ones.

  Sharing a core set of metadata results in Poky as an integration layer on top of OE-Core. You can see that in this figure. The Yocto Project combines various components such as BitBake, OE-Core, script “glue”, and documentation for its build system.

2.3.4. Reference Distribution (Poky)

Poky is the Yocto Project reference distribution. It contains the Open-Embedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] (BitBake and OE-Core) as well as a set of metadata to get you started building your own distribution. See the figure in "What is the Yocto Project?" section for an illustration that shows Poky and its relationship with other parts of the Yocto Project.

To use the Yocto Project tools and components, you can download (`clone`) Poky and use it to bootstrap your own distribution.

Note

Poky does not contain binary files. It is a working example of how to build your own custom Linux distribution from source.

You can read more about Poky in the "Reference Embedded Distribution (Poky)" section.

2.3.5. Packages for Finished Targets

The following lists components associated with packages for finished targets:

- Matchbox: Matchbox is an Open Source, base environment for the X Window System running on non-desktop, embedded platforms such as handhelds, set-top boxes, kiosks, and anything else for which screen space, input mechanisms, or system resources are limited.

Matchbox consists of a number of interchangeable and optional applications that you can tailor to a specific, non-desktop platform to enhance usability in constrained environments.

You can find the Matchbox source in the Yocto Project Source Repositories [http://git.yoctoproject.org].

- Opkg Open PacKaGe management (opkg) is a lightweight package management system based on the itsy package (ipkg) management system. Opkg is written in C and resembles Advanced Package Tool (APT) and Debian Package (dpkg) in operation.

  Opkg is intended for use on embedded Linux devices and is used in this capacity in the OpenEmbedded [http://www.openembedded.org/wiki/Main_Page] and OpenWrt [https://openwrt.org/] projects, as well as the Yocto Project.

  > **Note**
  > As best it can, opkg maintains backwards compatibility with ipkg and conforms to a subset of Debian's policy manual regarding control files.

2.3.6. Archived Components

The Build Appliance is a virtual machine image that enables you to build and boot a custom embedded Linux image with the Yocto Project using a non-Linux development system.

Historically, the Build Appliance was the second of three methods by which you could use the Yocto Project on a system that was not native to Linux.

1. Hob: Hob, which is now deprecated and is no longer available since the 2.1 release of the Yocto Project provided a rudimentary, GUI-based interface to the Yocto Project. Toaster has fully replaced Hob.

2. Build Appliance: Post Hob, the Build Appliance became available. It was never recommended that you use the Build Appliance as a day-to-day production development environment with the Yocto Project. Build Appliance was useful as a way to try out development in the Yocto Project environment.

3. CROPS: The final and best solution available now for developing using the Yocto Project on a system not native to Linux is with CROPS.

2.4. Development Methods

The Yocto Project development environment usually involves a Build Host [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#hardware-build-system-term] and target hardware. You use the Build Host to build images and develop applications, while you use the target hardware to test deployed software.

This section provides an introduction to the choices or development methods you have when setting up your Build Host. Depending on the your particular workflow preference and the type of operating system your Build Host runs, several choices exist that allow you to use the Yocto Project.

> **Note**
> For additional detail about the Yocto Project development environment, see the "The Yocto Project Development Environment" chapter.

- Native Linux Host: By far the best option for a Build Host. A system running Linux as its native operating system allows you to develop software by directly using the BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] tool. You can accomplish all aspects of development from a familiar shell of a supported Linux distribution.

  For information on how to set up a Build Host on a system running Linux as its native operating system, see the "Setting Up a Native Linux Host [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#setting-up-a-native-linux-host]" section in the Yocto Project Development Tasks Manual.

- CROss PlatformS (CROPS): Typically, you use CROPS [https://git.yoctoproject.org/cgit/cgit.cgi/crops/about/], which leverages Docker Containers [https://www.docker.com/], to set up a Build Host that is not running Linux (e.g. Microsoft® Windows™ or macOS®).

  > **Note**
  >
  > You can, however, use CROPS on a Linux-based system.

  CROPS is an open source, cross-platform development framework that provides an easily managed, extensible environment for building binaries targeted for a variety of architectures on Windows, macOS, or Linux hosts. Once the Build Host is set up using CROPS, you can prepare a shell environment to mimic that of a shell being used on a system natively running Linux.

  For information on how to set up a Build Host with CROPS, see the "Setting Up to Use CROss PlatformS (CROPS) [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#setting-up-to-use-crops]" section in the Yocto Project Development Tasks Manual.

- Toaster: Regardless of what your Build Host is running, you can use Toaster to develop software using the Yocto Project. Toaster is a web interface to the Yocto Project's Open-Embedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term]. The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. You can use Toaster to configure and start builds on multiple remote build servers.

  For information about and how to use Toaster, see the Toaster User Manual [http://www.yoctoproject.org/docs/3.0.3/toaster-manual/toaster-manual.html].

2.5. Reference Embedded Distribution (Poky)

"Poky", which is pronounced Pock-ee, is the name of the Yocto Project's reference distribution or Reference OS Kit. Poky contains the OpenEmbedded Build System [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] (BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] and OpenEmbedded-Core [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#oe-core]) as well as a set of metadata [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#metadata] to get you started building your own distro. In other words, Poky is a base specification of the functionality needed for a typical embedded system as well as the components from the Yocto Project that allow you to build a distribution into a usable binary image.

Poky is a combined repository of BitBake, OpenEmbedded-Core (which is found in `meta`), `meta-poky`, `meta-yocto-bsp`, and documentation provided all together and known to work well together. You can view these items that make up the Poky repository in the Source Repositories [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/].

> **Note**
>
> If you are interested in all the contents of the `poky` Git repository, see the "Top-Level Core Components [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#structure-core]" section in the Yocto Project Reference Manual.

The following figure illustrates what generally comprises Poky:

- BitBake is a task executor and scheduler that is the heart of the OpenEmbedded build system.
- `meta-poky`, which is Poky-specific metadata.
- `meta-yocto-bsp`, which are Yocto Project-specific Board Support Packages (BSPs).
- OpenEmbedded-Core (OE-Core) metadata, which includes shared configurations, global variable definitions, shared classes, packaging, and recipes. Classes define the encapsulation and inheritance of build logic. Recipes are the logical units of software and images to be built.
- Documentation, which contains the Yocto Project source files used to make the set of user manuals.

> **Note**
>
> While Poky is a "complete" distribution specification and is tested and put through QA, you cannot use it as a product "out of the box" in its current form.

To use the Yocto Project tools, you can use Git to clone (download) the Poky repository then use your local copy of the reference distribution to bootstrap your own distribution.

> **Note**
> Poky does not contain binary files. It is a working example of how to build your own custom Linux distribution from source.

Poky has a regular, well established, six-month release cycle under its own version. Major releases occur at the same time major releases (point releases) occur for the Yocto Project, which are typically in the Spring and Fall. For more information on the Yocto Project release schedule and cadence, see the "Yocto Project Releases and the Stable Release Process [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-release-process]" chapter in the Yocto Project Reference Manual.

Much has been said about Poky being a "default configuration." A default configuration provides a starting image footprint. You can use Poky out of the box to create an image ranging from a shell-accessible minimal image all the way up to a Linux Standard Base-compliant image that uses a GNOME Mobile and Embedded (GMAE) based reference user interface called Sato.

One of the most powerful properties of Poky is that every aspect of a build is controlled by the metadata. You can use metadata to augment these base image types by adding metadata layers that extend functionality. These layers can provide, for example, an additional software stack for an image type, add a board support package (BSP) for additional hardware, or even create a new image type.

Metadata is loosely grouped into configuration files or package recipes. A recipe is a collection of non-executable metadata used by BitBake to set variables or define additional build-time tasks. A recipe contains fields such as the recipe description, the recipe version, the license of the package and the upstream source repository. A recipe might also indicate that the build process uses autotools, make, distutils or any other build process, in which case the basic functionality can be defined by the classes it inherits from the OE-Core layer's class definitions in `./meta/classes`. Within a recipe you can also define additional tasks as well as task prerequisites. Recipe syntax through BitBake also supports both `_prepend` and `_append` operators as a method of extending task functionality. These operators inject code into the beginning or end of a task. For information on these BitBake operators, see the "Appending and Prepending (Override Style Syntax) [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#appending-and-prepending-override-style-syntax]" section in the BitBake User's Manual.

2.6. The OpenEmbedded Build System Workflow

The OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] uses a "workflow" to accomplish image and SDK generation. The following figure overviews that workflow:

Following is a brief summary of the "workflow":

1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location. The build system supports standard methods such as tarballs or source code repositories systems such as Git.
3. Once source code is downloaded, the build system extracts the sources into a local work area where patches are applied and common steps for configuring and compiling the software are run.
4. The build system then installs the software into a temporary staging area where the binary package format you select (DEB, RPM, or IPK) is used to roll up the software.
5. Different QA and sanity checks run throughout entire build process.
6. After the binaries are created, the build system generates a binary package feed that is used to create the final root file image.
7. The build system generates the file system image and a customized Extensible SDK (eSDK) for application development in parallel.

For a very detailed look at this workflow, see the "OpenEmbedded Build System Concepts" section.

2.7. Some Basic Terms

It helps to understand some basic fundamental terms when learning the Yocto Project. Although a list of terms exists in the "Yocto Project Terms [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-terms]" section of the Yocto Project Reference Manual, this section provides the definitions of some terms helpful for getting started:

- Configuration Files: Files that hold global definitions of variables, user-defined variables, and hardware configuration information. These files tell the Open-Embedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] what to build and what to put into the image to support a particular platform.

- Extensible Software Development Kit (eSDK): A custom SDK for application developers. This eSDK allows developers to incorporate their library and programming changes back into the image to make their code available to other application developers. For information on the eSDK, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html] manual.

- Layer: A collection of related recipes. Layers allow you to consolidate related metadata to customize your build. Layers also isolate information used when building for multiple architectures. Layers are hierarchical in their ability to override previous specifications. You can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. You can search the Layer Index for layers used within Yocto Project.

  For more detailed information on layers, see the "Understanding and Creating Layers [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#understanding-and-creating-layers]" section in the Yocto Project Development Tasks Manual. For a discussion specifically on BSP Layers, see the "BSP Layers [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html#bsp-layers]" section in the Yocto Project Board Support Packages (BSP) Developer's Guide.

- Metadata: A key element of the Yocto Project is the Metadata that is used to construct a Linux distribution and is contained in the files that the OpenEmbedded build system parses when building an image. In general, Metadata includes recipes, configuration files, and other information that refers to the build instructions themselves, as well as the data used to control what things get built and the effects of the build. Metadata also includes commands and data used to indicate what versions of software are used, from where they are obtained, and changes or additions to the software itself (patches or auxiliary files) that are used to fix bugs or customize the software for use in a particular situation. OpenEmbedded-Core is an important set of validated metadata.

- OpenEmbedded Build System: The terms "BitBake" and "build system" are sometimes used for the OpenEmbedded Build System.

  BitBake is a task scheduler and execution engine that parses instructions (i.e. recipes) and configuration data. After a parsing phase, BitBake creates a dependency tree to order the compilation, schedules the compilation of the included code, and finally executes the building of the specified custom Linux image (distribution). BitBake is similar to the `make` tool.

  During a build process, the build system tracks dependencies and performs a native or cross-compilation of the package. As a first step in a cross-build setup, the framework attempts to create a cross-compiler toolchain (i.e. Extensible SDK) suited for the target platform.

- OpenEmbedded-Core (OE-Core): OE-Core is metadata comprised of foundation recipes, classes, and associated files that are meant to be common among many different OpenEmbedded-derived systems, including the Yocto Project. OE-Core is a curated subset of an original repository developed by the OpenEmbedded community that has been pared down into a smaller, core set of continuously validated recipes. The result is a tightly controlled and quality-assured core set of recipes.

  You can see the Metadata in the `meta` directory of the Yocto Project Source Repositories [http://git.yoctoproject.org/cgit/cgit.cgi].

- Packages: In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e. a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. You "bake" something by running it through BitBake.

It is worth noting that the term "package" can, in general, have subtle meanings. For example, the packages referred to in the "Required Packages for the Build Host [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#required-packages-for-the-build-host]" section in the Yocto Project Reference Manual are compiled binaries that, when installed, add functionality to your Linux distribution.

Another point worth noting is that historically within the Yocto Project, recipes were referred to as packages - thus, the existence of several BitBake variables that are seemingly mis-named, (e.g. PR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PR], PV [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PV], and PE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PE]).

- Poky: Poky is a reference embedded distribution and a reference test configuration. Poky provides the following:
  - A base-level functional distro used to illustrate how to customize a distribution.
  - A means by which to test the Yocto Project components (i.e. Poky is used to validate the Yocto Project).
  - A vehicle through which you can download the Yocto Project.

  Poky is not a product level distro. Rather, it is a good starting point for customization.

  > **Note**
  > Poky is an integration layer on top of OE-Core.

- Recipe: The most common form of metadata. A recipe contains a list of settings and tasks (i.e. instructions) for building packages that are then used to build the binary image. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes as well as configuration and compilation options. Related recipes are consolidated into a layer.

Chapter 3. The Yocto Project Development Environment

This chapter takes a look at the Yocto Project development environment. The chapter provides Yocto Project Development environment concepts that help you understand how work is accomplished in an open source environment, which is very different as compared to work accomplished in a closed, proprietary environment.

Specifically, this chapter addresses open source philosophy, source repositories, workflows, Git, and licensing.

3.1. Open Source Philosophy

Open source philosophy is characterized by software development directed by peer production and collaboration through an active community of developers. Contrast this to the more standard centralized development models used by commercial software companies where a finite set of developers produces a product for sale using a defined set of procedures that ultimately result in an end product whose architecture and source material are closed to the public.

Open source projects conceptually have differing concurrent agendas, approaches, and production. These facets of the development process can come from anyone in the public (community) who has a stake in the software project. The open source environment contains new copyright, licensing, domain, and consumer issues that differ from the more traditional development environment. In an open source environment, the end product, source material, and documentation are all available to the public at no cost.

A benchmark example of an open source project is the Linux kernel, which was initially conceived and created by Finnish computer science student Linus Torvalds in 1991. Conversely, a good example of a non-open source project is the Windows® family of operating systems developed by Microsoft® Corporation.

Wikipedia has a good historical description of the Open Source Philosophy here [http://en.wikipedia.org/wiki/Open_source]. You can also find helpful information on how to participate in the Linux Community here [http://ldn.linuxfoundation.org/book/how-participate-linux-community].

3.2. The Development Host

A development host or build host [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#hardware-build-system-term] is key to using the Yocto Project. Because the goal of the Yocto Project is to develop images or applications that run on embedded hardware, development of those images and applications generally takes place on a system not intended to run the software - the development host.

You need to set up a development host in order to use it with the Yocto Project. Most find that it is best to have a native Linux machine function as the development host. However, it is possible to use a system that does not run Linux as its operating system as your development host. When you have a Mac or Windows-based system, you can set it up as the development host by using CROPS [https://git.yoctoproject.org/cgit/cgit.cgi/crops/about/], which leverages Docker Containers [https://www.docker.com/]. Once you take the steps to set up a CROPS machine, you effectively have access to a shell environment that is similar to what you see when using a Linux-based development host. For the steps needed to set up a system using CROPS, see the "Setting Up to Use CROss PlatformS (CROPS) [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#setting-up-to-use-crops]" section in the Yocto Project Development Tasks Manual.

If your development host is going to be a system that runs a Linux distribution, steps still exist that you must take to prepare the system for use with the Yocto Project. You need to be sure that the Linux distribution on the system is one that supports the Yocto Project. You also need to be sure that the correct set of host packages are installed that allow development using the Yocto Project. For the steps needed to set up a development host that runs Linux, see the "Setting Up a

Native Linux Host [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#setting-up-a-native-linux-host]" section in the Yocto Project Development Tasks Manual.

Once your development host is set up to use the Yocto Project, several methods exist for you to do work in the Yocto Project environment:

- Command Lines, BitBake, and Shells: Traditional development in the Yocto Project involves using the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term], which uses BitBake, in a command-line environment from a shell on your development host. You can accomplish this from a host that is a native Linux machine or from a host that has been set up with CROPS. Either way, you create, modify, and build images and applications all within a shell-based environment using components and tools available through your Linux distribution and the Yocto Project.

  For a general flow of the build procedures, see the "Building a Simple Image [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#dev-building-a-simple-image]" section in the Yocto Project Development Tasks Manual.

- Board Support Package (BSP) Development: Development of BSPs involves using the Yocto Project to create and test layers that allow easy development of images and applications targeted for specific hardware. To development BSPs, you need to take some additional steps beyond what was described in setting up a development host.

  The Yocto Project Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html] provides BSP-related development information. For specifics on development host preparation, see the "Preparing Your Build Host to Work With BSP Layers [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html#preparing-your-build-host-to-work-with-bsp-layers]" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

- Kernel Development: If you are going to be developing kernels using the Yocto Project you likely will be using `devtool`. A workflow using `devtool` makes kernel development quicker by reducing iteration cycle times.

  The Yocto Project Linux Kernel Development Manual [http://www.yoctoproject.org/docs/3.0.3/kernel-dev/kernel-dev.html] provides kernel-related development information. For specifics on development host preparation, see the "Preparing the Build Host to Work on the Kernel [http://www.yoctoproject.org/docs/3.0.3/kernel-dev/kernel-dev.html#preparing-the-build-host-to-work-on-the-kernel]" section in the Yocto Project Linux Kernel Development Manual.

- Using Toaster: The other Yocto Project development method that involves an interface that effectively puts the Yocto Project into the background is Toaster. Toaster provides an interface to the OpenEmbedded build system. The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. You can use Toaster to configure and start builds on multiple remote build servers.

  For steps that show you how to set up your development host to use Toaster and on how to use Toaster in general, see the Toaster User Manual [http://www.yoctoproject.org/docs/3.0.3/toaster-manual/toaster-manual.html].

3.3. Yocto Project Source Repositories

The Yocto Project team maintains complete source repositories for all Yocto Project files at http://git.yoctoproject.org. This web-based source code browser is organized into categories by function such as IDE Plugins, Matchbox, Poky, Yocto Linux Kernel, and so forth. From the interface, you can click on any particular item in the "Name" column and see the URL at the bottom of the page that you need to clone a Git repository for that particular item. Having a local Git repository of the Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory], which is usually named "poky", allows you to make changes, contribute to the history, and ultimately enhance the Yocto Project's tools, Board Support Packages, and so forth.

For any supported release of Yocto Project, you can also go to the Yocto Project Website [http://www.yoctoproject.org] and select the "DOWNLOADS" item from the "SOFTWARE" menu and get a released tarball of the `poky` repository, any supported BSP tarball, or Yocto Project tools. Unpacking these tarballs gives you a snapshot of the released files.

Notes

- The recommended method for setting up the Yocto Project Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory] and the files for supported BSPs (e.g., `meta-intel`) is to use Git to create a local copy of the upstream repositories.
- Be sure to always work in matching branches for both the selected BSP repository and the Source Directory (i.e. poky) repository. For example, if you have checked out the "master" branch of poky and you are going to use `meta-intel`, be sure to checkout the "master" branch of `meta-intel`.

In summary, here is where you can get the project files needed for development:

- Source Repositories: [http://git.yoctoproject.org] This area contains IDE Plugins, Matchbox, Poky, Poky Support, Tools, Yocto Linux Kernel, and Yocto Metadata Layers. You can create local copies of Git repositories for each of these areas.

  For steps on how to view and access these upstream Git repositories, see the "Accessing Source Repositories [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#accessing-source-repositories]" Section in the Yocto Project Development Tasks Manual.

- Index of /releases: [http://downloads.yoctoproject.org/releases/] This is an index of releases such as Poky, Pseudo, installers for cross-development toolchains, miscellaneous support and all released versions of Yocto Project in the form of images or tarballs. Downloading and extracting these files does not produce a local copy of the Git repository but rather a snapshot of a particular release or image.

For steps on how to view and access these files, see the "Accessing Index of Releases [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#accessing-index-of-releases]" section in the Yocto Project Development Tasks Manual.

- "DOWNLOADS" page for the Yocto Project Website [http://www.yoctoproject.org]:

  The Yocto Project website includes a "DOWNLOADS" page accessible through the "SOFTWARE" menu that allows you to download any Yocto Project release, tool, and Board Support Package (BSP) in tarball form. The tarballs are similar to those found in the Index of /releases: [http://downloads.yoctoproject.org/releases/] area.

  For steps on how to use the "DOWNLOADS" page, see the "Using the

Downloads Page [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#using-the-downloads-page]" section in the Yocto Project Development Tasks Manual.

3.4. Git Workflows and the Yocto Project

Developing using the Yocto Project likely requires the use of Git. Git is a free, open source distributed version control system used as part of many collaborative design environments. This section provides workflow concepts using the Yocto Project and Git. In particular, the information covers basic practices that describe roles and actions in a collaborative development environment.

Note

If you are familiar with this type of development environment, you might not want to read this section.

The Yocto Project files are maintained using Git in "branches" whose Git histories track every change and whose structures provide branches for all diverging functionality. Although there is no need to use Git, many open source projects do so.

For the Yocto Project, a key individual called the "maintainer" is responsible for the integrity of the "master" branch of a given Git repository. The "master" branch is the “upstream” repository from which final or most recent builds of a project occur. The maintainer is responsible for accepting changes from other developers and for organizing the underlying branch structure to reflect release strategies and so forth.

Note

For information on finding out who is responsible for (maintains) a particular area of code in the Yocto Project, see the "Submitting a Change to the Yocto Project [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#how-to-submit-a-change]" section of the Yocto Project Development Tasks Manual.

The Yocto Project poky Git repository also has an upstream contribution Git repository named `poky-contrib`. You can see all the branches in this repository using the web interface of the Source Repositories [http://git.yoctoproject.org] organized within the "Poky Support" area. These branches hold changes (commits) to the project that have been submitted or committed by the Yocto Project development team and by community members who contribute to the project. The maintainer determines if the changes are qualified to be moved from the "contrib" branches into the "master" branch of the Git repository.

Developers (including contributing community members) create and maintain cloned repositories of upstream branches. The cloned repositories are local to their development platforms and are used to develop changes. When a developer is satisfied with a particular feature or change, they "push" the change to the appropriate "contrib" repository.

Developers are responsible for keeping their local repository up-to-date with whatever upstream branch they are working against. They are also responsible for straightening out any conflicts that might arise within files that are being worked on simultaneously by more than one person. All this work is done locally on the development host before anything is pushed to a "contrib" area and examined at the maintainer’s level.

A somewhat formal method exists by which developers commit changes and push them into the "contrib" area and subsequently request that the maintainer include them into an upstream branch. This process is called “submitting a patch” or "submitting a change." For information on submitting patches and changes, see the "Submitting a Change to the Yocto Project [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#how-to-submit-a-change]" section in the Yocto Project Development Tasks Manual.

In summary, a single point of entry exists for changes into a "master" or development branch of the Git repository, which is controlled by the project’s maintainer. And, a set of developers exist who independently develop, test, and submit changes to "contrib" areas for the maintainer to examine. The maintainer then chooses which changes are going to become a permanent part of the project.

While each development environment is unique, there are some best practices or methods that help development run smoothly. The following list describes some of these practices. For more information about Git workflows, see the workflow topics in the Git Community Book [http://book.git-scm.com].

- Make Small Changes: It is best to keep the changes you commit small as compared to bundling many disparate changes into a single commit. This practice not only keeps things manageable but also allows the maintainer to more easily include or refuse changes.

- Make Complete Changes: It is also good practice to leave the repository in a state that allows you to still successfully build your project. In other words, do not commit half of a feature, then add the other half as a separate, later commit. Each commit should take you from one buildable project state to another buildable state.

- Use Branches Liberally: It is very easy to create, use, and delete local branches in your working Git repository on the development host. You can name these branches anything you like. It is helpful to give them names associated with the particular feature or change on which you are working. Once you are done with a feature or change and have merged it into your local master branch, simply discard the temporary branch.

- Merge Changes: The `git merge` command allows you to take the changes from one branch and fold them into another branch. This process is especially helpful when more than a single developer might be working on different parts of the same feature. Merging changes also automatically identifies any collisions or "conflicts" that might happen as a result of the same lines of code being altered by two different developers.

- Manage Branches: Because branches are easy to use, you should use a system where branches indicate varying levels of code readiness. For example, you can have a "work" branch to develop in, a "test" branch where the code or change is tested, a "stage" branch where changes are ready to be committed, and so forth. As your project develops, you can merge code across the branches to reflect ever-increasing stable states of the development.

- Use Push and Pull: The push-pull workflow is based on the concept of developers "pushing" local commits to a remote repository, which is usually a contribution repository. This workflow is also based on developers "pulling" known states of the project down into their local development repositories. The workflow easily allows you to pull changes submitted by other developers from the upstream repository into your work area ensuring that you have the most recent software on which to develop. The Yocto Project has two scripts named `create-pull-request` and `send-pull-request` that ship with the release to facilitate this workflow. You can find these scripts in the `scripts` folder of the Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory]. For information on how to use these scripts, see the "Using Scripts to Push a Change Upstream and Request a Pull [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#pushing-a-change-upstream]" section in the Yocto Project Development Tasks Manual.

- Patch Workflow: This workflow allows you to notify the maintainer through an email that you have a change (or patch) you would like considered for the "master" branch of the Git repository. To send this type of change, you format the patch and then send the email using the Git commands `git format-patch` and `git send-email`. For information on how to use these scripts, see the "Submitting a Change to the Yocto Project [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#how-to-submit-a-change]" section in the Yocto Project Development Tasks Manual.

3.5. Git

The Yocto Project makes extensive use of Git, which is a free, open source distributed version control system. Git supports distributed development, non-linear development, and can handle large projects. It is best that you have some fundamental understanding of how Git tracks projects and how to work with Git if you are going to use the Yocto Project for development. This section provides a quick overview of how Git works and provides you with a summary of some essential Git commands.

Notes

- For more information on Git, see http://git-scm.com/documentation.
- If you need to download Git, it is recommended that you add Git to your system through your distribution's "software store" (e.g. for Ubuntu, use the Ubuntu Software feature). For the Git download page, see http://git-scm.com/download.
- For information beyond the introductory nature in this section, see the "Locating Yocto Project Source Files [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#locating-yocto-project-source-files]" section in the Yocto Project Development Tasks Manual.

3.5.1. Repositories, Tags, and Branches

As mentioned briefly in the previous section and also in the "Git Workflows and the Yocto Project" section, the Yocto Project maintains source repositories at http://git.yoctoproject.org. If you look at this web-interface of the repositories, each item is a separate Git repository.

Git repositories use branching techniques that track content change (not files) within a project (e.g. a new feature or updated documentation). Creating a tree-like structure based on project divergence allows for excellent historical information over the life of a project. This methodology also allows for an environment from which you can do lots of local experimentation on projects as you develop changes or new features.

A Git repository represents all development efforts for a given project. For example, the Git repository poky contains all changes and developments for that repository over the course of its entire life. That means that all changes that make up all releases are captured. The repository maintains a complete history of changes.

You can create a local copy of any repository by "cloning" it with the `git clone` command. When you clone a Git repository, you end up with an identical copy of the repository on your development system. Once you have a local copy of a repository, you can take steps to develop locally. For examples on how to clone Git repositories, see the "Locating Yocto Project Source Files [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#locating-yocto-project-source-files]" section in the Yocto Project Development Tasks Manual.

It is important to understand that Git tracks content change and not files. Git uses "branches" to organize different development efforts. For example, the poky repository has several branches that include the current "zeus" branch, the "master" branch, and many branches for past Yocto Project releases. You can see all the branches by going to http://git.yoctoproject.org/cgit.cgi/poky/ and clicking on the `[...] [http://git.yoctoproject.org/cgit.cgi/poky/refs/heads]` link beneath the "Branch" heading.

Each of these branches represents a specific area of development. The "master" branch represents the current or most recent development. All other branches represent offshoots of the "master" branch.

When you create a local copy of a Git repository, the copy has the same set of branches as the original. This means you can use Git to create a local working area (also called a branch) that tracks a

specific development branch from the upstream source Git repository. in other words, you can define your local Git environment to work on any development branch in the repository. To help illustrate, consider the following example Git commands:

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git checkout -b zeus origin/zeus
```

In the previous example after moving to the home directory, the `git clone` command creates a local copy of the upstream `poky` Git repository. By default, Git checks out the "master" branch for your work. After changing the working directory to the new local repository (i.e. `poky`), the `git checkout` command creates and checks out a local branch named "zeus", which tracks the upstream "origin/zeus" branch. Changes you make while in this branch would ultimately affect the upstream "zeus" branch of the `poky` repository.

It is important to understand that when you create and checkout a local working branch based on a branch name, your local environment matches the "tip" of that particular development branch at the time you created your local branch, which could be different from the files in the "master" branch of the upstream repository. In other words, creating and checking out a local branch based on the "zeus" branch name is not the same as checking out the "master" branch in the repository. Keep reading to see how you create a local snapshot of a Yocto Project Release.

Git uses "tags" to mark specific changes in a repository branch structure. Typically, a tag is used to mark a special point such as the final change (or commit) before a project is released. You can see the tags used with the `poky` Git repository by going to http://git.yoctoproject.org/cgit.cgi/poky/ and clicking on the `[...] [http://git.yoctoproject.org/cgit.cgi/poky/refs/tags]` link beneath the "Tag" heading.

Some key tags for the `poky` repository are `jethro-14.0.3`, `morty-16.0.1`, `pyro-17.0.0`, and `zeus-22.0.3`. These tags represent Yocto Project releases.

When you create a local copy of the Git repository, you also have access to all the tags in the upstream repository. Similar to branches, you can create and checkout a local working Git branch based on a tag name. When you do this, you get a snapshot of the Git repository that reflects the state of the files when the change was made associated with that tag. The most common use is to checkout a working branch that matches a specific Yocto Project release. Here is an example:

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git fetch --tags
$ git checkout tags/rocko-18.0.0 -b my_rocko-18.0.0
```

In this example, the name of the top-level directory of your local Yocto Project repository is `poky`. After moving to the `poky` directory, the `git fetch` command makes all the upstream tags available locally in your repository. Finally, the `git checkout` command creates and checks out a branch named "my-rocko-18.0.0" that is based on the upstream branch whose "HEAD" matches the commit in the repository associated with the "rocko-18.0.0" tag. The files in your repository now exactly match that particular Yocto Project release as it is tagged in the upstream Git repository. It is important to understand that when you create and checkout a local working branch based on a tag, your environment matches a specific point in time and not the entire development branch (i.e. from the "tip" of the branch backwards).

3.5.2. Basic Commands

Git has an extensive set of commands that lets you manage changes and perform collaboration over the life of a project. Conveniently though, you can manage with a small set of basic operations and workflows once you understand the basic philosophy behind Git. You do not have to be an expert in Git to be functional. A good place to look for instruction on a minimal set of Git commands is here [http://git-scm.com/documentation].

The following list of Git commands briefly describes some basic Git operations as a way to get started. As with any set of commands, this list (in most cases) simply shows the base command and omits the many arguments it supports. See the Git documentation for complete descriptions and strategies on how to use these commands:

- `git init`: Initializes an empty Git repository. You cannot use Git commands unless you have a `.git` repository.
- `git clone`: Creates a local clone of a Git repository that is on equal footing with a fellow developer's Git repository or an upstream repository.
- `git add`: Locally stages updated file contents to the index that Git uses to track changes. You must stage all files that have changed before you can commit them.
- `git commit`: Creates a local "commit" that documents the changes you made. Only changes that have been staged can be committed. Commits are used for historical purposes, for determining if a maintainer of a project will allow the change, and for ultimately pushing the change from your local Git repository into the project's upstream repository.
- `git status`: Reports any modified files that possibly need to be staged and gives you a status of where you stand regarding local commits as compared to the upstream repository.
- `git checkout branch-name`: Changes your local working branch and in this form assumes the local branch already exists. This command is analogous to "cd".
- `git checkout –b working-branch upstream-branch`: Creates and checks out a working branch on your local machine. The local branch tracks the upstream branch. You can use your local branch to isolate your work. It is a good idea to use local branches when adding specific features or changes. Using isolated branches facilitates easy removal of changes if they do not work out.
- `git branch`: Displays the existing local branches associated with your local repository. The branch that you have currently checked out is noted with an asterisk character.
- `git branch -D branch-name`: Deletes an existing local branch. You need to be in a local branch other than the one you are deleting in order to delete `branch-name`.
- `git pull --rebase`: Retrieves information from an upstream Git repository and places it in your local Git repository. You use this command to make sure you are synchronized with the repository from which you are basing changes (.e.g. the "master" branch). The "--rebase" option ensures that any local commits you have in your branch are preserved at the top of your local branch.
- `git push repo-name local-branch:upstream-branch`: Sends all your committed local changes to the upstream Git repository that your local repository is tracking (e.g. a contribution repository). The maintainer of the project draws from these repositories to merge changes (commits) into the appropriate branch of project's upstream repository.
- `git merge`: Combines or adds changes from one local branch of your repository with another branch. When you create a local Git repository, the default branch is named "master". A typical workflow is to create a temporary branch that is based off "master" that you would use for isolated work. You would make your changes in that isolated branch, stage and commit them locally, switch to the "master" branch, and then use the `git merge` command to apply the changes from your isolated branch into the currently checked out branch (e.g. "master"). After the merge is complete and if you are done with working in that isolated branch, you can safely delete the isolated branch.
- `git cherry-pick commits`: Choose and apply specific commits from one branch into another branch. There are times when you might not be able to merge all the changes in one branch with another but need to pick out certain ones.
- `gitk`: Provides a GUI view of the branches and changes in your local Git repository. This command is a good way to graphically see where things have diverged in your local repository.

  ## Note

  You need to install the `gitk` package on your development system to use this command.

- `git log`: Reports a history of your commits to the repository. This report lists all commits regardless of whether you have pushed them upstream or not.

- `git diff`: Displays line-by-line differences between a local working file and the same file as understood by Git. This command is useful to see what you have changed in any given file.

3.6. Licensing

Because open source projects are open to the public, they have different licensing structures in place. License evolution for both Open Source and Free Software has an interesting history. If you are interested in this history, you can find basic information here:

- Open source license history [http://en.wikipedia.org/wiki/Open-source_license]
- Free software license history [http://en.wikipedia.org/wiki/Free_software_license]

In general, the Yocto Project is broadly licensed under the Massachusetts Institute of Technology (MIT) License. MIT licensing permits the reuse of software within proprietary software as long as the license is distributed with that software. MIT is also compatible with the GNU General Public License (GPL). Patches to the Yocto Project follow the upstream licensing scheme. You can find information on the MIT license here [http://www.opensource.org/licenses/mit-license.php]. You can find information on the GNU GPL here [http://www.opensource.org/licenses/LGPL-3.0].

When you build an image using the Yocto Project, the build process uses a known list of licenses to ensure compliance. You can find this list in the Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory] at `meta/files/common-licenses`. Once the build completes, the list of all licenses found and used during that build are kept in the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory] at `tmp/deploy/licenses`.

If a module requires a license that is not in the base list, the build process generates a warning during the build. These tools make it easier for a developer to be certain of the licenses with which their shipped products must comply. However, even with these tools it is still up to the developer to resolve potential licensing issues.

The base list of licenses used by the build process is a combination of the Software Package Data Exchange (SPDX) list and the Open Source Initiative (OSI) projects. SPDX Group [http://spdx.org] is a working group of the Linux Foundation that maintains a specification for a standard format for communicating the components, licenses, and copyrights associated with a software package. OSI [http://opensource.org] is a corporation dedicated to the Open Source Definition and the effort for reviewing and approving licenses that conform to the Open Source Definition (OSD).

You can find a list of the combined SPDX and OSI licenses that the Yocto Project uses in the `meta/files/common-licenses` directory in your Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory].

For information that can help you maintain compliance with various open source licensing during the lifecycle of a product created using the Yocto Project, see the "Maintaining Open Source License Compliance During Your Product's Lifecycle [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#maintaining-open-source-license-compliance-during-your-products-lifecycle]" section in the Yocto Project Development Tasks Manual.

Chapter 4. Yocto Project Concepts

This chapter provides explanations for Yocto Project concepts that go beyond the surface of "how-to" information and reference (or look-up) material. Concepts such as components, the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] workflow, cross-development toolchains, shared state cache, and so forth are explained.

4.1. Yocto Project Components

The BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] task executor together with various types of configuration files form the OpenEmbedded-Core [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#oe-core]. This section overviews these components by describing their use and how they interact.

BitBake handles the parsing and execution of the data files. The data itself is of various types:

- Recipes: Provides details about particular pieces of software.
- Class Data: Abstracts common build information (e.g. how to build a Linux kernel).
- Configuration Data: Defines machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

BitBake knows how to combine multiple data sources together and refers to each data source as a layer. For information on layers, see the "Understanding and Creating Layers [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#understanding-and-creating-layers]" section of the Yocto Project Development Tasks Manual.

Following are some brief details on these core components. For additional information on how these components interact during a build, see the "OpenEmbedded Build System Concepts" section.

4.1.1. BitBake

BitBake is the tool at the heart of the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] and is responsible for parsing the Metadata [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#metadata], generating a list of tasks from it, and then executing those tasks.

This section briefly introduces BitBake. If you want more information on BitBake, see the BitBake User Manual [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html].

To see a list of the options BitBake supports, use either of the following commands:

```
$ bitbake -h
$ bitbake --help
```

The most common usage for BitBake is `bitbake packagename`, where `packagename` is the name of the package you want to build (referred to as the "target"). The target often equates to the first part of a recipe's filename (e.g. "foo" for a recipe named `foo_1.3.0-r0.bb`). So, to process the `matchbox-desktop_1.2.3.bb` recipe file, you might type the following:

```
$ bitbake matchbox-desktop
```

Several different versions of `matchbox-desktop` might exist. BitBake chooses the one selected by the distribution configuration. You can get more details about how BitBake chooses between different target versions and providers in the "Preferences [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#bb-bitbake-preferences]" section of the BitBake User Manual.

BitBake also tries to execute any dependent tasks first. So for example, before building `matchbox-desktop`, BitBake would build a cross compiler and `glibc` if they had not already been built.

A useful BitBake option to consider is the `-k` or `--continue` option. This option instructs BitBake to try and continue processing the job as long as possible even after encountering an error. When an error occurs, the target that failed and those that depend on it cannot be remade. However, when you use this option other dependencies can still be processed.

4.1.2. Recipes

Files that have the `.bb` suffix are "recipes" files. In general, a recipe contains information about a single piece of software. This information includes the location from which to download the unaltered source, any source patches to be applied to that source (if needed), which special configuration options to apply, how to compile the source files, and how to package the compiled output.

The term "package" is sometimes used to refer to recipes. However, since the word "package" is used for the packaged output from the OpenEmbedded build system (i.e. `.ipk` or `.deb` files), this document avoids using the term "package" when referring to recipes.

4.1.3. Classes

Class files (`.bbclass`) contain information that is useful to share between recipes files. An example is the `autotools` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-autotools] class, which contains common settings for any application that Autotools uses. The "Classes [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes]" chapter in the Yocto Project Reference Manual provides details about classes and how to use them.

4.1.4. Configurations

The configuration files (`.conf`) define various configuration variables that govern the OpenEmbedded build process. These files fall into several areas that define machine configuration options, distribution configuration options, compiler tuning options, general common configuration options, and user configuration options in `conf/local.conf`, which is found in the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory].

4.2. Layers

Layers are repositories that contain related metadata (i.e. sets of instructions) that tell the OpenEmbedded build system how to build a target. Yocto Project's layer model facilitates collaboration, sharing, customization, and reuse within the Yocto Project development environment. Layers logically separate information for your project. For example, you can use a layer to hold all the configurations for a particular piece of hardware. Isolating hardware-specific configurations allows you to share other metadata by using a different layer where that metadata might be common across several pieces of hardware.

Many layers exist that work in the Yocto Project development environment. The Yocto Project Curated Layer Index [https://caffelli-staging.yoctoproject.org/software-overview/layers/] and OpenEmbedded Layer Index [http://layers.openembedded.org/layerindex/branch/master/layers/] both contain layers from which you can use or leverage.

By convention, layers in the Yocto Project follow a specific form. Conforming to a known structure allows BitBake to make assumptions during builds on where to find types of metadata. You can find procedures and learn about tools (i.e. `bitbake-layers`) for creating layers suitable for the Yocto Project in the "Understanding and Creating Layers [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#understanding-and-creating-layers]" section of the Yocto Project Development Tasks Manual.

4.3. OpenEmbedded Build System Concepts

This section takes a more detailed look inside the build process used by the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term], which is the build system specific to the Yocto Project. At the heart of the build system is BitBake, the task executor.

The following diagram represents the high-level workflow of a build. The remainder of this section expands on the fundamental input, output, process, and metadata logical blocks that make up the workflow.

In general, the build's workflow consists of several functional areas:

- User Configuration: metadata you can use to control the build process.
- Metadata Layers: Various layers that provide software, machine, and distro metadata.
- Source Files: Upstream releases, local projects, and SCMs.
- Build System: Processes under the control of BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term]. This block expands on how BitBake fetches source, applies patches, completes compilation, analyzes output for package generation, creates and tests packages, generates images, and generates cross-development tools.
- Package Feeds: Directories containing output packages (RPM, DEB or IPK), which are subsequently used in the construction of an image or Software Development Kit (SDK), produced by the build system. These feeds can also be copied and shared using a web server or other means to facilitate extending or updating existing images on devices at runtime if runtime package management is enabled.
- Images: Images produced by the workflow.
- Application Development SDK: Cross-development tools that are produced along with an image or separately with BitBake.

4.3.1. User Configuration

User configuration helps define the build. Through user configuration, you can tell BitBake the target architecture for which you are building the image, where to store downloaded source, and other build properties.

The following figure shows an expanded representation of the "User Configuration" box of the general workflow figure:

BitBake needs some basic configuration files in order to complete a build. These files are `*.conf` files. The minimally necessary ones reside as example files in the `build/conf` directory of the Source Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#source-directory]. For simplicity, this section refers to the Source Directory as the "Poky Directory."

When you clone the Poky [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#poky] Git repository or you download and unpack a Yocto Project release, you can set up the Source Directory to be named anything you want. For this discussion, the cloned repository uses the default name `poky`.

Note

The Poky repository is primarily an aggregation of existing repositories. It is not a canonical upstream source.

The `meta-poky` layer inside Poky contains a `conf` directory that has example configuration files. These example files are used as a basis for creating actual configuration files when you source `oe-init-build-env` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#structure-core-script], which is the build environment script.

Sourcing the build environment script creates a Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory] if one does not already exist. BitBake uses the Build Directory for all its work during builds. The Build Directory has a `conf` directory that contains default versions of your `local.conf` and `bblayers.conf` configuration files. These default configuration files are created only if versions do not already exist in the Build Directory at the time you source the build environment setup script.

Because the Poky repository is fundamentally an aggregation of existing repositories, some users might be familiar with running the `oe-init-build-env` script in the context of separate OpenEmbedded-Core [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#oe-core] and BitBake repositories rather than a single Poky repository. This discussion assumes the script is executed from within a cloned or unpacked version of Poky.

Depending on where the script is sourced, different sub-scripts are called to set up the Build Directory (Yocto or OpenEmbedded). Specifically, the script `scripts/oe-setup-builddir` inside the poky directory sets up the Build Directory and seeds the directory (if necessary) with configuration files appropriate for the Yocto Project development environment.

Note

The `scripts/oe-setup-builddir` script uses the `$TEMPLATECONF` variable to determine which sample configuration files to locate.

The `local.conf` file provides many basic variables that define a build environment. Here is a list of a few. To see the default configurations in a `local.conf` file created by the build environment script, see the `local.conf.sample` [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta-poky/conf/local.conf.sample] in the `meta-poky` layer:

- Target Machine Selection: Controlled by the `MACHINE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-MACHINE] variable.
- Download Directory: Controlled by the `DL_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DL_DIR] variable.
- Shared State Directory: Controlled by the `SSTATE_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SSTATE_DIR] variable.
- Build Output: Controlled by the `TMPDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TMPDIR] variable.
- Distribution Policy: Controlled by the `DISTRO` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DISTRO] variable.
- Packaging Format: Controlled by the `PACKAGE_CLASSES` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_CLASSES] variable.
- SDK Target Architecture: Controlled by the `SDKMACHINE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKMACHINE] variable.
- Extra Image Packages: Controlled by the `EXTRA_IMAGE_FEATURES` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-EXTRA_IMAGE_FEATURES] variable.

> **Note**
>
> Configurations set in the `conf/local.conf` file can also be set in the `conf/site.conf` and `conf/auto.conf` configuration files.

The `bblayers.conf` file tells BitBake what layers you want considered during the build. By default, the layers listed in this file include layers minimally needed by the build system. However, you must manually add any custom layers you have created. You can find more information on working with the `bblayers.conf` file in the "Enabling Your Layer [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#enabling-your-layer]" section in the Yocto Project Development Tasks Manual.

The files `site.conf` and `auto.conf` are not created by the environment initialization script. If you want the `site.conf` file, you need to create that yourself. The `auto.conf` file is typically created by an autobuilder:

- `site.conf`: You can use the `conf/site.conf` configuration file to configure multiple build directories. For example, suppose you had several build environments and they shared some common features. You can set these default build properties here. A good example is perhaps the packaging format to use through the PACKAGE_CLASSES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_CLASSES] variable.

  One useful scenario for using the `conf/site.conf` file is to extend your BBPATH [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-BBPATH] variable to include the path to a `conf/site.conf`. Then, when BitBake looks for Metadata using BBPATH, it finds the `conf/site.conf` file and applies your common configurations found in the file. To override configurations in a particular build directory, alter the similar configurations within that build directory's `conf/local.conf` file.

- `auto.conf`: The file is usually created and written to by an autobuilder. The settings put into the file are typically the same as you would find in the `conf/local.conf` or the `conf/site.conf` files.

You can edit all configuration files to further define any particular build environment. This process is represented by the "User Configuration Edits" box in the figure.

When you launch your build with the `bitbake target` command, BitBake sorts out the configurations to ultimately define your build environment. It is important to understand that the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term] reads the configuration files in a specific order: `site.conf`, `auto.conf`, and `local.conf`. And, the build system applies the normal assignment statement rules as described in the "Syntax and Operators [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#bitbake-user-manual-metadata]" chapter of the BitBake User Manual. Because the files are parsed in a specific order, variable assignments for the same variable could be affected. For example, if the `auto.conf` file and the `local.conf` set `variable1` to different values, because the build system parses `local.conf` after `auto.conf`, `variable1` is assigned the value from the `local.conf` file.

4.3.2. Metadata, Machine Configuration, and Policy Configuration

The previous section described the user configurations that define BitBake's global behavior. This section takes a closer look at the layers the build system uses to further control the build. These layers provide Metadata for the software, machine, and policies.

In general, three types of layer input exists. You can see them below the "User Configuration" box in the general workflow figure:

- Metadata (`.bb` + Patches): Software layers containing user-supplied recipe files, patches, and append files. A good example of a software layer might be the `meta-qt5` [https://github.com/meta-qt5/meta-qt5] layer from the OpenEmbedded Layer Index [http://layers.openembedded.org/layerindex/branch/master/layers/]. This layer is for version 5.0 of the popular Qt [https://wiki.qt.io/About_Qt] cross-platform application development framework for desktop, embedded and mobile.

- Machine BSP Configuration: Board Support Package (BSP) layers (i.e. "BSP Layer" in the following figure) providing machine-specific configurations. This type of information is specific to a particular target architecture. A good example of a BSP layer from the Poky Reference Distribution is the `meta-yocto-bsp` [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta-yocto-bsp] layer.

- Policy Configuration: Distribution Layers (i.e. "Distro Layer" in the following figure) providing top-level or general policies for the images or SDKs being built for a particular distribution. For example, in the Poky Reference Distribution the distro layer is the `meta-poky` [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta-poky] layer. Within the distro layer is a `conf/distro` directory that contains distro configuration files (e.g. `poky.conf` [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta-poky/conf/distro/poky.conf] that contain many policy configurations for the Poky distribution.

The following figure shows an expanded representation of these three layers from the general workflow figure:

In general, all layers have a similar structure. They all contain a licensing file (e.g. `COPYING.MIT`) if the layer is to be distributed, a `README` file as good practice and especially if the layer is to be distributed, a configuration directory, and recipe directories. You can learn about the general structure for layers used with the Yocto Project in the "Creating Your Own Layer [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#creating-your-own-layer]" section in the Yocto Project Development Tasks Manual. For a general discussion on layers and the many layers from which you can draw, see the "Layers" and "The Yocto Project Layer Model" sections both earlier in this manual.

If you explored the previous links, you discovered some areas where many layers that work with the Yocto Project exist. The Source Repositories [http://git.yoctoproject.org/] also shows layers categorized under "Yocto Metadata Layers."

> **Note**
> Layers exist in the Yocto Project Source Repositories that cannot be found in the OpenEmbedded Layer Index. These layers are either deprecated or experimental in nature.

BitBake uses the `conf/bblayers.conf` file, which is part of the user configuration, to find what layers it should be using as part of the build.

4.3.2.1. Distro Layer

The distribution layer provides policy configurations for your distribution. Best practices dictate that you isolate these types of configurations into their own layer. Settings you provide in `conf/distro/distro.conf` override similar settings that BitBake finds in your `conf/local.conf` file in the Build Directory.

The following list provides some explanation and references for what you typically find in the distribution layer:

- classes: Class files (`.bbclass`) hold common functionality that can be shared among recipes in the distribution. When your recipes inherit a class, they take on the settings and functions for that class. You can read more about class files in the "Classes [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes]" chapter of the Yocto Reference Manual.
- conf: This area holds configuration files for the layer (`conf/layer.conf`), the distribution (`conf/distro/distro.conf`), and any distribution-wide include files.
- recipes-*: Recipes and append files that affect common functionality across the distribution. This area could include recipes and append files to add distribution-specific configuration, initialization scripts, custom image recipes, and so forth. Examples of `recipes-*` directories are `recipes-core` and `recipes-extra`. Hierarchy and contents within a `recipes-*` directory can vary. Generally, these directories contain recipe files (`*.bb`), recipe append files (`*.bbappend`), directories that are distro-specific for configuration files, and so forth.

4.3.2.2. BSP Layer

The BSP Layer provides machine configurations that target specific hardware. Everything in this layer is specific to the machine for which you are building the image or the SDK. A common structure or form is defined for BSP layers. You can learn more about this structure in the Yocto Project Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/3.0.3/bsp-guide/bsp-guide.html].

> **Note**
> In order for a BSP layer to be considered compliant with the Yocto Project, it must meet some structural requirements.

The BSP Layer's configuration directory contains configuration files for the machine (`conf/machine/machine.conf`) and, of course, the layer (`conf/layer.conf`).

The remainder of the layer is dedicated to specific recipes by function: `recipes-bsp`, `recipes-core`, `recipes-graphics`, `recipes-kernel`, and so forth. Metadata can exist for multiple formfactors, graphics support systems, and so forth.

> **Note**
> While the figure shows several `recipes-*` directories, not all these directories appear in all BSP layers.

4.3.2.3. Software Layer

The software layer provides the Metadata for additional software packages used during the build. This layer does not include Metadata that is specific to the distribution or the machine, which are found in their respective layers.

This layer contains any recipes, append files, and patches, that your project needs.

4.3.3. Sources

In order for the OpenEmbedded build system to create an image or any target, it must be able to access source files. The general workflow figure represents source files using the "Upstream Project Releases", "Local Projects", and "SCMs (optional)" boxes. The figure represents mirrors, which also play a role in locating source files, with the "Source Materials" box.

The method by which source files are ultimately organized is a function of the project. For example, for released software, projects tend to use tarballs or other archived files that can capture the state of a release guaranteeing that it is statically represented. On the other hand, for a project that is more dynamic or experimental in nature, a project might keep source files in a repository controlled by a Source Control Manager (SCM) such as Git. Pulling source from a repository allows you to control the point in the repository (the revision) from which you want to build software. Finally, a combination of the two might exist, which would give the consumer a choice when deciding where to get source files.

BitBake uses the `SRC_URI` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SRC_URI] variable to point to source files regardless of their location. Each recipe must have a `SRC_URI` variable that points to the source.

Another area that plays a significant role in where source files come from is pointed to by the `DL_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DL_DIR] variable. This area is a cache that can hold previously downloaded source. You can also instruct the OpenEmbedded build system to create tarballs from Git repositories, which is not the default behavior, and store them in the `DL_DIR` by using the `BB_GENERATE_MIRROR_TARBALLS` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-BB_GENERATE_MIRROR_TARBALLS] variable.

Judicious use of a `DL_DIR` directory can save the build system a trip across the Internet when looking for files. A good method for using a download directory is to have `DL_DIR` point to an area outside of your Build Directory. Doing so allows you to safely delete the Build Directory if needed without fear of removing any downloaded source file.

The remainder of this section provides a deeper look into the source files and the mirrors. Here is a more detailed look at the source file area of the general workflow figure:

4.3.3.1. Upstream Project Releases

Upstream project releases exist anywhere in the form of an archived file (e.g. tarball or zip file). These files correspond to individual recipes. For example, the figure uses specific releases each for BusyBox, Qt, and Dbus. An archive file can be for any released product that can be built using a recipe.

4.3.3.2. Local Projects

Local projects are custom bits of software the user provides. These bits reside somewhere local to a project - perhaps a directory into which the user checks in items (e.g. a local directory containing a development source tree used by the group).

The canonical method through which to include a local project is to use the `externalsrc` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-externalsrc] class to include that local project. You use either the `local.conf` or a recipe's append file to override or set the recipe to point to the local directory on your disk to pull in the whole source tree.

4.3.3.3. Source Control Managers (Optional)

Another place from which the build system can get source files is with fetchers [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#bb-fetchers] employing various Source Control Managers (SCMs) such as Git or

Subversion. In such cases, a repository is cloned or checked out. The `do_fetch` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-fetch] task inside BitBake uses the `SRC_URI` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SRC_URI] variable and the argument's prefix to determine the correct fetcher module.

> **Note**
>
> For information on how to have the OpenEmbedded build system generate tarballs for Git repositories and place them in the `DL_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DL_DIR] directory, see the `BB_GENERATE_MIRROR_TARBALLS` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-BB_GENERATE_MIRROR_TARBALLS] variable in the Yocto Project Reference Manual.

When fetching a repository, BitBake uses the `SRCREV` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SRCREV] variable to determine the specific revision from which to build.

4.3.3.4. Source Mirror(s)

Two kinds of mirrors exist: pre-mirrors and regular mirrors. The `PREMIRRORS` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PREMIRRORS] and `MIRRORS` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-MIRRORS] variables point to these, respectively. BitBake checks pre-mirrors before looking upstream for any source files. Pre-mirrors are appropriate when you have a shared directory that is not a directory defined by the `DL_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DL_DIR] variable. A Pre-mirror typically points to a shared directory that is local to your organization.

Regular mirrors can be any site across the Internet that is used as an alternative location for source code should the primary site not be functioning for some reason or another.

4.3.4. Package Feeds

When the OpenEmbedded build system generates an image or an SDK, it gets the packages from a package feed area located in the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory]. The general workflow figure shows this package feeds area in the upper-right corner.

This section looks a little closer into the package feeds area used by the build system. Here is a more detailed look at the area:

Package feeds are an intermediary step in the build process. The OpenEmbedded build system provides classes to generate different package types, and you specify which classes to enable through the PACKAGE_CLASSES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_CLASSES] variable. Before placing the packages into package feeds, the build process validates them with generated output quality assurance checks through the insane [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-insane] class.

The package feed area resides in the Build Directory. The directory the build system uses to temporarily store packages is determined by a combination of variables and the particular package manager in use. See the "Package Feeds" box in the illustration and note the information to the right of that area. In particular, the following defines where package files are kept:

- DEPLOY_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR]: Defined as tmp/deploy in the Build Directory.
- DEPLOY_DIR_*: Depending on the package manager used, the package type sub-folder. Given RPM, IPK, or DEB packaging and tarball creation, the DEPLOY_DIR_RPM [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR_RPM], DEPLOY_DIR_IPK [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR_IPK], DEPLOY_DIR_DEB [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR_DEB], or DEPLOY_DIR_TAR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR_TAR], variables are used, respectively.

- PACKAGE_ARCH [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_ARCH]: Defines architecture-specific sub-folders. For example, packages could exist for the i586 or qemux86 architectures.

BitBake uses the do_package_write_* [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_deb] tasks to generate packages and place them into the package holding area (e.g. do_package_write_ipk for IPK packages). See the "do_package_write_deb [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_deb]", "do_package_write_ipk [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_ipk]", "do_package_write_rpm [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_rpm]", and "do_package_write_tar [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_tar]" sections in the Yocto Project Reference Manual for additional information. As an example, consider a scenario where an IPK packaging manager is being used and package architecture support for both i586 and qemux86 exist. Packages for the i586 architecture are placed in build/tmp/deploy/ipk/i586, while packages for the qemux86 architecture are placed in build/tmp/deploy/ipk/qemux86.

4.3.5. BitBake

The OpenEmbedded build system uses BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] to produce images and Software Development Kits (SDKs). You can see from the general workflow figure, the BitBake area consists of several functional areas. This section takes a closer look at each of those areas.

Note

Separate documentation exists for the BitBake tool. See the BitBake User Manual [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html] for reference material on BitBake.

4.3.5.1. Source Fetching

The first stages of building a recipe are to fetch and unpack the source code:

The `do_fetch` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-fetch] and `do_unpack` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-unpack] tasks fetch the source files and unpack them into the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory].

> **Note**
>
> For every local file (e.g. `file://`) that is part of a recipe's SRC_URI [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SRC_URI] statement, the OpenEmbedded build system takes a checksum of the file for the recipe and inserts the checksum into the signature for the `do_fetch` task. If any local file has been modified, the `do_fetch` task and all tasks that depend on it are re-executed.

By default, everything is accomplished in the Build Directory, which has a defined structure. For additional general information on the Build Directory, see the "`build/` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#structure-core-build]" section in the Yocto Project Reference Manual.

Each recipe has an area in the Build Directory where the unpacked source code resides. The S [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-S] variable points to this area for a recipe's unpacked source code. The name of that directory for any given recipe is defined from several different variables. The preceding figure and the following list describe the Build Directory's hierarchy:

- `TMPDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TMPDIR]: The base directory where the OpenEmbedded build system performs all its work during the build. The default base directory is the `tmp` directory.

- PACKAGE_ARCH [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_ARCH]: The architecture of the built package or packages. Depending on the eventual destination of the package or packages (i.e. machine architecture, build host [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#hardware-build-system-term], SDK, or specific machine), PACKAGE_ARCH varies. See the variable's description for details.

- TARGET_OS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TARGET_OS]: The operating system of the target device. A typical value would be "linux" (e.g. "qemux86-poky-linux").

- PN [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PN]: The name of the recipe used to build the package. This variable can have multiple meanings. However, when used in the context of input files, PN represents the the name of the recipe.

- WORKDIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR]: The location where the OpenEmbedded build system builds a recipe (i.e. does the work to create the package).

  - PV [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PV]: The version of the recipe used to build the package.

  - PR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PR]: The revision of the recipe used to build the package.

- S [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-S]: Contains the unpacked source files for a given recipe.

  - BPN [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-BPN]: The name of the recipe used to build the package. The BPN variable is a version of the PN variable but with common prefixes and suffixes removed.

  - PV [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PV]: The version of the recipe used to build the package.

Note

In the previous figure, notice that two sample hierarchies exist: one based on package architecture (i.e. PACKAGE_ARCH) and one based on a machine (i.e. MACHINE). The underlying structures are identical. The differentiator being what the OpenEmbedded build system is using as a build target (e.g. general architecture, a build host, an SDK, or a specific machine).

4.3.5.2. Patching

Once source code is fetched and unpacked, BitBake locates patch files and applies them to the source files:

The `do_patch` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-patch] task uses a recipe's `SRC_URI` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SRC_URI] statements and the `FILESPATH` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-FILESPATH] variable to locate applicable patch files.

Default processing for patch files assumes the files have either `*.patch` or `*.diff` file types. You can use `SRC_URI` parameters to change the way the build system recognizes patch files. See the `do_patch` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-patch] task for more information.

BitBake finds and applies multiple patches for a single recipe in the order in which it locates the patches. The `FILESPATH` variable defines the default set of directories that the build system uses to search for patch files. Once found, patches are applied to the recipe's source files, which are located in the `S` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-S] directory.

For more information on how the source directories are created, see the "Source Fetching" section. For more information on how to create patches and how the build system processes patches, see the "Patching Code [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#new-recipe-patching-code]" section in the Yocto Project Development Tasks Manual. You can also see the "Use `devtool modify` to Modify the Source of an Existing

Component [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html#sdk-devtool-use-devtool-modify-to-modify-the-source-of-an-existing-component]" section in the Yocto Project Application Development and the Extensible Software Development Kit (SDK) manual and the "Using Traditional Kernel Development to Patch the Kernel [http://www.yoctoproject.org/docs/3.0.3/kernel-dev/kernel-dev.html#using-traditional-kernel-development-to-patch-the-kernel]" section in the Yocto Project Linux Kernel Development Manual.

4.3.5.3. Configuration, Compilation, and Staging

After source code is patched, BitBake executes tasks that configure and compile the source code. Once compilation occurs, the files are copied to a holding area (staged) in preparation for packaging:

This step in the build process consists of the following tasks:

- `do_prepare_recipe_sysroot` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-prepare_recipe_sysroot]: This task sets up the two sysroots in `${WORKDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR]} (i.e. `recipe-sysroot` and `recipe-sysroot-native`) so that during the packaging phase the sysroots can contain the contents of the `do_populate_sysroot` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sysroot] tasks of the recipes on which the recipe containing the tasks depends. A sysroot exists for both the target and for the native binaries, which run on the host system.
- `do_configure`: This task configures the source by enabling and disabling any build-time and configuration options for the software being built. Configurations can come from the recipe itself as well as from an inherited class. Additionally, the software itself might configure itself depending on the target for which it is being built.

  The configurations handled by the `do_configure` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-configure] task are specific to configurations for the source code being built by the recipe.

If you are using the `autotools` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-autotools] class, you can add additional configuration options by using the `EXTRA_OECONF` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-EXTRA_OECONF] or `PACKAGECONFIG_CONFARGS` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGECONFIG_CONFARGS] variables. For information on how this variable works within that class, see the `autotools` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-autotools] class here [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/classes/autotools.bbclass].

- `do_compile`: Once a configuration task has been satisfied, BitBake compiles the source using the `do_compile` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-compile] task. Compilation occurs in the directory pointed to by the `B` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-B] variable. Realize that the `B` directory is, by default, the same as the `S` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-S] directory.

- `do_install`: After compilation completes, BitBake executes the `do_install` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-install] task. This task copies files from the `B` directory and places them in a holding area pointed to by the `D` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-D] variable. Packaging occurs later using files from this holding directory.

4.3.5.4. Package Splitting

After source code is configured, compiled, and staged, the build system analyzes the results and splits the output into packages:

The `do_package` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package] and `do_packagedata` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-packagedata] tasks combine to analyze the files found in the `D` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-D] directory and split them into subsets based on available packages and files. Analysis involves the following as well as other items: splitting out debugging symbols, looking at shared library dependencies between packages, and looking at package relationships.

The `do_packagedata` task creates package metadata based on the analysis such that the build system can generate the final packages. The `do_populate_sysroot` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sysroot] task stages (copies) a subset of the files installed by the `do_install` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-

manual.html#ref-tasks-install] task into the appropriate sysroot. Working, staged, and intermediate results of the analysis and package splitting process use several areas:

- PKGD [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PKGD]: The destination directory (i.e. package) for packages before they are split into individual packages.
- PKGDESTWORK [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PKGDESTWORK]: A temporary work area (i.e. pkgdata) used by the do_package task to save package metadata.
- PKGDEST [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PKGDEST]: The parent directory (i.e. packages-split) for packages after they have been split.
- PKGDATA_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PKGDATA_DIR]: A shared, global-state directory that holds packaging metadata generated during the packaging process. The packaging process copies metadata from PKGDESTWORK to the PKGDATA_DIR area where it becomes globally available.
- STAGING_DIR_HOST [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-STAGING_DIR_HOST]: The path for the sysroot for the system on which a component is built to run (i.e. recipe-sysroot).
- STAGING_DIR_NATIVE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-STAGING_DIR_NATIVE]: The path for the sysroot used when building components for the build host (i.e. recipe-sysroot-native).
- STAGING_DIR_TARGET [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-STAGING_DIR_TARGET]: The path for the sysroot used when a component that is built to execute on a system and it generates code for yet another machine (e.g. cross-canadian recipes).

The FILES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-FILES] variable defines the files that go into each package in PACKAGES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGES]. If you want details on how this is accomplished, you can look at package.bbclass [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/classes/package.bbclass].

Depending on the type of packages being created (RPM, DEB, or IPK), the do_package_write_* [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_deb] task creates the actual packages and places them in the Package Feed area, which is ${TMPDIR}/deploy. You can see the "Package Feeds" section for more detail on that part of the build process.

Note

Support for creating feeds directly from the deploy/* directories does not exist. Creating such feeds usually requires some kind of feed maintenance mechanism that would upload the new packages into an official package feed (e.g. the Ångström distribution). This functionality is highly distribution-specific and thus is not provided out of the box.

4.3.5.5. Image Generation

Once packages are split and stored in the Package Feeds area, the build system uses BitBake to generate the root filesystem image:

The image generation process consists of several stages and depends on several tasks and variables. The `do_rootfs` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-rootfs] task creates the root filesystem (file and directory structure) for an image. This task uses several key variables to help create the list of packages to actually install:

- `IMAGE_INSTALL` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_INSTALL]: Lists out the base set of packages from which to install from the Package Feeds area.

- PACKAGE_EXCLUDE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_EXCLUDE]: Specifies packages that should not be installed into the image.
- IMAGE_FEATURES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_FEATURES]: Specifies features to include in the image. Most of these features map to additional packages for installation.
- PACKAGE_CLASSES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_CLASSES]: Specifies the package backend (e.g. RPM, DEB, or IPK) to use and consequently helps determine where to locate packages within the Package Feeds area.
- IMAGE_LINGUAS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_LINGUAS]: Determines the language(s) for which additional language support packages are installed.
- PACKAGE_INSTALL [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_INSTALL]: The final list of packages passed to the package manager for installation into the image.

With IMAGE_ROOTFS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_ROOTFS] pointing to the location of the filesystem under construction and the PACKAGE_INSTALL variable providing the final list of packages to install, the root file system is created.

Package installation is under control of the package manager (e.g. dnf/rpm, opkg, or apt/dpkg) regardless of whether or not package management is enabled for the target. At the end of the process, if package management is not enabled for the target, the package manager's data files are deleted from the root filesystem. As part of the final stage of package installation, post installation scripts that are part of the packages are run. Any scripts that fail to run on the build host are run on the target when the target system is first booted. If you are using a read-only root filesystem [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#creating-a-read-only-root-filesystem], all the post installation scripts must succeed on the build host during the package installation phase since the root filesystem on the target is read-only.

The final stages of the do_rootfs task handle post processing. Post processing includes creation of a manifest file and optimizations.

The manifest file (.manifest) resides in the same directory as the root filesystem image. This file lists out, line-by-line, the installed packages. The manifest file is useful for the testimage [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-testimage*] class, for example, to determine whether or not to run specific tests. See the IMAGE_MANIFEST [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_MANIFEST] variable for additional information.

Optimizing processes that are run across the image include mklibs, prelink, and any other post-processing commands as defined by the ROOTFS_POSTPROCESS_COMMAND [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-ROOTFS_POSTPROCESS_COMMAND] variable. The mklibs process optimizes the size of the libraries, while the prelink process optimizes the dynamic linking of shared libraries to reduce start up time of executables.

After the root filesystem is built, processing begins on the image through the do_image [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-image] task. The build system runs any pre-processing commands as defined by the IMAGE_PREPROCESS_COMMAND [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_PREPROCESS_COMMAND] variable. This variable specifies a list of functions to call before the build system creates the final image output files.

The build system dynamically creates do_image_* tasks as needed, based on the image types specified in the IMAGE_FSTYPES [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_FSTYPES] variable. The process turns everything into an image file or a set of image files and can compress the root filesystem image to reduce the overall size of the image. The formats used for the root filesystem depend on the IMAGE_FSTYPES variable. Compression depends on whether the formats support compression.

As an example, a dynamically created task when creating a particular image type would take the following form:

```
do_image_type
```

So, if the type as specified by the IMAGE_FSTYPES were ext4, the dynamically generated task would be as follows:

```
do_image_ext4
```

The final task involved in image creation is the do_image_complete [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-image-complete] task. This task completes the image by applying any image post processing as defined through the IMAGE_POSTPROCESS_COMMAND [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_POSTPROCESS_COMMAND] variable. The variable specifies a list of functions to call once the build system has created the final image output files.

> **Note**
> The entire image generation process is run under Pseudo. Running under Pseudo ensures that the files in the root filesystem have correct ownership.

4.3.5.6. SDK Generation

The OpenEmbedded build system uses BitBake to generate the Software Development Kit (SDK) installer scripts for both the standard SDK and the extensible SDK (eSDK):

> **Note**
> For more information on the cross-development toolchain generation, see the "Cross-Development Toolchain Generation" section. For information on advantages gained when building a cross-development toolchain using the do_populate_sdk [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sdk] task, see the "Building an SDK Installer [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html#sdk-building-an-sdk-installer]" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

Like image generation, the SDK script process consists of several stages and depends on many variables. The do_populate_sdk [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sdk] and do_populate_sdk_ext [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sdk_ext] tasks use these key variables to help create the list of packages to actually install. For information on the variables listed in the figure, see the "Application Development SDK" section.

The do_populate_sdk task helps create the standard SDK and handles two parts: a target part and a host part. The target part is the part built for the target hardware and includes libraries and headers. The host part is the part of the SDK that runs on the SDKMACHINE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKMACHINE].

The do_populate_sdk_ext task helps create the extensible SDK and handles host and target parts differently than its counter part does for the standard SDK. For the extensible SDK, the task encapsulates the build system, which includes everything needed (host and target) for the SDK.

Regardless of the type of SDK being constructed, the tasks perform some cleanup after which a cross-development environment setup script and any needed configuration files are created. The final output is the Cross-development toolchain installation script (.sh file), which includes the environment setup script.

4.3.5.7. Stamp Files and the Rerunning of Tasks

For each task that completes successfully, BitBake writes a stamp file into the STAMPS_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-STAMPS_DIR]

directory. The beginning of the stamp file's filename is determined by the STAMP [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-STAMP] variable, and the end of the name consists of the task's name and current input checksum.

> ## Note
>
> This naming scheme assumes that BB_SIGNATURE_HANDLER [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#var-BB_SIGNATURE_HANDLER] is "OEBasicHash", which is almost always the case in current OpenEmbedded.

To determine if a task needs to be rerun, BitBake checks if a stamp file with a matching input checksum exists for the task. If such a stamp file exists, the task's output is assumed to exist and still be valid. If the file does not exist, the task is rerun.

> ## Note
>
> The stamp mechanism is more general than the shared state (sstate) cache mechanism described in the "Setscene Tasks and Shared State" section. BitBake avoids rerunning any task that has a valid stamp file, not just tasks that can be accelerated through the sstate cache.
>
> However, you should realize that stamp files only serve as a marker that some work has been done and that these files do not record task output. The actual task output would usually be somewhere in TMPDIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TMPDIR] (e.g. in some recipe's WORKDIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR].) What the sstate cache mechanism adds is a way to cache task output that can then be shared between build machines.

Since STAMPS_DIR is usually a subdirectory of TMPDIR, removing TMPDIR will also remove STAMPS_DIR, which means tasks will properly be rerun to repopulate TMPDIR.

If you want some task to always be considered "out of date", you can mark it with the nostamp [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#variable-flags] varflag. If some other task depends on such a task, then that task will also always be considered out of date, which might not be what you want.

For details on how to view information about a task's signature, see the "Viewing Task Variable Dependencies [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#dev-viewing-task-variable-dependencies]" section in the Yocto Project Development Tasks Manual.

4.3.5.8. Setscene Tasks and Shared State

The description of tasks so far assumes that BitBake needs to build everything and no available prebuilt objects exist. BitBake does support skipping tasks if prebuilt objects are available. These objects are usually made available in the form of a shared state (sstate) cache.

> ## Note
>
> For information on variables affecting sstate, see the SSTATE_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SSTATE_DIR] and SSTATE_MIRRORS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SSTATE_MIRRORS] variables.

The idea of a setscene task (i.e do_taskname_setscene) is a version of the task where instead of building something, BitBake can skip to the end result and simply place a set of files into specific locations as needed. In some cases, it makes sense to have a setscene task variant (e.g. generating package files in the do_package_write_* [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_deb] task). In other cases, it does not make sense (e.g. a do_patch [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-patch] task or a do_unpack [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-unpack] task) since the work involved would be equal to or greater than the underlying task.

In the build system, the common tasks that have setscene variants are do_package [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package], do_package_write_*, do_deploy [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-deploy], do_packagedata [http://www.yoctoproject.org/docs/3.0.3/

ref-manual/ref-manual.html#ref-tasks-packagedata], and `do_populate_sysroot` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-populate_sysroot]. Notice that these tasks represent most of the tasks whose output is an end result.

The build system has knowledge of the relationship between these tasks and other preceding tasks. For example, if BitBake runs `do_populate_sysroot_setscene` for something, it does not make sense to run any of the `do_fetch`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, and `do_install` tasks. However, if `do_package` needs to be run, BitBake needs to run those other tasks.

It becomes more complicated if everything can come from an sstate cache because some objects are simply not required at all. For example, you do not need a compiler or native tools, such as quilt, if nothing exists to compile or patch. If the `do_package_write_*` packages are available from sstate, BitBake does not need the `do_package` task data.

To handle all these complexities, BitBake runs in two phases. The first is the "setscene" stage. During this stage, BitBake first checks the sstate cache for any targets it is planning to build. BitBake does a fast check to see if the object exists rather than a complete download. If nothing exists, the second phase, which is the setscene stage, completes and the main build proceeds.

If objects are found in the sstate cache, the build system works backwards from the end targets specified by the user. For example, if an image is being built, the build system first looks for the packages needed for that image and the tools needed to construct an image. If those are available, the compiler is not needed. Thus, the compiler is not even downloaded. If something was found to be unavailable, or the download or setscene task fails, the build system then tries to install dependencies, such as the compiler, from the cache.

The availability of objects in the sstate cache is handled by the function specified by the `BB_HASHCHECK_FUNCTION` [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#var-BB_HASHCHECK_FUNCTION] variable and returns a list of available objects. The function specified by the `BB_SETSCENE_DEPVALID` [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#var-BB_SETSCENE_DEPVALID] variable is the function that determines whether a given dependency needs to be followed, and whether for any given relationship the function needs to be passed. The function returns a True or False value.

4.3.6. Images

The images produced by the build system are compressed forms of the root filesystem and are ready to boot on a target device. You can see from the general workflow figure that

BitBake output, in part, consists of images. This section takes a closer look at this output:

Note

For a list of example images that the Yocto Project provides, see the "Images [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-images]" chapter in the Yocto Project Reference Manual.

The build process writes images out to the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory] inside the `tmp/deploy/images/machine/` folder as shown in the figure. This folder contains any files expected to be loaded on the target device. The `DEPLOY_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR] variable points to the `deploy` directory, while the `DEPLOY_DIR_IMAGE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR_IMAGE] variable points to the appropriate directory containing images for the current configuration.

- `kernel-image`: A kernel binary file. The `KERNEL_IMAGETYPE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-KERNEL_IMAGETYPE] variable determines the naming scheme for the kernel image file. Depending on this variable, the file could begin with a variety of naming strings. The `deploy/images/machine` directory can contain multiple image files for the machine.
- `root-filesystem-image`: Root filesystems for the target device (e.g. `*.ext3` or `*.bz2` files). The `IMAGE_FSTYPES` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-IMAGE_FSTYPES] variable determines the root filesystem image type. The `deploy/images/machine` directory can contain multiple root filesystems for the machine.
- `kernel-modules`: Tarballs that contain all the modules built for the kernel. Kernel module tarballs exist for legacy purposes and can be suppressed by

setting the MODULE_TARBALL_DEPLOY [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-MODULE_TARBALL_DEPLOY] variable to "0". The deploy/images/machine directory can contain multiple kernel module tarballs for the machine.

- bootloaders: If applicable to the target machine, bootloaders supporting the image. The deploy/images/machine directory can contain multiple bootloaders for the machine.

- symlinks: The deploy/images/machine folder contains a symbolic link that points to the most recently built file for each machine. These links might be useful for external scripts that need to obtain the latest version of each file.

4.3.7. Application Development SDK

In the general workflow figure, the output labeled "Application Development SDK" represents an SDK. The SDK generation process differs depending on whether you build an extensible SDK (e.g. bitbake -c populate_sdk_ext imagename) or a standard SDK (e.g.

`bitbake -c populate_sdk imagename`). This section takes a closer look at this output:

The specific form of this output is a set of files that includes a self-extracting SDK installer (`*.sh`), host and target manifest files, and files used for SDK testing. When the SDK installer file is run, it installs the SDK. The SDK consists of a cross-development toolchain, a set of libraries and headers, and an SDK environment setup script. Running this installer essentially sets up your cross-development environment. You can think of the cross-toolchain as the "host" part because it runs on the SDK machine. You can think of the libraries and headers as the "target" part because they are built for the target hardware. The environment setup script is added so that you can initialize the environment before using the tools.

Notes

- The Yocto Project supports several methods by which you can set up this cross-development environment. These methods include downloading pre-built SDK installers or building and installing your own SDK installer.

- For background information on cross-development toolchains in the Yocto Project development environment, see the "Cross-Development Toolchain Generation" section.
- For information on setting up a cross-development environment, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html] manual.

All the output files for an SDK are written to the `deploy/sdk` folder inside the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory] as shown in the previous figure. Depending on the type of SDK, several variables exist that help configure these files. The following list shows the variables associated with an extensible SDK:

- `DEPLOY_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR]: Points to the `deploy` directory.
- `SDK_EXT_TYPE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_EXT_TYPE]: Controls whether or not shared state artifacts are copied into the extensible SDK. By default, all required shared state artifacts are copied into the SDK.
- `SDK_INCLUDE_PKGDATA` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_INCLUDE_PKGDATA]: Specifies whether or not packagedata is included in the extensible SDK for all recipes in the "world" target.
- `SDK_INCLUDE_TOOLCHAIN` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_INCLUDE_TOOLCHAIN]: Specifies whether or not the toolchain is included when building the extensible SDK.
- `SDK_LOCAL_CONF_WHITELIST` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_LOCAL_CONF_WHITELIST]: A list of variables allowed through from the build system configuration into the extensible SDK configuration.
- `SDK_LOCAL_CONF_BLACKLIST` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_LOCAL_CONF_BLACKLIST]: A list of variables not allowed through from the build system configuration into the extensible SDK configuration.
- `SDK_INHERIT_BLACKLIST` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_INHERIT_BLACKLIST]: A list of classes to remove from the `INHERIT` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-INHERIT] value globally within the extensible SDK configuration.

This next list, shows the variables associated with a standard SDK:

- `DEPLOY_DIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPLOY_DIR]: Points to the `deploy` directory.
- `SDKMACHINE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKMACHINE]: Specifies the architecture of the machine on which the cross-development tools are run to create packages for the target hardware.
- `SDKIMAGE_FEATURES` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKIMAGE_FEATURES]: Lists the features to include in the "target" part of the SDK.
- `TOOLCHAIN_HOST_TASK` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TOOLCHAIN_HOST_TASK]: Lists packages that make up the host part of the SDK (i.e. the part that runs on the `SDKMACHINE`). When you use `bitbake -c populate_sdk imagename` to create the SDK, a set of default packages apply. This variable allows you to add more packages.
- `TOOLCHAIN_TARGET_TASK` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TOOLCHAIN_TARGET_TASK]: Lists packages that make up the target part of the SDK (i.e. the part built for the target hardware).
- `SDKPATH` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKPATH]: Defines the default SDK installation path offered by the installation script.
- `SDK_HOST_MANIFEST` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_HOST_MANIFEST]: Lists all the installed packages that make up the host part of the SDK. This variable also plays a minor role for extensible SDK development as well. However, it is mainly used for the standard SDK.

- SDK_TARGET_MANIFEST [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_TARGET_MANIFEST]: Lists all the installed packages that make up the target part of the SDK. This variable also plays a minor role for extensible SDK development as well. However, it is mainly used for the standard SDK.

4.4. Cross-Development Toolchain Generation

The Yocto Project does most of the work for you when it comes to creating cross-development toolchains [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#cross-development-toolchain]. This section provides some technical background on how cross-development toolchains are created and used. For more information on toolchains, you can also see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html] manual.

In the Yocto Project development environment, cross-development toolchains are used to build images and applications that run on the target hardware. With just a few commands, the OpenEmbedded build system creates these necessary toolchains for you.

The following figure shows a high-level build environment regarding toolchain construction and use.

Most of the work occurs on the Build Host. This is the machine used to build images and generally work within the the Yocto Project environment. When you run BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] to create an image, the OpenEmbedded build

system uses the host gcc compiler to bootstrap a cross-compiler named gcc-cross. The gcc-cross compiler is what BitBake uses to compile source files when creating the target image. You can think of gcc-cross simply as an automatically generated cross-compiler that is used internally within BitBake only.

> **Note**
> The extensible SDK does not use gcc-cross-canadian since this SDK ships a copy of the OpenEmbedded build system and the sysroot within it contains gcc-cross.

The chain of events that occurs when gcc-cross is bootstrapped is as follows:

```
gcc -> binutils-cross -> gcc-cross-initial -> linux-libc-headers -> glibc-initial -> glibc -
```

- gcc: The build host's GNU Compiler Collection (GCC).
- binutils-cross: The bare minimum binary utilities needed in order to run the gcc-cross-initial phase of the bootstrap operation.
- gcc-cross-initial: An early stage of the bootstrap process for creating the cross-compiler. This stage builds enough of the gcc-cross, the C library, and other pieces needed to finish building the final cross-compiler in later stages. This tool is a "native" package (i.e. it is designed to run on the build host).
- linux-libc-headers: Headers needed for the cross-compiler.
- glibc-initial: An initial version of the Embedded GNU C Library (GLIBC) needed to bootstrap glibc.
- glibc: The GNU C Library.
- gcc-cross: The final stage of the bootstrap process for the cross-compiler. This stage results in the actual cross-compiler that BitBake uses when it builds an image for a targeted device.

  > **Note**
  > If you are replacing this cross compiler toolchain with a custom version, you must replace gcc-cross.

  This tool is also a "native" package (i.e. it is designed to run on the build host).
- gcc-runtime: Runtime libraries resulting from the toolchain bootstrapping process. This tool produces a binary that consists of the runtime libraries need for the targeted device.

You can use the OpenEmbedded build system to build an installer for the relocatable SDK used to develop applications. When you run the installer, it installs the toolchain, which contains the development tools (e.g., gcc-cross-canadian, binutils-cross-canadian, and other nativesdk-* tools), which are tools native to the SDK (i.e. native to SDK_ARCH [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDK_ARCH]), you need to cross-compile and test your software. The figure shows the commands you use to easily build out this toolchain. This cross-development toolchain is built to execute on the SDKMACHINE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKMACHINE], which might or might not be the same machine as the Build Host.

> **Note**
> If your target architecture is supported by the Yocto Project, you can take advantage of pre-built images that ship with the Yocto Project and already contain cross-development toolchain installers.

Here is the bootstrap process for the relocatable toolchain:

```
gcc -> binutils-crosssdk -> gcc-crosssdk-initial -> linux-libc-headers ->
   glibc-initial -> nativesdk-glibc -> gcc-crosssdk -> gcc-cross-canadian
```

- gcc: The build host's GNU Compiler Collection (GCC).

- `binutils-crosssdk`: The bare minimum binary utilities needed in order to run the `gcc-crosssdk-initial` phase of the bootstrap operation.
- `gcc-crosssdk-initial`: An early stage of the bootstrap process for creating the cross-compiler. This stage builds enough of the `gcc-crosssdk` and supporting pieces so that the final stage of the bootstrap process can produce the finished cross-compiler. This tool is a "native" binary that runs on the build host.
- `linux-libc-headers`: Headers needed for the cross-compiler.
- `glibc-initial`: An initial version of the Embedded GLIBC needed to bootstrap `nativesdk-glibc`.
- `nativesdk-glibc`: The Embedded GLIBC needed to bootstrap the `gcc-crosssdk`.
- `gcc-crosssdk`: The final stage of the bootstrap process for the relocatable cross-compiler. The `gcc-crosssdk` is a transitory compiler and never leaves the build host. Its purpose is to help in the bootstrap process to create the eventual `gcc-cross-canadian` compiler, which is relocatable. This tool is also a "native" package (i.e. it is designed to run on the build host).
- `gcc-cross-canadian`: The final relocatable cross-compiler. When run on the `SDKMACHINE` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SDKMACHINE], this tool produces executable code that runs on the target device. Only one cross-canadian compiler is produced per architecture since they can be targeted at different processor optimizations using configurations passed to the compiler through the compile commands. This circumvents the need for multiple compilers and thus reduces the size of the toolchains.

> **Note**
> For information on advantages gained when building a cross-development toolchain installer, see the "Building an SDK Installer [http://www.yoctoproject.org/docs/3.0.3/sdk-manual/sdk-manual.html#sdk-building-an-sdk-installer]" appendix in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

4.5. Shared State Cache

By design, the OpenEmbedded build system builds everything from scratch unless BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] can determine that parts do not need to be rebuilt. Fundamentally, building from scratch is attractive as it means all parts are built fresh and no possibility of stale data exists that can cause problems. When developers hit problems, they typically default back to building from scratch so they have a know state from the start.

Building an image from scratch is both an advantage and a disadvantage to the process. As mentioned in the previous paragraph, building from scratch ensures that everything is current and starts from a known state. However, building from scratch also takes much longer as it generally means rebuilding things that do not necessarily need to be rebuilt.

The Yocto Project implements shared state code that supports incremental builds. The implementation of the shared state code answers the following questions that were fundamental roadblocks within the OpenEmbedded incremental build support system:

- What pieces of the system have changed and what pieces have not changed?
- How are changed pieces of software removed and replaced?
- How are pre-built components that do not need to be rebuilt from scratch used when they are available?

For the first question, the build system detects changes in the "inputs" to a given task by creating a checksum (or signature) of the task's inputs. If the checksum changes, the system assumes the inputs have changed and the task needs to be rerun. For the second question, the shared state (sstate) code tracks which tasks add which output to the build process. This means the output from a given task can be removed, upgraded or otherwise manipulated. The third question is partly addressed by the solution for the second question assuming the build system can fetch the sstate objects from remote locations and install them if they are deemed to be valid.

Notes

- The build system does not maintain PR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PR] information as part of the shared state packages. Consequently, considerations exist that affect maintaining shared state feeds. For information on how the build system works with packages and can track incrementing PR information, see the "Automatically Incrementing a Binary Package Revision Number [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#automatically-incrementing-a-binary-package-revision-number]" section in the Yocto Project Development Tasks Manual.
- The code in the build system that supports incremental builds is not simple code. For techniques that help you work around issues related to shared state code, see the "Viewing Metadata Used to Create the Input Signature of a Shared State Task [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#dev-viewing-metadata-used-to-create-the-input-signature-of-a-shared-state-task]" and "Invalidating Shared State to Force a Task to Run [http://www.yoctoproject.org/docs/3.0.3/dev-manual/dev-manual.html#dev-invalidating-shared-state-to-force-a-task-to-run]" sections both in the Yocto Project Development Tasks Manual.

The rest of this section goes into detail about the overall incremental build architecture, the checksums (signatures), and shared state.

4.5.1. Overall Architecture

When determining what parts of the system need to be built, BitBake works on a per-task basis rather than a per-recipe basis. You might wonder why using a per-task basis is preferred over a per-recipe basis. To help explain, consider having the IPK packaging backend enabled and then switching to DEB. In this case, the `do_install` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-install] and `do_package` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package] task outputs are still valid. However, with a per-recipe approach, the build would not include the `.deb` files. Consequently, you would have to invalidate the whole build and rerun it. Rerunning everything is not the best solution. Also, in this case, the core must be "taught" much about specific tasks. This methodology does not scale well and does not allow users to easily add new tasks in layers or as external recipes without touching the packaged-staging core.

4.5.2. Checksums (Signatures)

The shared state code uses a checksum, which is a unique signature of a task's inputs, to determine if a task needs to be run again. Because it is a change in a task's inputs that triggers a rerun, the process needs to detect all the inputs to a given task. For shell tasks, this turns out to be fairly easy because the build process generates a "run" shell script for each task and it is possible to create a checksum that gives you a good idea of when the task's data changes.

To complicate the problem, there are things that should not be included in the checksum. First, there is the actual specific build path of a given task - the `WORKDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR]. It does not matter if the work directory changes because it should not affect the output for target packages. Also, the build process has the objective of making native or cross packages relocatable.

Note

Both native and cross packages run on the build host [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#hardware-build-system-term]. However, cross packages generate output for the target architecture.

The checksum therefore needs to exclude `WORKDIR`. The simplistic approach for excluding the work directory is to set `WORKDIR` to some fixed value and create the checksum for the "run" script.

Another problem results from the "run" scripts containing functions that might or might not get called. The incremental build solution contains code that figures out dependencies between shell functions. This code is used to prune the "run" scripts down to the minimum set, thereby alleviating this problem and making the "run" scripts much more readable as a bonus.

So far, solutions for shell scripts exist. What about Python tasks? The same approach applies even though these tasks are more difficult. The process needs to figure out what variables a Python function

accesses and what functions it calls. Again, the incremental build solution contains code that first figures out the variable and function dependencies, and then creates a checksum for the data used as the input to the task.

Like the WORKDIR case, situations exist where dependencies should be ignored. For these situations, you can instruct the build process to ignore a dependency by using a line like the following:

```
PACKAGE_ARCHS[vardepsexclude] = "MACHINE"
```

This example ensures that the PACKAGE_ARCHS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PACKAGE_ARCHS] variable does not depend on the value of MACHINE [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-MACHINE], even if it does reference it.

Equally, there are cases where you need to add dependencies BitBake is not able to find. You can accomplish this by using a line like the following:

```
PACKAGE_ARCHS[vardeps] = "MACHINE"
```

This example explicitly adds the MACHINE variable as a dependency for PACKAGE_ARCHS.

As an example, consider a case with in-line Python where BitBake is not able to figure out dependencies. When running in debug mode (i.e. using -DDD), BitBake produces output when it discovers something for which it cannot figure out dependencies. The Yocto Project team has currently not managed to cover those dependencies in detail and is aware of the need to fix this situation.

Thus far, this section has limited discussion to the direct inputs into a task. Information based on direct inputs is referred to as the "basehash" in the code. However, the question of a task's indirect inputs still exits - items already built and present in the Build Directory [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-directory]. The checksum (or signature) for a particular task needs to add the hashes of all the tasks on which the particular task depends. Choosing which dependencies to add is a policy decision. However, the effect is to generate a master checksum that combines the basehash and the hashes of the task's dependencies.

At the code level, a variety of ways exist by which both the basehash and the dependent task hashes can be influenced. Within the BitBake configuration file, you can give BitBake some extra information to help it construct the basehash. The following statement effectively results in a list of global variable dependency excludes (i.e. variables never included in any checksum):

```
BB_HASHBASE_WHITELIST ?= "TMPDIR FILE PATH PWD BB_TASKHASH BBPATH DL_DIR \
    SSTATE_DIR THISDIR FILESEXTRAPATHS FILE_DIRNAME HOME LOGNAME SHELL TERM \
    USER FILESPATH STAGING_DIR_HOST STAGING_DIR_TARGET COREBASE PRSERV_HOST \
    PRSERV_DUMPDIR PRSERV_DUMPFILE PRSERV_LOCKDOWN PARALLEL_MAKE \
    CCACHE_DIR EXTERNAL_TOOLCHAIN CCACHE CCACHE_DISABLE LICENSE_PATH SDKPKGSUFFIX"
```

The previous example excludes WORKDIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR] since that variable is actually constructed as a path within TMPDIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-TMPDIR], which is on the whitelist.

The rules for deciding which hashes of dependent tasks to include through dependency chains are more complex and are generally accomplished with a Python function. The code in meta/lib/oe/sstatesig.py shows two examples of this and also illustrates how you can insert your own policy into the system if so desired. This file defines the two basic signature generators OE-Core [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#oe-core] uses: "OEBasic" and "OEBasicHash". By default, a dummy "noop" signature handler is enabled in BitBake. This means that behavior is unchanged from previous versions. OE-Core uses the "OEBasicHash" signature handler by default through this setting in the bitbake.conf file:

```
BB_SIGNATURE_HANDLER ?= "OEBasicHash"
```

The "OEBasicHash" `BB_SIGNATURE_HANDLER` is the same as the "OEBasic" version but adds the task hash to the stamp files. This results in any metadata change that changes the task hash, automatically causing the task to be run again. This removes the need to bump PR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PR] values, and changes to metadata automatically ripple across the build.

It is also worth noting that the end result of these signature generators is to make some dependency and hash information available to the build. This information includes:

- `BB_BASEHASH_task-taskname`: The base hashes for each task in the recipe.
- `BB_BASEHASH_filename:taskname`: The base hashes for each dependent task.
- `BBHASHDEPS_filename:taskname`: The task dependencies for each task.
- `BB_TASKHASH`: The hash of the currently running task.

4.5.3. Shared State

Checksums and dependencies, as discussed in the previous section, solve half the problem of supporting a shared state. The other half of the problem is being able to use checksum information during the build and being able to reuse or rebuild specific components.

The `sstate` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-sstate] class is a relatively generic implementation of how to "capture" a snapshot of a given task. The idea is that the build process does not care about the source of a task's output. Output could be freshly built or it could be downloaded and unpacked from somewhere. In other words, the build process does not need to worry about its origin.

Two types of output exist. One type is just about creating a directory in `WORKDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR]. A good example is the output of either `do_install` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-install] or `do_package` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package]. The other type of output occurs when a set of data is merged into a shared directory tree such as the sysroot.

The Yocto Project team has tried to keep the details of the implementation hidden in `sstate` class. From a user's perspective, adding shared state wrapping to a task is as simple as this `do_deploy` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-deploy] example taken from the `deploy` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-deploy] class:

```
DEPLOYDIR = "${WORKDIR}/deploy-${PN}"
SSTATETASKS += "do_deploy"
do_deploy[sstate-inputdirs] = "${DEPLOYDIR}"
do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}"

python do_deploy_setscene () {
    sstate_setscene(d)
}
addtask do_deploy_setscene
do_deploy[dirs] = "${DEPLOYDIR} ${B}"
do_deploy[stamp-extra-info] = "${MACHINE_ARCH}"
```

The following list explains the previous example:

- Adding "do_deploy" to SSTATETASKS adds some required sstate-related processing, which is implemented in the `sstate` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-classes-sstate] class, to before and after the `do_deploy` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-deploy] task.

- The do_deploy[sstate-inputdirs] = "${DEPLOYDIR}" declares that do_deploy places its output in ${DEPLOYDIR} when run normally (i.e. when not using the sstate cache). This output becomes the input to the shared state cache.
- The do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}" line causes the contents of the shared state cache to be copied to ${DEPLOY_DIR_IMAGE}.

  ## Note

  If do_deploy is not already in the shared state cache or if its input checksum (signature) has changed from when the output was cached, the task runs to populate the shared state cache, after which the contents of the shared state cache is copied to ${DEPLOY_DIR_IMAGE}. If do_deploy is in the shared state cache and its signature indicates that the cached output is still valid (i.e. if no relevant task inputs have changed), then the contents of the shared state cache copies directly to ${DEPLOY_DIR_IMAGE} by the do_deploy_setscene task instead, skipping the do_deploy task.

- The following task definition is glue logic needed to make the previous settings effective:

  ```
  python do_deploy_setscene () {
      sstate_setscene(d)
  }
  addtask do_deploy_setscene
  ```

  sstate_setscene() takes the flags above as input and accelerates the do_deploy task through the shared state cache if possible. If the task was accelerated, sstate_setscene() returns True. Otherwise, it returns False, and the normal do_deploy task runs. For more information, see the "setscene [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#setscene]" section in the BitBake User Manual.

- The do_deploy[dirs] = "${DEPLOYDIR} ${B}" line creates ${DEPLOYDIR} and ${B} before the do_deploy task runs, and also sets the current working directory of do_deploy to ${B}. For more information, see the "Variable Flags [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#variable-flags]" section in the BitBake User Manual.

  ## Note

  In cases where sstate-inputdirs and sstate-outputdirs would be the same, you can use sstate-plaindirs. For example, to preserve the ${PKGD} and ${PKGDEST} output from the do_package [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package] task, use the following:

  ```
  do_package[sstate-plaindirs] = "${PKGD} ${PKGDEST}"
  ```

- The do_deploy[stamp-extra-info] = "${MACHINE_ARCH}" line appends extra metadata to the stamp file. In this case, the metadata makes the task specific to a machine's architecture. See "The Task List [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#ref-bitbake-tasklist]" section in the BitBake User Manual for more information on the stamp-extra-info flag.
- sstate-inputdirs and sstate-outputdirs can also be used with multiple directories. For example, the following declares PKGDESTWORK and SHLIBWORK as shared state input directories, which populates the shared state cache, and PKGDATA_DIR and SHLIBSDIR as the corresponding shared state output directories:

  ```
  do_package[sstate-inputdirs] = "${PKGDESTWORK} ${SHLIBSWORKDIR}"
  do_package[sstate-outputdirs] = "${PKGDATA_DIR} ${SHLIBSDIR}"
  ```

- These methods also include the ability to take a lockfile when manipulating shared state directory structures, for cases where file additions or removals are sensitive:

```
do_package[sstate-lockfile] = "${PACKAGELOCK}"
```

Behind the scenes, the shared state code works by looking in SSTATE_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SSTATE_DIR] and SSTATE_MIRRORS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-SSTATE_MIRRORS] for shared state files. Here is an example:

```
SSTATE_MIRRORS ?= "\
file://.* http://someserver.tld/share/sstate/PATH;downloadfilename=PATH \n \
file://.* file:///some/local/dir/sstate/PATH"
```

Note

The shared state directory (SSTATE_DIR) is organized into two-character subdirectories, where the subdirectory names are based on the first two characters of the hash. If the shared state directory structure for a mirror has the same structure as SSTATE_DIR, you must specify "PATH" as part of the URI to enable the build system to map to the appropriate subdirectory.

The shared state package validity can be detected just by looking at the filename since the filename contains the task checksum (or signature) as described earlier in this section. If a valid shared state package is found, the build process downloads it and uses it to accelerate the task.

The build processes use the *_setscene tasks for the task acceleration phase. BitBake goes through this phase before the main execution code and tries to accelerate any tasks for which it can find shared state packages. If a shared state package for a task is available, the shared state package is used. This means the task and any tasks on which it is dependent are not executed.

As a real world example, the aim is when building an IPK-based image, only the do_package_write_ipk [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_ipk] tasks would have their shared state packages fetched and extracted. Since the sysroot is not used, it would never get extracted. This is another reason why a task-based approach is preferred over a recipe-based approach, which would have to install the output from every task.

4.6. Automatically Added Runtime Dependencies

The OpenEmbedded build system automatically adds common types of runtime dependencies between packages, which means that you do not need to explicitly declare the packages using RDEPENDS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-RDEPENDS]. Three automatic mechanisms exist (shlibdeps, pcdeps, and depchains) that handle shared libraries, package configuration (pkg-config) modules, and -dev and -dbg packages, respectively. For other types of runtime dependencies, you must manually declare the dependencies.

- shlibdeps: During the do_package [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package] task of each recipe, all shared libraries installed by the recipe are located. For each shared library, the package that contains the shared library is registered as providing the shared library. More specifically, the package is registered as providing the soname [https://en.wikipedia.org/wiki/Soname] of the library. The resulting shared-library-to-package mapping is saved globally in PKGDATA_DIR [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PKGDATA_DIR] by the do_packagedata [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-packagedata] task.

  Simultaneously, all executables and shared libraries installed by the recipe are inspected to see what shared libraries they link against. For each shared library dependency that is found, PKGDATA_DIR is queried to see if some package (likely from a different recipe) contains the shared library. If such a package is found, a runtime dependency is added from the package that depends on the shared library to the package that contains the library.

  The automatically added runtime dependency also includes a version restriction. This version restriction specifies that at least the current version of the package that provides the shared library must be used, as if "package (>= version)" had been added to RDEPENDS. This forces an upgrade of

the package containing the shared library when installing the package that depends on the library, if needed.

If you want to avoid a package being registered as providing a particular shared library (e.g. because the library is for internal use only), then add the library to PRIVATE_LIBS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-PRIVATE_LIBS] inside the package's recipe.

- pcdeps: During the do_package task of each recipe, all pkg-config modules (*.pc files) installed by the recipe are located. For each module, the package that contains the module is registered as providing the module. The resulting module-to-package mapping is saved globally in PKGDATA_DIR by the do_packagedata task.

  Simultaneously, all pkg-config modules installed by the recipe are inspected to see what other pkg-config modules they depend on. A module is seen as depending on another module if it contains a "Requires:" line that specifies the other module. For each module dependency, PKGDATA_DIR is queried to see if some package contains the module. If such a package is found, a runtime dependency is added from the package that depends on the module to the package that contains the module.

  > **Note**
  > The pcdeps mechanism most often infers dependencies between -dev packages.

- depchains: If a package foo depends on a package bar, then foo-dev and foo-dbg are also made to depend on bar-dev and bar-dbg, respectively. Taking the -dev packages as an example, the bar-dev package might provide headers and shared library symlinks needed by foo-dev, which shows the need for a dependency between the packages.

  The dependencies added by depchains are in the form of RRECOMMENDS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-RRECOMMENDS].

  > **Note**
  > By default, foo-dev also has an RDEPENDS-style dependency on foo, because the default value of RDEPENDS_${PN}-dev (set in bitbake.conf) includes "${PN}".

  To ensure that the dependency chain is never broken, -dev and -dbg packages are always generated by default, even if the packages turn out to be empty. See the ALLOW_EMPTY [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-ALLOW_EMPTY] variable for more information.

The do_package task depends on the do_packagedata task of each recipe in DEPENDS [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-DEPENDS] through use of a [deptask [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#variable-flags]] declaration, which guarantees that the required shared-library/module-to-package mapping information will be available when needed as long as DEPENDS has been correctly set.

4.7. Fakeroot and Pseudo

Some tasks are easier to implement when allowed to perform certain operations that are normally reserved for the root user (e.g. do_install [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-install], do_package_write* [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-package_write_deb], do_rootfs [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-rootfs], and do_image* [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#ref-tasks-image]). For example, the do_install task benefits from being able to set the UID and GID of installed files to arbitrary values.

One approach to allowing tasks to perform root-only operations would be to require BitBake [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#bitbake-term] to run as root. However, this method is cumbersome and has security issues. The approach that is actually used is to run tasks that benefit from root privileges in a "fake" root environment. Within this environment, the task and its child processes believe that they are running as the root user, and see an internally consistent view of the filesystem. As long as generating the final output (e.g. a package or an image) does not

require root privileges, the fact that some earlier steps ran in a fake root environment does not cause problems.

The capability to run tasks in a fake root environment is known as "fakeroot [http://man.he.net/man1/fakeroot]", which is derived from the BitBake keyword/variable flag that requests a fake root environment for a task.

In the OpenEmbedded build system [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#build-system-term], the program that implements fakeroot is known as Pseudo [https://www.yoctoproject.org/software-item/pseudo/]. Pseudo overrides system calls by using the environment variable `LD_PRELOAD`, which results in the illusion of running as root. To keep track of "fake" file ownership and permissions resulting from operations that require root permissions, Pseudo uses an SQLite 3 database. This database is stored in `${WORKDIR` [http://www.yoctoproject.org/docs/3.0.3/ref-manual/ref-manual.html#var-WORKDIR]`}/pseudo/files.db` for individual recipes. Storing the database in a file as opposed to in memory gives persistence between tasks and builds, which is not accomplished using fakeroot.

Caution

If you add your own task that manipulates the same files or directories as a fakeroot task, then that task also needs to run under fakeroot. Otherwise, the task cannot run root-only operations, and cannot see the fake file ownership and permissions set by the other task. You need to also add a dependency on `virtual/fakeroot-native:do_populate_sysroot`, giving the following:

```
fakeroot do_mytask () {
    ...
}
do_mytask[depends] += "virtual/fakeroot-native:do_populate_sysroot"
```

For more information, see the `FAKEROOT*` [http://www.yoctoproject.org/docs/3.0.3/bitbake-user-manual/bitbake-user-manual.html#var-FAKEROOT] variables in the BitBake User Manual. You can also reference the "Why Not Fakeroot? [https://github.com/wrpseudo/pseudo/wiki/WhyNotFakeroot]" article for background information on Fakeroot and Pseudo.