Scott Rifenbark, Intel Corporation **<scott.m.rifenbark@intel.com>**

by Scott Rifenbark

Manual Notes

- This version of The Yocto Project Development Manual is for the 1.2.2 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 Development Manual

1.1. Introduction

Welcome to the Yocto Project Development Manual! This manual gives you an idea of how to use the Yocto Project to develop embedded Linux images and user-space applications to run on targeted devices. Reading this manual gives you an overview of image, kernel, and user-space application development using the Yocto Project. Because much of the information in this manual is general, it contains many references to other sources where you can find more detail. For example, detailed information on Git, repositories and open source in general can be found in many places. Another example is how to get set up to use the Yocto Project, which our Yocto Project Quick Start covers.

The Yocto Project Development Manual, however, does provide detailed examples on how to create a Board Support Package (BSP), change the kernel source code, and re-configure the kernel. You can find this information in the appendices of the manual.

1.2. What this Manual Provides

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

- Information that lets you get set up to develop using the Yocto Project.
- Information to help developers who are new to the open source environment and to the distributed revision control system Git, which the Yocto Project uses.
- An understanding of common end-to-end development models and tasks.
- Development case overviews for both system development and user-space applications.
- An overview and understanding of the emulation environment used with the Yocto Project (QEMU).
- An understanding of basic kernel architecture and concepts.
- Many references to other sources of related information.

1.3. What this Manual Does Not Provide

This manual will not give you the following:

- Step-by-step instructions if those instructions exist in other Yocto Project documentation. For example, the Application Development Toolkit (ADT) User’s Guide contains detailed instruction on how to obtain and configure the Eclipse™ Yocto Plug-in.
- 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/1.2.2/poky-ref-manual/poky-ref-manual.html].
- Detailed public information that is not specific to the Yocto Project. For example, exhaustive information on how to use Git is covered better through the Internet than in this manual.

1.4. Other Information

Because this manual presents overview information for many different topics, you will need to supplement it with other information. The following list presents other sources of information you might find helpful:

- The Yocto Project Website [http://www.yoctoproject.org]: The home page for the Yocto Project provides lots of information on the project as well as links to software and documentation.

- The Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html]: This short document lets you get started with the Yocto Project quickly and start building an image.
- The Yocto Project Reference Manual [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html]: This manual is a reference guide to the Yocto Project build component known as "Poky." The manual also contains a reference chapter on Board Support Package (BSP) layout.
- The Yocto Project Application Development Toolkit (ADT) User's Guide [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html]: This guide provides information that lets you get going with the ADT to develop projects using the Yocto Project.
- The Yocto Project Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html]: This guide defines the structure for BSP components. Having a commonly understood structure encourages standardization.
- The Yocto Project Kernel Architecture and Use Manual [http://www.yoctoproject.org/docs/1.2.2/kernel-manual/kernel-manual.html]: This manual describes the architecture of the Yocto Project kernel and provides some work flow examples.
- Yocto Eclipse Plug-in [http://www.youtube.com/watch?v=3ZlOu-gLsh0]: A step-by-step instructional video that demonstrates how an application developer uses Yocto Plug-in features within the Eclipse IDE.
- FAQ [https://wiki.yoctoproject.org/wiki/FAQ]: A list of commonly asked questions and their answers.
- Release Notes [http://www.yoctoproject.org/download/yocto/yocto-project-1.1-release-notes-poky-7.0.2]: Features, updates and known issues for the current release of the Yocto Project.
- Hob [http://www.yoctoproject.org/projects/hob]: A graphical user interface for BitBake. Hob's primary goal is to enable a user to perform common tasks more easily.
- Build Appliance [http://www.yoctoproject.org/documentation/build-appliance]: Allows you to build and boot a custom embedded Linux image with the Yocto Project using a non-Linux development system.
- Bugzilla [http://bugzilla.yoctoproject.org]: The bug tracking application the Yocto Project uses. If you find problems with the Yocto Project, you should report them using this application.
- Yocto Project Mailing Lists: To subscribe to the Yocto Project mailing lists, click on the following URLs and follow the instructions:
  - http://lists.yoctoproject.org/listinfo/yocto for a Yocto Project Discussions mailing list.
  - http://lists.yoctoproject.org/listinfo/poky for a Yocto Project Discussions mailing list about the Poky build system.
  - http://lists.yoctoproject.org/listinfo/yocto-announce for a mailing list to receive offical Yocto Project announcements for developments and as well as Yocto Project milestones.
- Internet Relay Chat (IRC): Two IRC channels on freenode are available for Yocto Project and Poky discussions: `#yocto` and `#poky`.
- OpenedHand [http://o-hand.com]: The company where the Yocto Project build system Poky was first developed. OpenedHand has since been acquired by Intel Corporation.
- Intel Corporation [http://www.intel.com/]: The company that acquired OpenedHand in 2008 and continues development on the Yocto Project.
- OpenEmbedded [http://www.openembedded.org]: The upstream, generic, embedded distribution the Yocto Project build system (Poky) derives from and to which it contributes.
- BitBake [http://developer.berlios.de/projects/bitbake/]: The tool used to process Yocto Project metadata.
- BitBake User Manual [http://bitbake.berlios.de/manual/]: A comprehensive guide to the BitBake tool.

- Pimlico [http://pimlico-project.org/]: A suite of lightweight Personal Information Management (PIM) applications designed primarily for handheld and mobile devices.
- QEMU [http://wiki.qemu.org/Index.html]: An open-source machine emulator and virtualizer.

Chapter 2. Getting Started with the Yocto Project

This chapter introduces the Yocto Project and gives you an idea of what you need to get started. You can find enough information to set up your development host and build or use images for hardware supported by the Yocto Project by reading The Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html].

The remainder of this chapter summarizes what is in the Yocto Project Quick Start and provides some higher-level concepts you might want to consider.

2.1. Introducing the Yocto Project

The Yocto Project is an open-source collaboration project focused on embedded Linux development. The project currently provides a build system, which is sometimes referred to as "Poky", and provides various ancillary tools suitable for the embedded developer. The Yocto Project also features the Sato reference User Interface, which is optimized for stylus driven, low-resolution screens.

You can use the Yocto Project build system, which uses BitBake [http://bitbake.berlios.de/manual/], to develop complete Linux images and associated user-space applications for architectures based on ARM, MIPS, PowerPC, x86 and x86-64. While the Yocto Project does not provide a strict testing framework, it does provide or generate for you artifacts that let you perform target-level and emulated testing and debugging. Additionally, if you are an Eclipse™ IDE user, you can install an Eclipse Yocto Plug-in to allow you to develop within that familiar environment.

2.2. Getting Set Up

Here is what you need to get set up to use the Yocto Project:

- Host System: You should have a reasonably current Linux-based host system. You will have the best results with a recent release of Fedora, OpenSUSE, or Ubuntu as these releases are frequently tested against the Yocto Project and officially supported. You should also have about 100 gigabytes of free disk space for building images.
- Packages: The Yocto Project requires certain packages exist on your development system (e.g. Python 2.6 or 2.7). See "The Packages [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#packages]" section in the Yocto Project Quick start for the exact package requirements and the installation commands to install them for the supported distributions.
- Yocto Project Release: You need a release of the Yocto Project. You can get set up with local Yocto Project Files one of two ways depending on whether you are going to be contributing back into the Yocto Project source repository or not.

  > **Note**
  >
  > Regardless of the method you use, this manual refers to the resulting hierarchical set of files as the "Yocto Project Files" or the "Yocto Project File Structure."

  - Tarball Extraction: If you are not going to contribute back into the Yocto Project, you can simply download the Yocto Project release you want from the website's download page [http://www.yoctoproject.org/download]. Once you have the tarball, just extract it into a directory of your choice.

    For example, the following command extracts the Yocto Project 1.2.2 release tarball into the current working directory and sets up the Yocto Project file structure with a top-level directory named `poky-denzil-7.0.2`:

    ```
    $ tar xfj poky-denzil-7.0.2.tar.bz2
    ```

    This method does not produce a Git repository. Instead, you simply end up with a local snapshot of the Yocto Project files that are based on the particular release in the tarball.

- Git Repository Method: If you are going to be contributing back into the Yocto Project or you simply want to keep up with the latest developments, you should use Git commands to set up a local Git repository of the Yocto Project Files. Doing so creates a Git repository with a complete history of changes and allows you to easily submit your changes upstream to the project. Because you cloned the repository, you have access to all the Yocto Project development branches and tag names used in the upstream repository.

  The following transcript shows how to clone the Yocto Project Files' Git repository into the current working directory.

  ### Note

  The name of the Yocto Project Files Git repository in the Yocto Project Files Source Repositories is poky. You can view the Yocto Project Source Repositories at http://git.yoctoproject.org/cgit.cgi

  The command creates the local repository in a directory named poky. For information on Git used within the Yocto Project, see the "Git" section.

  ```
  $ git clone git://git.yoctoproject.org/poky
  Initialized empty Git repository in /home/scottrif/poky/.git/
  remote: Counting objects: 141863, done.
  remote: Compressing objects: 100% (38624/38624), done.
  remote: Total 141863 (delta 99661), reused 141816 (delta 99614)
  Receiving objects: 100% (141863/141863), 76.64 MiB | 126 KiB/s, done.
  Resolving deltas: 100% (99661/99661), done.
  ```

  For another example of how to set up your own local Git repositories, see this wiki page [https://wiki.yoctoproject.org/wiki/Transcript:_from_git_checkout_to_meta-intel_BSP], which describes how to create both poky and `meta-intel` Git repositories.

- Linux Yocto Kernel: If you are going to be making modifications to a supported Linux Yocto kernel, you need to establish local copies of the source. You can find Git repositories of supported Linux Yocto Kernels organized under "Yocto Linux Kernel" in the Yocto Project Source Repositories at http://git.yoctoproject.org/cgit.cgi.

  This setup involves creating a bare clone of the Linux Yocto kernel and then copying that cloned repository. You can create the bare clone and the copy of the bare clone anywhere you like. For simplicity, it is recommended that you create these structures outside of the Yocto Project Files Git repository.

  As an example, the following transcript shows how to create the bare clone of the `linux-yocto-3.2` kernel and then create a copy of that clone.

  ### Note

  When you have a local Linux Yocto kernel Git repository, you can reference that repository rather than the upstream Git repository as part of the `clone` command. Doing so can speed up the process.

  In the following example, the bare clone is named `linux-yocto-3.2.git`, while the copy is named `my-linux-yocto-3.2-work`:

  ```
  $ git clone --bare git://git.yoctoproject.org/linux-yocto-3.2 linux-yocto-3.2.git
  Initialized empty Git repository in /home/scottrif/linux-yocto-3.2.git/
  remote: Counting objects: 2468027, done.
  remote: Compressing objects: 100% (392255/392255), done.
  remote: Total 2468027 (delta 2071693), reused 2448773 (delta 2052498)
  Receiving objects: 100% (2468027/2468027), 530.46 MiB | 129 KiB/s, done.
  Resolving deltas: 100% (2071693/2071693), done.
  ```

  Now create a clone of the bare clone just created:

```
$ git clone linux-yocto-3.2.git my-linux-yocto-3.2-work
Initialized empty Git repository in /home/scottrif/my-linux-yocto-3.2-work/.git/
Checking out files: 100% (37619/37619), done.
```

- The `poky-extras` Git Repository: The `poky-extras` Git repository contains metadata needed only if you are modifying and building the kernel image. In particular, it contains the kernel BitBake append (`.bbappend`) files that you edit to point to your locally modified kernel source files and to build the kernel image. Pointing to these local files is much more efficient than requiring a download of the source files from upstream each time you make changes to the kernel.

  You can find the `poky-extras` Git Repository in the "Yocto Metadata Layers" area of the Yocto Project Source Repositories at http://git.yoctoproject.org/cgit.cgi. It is good practice to create this Git repository inside the Yocto Project files Git repository.

  Following is an example that creates the `poky-extras` Git repository inside the Yocto Project files Git repository, which is named poky in this case:

  ```
  $ git clone git://git.yoctoproject.org/poky-extras poky-extras
  Initialized empty Git repository in /home/scottrif/poky/poky-extras/.git/
  remote: Counting objects: 618, done.
  remote: Compressing objects: 100% (558/558), done.
  remote: Total 618 (delta 192), reused 307 (delta 39)
  Receiving objects: 100% (618/618), 526.26 KiB | 111 KiB/s, done.
  Resolving deltas: 100% (192/192), done.
  ```

- Supported Board Support Packages (BSPs): The Yocto Project provides a layer called `meta-intel` and it is maintained in its own separate Git repository. The `meta-intel` layer contains many supported BSP Layers [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-layers].

  Similar considerations exist for setting up the `meta-intel` layer. You can get set up for BSP development one of two ways: tarball extraction or with a local Git repository. It is a good idea to use the same method used to set up the Yocto Project Files. Regardless of the method you use, the Yocto Project uses the following BSP layer naming scheme:

  ```
  meta-<BSP_name>
  ```

  where <BSP_name> is the recognized BSP name. Here are some examples:

  ```
  meta-crownbay
  meta-emenlow
  meta-n450
  ```

  See the "BSP Layers [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-layers]" section in the Yocto Project Board Support Package (BSP) Developer's Guide for more information on BSP Layers.

  - Tarball Extraction: You can download any released BSP tarball from the same download site [http://www.yoctoproject.org/download] used to get the Yocto Project release. Once you have the tarball, just extract it into a directory of your choice. Again, this method just produces a snapshot of the BSP layer in the form of a hierarchical directory structure.

  - Git Repository Method: If you are working with a Yocto Project Files Git repository, you should also use this method to set up the `meta-intel` Git repository. You can locate the `meta-intel` Git repository in the "Yocto Metadata Layers" area of the Yocto Project Source Repositories at http://git.yoctoproject.org/cgit.cgi.

Typically, you set up the `meta-intel` Git repository inside the Yocto Project Files Git repository. For example, the following transcript shows the steps to clone the `meta-intel` Git repository inside the poky Git repository.

```
$ git clone git://git.yoctoproject.org/meta-intel.git
Initialized empty Git repository in /home/scottrif/poky/meta-intel/.git/
remote: Counting objects: 3380, done.
remote: Compressing objects: 100% (2750/2750), done.
remote: Total 3380 (delta 1689), reused 227 (delta 113)
Receiving objects: 100% (3380/3380), 1.77 MiB | 128 KiB/s, done.
Resolving deltas: 100% (1689/1689), done.
```

The same wiki page [https://wiki.yoctoproject.org/wiki/Transcript:_from_git_checkout_to_meta-intel_BSP] referenced earlier covers how to set up the `meta-intel` Git repository.

- Eclipse Yocto Plug-in: If you are developing applications using the Eclipse Integrated Development Environment (IDE), you will need this plug-in. See the "Setting up the Eclipse IDE [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html#setting-up-the-eclipse-ide]" section in the Yocto Application Development Toolkit (ADT) User’s Guide for more information.

2.3. Building Images

The build process creates an entire Linux distribution, including the toolchain, from source. For more information on this topic, see the "Building an Image [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#building-image]" section in the Yocto Project Quick Start.

The build process is as follows:

1. Make sure you have the Yocto Project files as described in the previous section.
2. Initialize the build environment by sourcing a build environment script.
3. Optionally ensure the `/conf/local.conf` configuration file, which is found in the Yocto Project Build Directory, is set up how you want it. This file defines many aspects of the build environment including the target machine architecture through the `MACHINE [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-MACHINE]` variable, the development machine's processor use through the `BB_NUMBER_THREADS [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BB_NUMBER_THREADS]` and `PARALLEL_MAKE [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PARALLEL_MAKE]` variables, and a centralized tarball download directory through the `DL_DIR [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-DL_DIR]` variable.
4. Build the image using the `bitbake` command. If you want information on BitBake, see the user manual at http://docs.openembedded.org/bitbake/html.
5. Run the image either on the actual hardware or using the QEMU emulator.

2.4. Using Pre-Built Binaries and QEMU

Another option you have to get started is to use pre-built binaries. This scenario is ideal for developing software applications to run on your target hardware. To do this, you need to install the stand-alone Yocto Project cross-toolchain tarball and then download the pre-built kernel that you will boot in the QEMU emulator. Next, you must download and extract the target root filesystem for your target machine’s architecture. Finally, you set up the environment to emulate the hardware and then start the QEMU emulator.

You can find details on all these steps in the "Using Pre-Built Binaries and QEMU [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#using-pre-built]" section of the Yocto Project Quick Start.

Chapter 3. The Yocto Project Open Source Development Environment

This chapter helps you understand the Yocto Project as an open source development project. In general, working in an open source environment is very different from working in a closed, proprietary environment. Additionally, the Yocto Project uses specific tools and constructs as part of its development environment. This chapter specifically addresses open source philosophy, licensing issues, code repositories, the open source distributed version control system Git, and best practices using the Yocto Project.

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 produce 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) that 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. Using The Yocto Project in a Team Environment

It might not be immediately clear how you can use the Yocto Project in a team environment, or scale it for a large team of developers. The specifics of any situation determine the best solution. Granted that the Yocto Project offers immense flexibility regarding this, practices do exist that experience has shown work well.

The core component of any development effort with the Yocto Project is often an automated build and testing framework along with an image generation process. You can use these core components to check that the metadata can be built, highlight when commits break the build, and provide up-to-date images that allow developers to test the end result and use it as a base platform for further development. Experience shows that buildbot is a good fit for this role. What works well is to configure buildbot to make two types of builds: incremental and full (from scratch). See the buildbot for the Yocto Project [http://autobuilder.yoctoproject.org:8010/] for an example implementation that uses buildbot.

You can tie incremental builds to a commit hook that triggers the build each time a commit is made to the metadata. This practice results in useful acid tests that determine whether a given commit breaks the build in some serious way. Associating a build to a commit can catch a lot of simple errors. Furthermore, the tests are fast so developers can get quick feedback on changes.

Full builds build and test everything from the ground up. These types of builds usually happen at predetermined times like during the night when the machine load is low.

Most teams have many pieces of software undergoing active development at any given time. You can derive large benefits by putting these pieces under the control of a source control system that is compatible with the Yocto Project (i.e. Git or Subversion (SVN). You can then set the autobuilder to pull the latest revisions of the packages and test the latest commits by the builds. This practice quickly highlights issues. The Yocto Project easily supports testing configurations that use both a stable known good revision and a floating revision. The Yocto Project can also take just the changes from specific source control branches. This capability allows you to track and test specific changes.

Perhaps the hardest part of setting this up is defining the software project or the Yocto Project metadata policies that surround the different source control systems. Of course circumstances will be different in each case. However, this situation reveals one of the Yocto Project's advantages - the system itself does not force any particular policy on users, unlike a lot of build systems. The system allows the best policies to be chosen for the given circumstances.

3.3. Yocto Project Source Repositories

The Yocto Project team maintains complete source repositories for all Yocto Project files at http://git.yoctoproject.org/cgit/cgit.cgi. 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 set up a Git repository for that particular item. Having a local Git repository of the Yocto Project files allows you to make changes, contribute to the history, and ultimately enhance the Yocto Project's tools, Board Support Packages, and so forth.

Conversely, if you are a developer that is not interested in contributing back to the Yocto Project, you have the ability to simply download and extract release tarballs and use them within the Yocto Project environment. All that is required is a particular release of the Yocto Project and your application source code.

For any supported release of Yocto Project, you can go to the Yocto Project website’s download page [http://www.yoctoproject.org/download] and get a tarball of the release. You can also go to this site to download any supported BSP tarballs. Unpacking the tarball gives you a hierarchical directory structure of Yocto Project files that lets you develop using the Yocto Project.

Once you are set up through either tarball extraction or creation of Git repositories, you are ready to develop.

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

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

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

- Yocto Project Download Page [http://www.yoctoproject.org/download] This page on the Yocto Project website allows you to download any Yocto Project release or 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.

3.4. Yocto Project Terms

Following is a list of terms and definitions users new to the Yocto Project development environment might find helpful. While some of these terms are universal, the list includes them just in case:

- Append Files: Files that append build information to a recipe file. Append files are known as BitBake append files and `.bbappend` files. The Yocto Project build system expects every append file to have a corresponding and underlying recipe (`.bb`) file. Furthermore, the append file and the underlying recipe must have the same root filename. The filenames can differ only in the file type suffix used (e.g. `formfactor_0.0.bb` and `formfactor_0.0.bbappend`).

  Information in append files overrides the information in the similarly-named recipe file. For examples of `.bbappend` file in use, see the "Using .bbappend Files" and "Changing `recipes-kernel`" sections.

- BitBake: The task executor and scheduler used by the Yocto Project to build images. For more information on BitBake, see the BitBake documentation [http://bitbake.berlios.de/manual/].

- Classes: Files that provide for logic encapsulation and inheritance allowing commonly used patterns to be defined once and easily used in multiple recipes. Class files end with the `.bbclass` filename extension.

- Configuration File: Configuration information in various `.conf` files provides global definitions of variables. The `conf/local.conf` configuration file in the Yocto Project Build Directory contains user-defined variables that affect each build. The `meta-yocto/conf/distro/poky.conf` configuration file defines Yocto ‘distro’ configuration variables used only when building with this policy. Machine configuration files, which are located throughout the Yocto Project file structure, define variables for specific hardware and are only used when building for that target (e.g. the `machine/beagleboard.conf` configuration file defines variables for the Texas Instruments ARM Cortex-A8 development board). Configuration files end with a `.conf` filename extension.

- Cross-Development Toolchain: A collection of software development tools and utilities that allow you to develop software for targeted architectures. This toolchain contains cross-compilers, linkers, and debuggers that are specific to an architecture. You can use the Yocto Project to build cross-development toolchains in tarball form that when unpacked contain the development tools you need to cross-compile and test your software. The Yocto Project ships with images that contain toolchains for supported architectures as well. Sometimes this toolchain is referred to as the meta-toolchain.

- Image: An image is the result produced when BitBake processes a given collection of recipes and related metadata. Images are the binary output that runs on specific hardware and for specific use cases. For a list of the supported image types that the Yocto Project provides, see the "Reference: Images [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-images]" appendix in The Yocto Project Reference Manual.
- Layer: A collection of recipes representing the core, a BSP, or an application stack. For a discussion on BSP Layers, see the "BSP Layers [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-layers]" section in the Yocto Project Board Support Packages (BSP) Developer's Guide.
- Metadata: The files that BitBake parses when building an image. Metadata includes recipes, classes, and configuration files.
- OE-Core: A core set of metadata originating with OpenEmbedded (OE) that is shared between OE and the Yocto Project. This metadata is found in the `meta` directory of the Yocto Project files.
- Package: The packaged output from a baked recipe. A package is generally the compiled binaries produced from the recipe's sources. You 'bake' something by running it through BitBake.
- Poky: The build tool that the Yocto Project uses to create images.
- Recipe: A set of instructions for building packages. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes, and they also contain configuration and compilation options. Recipes contain the logical unit of execution, the software/images to build, and use the `.bb` file extension.
- Tasks: Arbitrary groups of software Recipes. You simply use Tasks to hold recipes that, when built, usually accomplish a single task. For example, a task could contain the recipes for a company's proprietary or value-add software. Or, the task could contain the recipes that enable graphics. A task is really just another recipe. Because task files are recipes, they end with the `.bb` filename extension.
- Upstream: A reference to source code or repositories that are not local to the development system but located in a master area that is controlled by the maintainer of the source code. For example, in order for a developer to work on a particular piece of code, they need to first get a copy of it from an "upstream" source.
- Yocto Project Files: This term refers to the directory structure created as a result of either downloading and unpacking a Yocto Project release tarball or setting up a Git repository by cloning `git://git.yoctoproject.org/poky`. Sometimes the term "the Yocto Project Files structure" is used as well.

  The Yocto Project Files contain BitBake, Documentation, metadata and other files that all support the development environment. Consequently, you must have the Yocto Project Files in place on your development system in order to do any development using the Yocto Project.

  The name of the top-level directory of the Yocto Project Files structure is derived from the Yocto Project release tarball. For example, downloading and unpacking `poky-denzil-7.0.2.tar.bz2` results in a Yocto Project file structure whose Yocto Project source directory is named `poky-denzil-7.0.2`. If you create a Git repository, then you can name the repository anything you like. Throughout much of the documentation, the name of the Git repository is used as the name for the local folder. So, for example, cloning the `poky` Git repository results in a local Git repository also named `poky`.

  It is important to understand the differences between Yocto Project Files created by unpacking a release tarball as compared to cloning `git://git.yoctoproject.org/poky`. When you unpack a tarball, you have an exact copy of the files based on the time of release - a fixed release point. Any changes you make to your local Yocto Project Files are on top of the release. On the other hand, when you clone the Yocto Project Git repository, you have an active development repository. In this case, any local changes you make to the Yocto Project can be later applied to active development branches of the upstream Yocto Project Git repository.

Finally, if you want to track a set of local changes while starting from the same point as a release tarball, you can create a local Git branch that reflects the exact copy of the files at the time of their release. You do this using Git tags that are part of the repository.

For more information on concepts around Git repositories, branches, and tags, see the "Repositories, Tags, and Branches" section.

- Yocto Project Build Directory: This term refers to the area used by the Yocto Project for builds. The area is created when you `source` the Yocto Project setup environment script that is found in the Yocto Project files area (i.e. `oe-init-build-env`). The `TOPDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-TOPDIR] variable points to the build directory.

  You have a lot of flexibility when creating the Yocto Project Build Directory. Following are some examples that show how to create the directory:

  - Create the build directory in your current working directory and name it `build`. This is the default behavior.

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

  - Provide a directory path and specifically name the build directory. This next example creates a build directory named `YP-7.0.2` in your home directory within the directory `mybuilds`. If `mybuilds` does not exist, the directory is created for you:

    ```
    $ source poky-denzil-7.0.2/oe-init-build-env $HOME/mybuilds/YP-7.0.2
    ```

  - Provide an existing directory to use as the build directory. This example uses the existing `mybuilds` directory as the build directory.

    ```
    $ source poky-denzil-7.0.2/oe-init-build-env $HOME/mybuilds/
    ```

3.5. 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 at 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 Yocto Project, the build process uses a known list of licenses to ensure compliance. You can find this list in the Yocto Project files 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 Yocto Project Build Directory at `tmp/deploy/images/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 are OSD-conformant.

You can find a list of the combined SPDX and OSI licenses that the Yocto Project uses here [http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/files/common-licenses]. This wiki page discusses the license infrastructure used by the Yocto Project.

3.6. Git

The Yocto Project uses 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 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.

For more information on Git, see http://git-scm.com/documentation. If you need to download Git, go to http://git-scm.com/download.

3.6.1. Repositories, Tags, and Branches

As mentioned earlier in section "Yocto Project Source Repositories", the Yocto Project maintains source repositories at http://git.yoctoproject.org/cgit.cgi. 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 in which you can do lots of local experimentation on a project 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 Poky 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 section "Getting Set Up" earlier in this 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 `laverne`, `bernard`, `edison`, `denzil` and `master` branches among others. 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 off-shoots 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 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, here is a set of commands that creates a local copy of the poky Git repository and then creates and checks out a local Git branch that tracks the Yocto Project 1.2.2 Release (denzil) development:

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

In this example, the name of the top-level directory of your local Yocto Project Files Git repository is poky. And, the name of the local working area (or local branch) you have created and checked out is named `denzil`. The files in your repository now reflect the same files that are in the `denzil` development branch of the Yocto Project's 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 development branch at the time you created your local branch, which could be different than the files at the time of a similarly named release. In other words, creating and checking out a local branch based on the `denzil` branch name is not the same as creating and checking out a local branch based on the `denzil-1.2.2` release. 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. Typically, a tag is used to mark a special point such as the final change 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 are `laverne-4.0`, `bernard-5.0`, and `denzil-7.0.2`. These tags represent Yocto Project releases.

When you create a local copy of the Git repository, you also have access to all the tags. 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 checkout -b my-denzil-7.0.2 denzil-7.0.2
```

In this example, the name of the top-level directory of your local Yocto Project Files Git repository is poky. And, the name of the local branch you have created and checked out is `my-denzil-7.0.2`. The files in your repository now exactly match the Yocto Project 1.2.2 Release tag (`denzil-7.0.2`). 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 a development branch.

3.6.2. 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]. If you need to download Git, you can do so here [http://git-scm.com/download].

If you don’t know much about Git, we suggest you educate yourself by visiting the links previously mentioned.

The following list 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 they support. 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 clone of a repository. During collaboration, this command allows you to create a local repository that is on equal footing with a fellow developer’s repository.
- `git add`: Adds updated file contents to the index that Git uses to track changes. You must add all files that have changed before you can commit them.
- `git commit`: Creates a “commit” that documents the changes you made. 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 (or master) repository.

- `git status`: Reports any modified files that possibly need to be added and committed.
- `git checkout <branch-name>`: Changes your working branch. This command is analogous to "cd".
- `git checkout –b <working-branch>`: Creates a working branch on your local machine where you can isolate work. It is a good idea to use local branches when adding specific features or changes. This way if you don't like what you have done you can easily get rid of the work.
- `git branch`: Reports existing branches and tells you which branch in which you are currently working.
- `git branch -D <branch-name>`: Deletes an existing branch. You need to be in a branch other than the one you are deleting in order to delete <branch-name>.
- `git pull`: 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 repository).
- `git push`: Sends all your local changes you have committed to an upstream Git repository (e.g. a contribution repository). The maintainer of the project draws from these repositories when adding your changes to the project's master 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 for isolated work, make and commit your changes, switch to your local master branch, merge the changes from the temporary branch into the local master branch, and then delete the temporary branch.
- `git cherry-pick`: 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.
- `git log`: Reports a history of your changes to the repository.
- `git diff`: Displays line-by-line differences between your local working files and the same files in the upstream Git repository that your branch currently tracks.

3.7. Workflows

This section provides some overview on workflows using Git. In particular, the information covers basic practices that describe roles and actions in a collaborative development environment. Again, if you are familiar with this type of development environment, you might want to just skip this section.

The Yocto Project files are maintained using Git in a "master" branch whose Git history tracks every change and whose structure provides 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 "master" branch of the Git repository. The "master" branch is the "upstream" repository where the final builds of the project occur. The maintainer is responsible for allowing changes in from other developers and for organizing the underlying branch structure to reflect release strategies and so forth.

Note

You can see who is the maintainer for Yocto Project files by examining the `distro_tracking_fields.inc` file in the Yocto Project `meta/conf/distro/include` directory.

The project also has contribution repositories known as "contrib" areas. These areas temporarily hold changes to the project that have been submitted or committed by the Yocto Project development team and by community members that contribute to the project. The maintainer determines if the changes are qualified to be moved from the "contrib" areas into the "master" branch of the Git repository.

Developers (including contributing community members) create and maintain cloned repositories of the upstream "master" branch. These 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 changes to the appropriate "contrib" repository.

Developers are responsible for keeping their local repository up-to-date with "master". 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 developer’s machine 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 "master" This process is called “submitting a patch” or “submitting a change.”

To summarize the environment: we have a single point of entry for changes into the project’s "master" branch of the Git repository, which is controlled by the project’s maintainer. And, we have a set of developers 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.

  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 in 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. 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, simply discard the 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 local Yocto Project files Git repository in the `scripts` directory.
- 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`. You can find information on how to submit later in this chapter.

3.8. Tracking Bugs

The Yocto Project uses its own implementation of Bugzilla [http://www.bugzilla.org/about/] to track bugs. Implementations of Bugzilla work well for group development because they track bugs and code changes, can be used to communicate changes and problems with developers, can be used to submit and review patches, and can be used to manage quality assurance. The home page for the Yocto Project implementation of Bugzilla is http://bugzilla.yoctoproject.org.

Sometimes it is helpful to submit, investigate, or track a bug against the Yocto Project itself such as when discovering an issue with some component of the build system that acts contrary to the documentation or your expectations. Following is the general procedure for submitting a new bug using the Yocto Project Bugzilla. You can find more information on defect management, bug tracking, and feature request processes all accomplished through the Yocto Project Bugzilla on the wiki page here [https://wiki.yoctoproject.org/wiki/Bugzilla_Configuration_and_Bug_Tracking].

1. Always use the Yocto Project implementation of Bugzilla to submit a bug.
2. When submitting a new bug, be sure to choose the appropriate Classification, Product, and Component for which the issue was found. Defects for Yocto Project fall into one of four classifications: Yocto Projects, Infrastructure, Poky, and Yocto Metadata Layers. Each of these Classifications break down into multiple Products and, in some cases, multiple Components.
3. Use the bug form to choose the correct Hardware and Architecture for which the bug applies.
4. Indicate the Yocto Project version you were using when the issue occurred.
5. Be sure to indicate the Severity of the bug. Severity communicates how the bug impacted your work.
6. Provide a brief summary of the issue. Try to limit your summary to just a line or two and be sure to capture the essence of the issue.
7. Provide a detailed description of the issue. You should provide as much detail as you can about the context, behavior, output, and so forth that surround the issue. You can even attach supporting files for output or log by using the "Add an attachment" button.
8. Submit the bug by clicking the "Submit Bug" button.

Note

Bugs in the Yocto Project Bugzilla follow naming convention: `[YOCTO #<number>]`, where <number> is the assigned defect ID used in Bugzilla. So, for example, a valid way to refer to a defect would be `[YOCTO #1011]`. This convention becomes important if you are submitting patches against the Yocto Project code itself.

3.9. How to Submit a Change

Contributions to the Yocto Project are very welcome. Because the Yocto Project is extremely configurable and flexible, we recognize that developers will want to extend, configure or optimize it for their specific uses. You should send patches to the appropriate Yocto Project mailing list to get them in front of the Yocto Project Maintainer. For a list of the Yocto Project mailing lists, see the "Mailing lists [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#resources-mailinglist]" section in The Yocto Project Reference Manual.

The following is some guidance on which mailing list to use for what type of defect:

- For defects against the Yocto Project build system Poky, send your patch to the http://lists.yoctoproject.org/listinfo/poky mailing list. This mailing list corresponds to issues that are not specific to the Yocto Project but are part of the OE-core. For example, a defect against anything in the `meta` layer or the BitBake Manual could be sent to this mailing list.
- For defects against Yocto-specific layers, tools, and Yocto Project documentation use the http://lists.yoctoproject.org/listinfo/yocto mailing list. This mailing list corresponds to Yocto-specific areas such as `meta-yocto`, `meta-intel`, `linux-yocto`, and `documentation`.

When you send a patch, be sure to include a "Signed-off-by:" line in the same style as required by the Linux kernel. Adding this line signifies the developer has agreed to the Developer's Certificate of Origin 1.1 as follows:

```
     Developer's Certificate of Origin 1.1

     By making a contribution to this project, I certify that:

     (a) The contribution was created in whole or in part by me and I
         have the right to submit it under the open source license
         indicated in the file; or

     (b) The contribution is based upon previous work that, to the best
         of my knowledge, is covered under an appropriate open source
         license and I have the right under that license to submit that
         work with modifications, whether created in whole or in part
         by me, under the same open source license (unless I am
         permitted to submit under a different license), as indicated
         in the file; or

     (c) The contribution was provided directly to me by some other
         person who certified (a), (b) or (c) and I have not modified
         it.

     (d) I understand and agree that this project and the contribution
         are public and that a record of the contribution (including all
         personal information I submit with it, including my sign-off) is
         maintained indefinitely and may be redistributed consistent with
         this project or the open source license(s) involved.
```

A Poky contributions tree (`poky-contrib`, `git://git.yoctoproject.org/poky-contrib.git`) exists for contributors to stage contributions. If people desire such access, please ask on the mailing list. Usually, the Yocto Project team will grant access to anyone with a proven track record of good patches.

In a collaborative environment, it is necessary to have some sort of standard or method through which you submit changes. Otherwise, things could get quite chaotic. One general practice to follow is to make small, controlled changes to the Yocto Project. Keeping changes small and isolated lets you best keep pace with future Yocto Project changes.

When you create a commit, you must follow certain standards established by the Yocto Project development team. For each commit, you must provide a single-line summary of the change and you almost always provide a more detailed description of what you did (i.e. the body of the commit). The only exceptions for not providing a detailed description would be if your change is a simple, self-explanatory change that needs no description. Here are the Yocto Project commit message guidelines:

- Provide a single-line, short summary of the change. This summary is typically viewable by source control systems. Thus, providing something short and descriptive that gives the reader a summary of the change is useful when viewing a list of many commits.

- For the body of the commit message, provide detailed information that describes what you changed, why you made the change, and the approach you used. Provide as much detail as you can in the body of the commit message.
- If the change addresses a specific bug or issue that is associated with a bug-tracking ID, prefix your detailed description with the bug or issue ID. For example, the Yocto Project tracks bugs using a bug-naming convention. Any commits that address a bug must start with the bug ID in the description as follows:

```
     YOCTO #<bug-id>:  <Detailed description of commit>
```

You can find more guidance on creating well-formed commit messages at this OpenEmbedded wiki page: http://www.openembedded.org/wiki/Commit_Patch_Message_Guidelines.

Following are general instructions for both pushing changes upstream and for submitting changes as patches.

3.9.1. Pushing a Change Upstream and Requesting a Pull

The basic flow for pushing a change to an upstream "contrib" Git repository is as follows:

- Make your changes in your local Git repository.
- Stage your commit (or change) by using the `git add` command.
- Commit the change by using the `git commit` command and push it to the "contrib" repository. Be sure to provide a commit message that follows the project’s commit standards as described earlier.
- Notify the maintainer that you have pushed a change by making a pull request. The Yocto Project provides two scripts that conveniently let you generate and send pull requests to the Yocto Project. These scripts are `create-pull-request` and `send-pull-request`. You can find these scripts in the `scripts` directory of the Yocto Project file structure.

  For help on using these scripts, simply provide the `--help` argument as follows:

```
     $ ~/poky/scripts/create-pull-request --help
     $ ~/poky/scripts/send-pull-request --help
```

You can find general Git information on how to push a change upstream in the Git Community Book [http://book.git-scm.com/3_distributed_workflows.html].

3.9.2. Submitting a Patch Through Email

If you have a just a few changes, you can commit them and then submit them as an email to the maintainer. Here is a general procedure:

- Make your changes in your local Git repository.
- Stage your commit (or change) by using the `git add` command.
- Commit the change by using the `git commit --signoff` command. Using the `--signoff` option identifies you as the person making the change and also satisfies the Developer's Certificate of Origin (DCO) shown earlier.

  When you form a commit you must follow certain standards established by the Yocto Project development team. See the earlier section "How to Submit a Change" for Yocto Project commit message standards.
- Format the commit into an email message. To format commits, use the `git format-patch` command. When you provide the command, you must include a revision list or a number of patches as part of the command. For example, these two commands each take the most recent single commit and format it as an email message in the current directory:

```
     $ git format-patch -1
     $ git format-patch HEAD~
```

After the command is run, the current directory contains a numbered `.patch` file for the commit.

If you provide several commits as part of the command, the `git format-patch` command produces a numbered series of files in the current directory – one for each commit. If you have more than one patch, you should also use the `--cover` option with the command, which generates a cover letter as the first "patch" in the series. You can then edit the cover letter to provide a description for the series of patches. For information on the `git format-patch` command, see `GIT_FORMAT_PATCH(1)` displayed using the `man git-format-patch` command.

- Import the files into your mail client by using the `git send-email` command.

Note

In order to use `git send-email`, you must have the the proper Git packages installed. For Ubuntu and Fedora the package is `git-email`.

The `git send-email` command sends email by using a local or remote Mail Transport Agent (MTA) such as `msmtp`, `sendmail`, or through a direct `smtp` configuration in your Git `config` file. If you are submitting patches through email only, it is very important that you submit them without any whitespace or HTML formatting that either you or your mailer introduces. The maintainer that receives your patches needs to be able to save and apply them directly from your emails. A good way to verify that what you are sending will be applicable by the maintainer is to do a dry run and send them to yourself and then save and apply them as the maintainer would.

The `git send-email` command is the preferred method for sending your patches since there is no risk of compromising whitespace in the body of the message, which can occur when you use your own mail client. The command also has several options that let you specify recipients and perform further editing of the email message. For information on how to use the `git send-email` command, use the `man git-send-email` command.

Chapter 4. Common Tasks

This chapter describes standard tasks such as adding new software packages, extending or customizing images or porting the Yocto Project to new hardware (adding a new machine). The chapter also describes ways to modify package source code, combine multiple versions of library files into a single image, and handle a package name alias. Finally, the chapter contains advice about how to make changes to the Yocto Project to achieve the best results.

4.1. Understanding and Creating Layers

The Yocto Project build system supports organizing metadata into multiple layers. Layers allow you to isolate different types of customizations from each other. You might find it tempting to keep everything in one layer when working on a single project. However, the more modular you organize your metadata, the easier it is to cope with future changes.

To illustrate how layers are used to keep things modular, consider machine customizations. These types of customizations typically reside in 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 a 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.

4.1.1. Yocto Project Layers

The Yocto Project contains several layers right out of the box. You can easily identify a layer in the Yocto Project by the name of its folder. Folders that are layers begin with the string `meta`. For example, when you set up the Yocto Project Files structure, you will see several layers: `meta`, `meta-demoapps`, `meta-skeleton`, and `meta-yocto`. Each of these folders is a layer.

Furthermore, if you set up a local copy of the `meta-intel` Git repository and then explore that folder, you will discover many BSP layers within the `meta-intel` layer. For more information on BSP layers, see the "BSP Layers [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-layers]" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

4.1.2. Creating Your Own Layer

It is very easy to create your own layer to use with the Yocto Project. Follow these general steps to create your layer:

1. Check Existing Layers: Before creating a new layer, you should be sure someone has not already created a layer containing the metadata you need. You can see the `LayerIndex` [http://www.openembedded.org/wiki/LayerIndex] for a list of layers from the OpenEmbedded community that can be used in the Yocto Project.

2. Create a Directory: Create the directory for your layer. Traditionally, prepend the name of the folder with the string `meta`. For example:

   ```
   meta-mylayer
   meta-GUI_xyz
   meta-mymachine
   ```

3. Create a Layer Configuration File: Inside your new layer folder, you need to create a `conf/layer.conf` file. It is easiest to take an existing layer configuration file and copy that to your layer's `conf` directory and then modify the file as needed.

   The `meta-yocto/conf/layer.conf` file demonstrates the required syntax:

```
    # We have a conf and classes directory, add to BBPATH
    BBPATH := "${LAYERDIR}:${BBPATH}"

    # We have a packages directory, add to BBFILES
    BBFILES := "${BBFILES} ${LAYERDIR}/recipes-*/*/*.bb \
                ${LAYERDIR}/recipes-*/*/*.bbappend"

    BBFILE_COLLECTIONS += "yocto"
    BBFILE_PATTERN_yocto := "^${LAYERDIR}/"
    BBFILE_PRIORITY_yocto = "5"
```

In the previous example, the recipes for the layers are added to `BBFILES` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBFILES]. The `BBFILE_COLLECTIONS` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBFILE_COLLECTIONS] variable is then appended with the layer name. The `BBFILE_PATTERN` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBFILE_PATTERN] variable is set to a regular expression and is used to match files from `BBFILES` into a particular layer. In this case, immediate expansion of `LAYERDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-LAYERDIR] sets `BBFILE_PATTERN` to the layer's path. The `BBFILE_PRIORITY` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBFILE_PRIORITY] variable then assigns a priority to the layer. Applying priorities is useful in situations where the same package might appear in multiple layers and allows you to choose what layer should take precedence.

Note the use of the `LAYERDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-LAYERDIR] variable with the immediate expansion operator. The `LAYERDIR` variable expands to the directory of the current layer and requires the immediate expansion operator so that BitBake does not wait to expand the variable when it's parsing a different directory.

Through the use of the `BBPATH` variable, BitBake locates `.bbclass` files, configuration files, and files that are included with `include` and `require` statements. For these cases, BitBake uses the first file with the matching name found in `BBPATH`. This is similar to the way the `PATH` variable is used for binaries. We recommend, therefore, that you use unique `.bbclass` and configuration file names in your custom layer.

4. Add Content: Depending on the type of layer, add the content. If the layer adds support for a machine, add the machine configuration in a `conf/machine/` file within the layer. If the layer adds distro policy, add the distro configuration in a `conf/distro/` file with the layer. If the layer introduces new recipes, put the recipes you need in `recipes-*` subdirectories within the layer.

To create layers that are easier to maintain, you should consider the following:

- Avoid "overlaying" entire recipes from other layers in your configuration. In other words, don't copy an entire recipe into your layer and then modify it. Use `.bbappend` files to override the parts of the recipe you need to modify.

- Avoid duplicating include files. Use `.bbappend` files for each recipe that uses an include file. Or, if you are introducing a new recipe that requires the included file, use the path relative to the original layer directory to refer to the file. For example, use `require recipes-core/somepackage/somefile.inc` instead of `require somefile.inc`. If you're finding you have to overlay the include file, it could indicate a deficiency in the include file in the layer to which it originally belongs. If this is the case, you need to address that deficiency instead of overlaying the include file. For example, consider how Qt 4 database support plugins are configured. The Yocto Project does not have MySQL or PostgreSQL, however OpenEmbedded's layer `meta-oe` does. Consequently, `meta-oe` uses `.bbappend` files to modify the `QT_SQL_DRIVER_FLAGS` variable to enable the appropriate plugins. This variable was added to the `qt4.inc` include file in The Yocto Project specifically to allow the `meta-oe` layer to be able to control which plugins are built.

We also recommend the following:

- Store custom layers in a Git repository that uses the `meta-<layer_name>` format.

- Clone the repository alongside other `meta` directories in Yocto Project Files.

Following these recommendations keeps your Yocto Project files area and its configuration entirely inside the Yocto Project's core base.

4.1.3. Enabling Your Layer

Before the Yocto Project build system can use your new layer, you need to enable it. To enable your layer, simply add your layer's path to the BBLAYERS [`http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBLAYERS`] variable in your `conf/bblayers.conf` file, which is found in the Yocto Project Build Directory. The following example shows how to enable a layer named `meta-mylayer`:

```
     LCONF_VERSION = "1"

     BBFILES ?= ""
     BBLAYERS = " \
       /path/to/poky/meta \
       /path/to/poky/meta-yocto \
       /path/to/poky/meta-mylayer \
       "
```

BitBake parses each `conf/layer.conf` file as specified in the BBLAYERS variable within the `conf/bblayers.conf` file. During the processing of each `conf/layer.conf` file, BitBake adds the recipes, classes and configurations contained within the particular layer to the Yocto Project.

4.1.4. Using .bbappend Files

Recipes used to append metadata to other recipes are called BitBake append files. BitBake append files use the `.bbappend` file type suffix, while underlying recipes to which metadata is being appended use the `.bb` file type suffix.

A `.bbappend` file allows your layer to make additions or changes to the content of another layer's recipe without having to copy the other recipe into your layer. Your `.bbappend` file resides in your layer, while the underlying `.bb` recipe file to which you are appending metadata resides in a different layer.

Append files files must have the same name as the underlying recipe. For example, the append file `someapp_1.2.2.bbappend` must apply to `someapp_1.2.2.bb`. This means the original recipe and append file names are version number specific. If the underlying recipe is renamed to update to a newer version, the corresponding `.bbappend` file must be renamed as well. During the build process, BitBake displays an error on starting if it detects a `.bbappend` file that does not have an underlying recipe with a matching name.

Being able to append information to an existing recipe not only avoids duplication, but also automatically applies recipe changes in a different layer to your layer. If you were copying recipes, you would have to manually merge changes as they occur.

As an example, consider the main formfactor recipe and a corresponding formfactor append file both from the Yocto Project Files. Here is the main formfactor recipe, which is named `formfactor_0.0.bb` and located in the meta layer at `meta/recipes-bsp/formfactor`:

```
     DESCRIPTION = "Device formfactor information"
     SECTION = "base"
     LICENSE = "MIT"
     LIC_FILES_CHKSUM = "file://${COREBASE}/LICENSE;md5=3f40d7994397109285ec7b81fdeb3b58 \
                         file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420
     PR = "r20"

     SRC_URI = "file://config file://machconfig"
     S = "${WORKDIR}"
```

```
PACKAGE_ARCH = "${MACHINE_ARCH}"
INHIBIT_DEFAULT_DEPS = "1"

do_install() {
 # Only install file if it has a contents
        install -d ${D}${sysconfdir}/formfactor/
        install -m 0644 ${S}/config ${D}${sysconfdir}/formfactor/
 if [ -s "${S}/machconfig" ]; then
         install -m 0644 ${S}/machconfig ${D}${sysconfdir}/formfactor/
 fi
}
```

Here is the append file, which is named `formfactor_0.0.bbappend` and is from the Crown Bay BSP Layer named `meta-intel/meta-crownbay`. The file is in `recipes-bsp/formfactor`:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

PRINC = "1"
```

This example adds or overrides files in `SRC_URI` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI] within a `.bbappend` by extending the path BitBake uses to search for files. The most reliable way to do this is by prepending the `FILESEXTRAPATHS` variable. For example, if you have your files in a directory that is named the same as your package (`PN` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PN]), you can add this directory by adding the following to your `.bbappend` file:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

Using the immediate expansion assignment operator `:=` is important because of the reference to `THISDIR`. The trailing colon character is important as it ensures that items in the list remain colon-separated.

Note

BitBake automatically defines the `THISDIR` variable. You should never set this variable yourself. Using `_prepend` ensures your path will be searched prior to other paths in the final list.

For another example on how to use a `.bbappend` file, see the "Changing `recipes-kernel`" section.

4.1.5. Prioritizing Your Layer

Each layer is assigned a priority value. Priority values control which layer takes precedence if there are recipe files with the same name in multiple layers. For these cases, the recipe file from the layer with a higher priority number taking precedence. Priority values also affect the order in which multiple `.bbappend` files for the same recipe are applied. You can either specify the priority manually, or allow the build system to calculate it based on the layer's dependencies.

To specify the layer's priority manually, use the `BBFILE_PRIORITY` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBFILE_PRIORITY] variable. For example:

```
BBFILE_PRIORITY := "1"
```

Note

It is possible for a recipe with a lower version number `PV` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PV] in a layer that has a higher priority to take precedence.

Also, the layer priority does not currently affect the precedence order of `.conf` or `.bbclass` files. Future versions of BitBake might address this.

4.1.6. Managing Layers

You can use the BitBake layer management tool to provide a view into the structure of recipes across a multi-layer project. Being able to generate output that reports on configured layers with their paths and priorities and on `.bbappend` files and their applicable recipes can help to reveal potential problems.

Use the following form when running the layer management tool.

```
     $ bitbake-layers <command> [arguments]
```

The following list describes the available commands:

- `help:` Displays general help or help on a specified command.
- `show-layers:` Show the current configured layers.
- `show-recipes:` Lists available recipes and the layers that provide them.
- `show-overlayed:` Lists overlayed recipes. A recipe is overlayed when a recipe with the same name exists in another layer that has a higher layer priority.
- `show-appends:` Lists `.bbappend` files and the recipe files to which they apply.
- `flatten:` Flattens the layer configuration into a separate output directory. Flattening your layer configuration builds a "flattened" directory that contains the contents of all layers, with any overlayed recipes removed and any `.bbappend` files appended to the corresponding recipes. You might have to perform some manual cleanup of the flattened layer as follows:
    - Non-recipe files (such as patches) are overwritten. The flatten command shows a warning for these files.
    - Anything beyond the normal layer setup has been added to the `layer.conf` file. Only the lowest priority layer's `layer.conf` is used.
    - Overridden and appended items from `.bbappend` files need to be cleaned up. The contents of each `.bbappend` end up in the flattened recipe. However, if there are appended or changed variable values, you need to tidy these up yourself. Consider the following example. Here, the `bitbake-layers` command adds the line `#### bbappended ...` so that you know where the following lines originate:

```
     ...
     DESCRIPTION = "A useful utility"
     ...
     EXTRA_OECONF = "--enable-something"
     ...

     #### bbappended from meta-anotherlayer ####

     DESCRIPTION = "Customized utility"
     EXTRA_OECONF += "--enable-somethingelse"
```

    Ideally, you would tidy up these utilities as follows:

```
     ...
     DESCRIPTION = "Customized utility"
     ...
     EXTRA_OECONF = "--enable-something --enable-somethingelse"
```

```
    ...
```

4.2. Customizing Images

You can customize Yocto Project images to satisfy particular requirements. This section describes several methods and provides guidelines for each.

4.2.1. Customizing Images Using Custom .bb Files

One way to get additional software into an image is to create a custom image. The following example shows the form for the two lines you need:

```
IMAGE_INSTALL = "task-core-x11-base package1 package2"

inherit core-image
```

By creating a custom image, a developer has total control over the contents of the image. It is important to use the correct names of packages in the `IMAGE_INSTALL [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_INSTALL]` variable. You must use the OpenEmbedded notation and not the Debian notation for the names (e.g. `eglibc-dev` instead of `libc6-dev`).

The other method for creating a custom image is to base it on an existing image. For example, if you want to create an image based on `core-image-sato` but add the additional package `strace` to the image, copy the `meta/recipes-sato/images/core-image-sato.bb` to a new `.bb` and add the following line to the end of the copy:

```
IMAGE_INSTALL += "strace"
```

4.2.2. Customizing Images Using Custom Tasks

For complex custom images, the best approach is to create a custom task package that is used to build the image or images. A good example of a tasks package is `meta/recipes-core/tasks/task-core-boot.bb` The `PACKAGES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PACKAGES]` variable lists the task packages to build along with the complementary `-dbg` and `-dev` packages. For each package added, you can use `RDEPENDS [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-RDEPENDS]` and `RRECOMMENDS [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-RRECOMMENDS]` entries to provide a list of packages the parent task package should contain. Following is an example:

```
DESCRIPTION = "My Custom Tasks"

PACKAGES = "\
    task-custom-apps \
    task-custom-apps-dbg \
    task-custom-apps-dev \
    task-custom-tools \
    task-custom-tools-dbg \
    task-custom-tools-dev \
    "

RDEPENDS_task-custom-apps = "\
    dropbear \
    portmap \
    psplash"
```

```
RDEPENDS_task-custom-tools = "\
    oprofile \
    oprofileui-server \
    lttng-control \
    lttng-viewer"

RRECOMMENDS_task-custom-tools = "\
    kernel-module-oprofile"
```

In the previous example, two task packages are created with their dependencies and their recommended package dependencies listed: `task-custom-apps`, and `task-custom-tools`. To build an image using these task packages, you need to add `task-custom-apps` and/or `task-custom-tools` to `IMAGE_INSTALL [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_INSTALL]`. For other forms of image dependencies see the other areas of this section.

4.2.3. Customizing Images Using Custom **`IMAGE_FEATURES`** and **`EXTRA_IMAGE_FEATURES`**

Ultimately users might want to add extra image features to the set used by Yocto Project with the `IMAGE_FEATURES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_FEATURES]` variable. To create these features, the best reference is `meta/classes/core-image.bbclass`, which shows how the Yocto Project achieves this. In summary, the file looks at the contents of the `IMAGE_FEATURES` variable and then maps that into a set of tasks or packages. Based on this information the `IMAGE_INSTALL [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_INSTALL]` variable is generated automatically. Users can add extra features by extending the class or creating a custom class for use with specialized image `.bb` files. You can also add more features by configuring the `EXTRA_IMAGE_FEATURES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-EXTRA_IMAGE_FEATURES]` variable in the `local.conf` file found in the Yocto Project files located in the build directory.

The Yocto Project ships with two SSH servers you can use in your images: Dropbear and OpenSSH. Dropbear is a minimal SSH server appropriate for resource-constrained environments, while OpenSSH is a well-known standard SSH server implementation. By default, the `core-image-sato` image is configured to use Dropbear. The `core-image-basic` and `core-image-lsb` images both include OpenSSH. To change these defaults, edit the `IMAGE_FEATURES` variable so that it sets the image you are working with to include `ssh-server-dropbear` or `ssh-server-openssh`.

4.2.4. Customizing Images Using**`local.conf`**

It is possible to customize image contents by using variables from your local configuration in your `conf/local.conf` file. Because it is limited to local use, this method generally only allows you to add packages and is not as flexible as creating your own customized image. When you add packages using local variables this way, you need to realize that these variable changes affect all images at the same time and might not be what you require.

The simplest way to add extra packages to all images is by using the `IMAGE_INSTALL [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_INSTALL]` variable with the _append operator:

```
IMAGE_INSTALL_append = " strace"
```

Use of the syntax is important. Specifically, the space between the quote and the package name, which is `strace` in this example. This space is required since the _append operator does not add the space.

Furthermore, you must use _append instead of the += operator if you want to avoid ordering issues. The reason for this is because doing so unconditionally appends to the variable and avoids ordering problems due to the variable being set in image recipes and `.bbclass` files with operators like ?=. Using _append ensures the operation takes affect.

As shown in its simplest use, `IMAGE_INSTALL_append` affects all images. It is possible to extend the syntax so that the variable applies to a specific image only. Here is an example:

```
     IMAGE_INSTALL_append_pn-core-image-minimal = " strace"
```

This example adds `strace` to `core-image-minimal` only.

You can add packages using a similar approach through the `CORE_IMAGE_EXTRA_INSTALL [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-CORE_IMAGE_EXTRA_INSTALL]` variable. If you use this variable, only `core-image-*` images are affected.

4.3. Adding a Package

To add a package into the Yocto Project you need to write a recipe for it. Writing a recipe means creating a `.bb` file that sets some variables. For information on variables that are useful for recipes and for information about recipe naming issues, see the "Required [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-varlocality-recipe-required]" section of the Yocto Project Reference Manual.

Before writing a recipe from scratch, it is often useful to check whether someone else has written one already. OpenEmbedded is a good place to look as it has a wider scope and range of packages. Because the Yocto Project aims to be compatible with OpenEmbedded, most recipes you find there should work in Yocto Project.

For new packages, the simplest way to add a recipe is to base it on a similar pre-existing recipe. The sections that follow provide some examples that show how to add standard types of packages.

4.3.1. Single .c File Package (Hello World!)

Building an application from a single file that is stored locally (e.g. under `files/`) requires a recipe that has the file listed in the `SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI]` variable. Additionally, you need to manually write the `do_compile` and `do_install` tasks. The `S [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S]` variable defines the directory containing the source code, which is set to `WORKDIR [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-WORKDIR]` in this case - the directory BitBake uses for the build.

```
     DESCRIPTION = "Simple helloworld application"
     SECTION = "examples"
     LICENSE = "MIT"
     LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
     PR = "r0"

     SRC_URI = "file://helloworld.c"

     S = "${WORKDIR}"

     do_compile() {
      ${CC} helloworld.c -o helloworld
     }

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

By default, the `helloworld`, `helloworld-dbg`, and `helloworld-dev` packages are built. For information on how to customize the packaging process, see the "Splitting an Application into Multiple Packages" section.

4.3.2. Autotooled Package

Applications that use Autotools such as autoconf and automake require a recipe that has a source archive listed in SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI] and also inherits Autotools, which instructs BitBake to use the autotools.bbclass file, which contains the definitions of all the steps needed to build an Autotool-based application. The result of the build is automatically packaged. And, if the application uses NLS for localization, packages with local information are generated (one package per language). Following is one example: (hello_2.3.bb)

```
     DESCRIPTION = "GNU Helloworld application"
     SECTION = "examples"
     LICENSE = "GPLv2+"
     LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe"
     PR = "r0"

     SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"

     inherit autotools gettext
```

The variable LIC_FILES_CHKSUM [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-LIC_FILES_CHKSUM] is used to track source license changes as described in the "Track License Changes [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#usingpoky-configuring-LIC_FILES_CHKSUM]" section. You can quickly create Autotool-based recipes in a manner similar to the previous example.

4.3.3. Makefile-Based Package

Applications that use GNU make also require a recipe that has the source archive listed in SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI]. You do not need to add a do_compile step since by default BitBake starts the make command to compile the application. If you need additional make options you should store them in the EXTRA_OEMAKE [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-EXTRA_OEMAKE] variable. BitBake passes these options into the make GNU invocation. Note that a do_install task is still required. Otherwise BitBake runs an empty do_install task by default.

Some applications might require extra parameters to be passed to the compiler. For example, the application might need an additional header path. You can accomplish this by adding to the CFLAGS [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-CFLAGS] variable. The following example shows this:

```
     CFLAGS_prepend = "-I ${S}/include "
```

In the following example, mtd-utils is a makefile-based package:

```
     DESCRIPTION = "Tools for managing memory technology devices."
     SECTION = "base"
     DEPENDS = "zlib lzo e2fsprogs util-linux"
     HOMEPAGE = "http://www.linux-mtd.infradead.org/"
     LICENSE = "GPLv2+"
     LIC_FILES_CHKSUM = "file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3 \
                    file://include/common.h;beginline=1;endline=17;md5=ba05b07912a44ea2bf81ce4093

     SRC_URI = "git://git.infradead.org/mtd-utils.git;protocol=git;tag=995cfe51b0a3cf32f381c140bf
       file://add-exclusion-to-mkfs-jffs2-git-2.patch"

     S = "${WORKDIR}/git/"
```

```
PR = "r1"

EXTRA_OEMAKE = "'CC=${CC}' 'RANLIB=${RANLIB}' 'AR=${AR}' \
   'CFLAGS=${CFLAGS} -I${S}/include -DWITHOUT_XATTR' 'BUILDDIR=${S}'"

do_install () {
 oe_runmake install DESTDIR=${D} SBINDIR=${sbindir} MANDIR=${mandir} \
       INCLUDEDIR=${includedir}
 install -d ${D}${includedir}/mtd/
 for f in ${S}/include/mtd/*.h; do
  install -m 0644 $f ${D}${includedir}/mtd/
 done
}

PARALLEL_MAKE = ""

BBCLASSEXTEND = "native"
```

If your sources are available as a tarball instead of a Git repository, you will need to provide the URL to the tarball as well as an md5 or sha256 sum of the download. Here is an example:

```
SRC_URI="ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.4.9.tar.bz2"
SRC_URI[md5sum]="82b8e714b90674896570968f70ca778b"
```

You can generate the md5 or sha256 sums by using the `md5sum` or `sha256sum` commands with the target file as the only argument. Here is an example:

```
$ md5sum mtd-utils-1.4.9.tar.bz2
82b8e714b90674896570968f70ca778b mtd-utils-1.4.9.tar.bz2
```

4.3.4. Splitting an Application into Multiple Packages

You can use the variables `PACKAGES` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PACKAGES] and `FILES` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-FILES] to split an application into multiple packages.

Following is an example that uses the `libXpm` recipe. By default, this recipe generates a single package that contains the library along with a few binaries. You can modify the recipe to split the binaries into separate packages:

```
require xorg-lib-common.inc

DESCRIPTION = "X11 Pixmap library"
LICENSE = "X-BSD"
LIC_FILES_CHKSUM = "file://COPYING;md5=3e07763d16963c3af12db271a31abaa5"
DEPENDS += "libxext libsm libxt"
PR = "r3"
PE = "1"

XORG_PN = "libXpm"

PACKAGES =+ "sxpm cxpm"
FILES_cxpm = "${bindir}/cxpm"
FILES_sxpm = "${bindir}/sxpm"
```

In the previous example, we want to ship the `sxpm` and `cxpm` binaries in separate packages. Since `bindir` would be packaged into the main `PN` [http://www.yoctoproject.org/

docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PN] package by default, we prepend the PACKAGES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PACKAGES] variable so additional package names are added to the start of list. This results in the extra FILES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-FILES]_* variables then containing information that define which files and directories go into which packages. Files included by earlier packages are skipped by latter packages. Thus, the main PN [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PN] package does not include the above listed files.

4.3.5. Including Static Library Files

If you are building a library and the library offers static linking, you can control which static library files (*.a files) get included in the built library.

The PACKAGES and FILES_* variables in the meta/conf/bitbake.conf configuration file define how files installed by the do_install task are packaged. By default, the PACKAGES variable contains ${PN}-staticdev, which includes all static library files.

Note

Previously released versions of the Yocto Project defined the static library files through ${PN}-dev.

Following, is part of the BitBake configuration file. You can see where the static library files are defined:

```
PACKAGES = "${PN}-dbg ${PN} ${PN}-doc ${PN}-dev ${PN}-staticdev ${PN}-locale"
PACKAGES_DYNAMIC = "${PN}-locale-*"
FILES = ""

FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS} \
            ${sysconfdir} ${sharedstatedir} ${localstatedir} \
            ${base_bindir}/* ${base_sbindir}/* \
            ${base_libdir}/*${SOLIBS} \
            ${datadir}/${BPN} ${libdir}/${BPN}/* \
            ${datadir}/pixmaps ${datadir}/applications \
            ${datadir}/idl ${datadir}/omf ${datadir}/sounds \
            ${libdir}/bonobo/servers"

FILES_${PN}-doc = "${docdir} ${mandir} ${infodir} ${datadir}/gtk-doc \
            ${datadir}/gnome/help"
SECTION_${PN}-doc = "doc"

FILES_${PN}-dev = "${includedir} ${libdir}/lib*${SOLIBSDEV} ${libdir}/*.la \
                ${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
                ${datadir}/aclocal ${base_libdir}/*.o"
SECTION_${PN}-dev = "devel"
ALLOW_EMPTY_${PN}-dev = "1"
RDEPENDS_${PN}-dev = "${PN} (= ${EXTENDPKGV})"

FILES_${PN}-staticdev = "${libdir}/*.a ${base_libdir}/*.a"
SECTION_${PN}-staticdev = "devel"
RDEPENDS_${PN}-staticdev = "${PN}-dev (= ${EXTENDPKGV})"
```

4.3.6. Post Install Scripts

To add a post-installation script to a package, add a pkg_postinst_PACKAGENAME() function to the .bb file and use PACKAGENAME as the name of the package you want to attach to the postinst script. Normally PN [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PN] can be used, which automatically expands to PACKAGENAME. A post-installation function has the following structure:

```
pkg_postinst_PACKAGENAME () {
#!/bin/sh -e
# Commands to carry out
}
```

The script defined in the post-installation function is called when the root filesystem is created. If the script succeeds, the package is marked as installed. If the script fails, the package is marked as unpacked and the script is executed when the image boots again.

Sometimes it is necessary for the execution of a post-installation script to be delayed until the first boot. For example, the script might need to be executed on the device itself. To delay script execution until boot time, use the following structure in the post-installation script:

```
pkg_postinst_PACKAGENAME () {
#!/bin/sh -e
if [ x"$D" = "x" ]; then
     # Actions to carry out on the device go here
else
     exit 1
fi
}
```

The previous example delays execution until the image boots again because the `D` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-D] variable points to the directory containing the image when the root filesystem is created at build time but is unset when executed on the first boot.

4.4. Porting the Yocto Project to a New Machine

Adding a new machine to the Yocto Project is a straightforward process. This section provides information that gives you an idea of the changes you must make. The information covers adding machines similar to those the Yocto Project already supports. Although well within the capabilities of the Yocto Project, adding a totally new architecture might require changes to `gcc/eglibc` and to the site information, which is beyond the scope of this manual.

For a complete example that shows how to add a new machine to the Yocto Project, see the "BSP Development Example [http://www.yoctoproject.org/docs/1.2.2/dev-manual/dev-manual.html#dev-manual-bsp-appendix]" in Appendix A.

4.4.1. Adding the Machine Configuration File

To add a machine configuration you need to add a `.conf` file with details of the device being added to the `conf/machine/` file. The name of the file determines the name the Yocto Project uses to reference the new machine.

The most important variables to set in this file are as follows:

- `TARGET_ARCH` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-TARGET_ARCH] (e.g. "arm")
- `PREFERRED_PROVIDER` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PREFERRED_PROVIDER]_virtual/kernel (see below)
- `MACHINE_FEATURES` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-MACHINE_FEATURES] (e.g. "apm screen wifi")

You might also need these variables:

- `SERIAL_CONSOLE` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SERIAL_CONSOLE] (e.g. "115200 ttyS0")
- `KERNEL_IMAGETYPE` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-KERNEL_IMAGETYPE] (e.g. "zImage")

- IMAGE_FSTYPES [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-IMAGE_FSTYPES] (e.g. "tar.gz jffs2")

You can find full details on these variables in the reference section. You can leverage many existing machine .conf files from meta/conf/machine/.

4.4.2. Adding a Kernel for the Machine

The Yocto Project needs to be able to build a kernel for the machine. You need to either create a new kernel recipe for this machine, or extend an existing recipe. You can find several kernel examples in the Yocto Project file's meta/recipes-kernel/linux directory that you can use as references.

If you are creating a new recipe, normal recipe-writing rules apply for setting up a SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI]. Thus, you need to specify any necessary patches and set S [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S] to point at the source code. You need to create a configure task that configures the unpacked kernel with a defconfig. You can do this by using a make defconfig command or, more commonly, by copying in a suitable defconfig file and and then running make oldconfig. By making use of inherit kernel and potentially some of the linux-*.inc files, most other functionality is centralized and the the defaults of the class normally work well.

If you are extending an existing kernel, it is usually a matter of adding a suitable defconfig file. The file needs to be added into a location similar to defconfig files used for other machines in a given kernel. A possible way to do this is by listing the file in the SRC_URI and adding the machine to the expression in COMPATIBLE_MACHINE [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-COMPATIBLE_MACHINE]:

```
     COMPATIBLE_MACHINE = '(qemux86|qemumips)'
```

4.4.3. Adding a Formfactor Configuration File

A formfactor configuration file provides information about the target hardware for which the Yocto Project is building and information that the Yocto Project cannot obtain from other sources such as the kernel. Some examples of information contained in a formfactor configuration file include framebuffer orientation, whether or not the system has a keyboard, the positioning of the keyboard in relation to the screen, and the screen resolution.

The Yocto Project uses reasonable defaults in most cases, but if customization is necessary you need to create a machconfig file in the Yocto Project file's meta/recipes-bsp/formfactor/files directory. This directory contains directories for specific machines such as qemuarm and qemux86. For information about the settings available and the defaults, see the meta/recipes-bsp/formfactor/files/config file found in the same area. Following is an example for qemuarm:

```
     HAVE_TOUCHSCREEN=1
     HAVE_KEYBOARD=1

     DISPLAY_CAN_ROTATE=0
     DISPLAY_ORIENTATION=0
     #DISPLAY_WIDTH_PIXELS=640
     #DISPLAY_HEIGHT_PIXELS=480
     #DISPLAY_BPP=16
     DISPLAY_DPI=150
     DISPLAY_SUBPIXEL_ORDER=vrgb
```

4.5. Modifying Temporary Source Code

Although the Yocto Project is typically used to build software, you might find it helpful during development to modify the temporary source code used by recipes to build packages. For example, suppose you are developing a patch and you need to experiment a bit to figure out your solution. After you have initially built the package, you can iteratively tweak the source code, which is located in the Yocto Project's Build Directory, and then you can force a re-compile and quickly test your altered code. Once you settle on a solution, you can then preserve your changes in the form of patches. You can accomplish these steps all within either a Quilt [http://savannah.nongnu.org/projects/quilt] or Git workflow.

4.5.1. Finding the Temporary Source Code

During a build, the unpacked temporary source code used by recipes to build packages is available in the Yocto Project Build Directory as defined by the `S` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S] variable. Below is the default value for the `S` variable as defined in the `meta/conf/bitbake.conf` configuration file in the Yocto Project Files:

```
S = ${WORKDIR}/${BP}
```

You should be aware that many recipes override the `S` variable. For example, recipes that fetch their source from Git usually set `S` to `${WORKDIR}/git`.

Note

BP represents the "Base Package", which is the base package name and the package version:

```
BP = ${BPN}-${PV}
```

The path to the work directory for the recipe (`WORKDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-WORKDIR]) depends on the package name and the architecture of the target device. For example, here is the work directory for packages whose targets are not device-dependent:

```
${TMPDIR}/work/${PACKAGE_ARCH}-poky-${TARGET_OS}/${PN}-${PV}-${PR}
```

Let's look at an example without variables. Assuming a Yocto Project Files top-level directory named poky and a default Yocto Project Build Directory of `poky/build`, the following is the work directory for the `acl` package:

```
~/poky/build/tmp/work/i586-poky-linux/acl-2.2.51-r3
```

If your package is dependent on the target device, the work directory varies slightly:

```
${TMPDIR}/work/${MACHINE}-poky-${TARGET_OS}/${PN}-${PV}-${PR}
```

Again, assuming a Yocto Project Files top-level directory named poky and a default Yocto Project Build Directory of `poky/build`, the following is the work directory for the `acl` package that is being built for a MIPS-based device:

```
~/poky/build/tmp/work/mips-poky-linux/acl-2.2.51-r2
```

Note

To better understand how the Yocto Project build system resolves directories during the build process, see the glossary entries for the `WORKDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-WORKDIR], `TMPDIR` [http://www.yoctoproject.org/docs/1.2.2/

poky-ref-manual/poky-ref-manual.html#var-TMPDIR], `TOPDIR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-TOPDIR], `PACKAGE_ARCH` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PACKAGE_ARCH], `TARGET_OS` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-TARGET_OS], `PN` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PN], `PV` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PV], and `PR` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PR] variables in the Yocto Project Reference Manual.

Now that you know where to locate the directory that has the temporary source code, you can use a Quilt or Git workflow to make your edits, test the changes, and preserve the changes in the form of patches.

4.5.2. Using a Quilt Workflow

Quilt [http://savannah.nongnu.org/projects/quilt] is a powerful tool that allows you to capture source code changes without having a clean source tree. This section outlines the typical workflow you can use to modify temporary source code, test changes, and then preserve the changes in the form of a patch all using Quilt.

Follow these general steps:

1. Find the Source Code: The temporary source code used by the Yocto Project build system is kept in the Yocto Project Build Directory. See the "Finding the Temporary Source Code" section to learn how to locate the directory that has the temporary source code for a particular package.

2. Change Your Working Directory: You need to be in the directory that has the temporary source code. That directory is defined by the S [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S] variable.

3. Create a New Patch: Before modifying source code, you need to create a new patch. To create a new patch file, use `quilt new` as below:

```
     $ quilt new my_changes.patch
```

4. Notify Quilt and Add Files: After creating the patch, you need to notify Quilt about the files you will be changing. Add the files you will be modifying into the patch you just created:

```
     $ quilt add file1.c file2.c file3.c
```

5. Edit the Files: Make the changes to the temporary source code.

6. Test Your Changes: Once you have modified the source code, the easiest way to test your changes is by calling the `compile` task as shown in the following example:

```
     $ bitbake -c compile -f <name_of_package>
```

   The `-f` or `--force` option forces re-execution of the specified task. If you find problems with your code, you can just keep editing and re-testing iteratively until things work as expected.

   ### Note

   All the modifications you make to the temporary source code disappear once you `-c clean` or `-c cleanall` with BitBake for the package. Modifications will also disappear if you use the `rm_work` feature as described in the "Building an Image [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#building-image]" section of the Yocto Project Quick Start.

7. Generate the Patch: Once your changes work as expected, you need to use Quilt to generate the final patch that contains all your modifications.

```
$ quilt refresh
```

At this point the `my_changes.patch` file has all your edits made to the `file1.c`, `file2.c`, and `file3.c` files.

You can find the resulting patch file in the `patches/` subdirectory of the source (S) directory.

8. Copy the Patch File: For simplicity, copy the patch file into a directory named `files`, which you can create in the same directory as the recipe. Placing the patch here guarantees that the Yocto Project build system will find the patch. Next, add the patch into the `SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI]` of the recipe. Here is an example:

```
SRC_URI += "file://my_changes.patch"
```

9. Increment the Package Revision Number: Finally, don't forget to 'bump' the `PR [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PR]` value in the same recipe since the resulting packages have changed.

4.5.3. Using a Git Workflow

Git is an even more powerful tool that allows you to capture source code changes without having a clean source tree. This section outlines the typical workflow you can use to modify temporary source code, test changes, and then preserve the changes in the form of a patch all using Git. For general information on Git as it is used in the Yocto Project, see the "Git" section.

Note

This workflow uses Git only for its ability to manage local changes to the source code and produce patches independent of any version control used on the Yocto Project Files.

Follow these general steps:

1. Find the Source Code: The temporary source code used by the Yocto Project build system is kept in the Yocto Project Build Directory. See the "Finding the Temporary Source Code" section to learn how to locate the directory that has the temporary source code for a particular package.

2. Change Your Working Directory: You need to be in the directory that has the temporary source code. That directory is defined by the S [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S] variable.

3. Initialize a Git Repository: Use the `git init` command to initialize a new local repository that is based on the work directory:

```
$ git init
```

4. Stage all the files: Use the `git add *` command to stage all the files in the source code directory so that they can be committed:

```
$ git add *
```

5. Commit the Source Files: Use the `git commit` command to initially commit all the files in the work directory:

```
$ git commit
```

At this point, your Git repository is aware of all the source code files. Any edits you now make to files will be tracked by Git.

6. Edit the Files: Make the changes to the temporary source code.

7. Test Your Changes: Once you have modified the source code, the easiest way to test your changes is by calling the `compile` task as shown in the following example:

   ```
   $ bitbake -c compile -f <name_of_package>
   ```

   The `-f` or `--force` option forces re-execution of the specified task. If you find problems with your code, you can just keep editing and re-testing iteratively until things work as expected.

   ### Note

   All the modifications you make to the temporary source code disappear once you `-c clean` or `-c cleanall` with BitBake for the package. Modifications will also disappear if you use the `rm_work` feature as described in the "Building an Image [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#building-image]" section of the Yocto Project Quick Start.

8. See the List of Files You Changed: Use the `git status` command to see what files you have actually edited. The ability to have Git track the files you have changed is an advantage that this workflow has over the Quilt workflow. Here is the Git command to list your changed files:

   ```
   $ git status
   ```

9. Stage the Modified Files: Use the `git add` command to stage the changed files so they can be committed as follows:

   ```
   $ git add file1.c file2.c file3.c
   ```

10. Commit the Staged Files and View Your Changes: Use the `git commit` command to commit the changes to the local repository. Once you have committed the files, you can use the `git log` command to see your changes:

    ```
    $ git commit
    $ git log
    ```

11. Generate the Patch: Once the changes are committed, use the `git format-patch` command to generate a patch file:

    ```
    $ git format-patch HEAD~1
    ```

    The `HEAD~1` part of the command causes Git to generate the patch file for the most recent commit.

    At this point, the patch file has all your edits made to the `file1.c`, `file2.c`, and `file3.c` files. You can find the resulting patch file in the current directory. The patch file ends with `.patch`.

12. Copy the Patch File: For simplicity, copy the patch file into a directory named `files`, which you can create in the same directory as the recipe. Placing the patch here guarantees that the Yocto Project build system will find the patch. Next, add the patch into the `SRC_URI [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI]` of the recipe. Here is an example:

    ```
    SRC_URI += "file://my_changes.patch"
    ```

13. Increment the Package Revision Number: Finally, don't forget to 'bump' the PR [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PR] value in the same recipe since the resulting packages have changed.

4.6. Combining Multiple Versions of Library Files into One Image

The build system offers the ability to build libraries with different target optimizations or architecture formats and combine these together into one system image. You can link different binaries in the image against the different libraries as needed for specific use cases. This feature is called "Multilib."

An example would be where you have most of a system compiled in 32-bit mode using 32-bit libraries, but you have something large, like a database engine, that needs to be a 64-bit application and use 64-bit libraries. Multilib allows you to get the best of both 32-bit and 64-bit libraries.

While the Multilib feature is most commonly used for 32 and 64-bit differences, the approach the build system uses facilitates different target optimizations. You could compile some binaries to use one set of libraries and other binaries to use other different sets of libraries. The libraries could differ in architecture, compiler options, or other optimizations.

This section overviews the Multilib process only. For more details on how to implement Multilib, see the Multilib [https://wiki.yoctoproject.org/wiki/Multilib] wiki page.

4.6.1. Preparing to use Multilib

User-specific requirements drive the Multilib feature, Consequently, there is no one "out-of-the-box" configuration that likely exists to meet your needs.

In order to enable Multilib, you first need to ensure your recipe is extended to support multiple libraries. Many standard recipes are already extended and support multiple libraries. You can check in the `meta/conf/multilib.conf` configuration file in the Yocto Project files directory to see how this is done using the `BBCLASSEXTEND` variable. Eventually, all recipes will be covered and this list will be unneeded.

For the most part, the Multilib class extension works automatically to extend the package name from `${PN}` to `${MLPREFIX}${PN}`, where `MLPREFIX` is the particular multilib (e.g. "lib32-" or "lib64-"). Standard variables such as `DEPENDS`, `RDEPENDS`, `RPROVIDES`, `RRECOMMENDS`, `PACKAGES`, and `PACKAGES_DYNAMIC` are automatically extended by the system. If you are extending any manual code in the recipe, you can use the `${MLPREFIX}` variable to ensure those names are extended correctly. This automatic extension code resides in `multilib.bbclass`.

4.6.2. Using Multilib

After you have set up the recipes, you need to define the actual combination of multiple libraries you want to build. You accomplish this through your `local.conf` configuration file in the Yocto Project Build Directory. An example configuration would be as follows:

```
MACHINE = "qemux86-64"
require conf/multilib.conf
MULTILIBS = "multilib:lib32"
DEFAULTTUNE_virtclass-multilib-lib32 = "x86"
IMAGE_INSTALL = "lib32-connman"
```

This example enables an additional library named `lib32` alongside the normal target packages. When combining these "lib32" alternatives, the example uses "x86" for tuning. For information on this particular tuning, see `meta/conf/machine/include/ia32/arch-ia32.inc`.

The example then includes `lib32-connman` in all the images, which illustrates one method of including a multiple library dependency. You can use a normal image build to include this dependency, for example:

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

You can also build Multilib packages specifically with a command like this:

```
     $  bitbake lib32-connman
```

4.6.3. Additional Implementation Details

Different packaging systems have different levels of native Multilib support. For the RPM Package Management System, the following implementation details exist:

- A unique architecture is defined for the Multilib packages, along with creating a unique deploy folder under `tmp/deploy/rpm` in the Yocto Project Build Directory. For example, consider `lib32` in a `qemux86-64` image. The possible architectures in the system are "all", "qemux86_64", "lib32_qemux86_64", and "lib32_x86".
- The `${MLPREFIX}` variable is stripped from `${PN}` during RPM packaging. The naming for a normal RPM package and a Multilib RPM package in a `qemux86-64` system resolves to something similar to `bash-4.1-r2.x86_64.rpm` and `bash-4.1.r2.lib32_x86.rpm`, respectively.
- When installing a Multilib image, the RPM backend first installs the base image and then installs the Multilib libraries.
- The build system relies on RPM to resolve the identical files in the two (or more) Multilib packages.

For the IPK Package Management System, the following implementation details exist:

- The `${MLPREFIX}` is not stripped from `${PN}` during IPK packaging. The naming for a normal RPM package and a Multilib IPK package in a `qemux86-64` system resolves to something like `bash_4.1-r2.x86_64.ipk` and `lib32-bash_4.1-rw_x86.ipk`, respectively.
- The IPK deploy folder is not modified with `${MLPREFIX}` because packages with and without the Multilib feature can exist in the same folder due to the `${PN}` differences.
- IPK defines a sanity check for Multilib installation using certain rules for file comparison, overridden, etc.

4.7. Configuring the Kernel

Configuring the Linux Yocto kernel consists of making sure the `.config` file has all the right information in it for the image you are building. You can use the `menuconfig` tool and configuration fragments to make sure your `.config` file is just how you need it. This section describes how to use `menuconfig`, create and use configuration fragments, and how to interactively tweak your `.config` file to create the leanest kernel configuration file possible.

For concepts on kernel configuration, see the "Kernel Configuration [http://www.yoctoproject.org/docs/1.2.2/kernel-manual/kernel-manual.html#kernel-configuration]" section in the Yocto Project Kernel Architecture and Use Manual.

4.7.1. Using **`menuconfig`**

The easiest way to define kernel configurations is to set them through the `menuconfig` tool. For general information on `menuconfig`, see http://en.wikipedia.org/wiki/Menuconfig.

To use the `menuconfig` tool in the Yocto Project development environment, you must build the tool using BitBake. The following commands build and invoke `menuconfig` assuming the Yocto Project files top-level directory is `~/poky`:

```
     $ cd ~/poky
     $ source oe-init-build-env
     $ bitbake linux-yocto -c menuconfig
```

Once `menuconfig` comes up, its standard interface allows you to examine and configure all the kernel configuration parameters. Once you have made your changes, simply exit the tool and save your changes to create an updated version of the `.config` configuration file.

For an example that shows how to change the `SMP_CONFIG` parameter using `menuconfig`, see the "Changing the `CONFIG_SMP` Configuration Using `menuconfig`" section.

4.7.2. Creating Config Fragments

Configuration fragments are simply kernel options that appear in a file. Syntactically, the configuration statement is identical to what would appear in the `.config`. For example, issuing the following from the shell would create a config fragment file named `my_smp.cfg` that enables multi-processor support within the kernel:

```
     $ echo "CONFIG_SMP=y" >> my_smp.cfg
```

Where do you put your configuration files? You can place these configuration files in the same area pointed to by `SRC_URI`. The Yocto Project build process will pick up the configuration and add it to the kernel's configuration. For example, assume you add the following to your `linux-yocto_3.0.bbappend` file:

```
     file://my_smp.cfg
```

You would put the config fragment file `my_smp.cfg` in a sub-directory with the same root name (`linux-yocto`) beneath the directory that contains your `linux-yocto_3.0.bbappend` file and the build system will pick up and apply the fragment.

4.7.3. Fine-tuning the Kernel Configuration File

You can make sure the `.config` is as lean or efficient as possible by reading the output of the kernel configuration fragment audit, noting any issues, making changes to correct the issues, and then repeating.

As part of the Linux Yocto kernel build process, the `kernel_configcheck` task runs. This task validates the kernel configuration by checking the final `.config` file against the input files. During the check, the task produces warning messages for the following issues:

- Requested options that did not make the final `.config` file.
- Configuration items that appear twice in the same configuration fragment.
- Configuration items tagged as 'required' were overridden.
- A board overrides a non-board specific option.
- Listed options not valid for the kernel being processed. In other words, the option does not appear anywhere.

Note

The `kernel_configcheck` task can also optionally report if an option is overridden during processing.

For each output warning, a message points to the file that contains a list of the options and a pointer to the config fragment that defines them. Collectively, the files are the key to streamlining the configuration.

To streamline the configuration, do the following:

1. Start with a full configuration that you know works - it builds and boots successfully. This configuration file will be your baseline.

2. Separately run the `configme` and `kernel_configcheck` tasks.
3. Take the resulting list of files from the `kernel_configcheck` task warnings and do the following:
   - Drop values that are redefined in the fragment but do not change the final `.config` file.
   - Analyze and potentially drop values from the `.config` file that override required configurations.
   - Analyze and potentially remove non-board specific options.
   - Remove repeated and invalid options.
4. After you have worked through the output of the kernel configuration audit, you can re-run the `configme` and `kernel_configcheck` tasks to see the results of your changes. If you have more issues, you can deal with them as described in the previous step.

Iteratively working through steps two through four eventually yields a minimal, streamlined configuration file. Once you have the best `.config`, you can build the Linux Yocto kernel.

4.8. Updating Existing Images

Often, rather than re-flashing a new image, you might wish to install updated packages into an existing running system. You can do this by first sharing the `tmp/deploy/ipk/` directory through a web server and then by changing `/etc/opkg/base-feeds.conf` to point at the shared server. Following is an example:

```
     $ src/gz all http://www.mysite.com/somedir/deploy/ipk/all
     $ src/gz armv7a http://www.mysite.com/somedir/deploy/ipk/armv7a
     $ src/gz beagleboard http://www.mysite.com/somedir/deploy/ipk/beagleboard
```

4.9. Incrementing a Package Revision Number

If a committed change results in changing the package output, then the value of the PR `[http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PR]` variable needs to be increased (or "bumped") as part of that commit. This means that for new recipes you must be sure to add the PR variable and set its initial value equal to "r0". Failing to define PR makes it easy to miss when you bump a package. Note that you can only use integer values following the "r" in the PR variable.

If you are sharing a common `.inc` file with multiple recipes, you can also use the `INC_PR` `[http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-INC_PR]` variable to ensure that the recipes sharing the `.inc` file are rebuilt when the `.inc` file itself is changed. The `.inc` file must set `INC_PR` (initially to "r0"), and all recipes referring to it should set PR to "$(INC_PR).0" initially, incrementing the last number when the recipe is changed. If the `.inc` file is changed then its `INC_PR` should be incremented.

When upgrading the version of a package, assuming the PV `[http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PV]` changes, the PR variable should be reset to "r0" (or "$(INC_PR).0" if you are using `INC_PR`).

Usually, version increases occur only to packages. However, if for some reason PV changes but does not increase, you can increase the PE `[http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-PE]` variable (Package Epoch). The PE variable defaults to "0".

Version numbering strives to follow the Debian Version Field Policy Guidelines [http://www.debian.org/doc/debian-policy/ch-controlfields.html]. These guidelines define how versions are compared and what "increasing" a version means.

There are two reasons for following the previously mentioned guidelines. First, to ensure that when a developer updates and rebuilds, they get all the changes to the repository and do not have to remember to rebuild any sections. Second, to ensure that target users are able to upgrade their devices using package manager commands such as `opkg upgrade` (or similar commands for dpkg/apt or rpm-based systems).

The goal is to ensure the Yocto Project has packages that can be upgraded in all cases.

4.10. Handling a Package Name Alias

Sometimes a package name you are using might exist under an alias or as a similarly named package in a different distribution. The Yocto Project implements a `distro_check` task that automatically connects to major distributions and checks for these situations. If the package exists under a different name in a different distribution, you get a `distro_check` mismatch. You can resolve this problem by defining a per-distro recipe name alias using the `DISTRO_PN_ALIAS` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-DISTRO_PN_ALIAS] variable.

Following is an example that shows how you specify the `DISTRO_PN_ALIAS` variable:

```
DISTRO_PN_ALIAS_pn-PACKAGENAME = "distro1=package_name_alias1 \
                                  distro2=package_name_alias2 \
                                  distro3=package_name_alias3 \
                                  ..."
```

If you have more than one distribution alias, separate them with a space. Note that the Yocto Project currently automatically checks the Fedora, OpenSuSE, Debian, Ubuntu, and Mandriva distributions for source package recipes without having to specify them using the `DISTRO_PN_ALIAS` variable. For example, the following command generates a report that lists the Linux distributions that include the sources for each of the Yocto Project recipes.

```
$ bitbake world -f -c distro_check
```

The results are stored in the `build/tmp/log/distro_check-${DATETIME}.results` file found in the Yocto Project files area.

4.11. Building Software from an External Source

By default, the Yocto Project build system does its work from within the Yocto Project Build Directory. The build process involves fetching the source files, unpacking them, and then patching them if necessary before the build takes place.

Situations exist where you might want to build software from source files that are external to and thus outside of the Yocto Project Files. For example, suppose you have a project that includes a new BSP with a heavily customized kernel, a very minimal image, and some new user-space recipes. And, you want to minimize the exposure to the Yocto Project build system to the development team so that they can focus on their project and maintain everyone's workflow as much as possible. In this case, you want a kernel source directory on the development machine where the development occurs. You want the recipe's `SRC_URI` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-SRC_URI] variable to point to the external directory and use it as is, not copy it.

To build from software that comes from an external source, all you need to do is change your recipe so that it inherits the `externalsrc.bbclass` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-classes-externalsrc] class and then sets the `S` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-S] variable to point to your external source code. Here are the statements to put in your recipe:

```
inherit externalsrc
S = "/some/path/to/your/package/source"
```

It is important to know that the `externalsrc.bbclass` assumes that the source directory `S` and the build directory `B` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-B] are different even though by default these directories are the same. This assumption is important because it supports building different variants of the recipe by using the `BBCLASSEXTEND` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-

BBCLASSEXTEND] variable. You could allow the build directory to be the same as the source directory but you would not be able to build more than one variant of the recipe. Consequently, if you are building multiple variants of the recipe, you need to establish a build directory that is different than the source directory.

4.12. Excluding Recipes From the Build

You might find that there are groups of recipes you want to filter out of the build process. For example, recipes you know you will never use or want should not be part of the build. Removing these recipes from parsing speeds up parts of the build.

It is possible to filter or mask out `.bb` and `.bbappend` files. You can do this by providing an expression with the BBMASK `[http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-BBMASK]` variable. Here is an example:

```
     BBMASK = ".*/meta-mymachine/recipes-maybe/"
```

Here, all `.bb` and `.bbappend` files in the directory that match the expression are ignored during the build process.

Chapter 5. Common Development Models

Many development models exist for which you can use the Yocto Project. However, for the purposes of this manual we are going to focus on two common models: System Development and User Application Development. System Development covers Board Support Package (BSP) development and kernel modification or configuration. User Application Development covers development of applications that you intend to run on some target hardware.

This chapter presents overviews of both system and application models. If you want to examine specific examples of the system development models, see the "BSP Development Example" appendix and the "Kernel Modification Example" appendix. For a user-space application development example that uses the Eclipse™ IDE, see the The Yocto Project Application Development Toolkit (ADT) User's Guide [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html].

Aside from these two models, this chapter will also briefly introduce and discuss development using Hob [http://www.yoctoproject.org/projects/hob], which is a graphical interface to the Yocto Project build system.

5.1. System Development Workflow

System development involves modification or creation of an image that you want to run on a specific hardware target. Usually, when you want to create an image that runs on embedded hardware, the image does not require the same number of features that a full-fledged Linux distribution provides. Thus, you can create a much smaller image that is designed to use only the hardware features for your particular hardware.

To help you understand how system development works in the Yocto Project, this section covers two types of image development: BSP creation and kernel modification or configuration.

5.1.1. Developing a Board Support Package (BSP)

A BSP is a package of recipes that, when applied, during a build results in an image that you can run on a particular board. Thus, the package, when compiled into the new image, supports the operation of the board.

Note

For a brief list of terms used when describing the development process in the Yocto Project, see the "Yocto Project Terms" section.

The remainder of this section presents the basic steps used to create a BSP based on an existing BSP that ships with the Yocto Project. You can reference the "BSP Development Example" appendix for a detailed example that uses the Crown Bay BSP as a base BSP from which to start.

The following illustration and list summarize the BSP creation general workflow.

1. Set up your host development system to support development using the Yocto Project: See the "The Linux Distributions [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#the-linux-distro]" and the "The Packages [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#packages]" sections both in the Yocto Project Quick Start for requirements.

2. Establish a local copy of the Yocto Project files on your system: You need to have the Yocto Project files available on your host system. Having the Yocto Project files on your system gives you access to the build process and to the tools you need. For information on how to get these files, see the "Getting Setup" section.

3. Establish a local copy of the base BSP files: Having the BSP files on your system gives you access to the build process and to the tools you need for creating a BSP. For information on how to get these files, see the "Getting Setup" section.

4. Choose a Yocto Project-supported BSP as your base BSP: The Yocto Project ships with several BSPs that support various hardware. It is best to base your new BSP on an existing BSP rather than

create all the recipes and configuration files from scratch. While it is possible to create everything from scratch, basing your new BSP on something that is close is much easier. Or, at a minimum, leveraging off an existing BSP gives you some structure with which to start.

At this point you need to understand your target hardware well enough to determine which existing BSP it most closely matches. Things to consider are your hardware's on-board features, such as CPU type and graphics support. You should look at the README files for supported BSPs to get an idea of which one you could use. A generic Intel® Atom™-based BSP to consider is the Crown Bay that does not support the Intel® Embedded Media Graphics Driver (EMGD). The remainder of this example uses that base BSP.

To see the supported BSPs, go to the Download [http://www.yoctoproject.org/download] page on the Yocto Project website and click on "BSP Downloads."

5. Create your own BSP layer: Layers are ideal for isolating and storing work for a given piece of hardware. A layer is really just a location or area in which you place the recipes for your BSP. In fact, a BSP is, in itself, a special type of layer.

   Another example that illustrates a layer is an application. Suppose you are creating an application that has library or other dependencies in order for it to compile and run. The layer, in this case, would be where all the recipes that define those dependencies are kept. The key point for a layer is that it is an isolated area that contains all the relevant information for the project that the Yocto Project build system knows about. For more information on layers, see the "Understanding and Creating Layers" section. For more information on BSP layers, see the "BSP Layers [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-layers]" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

   ### Note

   The Yocto Project supports four BSPs that are part of the Yocto Project release: `atom-pc`, `beagleboard`, `mpc8315e`, and `routerstationpro`. The recipes and configurations for these four BSPs are located and dispersed within the Yocto Project Files. On the other hand, BSP layers for Crown Bay, Emenlow, Jasper Forest, N450, Cedar Trail, Fish River, Fish River Island II, Romley, sys940x, tlk, and Sugar Bay exist in their own separate layers within the larger `meta-intel` layer.

   When you set up a layer for a new BSP, you should follow a standard layout. This layout is described in the section "Example Filesystem Layout [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-filelayout]" section of the Board Support Package (BSP) Development Guide. In the standard layout, you will notice a suggested structure for recipes and configuration information. You can see the standard layout for the Crown Bay BSP in this example by examining the directory structure of the `meta-crownbay` layer inside the local Yocto Project files.

6. Make configuration changes to your new BSP layer: The standard BSP layer structure organizes the files you need to edit in `conf` and several `recipes-*` directories within the BSP layer. Configuration changes identify where your new layer is on the local system and identify which kernel you are going to use.

7. Make recipe changes to your new BSP layer: Recipe changes include altering recipes (`.bb` files), removing recipes you don't use, and adding new recipes that you need to support your hardware.

8. Prepare for the build: Once you have made all the changes to your BSP layer, there remains a few things you need to do for the Yocto Project build system in order for it to create your image. You need to get the build environment ready by sourcing an environment setup script and you need to be sure two key configuration files are configured appropriately.

   The entire process for building an image is overviewed in the section "Building an Image [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#building-image]" section of the Yocto Project Quick Start. You might want to reference this information.

9. Build the image: The Yocto Project uses the BitBake tool to build images based on the type of image you want to create. You can find more information on BitBake here [http://bitbake.berlios.de/manual/].

   The build process supports several types of images to satisfy different needs. See the "Reference: Images [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-images]" appendix in The Yocto Project Reference Manual for information on supported images.

You can view a video presentation on "Building Custom Embedded Images with Yocto" at Free Electrons [http://free-electrons.com/blog/elc-2011-videos]. You can also find supplemental information in The Board Support Package (BSP) Development Guide [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html]. Finally, there is wiki page write up of the example also located here [https://wiki.yoctoproject.org/wiki/Transcript:_creating_one_generic_Atom_BSP_from_another] that you might find helpful.

5.1.2. Modifying the Kernel

Kernel modification involves changing the Linux Yocto kernel, which could involve changing configuration options as well as adding new kernel recipes. Configuration changes can be added in the form of configuration fragments, while recipe modification comes through the kernel's `recipes-kernel` area in a kernel layer you create.

The remainder of this section presents a high-level overview of the Linux Yocto kernel architecture and the steps to modify the Linux Yocto kernel. For a complete discussion of the kernel, see The Yocto Project Kernel Architecture and Use Manual [http://www.yoctoproject.org/docs/1.2.2/kernel-manual/kernel-manual.html]. You can reference the appendix "Kernel Modification Example" for a detailed example that changes the configuration of a kernel.

5.1.2.1. Kernel Overview

Traditionally, when one thinks of a patched kernel, they think of a base kernel source tree and a fixed structure that contains kernel patches. The Yocto Project, however, employs mechanisms, that in a sense, result in a kernel source generator. By the end of this section, this analogy will become clearer.

You can find a web interface to the Linux Yocto kernel source repositories at http://git.yoctoproject.org. If you look at the interface, you will see to the left a grouping of Git repositories titled "Yocto Linux Kernel." Within this group, you will find several kernels supported by the Yocto Project:

- `linux-yocto-2.6.34` - The stable Linux Yocto kernel that is based on the Linux 2.6.34 release.
- `linux-yocto-2.6.37` - The stable Linux Yocto kernel that is based on the Linux 2.6.37 release.
- `linux-yocto-3.0` - The stable Linux Yocto kernel that is based on the Linux 3.0 release.
- `linux-yocto-3.0-1.1.x` - The stable Linux Yocto kernel to use with the Yocto Project Release 1.1.x. This kernel is based on the Linux 3.0 release
- `linux-yocto-3.2` - The stable Linux Yocto kernel to use with the Yocto Project Release 1.2. This kernel is based on the Linux 3.2 release
- `linux-yocto-dev` - A development kernel based on the latest upstream release candidate available.

The kernels are maintained using the Git revision control system that structures them using the familiar "tree", "branch", and "leaf" scheme. Branches represent diversions from general code to more specific code, while leaves represent the end-points for a complete and unique kernel whose source files when gathered from the root of the tree to the leaf accumulate to create the files necessary for a specific piece of hardware and its features. The following figure displays this concept:

Within the figure, the "Kernel.org Branch Point" represents the point in the tree where a supported base kernel is modified from the Linux kernel. For example, this could be the branch point for the `linux-yocto-3.0` kernel. Thus, everything further to the right in the structure is based on the `linux-yocto-3.0` kernel. Branch points to right in the figure represent where the `linux-yocto-3.0` kernel is modified for specific hardware or types of kernels, such as real-time kernels. Each leaf thus represents the end-point for a kernel designed to run on a specific targeted device.

The overall result is a Git-maintained repository from which all the supported Yocto Project kernel types can be derived for all the supported Yocto Project devices. A big advantage to this scheme is the sharing of common features by keeping them in "larger" branches within the tree. This practice eliminates redundant storage of similar features shared among kernels.

Note

Keep in mind the figure does not take into account all the supported Linux Yocto kernel types, but rather shows a single generic kernel just for conceptual purposes. Also keep in mind that this structure represents the Yocto Project source repositories that are either pulled from during the build or established on the host development system prior to the build by either cloning a particular kernel's Git repository or by downloading and unpacking a tarball.

Storage of all the available kernel source code is one thing, while representing the code on your host development system is another. Conceptually, you can think of the Yocto Project kernel source repositories as all the source files necessary for all the supported kernels. As a developer, you are just interested in the source files for the kernel on on which you are working. And, furthermore, you need them available on your host system.

You make kernel source code available on your host development system by using Git to create a bare clone of the Linux Yocto kernel Git repository in which you are interested. Then, you use Git again to clone a copy of that bare clone. This copy represents the directory structure on your host system that is particular to the kernel you want. These are the files you actually modify to change the kernel. See the Linux Yocto Kernel item earlier in this manual for an example of how to set up the kernel source directory structure on your host system.

This next figure illustrates how the kernel source files might be arranged on your host system.

In the previous figure, the file structure on the left represents the bare clone set up to track the Yocto Project kernel Git repository. The structure on the right represents the copy of the bare clone. When you make modifcations to the kernel source code, this is the area in which you work. Once you make corrections, you must use Git to push the committed changes to the bare clone. The example in Section B.1, “Modifying the Kernel Source Code”provides a detailed example.

What happens during the build? When you build the kernel on your development system all files needed for the build are taken from the Yocto Project source repositories pointed to by the `SRC_URI` variable and gathered in a temporary work area where they are subsequently used to create the unique kernel. Thus, in a sense, the process constructs a local source tree specific to your kernel to generate the new kernel image - a source generator if you will.
The following figure shows the temporary file structure created on your host system when the build occurs. This build directory contains all the source files used during the build.

Again, for a complete discussion of the Yocto Project kernel's architcture and its branching strategy, see The Yocto Project Kernel Architecture and Use Manual [http://www.yoctoproject.org/docs/1.2.2/kernel-manual/kernel-manual.html]. You can also reference the "Modifying the Kernel Source Code" section for a detailed example that modifies the kernel.

5.1.2.2. Kernel Modification Workflow

This illustration and the following list summarizes the kernel modification general workflow.

1. Set up your host development system to support development using the Yocto Project: See "The Linux Distributions [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#the-linux-distro]" and "The Packages [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#packages]" sections both in the Yocto Project Quick Start for requirements.

2. Establish a local copy of the Yocto Project files on your system: Having the Yocto Project files on your system gives you access to the build process and tools you need. For information on how to get these files, see the bulleted item "Yocto Project Release" earlier in this manual.

3. Set up the `poky-extras` Git repository: This repository is the area for your configuration fragments, new kernel recipes, and the kernel `.bbappend` file used during the build. It is good practice to set this repository up inside the local Yocto Project files Git repository. For information on how to get these files, see the bulleted item "The `poky-extras` Git Repository" earlier in this manual.

Note

While it is certainly possible to modify the kernel without involving a local Git repository, the suggested workflow for kernel modification using the Yocto Project does use a Git repository.

4. Establish a local copy of the Linux Yocto kernel files on your system: In order to make modifications to the kernel you need two things: a bare clone of the Linux Yocto kernel you are modifying and a copy of that bare clone. The bare clone is required by the build process and is the area to which you push your kernel source changes (pulling does not work with bare clones). The copy of the bare clone is a local Git repository that contains all the kernel's source files. You make your changes to the files in this copy of the bare clone. For information on how to set these two items up, see the bulleted item "Linux Yocto Kernel" earlier in this manual.

5. Make changes to the kernel source code if applicable: Modifying the kernel does not always mean directly changing source files. However, if you have to do this, you make the changes in the local Git repository you set up to hold the source files (i.e. the copy of the bare clone). Once the changes are made, you need to use Git commands to commit the changes and then push them to the bare clone.

6. Make kernel configuration changes if applicable: If your situation calls for changing the kernel's configuration, you can use `menuconfig` to enable and disable kernel configurations. Using `menuconfig` allows you to interactively develop and test the configuration changes you are making to the kernel. When saved, changes using `menuconfig` update the kernel's `.config`. Try to resist the temptation of directly editing the `.config` file found in the Yocto Project Build Directory at `tmp/sysroots/<machine-name>/kernel`. Doing so, can produce unexpected results when the Yocto Project build system regenerates the configuration file.

   Once you are satisfied with the configuration changes made using `menuconfig`, you can directly examine the `.config` file against a saved original and gather those changes into a config fragment to be referenced from within the kernel's `.bbappend` file.

7. Add or extend kernel recipes if applicable: The standard layer structure organizes recipe files inside the `meta-kernel-dev` layer that is within the `poky-extras` Git repository. If you need to add new kernel recipes, you add them within this layer. Also within this area, you will find the `.bbappend` file that appends information to the kernel's recipe file used during the build.

8. Prepare for the build: Once you have made all the changes to your kernel (configurations, source code changes, recipe additions, or recipe changes), there remains a few things you need to do in order for the Yocto Project build system (BitBake) to create your image. If you have not done so, you need to get the build environment ready by sourcing the environment setup script described earlier. You also need to be sure two key configuration files (`local.conf` and `bblayers.conf`) are configured appropriately.

   The entire process for building an image is overviewed in the "Building an Image [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#building-image]" section of the Yocto Project Quick Start. You might want to reference this information. Also, you should look at the detailed examples found in the appendices at at the end of this manual.

9. Build the image: The Yocto Project build system Poky uses the BitBake tool to build images based on the type of image you want to create. You can find more information on BitBake here [http://bitbake.berlios.de/manual/].

   The build process supports several types of images to satisfy different needs. See the appendix "Reference: Images [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-images]" in The Yocto Project Reference Manual for information on supported images.

10. Make your configuration changes available in the kernel layer: Up to this point, all the configuration changes to the kernel have been done and tested iteratively. Once they are tested and ready to go, you can move them into the kernel layer, which allows you to distribute the layer.

11. If applicable, share your in-tree changes: If the changes you made are suited for all Linux Yocto users, you might want to send them on for inclusion into the Linux Yocto Git repository. If the changes are accepted, the Yocto Project Maintainer pulls them into the master branch of the kernel tree. Doing so makes them available to everyone using the kernel.

5.2. Application Development Workflow

Application development involves creation of an application that you want to be able to run on your target hardware, which is running a Linux Yocto image. The Yocto Project provides an Application Development Toolkit (ADT) that facilitates quick development and integration of your application into its run-time environment. Using the ADT you can employ cross-development toolchains designed for your target hardware to compile and link your application. You can then deploy your application to the actual hardware or to the QEMU emulator for testing. If you are familiar with the popular Eclipse IDE, you can use an Eclipse Yocto Plug-in to allow you to develop, deploy, and test your application all from within Eclipse.

While we strongly suggest using the Yocto Project ADT to develop your application, you might not want to. If this is the case, you can still use pieces of the Yocto Project for your development process. However, because the process can vary greatly, this manual does not provide detail on the process.

5.2.1. Workflow Using the ADT and Eclipse™

To help you understand how application development works in the Yocto Project ADT environment, this section provides an overview of the general development process. If you want to see a detailed example of the process as it is used from within the Eclipse IDE, see The Application Development Toolkit (ADT) User's Manual [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html].

This illustration and the following list summarizes the application development general workflow.

1. Prepare the Host System for the Yocto Project: See "The Linux Distributions [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#the-linux-distro]" and "The Packages [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#packages]" sections both in the Yocto Project Quick Start for requirements.

2. Secure the Linux Yocto Kernel Target Image: You must have a target kernel image that has been built using the Yocto Project.

Depending on whether the Yocto Project has a pre-built image that matches your target architecture and where you are going to run the image while you develop your application (QEMU or real hardware), the area from which you get the image differs.

- Download the image from `machines` [http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.2/machines] if your target architecture is supported and you are going to develop and test your application on actual hardware.
- Download the image from the `machines/qemu` [http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.2/machines/qemu] if your target architecture is supported and you are going to develop and test your application using the QEMU emulator.
- Build your image if you cannot find a pre-built image that matches your target architecture. If your target architecture is similar to a supported architecture, you can modify the kernel image before you build it. See the "Kernel Modification Workflow" section earlier in this manual for information on how to create a modified Linux Yocto kernel.

For information on pre-built kernel image naming schemes for images that can run on the QEMU emulator, see the "Using Pre-Built Binaries and QEMU [http://www.yoctoproject.org/docs/1.2.2/yocto-project-qs/yocto-project-qs.html#using-pre-built]" section in the Yocto Project Quick Start.

3. Install the ADT: The ADT provides a target-specific cross-development toolchain, the root filesystem, the QEMU emulator, and other tools that can help you develop your application. While it is possible to get these pieces separately, the Yocto Project provides an easy method. You can get these pieces by running an ADT installer script, which is configurable. For information on how to install the ADT, see the "Using the ADT Installer [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html#using-the-adt-installer]" section in the Yocto Project Application Development (ADT) User's Manual.
4. If Applicable, Secure the Target Root Filesystem: If you choose not to install the ADT using the ADT Installer, you need to find and download the appropriate root filesystems. You can find these tarballs in the same areas used for the kernel images. Depending on the type of image you are running, the root filesystem you need differs. For example, if you are developing an application that runs on an image that supports Sato, you need to get root filesystem that supports Sato.
5. Create and Build your Application: At this point, you need to have source files for your application. Once you have the files, you can use the Eclipse IDE to import them and build the project. If you are not using Eclipse, you need to use the cross-development tools you have installed to create the image.
6. Deploy the Image with the Application: If you are using the Eclipse IDE, you can deploy your image to the hardware or to QEMU through the project's preferences. If you are not using the Eclipse IDE, then you need to deploy the application using other methods to the hardware. Or, if you are using QEMU, you need to use that tool and load your image in for testing.
7. Test and Debug the Application: Once your application is deployed, you need to test it. Within the Eclipse IDE, you can use the debubbing environment along with the set of user-space tools installed along with the ADT to debug your application. Of course, the same user-space tools are available separately to use if you choose not to use the Eclipse IDE.

5.2.2. Workflow Without ADT

If you want to develop an application outside of the Yocto Project ADT environment, you can still employ the cross-development toolchain, the QEMU emulator, and a number of supported target image files. You just need to follow these general steps:

1. Install the cross-development toolchain for your target hardware: For information on how to install the toolchain, see the "Using a Cross-Toolchain Tarball [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html#using-an-existing-toolchain-tarball]" section in the Yocto Project Application Development (ADT) User's Manual.
2. Download the Target Image: The Yocto Project supports several target architectures and has many pre-built kernel images and root filesystem images.

   If you are going to develop your application on hardware, go to the `machines` [http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.2/machines] download area and choose a target machine area from which to download the kernel image and root filesystem. This download area could have several files in it that support development using actual hardware. For example, the area might contain `.hddimg` files that combine the kernel image with the filesystem, boot loaders, etc. Be sure to get the files you need for your particular development process.

If you are going to develop your application and then run and test it using the QEMU emulator, go to the `machines/qemu` [http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.2/machines/qemu] download area. From this area, go down into the directory for your target architecture (e.g. `qemux86_64` for an Intel®-based 64-bit architecture). Download kernel, root filesystem, and any other files you need for your process.

> **Note**
> In order to use the root filesystem in QEMU, you need to extract it. See the "Extracting the Root Filesystem [http://www.yoctoproject.org/docs/1.2.2/adt-manual/adt-manual.html#extracting-the-root-filesystem]" section for information on how to extract the root filesystem.

3. Develop and Test your Application: At this point, you have the tools to develop your application. If you need to separately install and use the QEMU emulator, you can go to QEMU Home Page [http://www.qemu.org] to download and learn about the emulator.

5.3. Image Development Using Hob

The Hob [http://www.yoctoproject.org/projects/hob] is a graphical user interface for the Yocto Project build system based on BitBake. You can use the Hob to build custom operating system images within the Yocto Project build environment. Hob simply provides a friendly interface over the build system used during system development. In other words, building images with the Hob lets you take care of common Yocto Project build tasks more easily.

For a better understanding of Hob, see the project page at http://www.yoctoproject.org/projects/hob on the Yocto Project website. The page has a short introductory training video on Hob. The following lists some features of Hob:

- You can setup and run Hob using these commands:

```
     $ source oe-init-build-env
     $ hob
```

- You can set the `MACHINE` [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-MACHINE] for which you are building the image.
- You can modify various policy settings such as the package format used to build with, the parrallelism BitBake uses, whether or not to build an external toolchain, and which host to build against.
- You can manage layers.
- You can select a base image and then add extra packages for your custom build.
- You can launch and monitor the build from within Hob.

Appendix A. BSP Development Example

This appendix provides a complete BSP development example. The example assumes the following:

- No previous preparation or use of the Yocto Project.
- Use of the Crown Bay Board Support Package (BSP) as a "base" BSP from which to work. The example begins with the Crown Bay BSP as the starting point but ends by building a new 'atom-pc' BSP, which was based on the Crown Bay BSP.
- Shell commands assume `bash`
- Example was developed on an Intel-based Core i7 platform running Ubuntu 10.04 LTS released in April of 2010.

A.1. Getting Local Yocto Project Files and BSP Files

You need to have the Yocto Project files available on your host system. You can get files through tarball extraction or by cloning the poky Git repository. The following paragraphs describe both methods. For additional information, see the bulleted item "Yocto Project Release".

As mentioned, one way to get the Yocto Project files is to use Git to clone the poky repository. These commands create a local copy of the Git repository. By default, the top-level directory of the repository is named poky:

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

Alternatively, you can start with the downloaded Poky "denzil" tarball. These commands unpack the tarball into a Yocto Project File directory structure. By default, the top-level directory of the file structure is named `poky-denzil-7.0.2`:

```
$ tar xfj poky-denzil-7.0.2.tar.bz2
$ cd poky-denzil-7.0.2
```

Note

If you're using the tarball method, you can ignore all the following steps that ask you to carry out Git operations. You already have the results of those operations in the form of the denzil release tarballs. Consequently, there is nothing left to do other than extract those tarballs into the proper locations.

Once you expand the released tarball, you have a snapshot of the Git repository that represents a specific release. Fundamentally, this is different than having a local copy of the Yocto Project Git repository. Given the tarball method, changes you make are building on top of a release. With the Git repository method you have the ability to track development and keep changes in revision control. See the "Repositories, Tags, and Branches" section for more discussion around these differences.

With the local poky Git repository set up, you have all the development branches available to you from which you can work. Next, you need to be sure that your local repository reflects the exact release in which you are interested. From inside the repository you can see the development branches that represent areas of development that have diverged from the main (master) branch at some point, such as a branch to track a maintenance release's development. You can also see the tag names used to mark snapshots of stable releases or points in the repository. Use the following commands to list out the branches and the tags in the repository, respectively.

```
     $ git branch -a
     $ git tag -l
```

For this example, we are going to use the Yocto Project 1.2.2 Release, which is code named "denzil". To make sure we have a local area (branch in Git terms) on our machine that reflects the 1.2.2 release, we can use the following commands:

```
     $ cd ~/poky
     $ git fetch --tags
     $ git checkout denzil-7.0.2 -b denzil
     Switched to a new branch 'denzil'
```

The `git fetch --tags` is somewhat redundant since you just set up the repository and should have all the tags. The `fetch` command makes sure all the tags are available in your local repository. The Git `checkout` command with the `-b` option creates a local branch for you named `denzil`. Your local branch begins in the same state as the Yocto Project 1.2.2 released tarball marked with the `denzil-7.0.2` tag in the source repositories.

A.2. Choosing a Base BSP

For this example, the base BSP is the Intel® Atom™ Processor E660 with Intel Platform Controller Hub EG20T Development Kit, which is otherwise referred to as "Crown Bay." The BSP layer is `meta-crownbay`. The base BSP is simply the BSP we will be using as a starting point, so don't worry if you don't actually have Crown Bay hardware. The remainder of the example transforms the base BSP into a BSP that should be able to boot on generic atom-pc (netbook) hardware.

For information on how to choose a base BSP, see "Developing a Board Support Package (BSP)".

A.3. Getting Your Base BSP

You need to have the base BSP layer on your development system. Similar to the local Yocto Project Files, you can get the BSP layer in a couple of different ways: download the BSP tarball and extract it, or set up a local Git repository that has the Yocto Project BSP layers. You should use the same method that you used to get the local Yocto Project files earlier. See "Getting Setup" for information on how to get the BSP files.

This example assumes the BSP layer will be located within a directory named `meta-intel` contained within the poky parent directory. The following steps will automatically create the `meta-intel` directory and the contained `meta-crownbay` starting point in both the Git and the tarball cases.

If you're using the Git method, you could do the following to create the starting layout after you have made sure you are in the poky directory created in the previous steps:

```
     $ git clone git://git.yoctoproject.org/meta-intel.git
     $ cd meta-intel
```

Alternatively, you can start with the downloaded Crown Bay tarball. You can download the denzil version of the BSP tarball from the Download [http://www.yoctoproject.org/download] page of the Yocto Project website. Here is the specific link for the tarball needed for this example: http://downloads.yoctoproject.org/releases/yocto/yocto-1.2.2/machines/crownbay-noemgd/crownbay-noemgd-denzil-7.0.2.tar.bz2. Again, be sure that you are already in the poky directory as described previously before installing the tarball:

```
     $ tar xfj crownbay-noemgd-denzil-7.0.2.tar.bz2
     $ cd meta-intel
```

The meta-intel directory contains all the metadata that supports BSP creation. If you're using the Git method, the following step will switch to the denzil metadata. If you're using the tarball method, you already have the correct metadata and can skip to the next step. Because meta-intel is its own Git repository, you will want to be sure you are in the appropriate branch for your work. For this example we are going to use the denzil branch.

```
$ git checkout -b denzil origin/denzil
Branch denzil set up to track remote branch denzil from origin.
Switched to a new branch 'denzil'
```

A.4. Making a Copy of the Base BSP to Create Your New BSP Layer

Now that you have the local Yocto Project files and the base BSP files, you need to create a new layer for your BSP. To create your BSP layer, you simply copy the meta-crownbay layer to a new layer.

For this example, the new layer will be named meta-mymachine. The name should follow the BSP layer naming convention, which is meta-<name>. The following assumes your working directory is meta-intel inside the local Yocto Project files. To start your new layer, just copy the new layer alongside the existing BSP layers in the meta-intel directory:

```
$ cp -a meta-crownbay/ meta-mymachine
```

A.5. Making Changes to Your BSP

Right now you have two identical BSP layers with different names: meta-crownbay and meta-mymachine. You need to change your configurations so that they work for your new BSP and your particular hardware. The following sections look at each of these areas of the BSP.

A.5.1. Changing the BSP Configuration

We will look first at the configurations, which are all done in the layer’s conf directory.

First, since in this example the new BSP will not support EMGD, we will get rid of the crownbay.conf file and then rename the crownbay-noemgd.conf file to mymachine.conf. Much of what we do in the configuration directory is designed to help the Yocto Project build system work with the new layer and to be able to find and use the right software. The following two commands result in a single machine configuration file named mymachine.conf.

```
$ rm meta-mymachine/conf/machine/crownbay.conf
$ mv meta-mymachine/conf/machine/crownbay-noemgd.conf \
meta-mymachine/conf/machine/mymachine.conf
```

Next, we need to make changes to the mymachine.conf itself. The only changes we want to make for this example are to the comment lines. Changing comments, of course, is never strictly necessary, but it's alway good form to make them reflect reality as much as possible. Here, simply substitute the Crown Bay name with an appropriate name for the BSP (mymachine in this case) and change the description to something that describes your hardware.

Note that inside the mymachine.conf is the PREFERRED_VERSION_linux-yocto statement. This statement identifies the kernel that the BSP is going to use. In this case, the BSP is using linux-yocto, which is the current Linux Yocto kernel based on the Linux 3.2 release.

The next configuration file in the new BSP layer we need to edit is meta-mymachine/conf/layer.conf. This file identifies build information needed for the new layer. You can see the "Layer

Configuration File [http://www.yoctoproject.org/docs/1.2.2/bsp-guide/bsp-guide.html#bsp-filelayout-layer]" section in The Board Support Packages (BSP) Development Guide for more information on this configuration file. Basically, we are changing the existing statements to work with our BSP.

The file contains these statements that reference the Crown Bay BSP:

```
BBFILE_COLLECTIONS += "crownbay"
BBFILE_PATTERN_crownbay := "^${LAYERDIR}/"
BBFILE_PRIORITY_crownbay = "6"

LAYERDEPENDS_crownbay = "intel"
```

Simply substitute the machine string name `crownbay` with the new machine name `mymachine` to get the following:

```
BBFILE_COLLECTIONS += "mymachine"
BBFILE_PATTERN_mymachine := "^${LAYERDIR}/"
BBFILE_PRIORITY_mymachine = "6"

LAYERDEPENDS_mymachine = "intel"
```

A.5.2. Changing the Recipes in Your BSP

Now we will take a look at the recipes in your new layer. The standard BSP structure has areas for BSP, graphics, core, and kernel recipes. When you create a BSP, you use these areas for appropriate recipes and append files. Recipes take the form of `.bb` files, while append files take the form of `.bbappend` files. If you want to leverage the existing recipes the Yocto Project build system uses but change those recipes, you can use `.bbappend` files. All new recipes and append files for your layer must go in the layer's `recipes-bsp`, `recipes-kernel`, `recipes-core`, and `recipes-graphics` directories.

A.5.2.1. Changing **`recipes-bsp`**

First, let's look at `recipes-bsp`. For this example we are not adding any new BSP recipes. And, we only need to remove the formfactor we do not want and change the name of the remaining one that doesn't support EMGD. These commands take care of the `recipes-bsp` recipes:

```
$ rm -rf meta-mymachine/recipes-bsp/formfactor/formfactor/crownbay
$ mv meta-mymachine/recipes-bsp/formfactor/formfactor/crownbay-noemgd/ \
meta-mymachine/recipes-bsp/formfactor/formfactor/mymachine
```

A.5.2.2. Changing **`recipes-graphics`**

Now let's look at `recipes-graphics`. For this example we want to remove anything that supports EMGD and be sure to rename remaining directories appropriately. The following commands clean up the `recipes-graphics` directory:

```
$ rm -rf meta-mymachine/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay
$ mv meta-mymachine/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd \
   meta-mymachine/recipes-graphics/xorg-xserver/xserver-xf86-config/mymachine
```

At this point the `recipes-graphics` directory just has files that support Video Electronics Standards Association (VESA) graphics modes and not EMGD.

A.5.2.3. Changing **`recipes-core`**

Now let's look at changes in `recipes-core`. The file `task-core-tools.bbappend` in `recipes-core/tasks` appends the similarly named recipe located in the local Yocto Project Files at `meta/recipes-`

core/tasks. The append file in our layer right now is Crown Bay-specific and supports EMGD and non-EMGD. Here are the contents of the file:

```
RRECOMMENDS_task-core-tools-profile_append_crownbay = " systemtap"
RRECOMMENDS_task-core-tools-profile_append_crownbay-noemgd = " systemtap"
```

The RRECOMMENDS statements list packages that extend usability. The first RRECOMMENDS statement can be removed, while the second one can be changed to reflect meta-mymachine:

```
RRECOMMENDS_task-core-tools-profile_append_mymachine = " systemtap"
```

A.5.2.4. Changing **recipes-kernel**

Finally, let's look at recipes-kernel changes. Recall that the BSP uses the linux-yocto kernel as determined earlier in the mymachine.conf. The recipe for that kernel is not located in the BSP layer but rather in the local Yocto Project files at meta/recipes-kernel/linux and is named linux-yocto_3.2.bb. The SRCREV_machine and SRCREV_meta statements point to the exact commits used by the Yocto Project development team in their source repositories that identify the right kernel for our hardware. In other words, the SRCREV values are simply Git commit IDs that identify which commit on each of the kernel branches (machine and meta) will be checked out and used to build the kernel.

However, in the meta-mymachine layer in recipes-kernel/linux resides a .bbappend file named linux-yocto_3.2.bbappend that appends information to the recipe of the same name in meta/recipes-kernel/linux. Thus, the SRCREV statements in the append file override the more general statements found in meta.

The SRCREV statements in the append file currently identify the kernel that supports the Crown Bay BSP with and without EMGD support. Here are the statements:

> **Note**
>
> The commit ID strings used in this manual might not match the actual commit ID strings found in the linux-yocto_3.2.bbappend file. For the example, this difference does not matter.

```
SRCREV_machine_pn-linux-yocto_crownbay ?= \
   "211fc7f4d10ec2b82b424286aabbaff9254b7cbd"
SRCREV_meta_pn-linux-yocto_crownbay ?= \
   "514847185c78c07f52e02750fbe0a03ca3a31d8f"

SRCREV_machine_pn-linux-yocto_crownbay-noemgd ?= \
   "211fc7f4d10ec2b82b424286aabbaff9254b7cbd"
SRCREV_meta_pn-linux-yocto_crownbay-noemgd ?= \
   "514847185c78c07f52e02750fbe0a03ca3a31d8f"
```

You will notice that there are two pairs of SRCREV statements. The top pair identifies the kernel that supports EMGD, which we don’t care about in this example. The bottom pair identifies the kernel that we will use: linux-yocto. At this point though, the unique commit strings all are still associated with Crown Bay and not meta-mymachine.

To fix this situation in linux-yocto_3.2.bbappend, we delete the two SRCREV statements that support EMGD (the top pair). We also change the remaining pair to specify mymachine and insert the commit identifiers to identify the kernel in which we are interested, which will be based on the atom-pc-standard kernel. In this case, because we're working with the denzil branch of everything, we need to use the SRCREV values for the atom-pc branch that are associated with the denzil release. To find those values, we need to find the SRCREV values that denzil uses for the atom-pc branch, which we find in the poky/meta-yocto/recipes-kernel/linux/linux-yocto_3.2.bbappend file.

The machine SRCREV we want is in the SRCREV_machine_atom-pc variable. The meta SRCREV isn't specified in this file, so it must be specified in the base kernel recipe in the poky/meta/recipes-kernel/linux/linux-yocto_3.2.bb file, in the SRCREV_meta variable found there. Here are the final SRCREV statements:

```
SRCREV_machine_pn-linux-yocto_mymachine ?= \
   "f29531a41df15d74be5ad47d958e4117ca9e489e"
SRCREV_meta_pn-linux-yocto_mymachine ?= \
   "b14a08f5c7b469a5077c10942f4e1aec171faa9d"
```

In this example, we're using the SRCREV values we found already captured in the denzil release because we're creating a BSP based on denzil. If, instead, we had based our BSP on the master branches, we would want to use the most recent SRCREV values taken directly from the kernel repo. We will not be doing that for this example. However, if you do base a future BSP on master and if you are familiar with Git repositories, you probably won't have trouble locating the exact commit strings in the Yocto Project source repositories you need to change the SRCREV statements. You can find all the `machine` and `meta` branch points (commits) for the `linux-yocto-3.2` kernel at http://git.yoctoproject.org/cgit/cgit.cgi/linux-yocto-3.2.

If you need a little more assistance after going to the link then do the following:

1. Expand the list of branches by clicking […]

2. Click on the `standard/default/common-pc/atom-pc` branch

3. Click on the commit column header to view the top commit

4. Copy the commit string for use in the `linux-yocto_3.2.bbappend` file

For the SRCREV statement that points to the `meta` branch use the same procedure except expand the `meta` branch in step 2 above.

Also in the `linux-yocto_3.2.bbappend` file are `COMPATIBLE_MACHINE`, `KMACHINE`, and `KBRANCH` statements. Two sets of these exist: one set supports EMGD and one set does not. Because we are not interested in supporting EMGD those three can be deleted. The remaining three must be changed so that `mymachine` replaces `crownbay-noemgd` and `crownbay`. Because we are using the `atom-pc` branch for this new BSP, we can also find the exact branch we need for the `KMACHINE` and `KBRANCH` variables in our new BSP from the value we find in the `poky/meta-yocto/recipes-kernel/linux/linux-yocto_3.2.bbappend` file we looked at in a previous step. In this case, the values we want are in the `KMACHINE_atom-pc` variable and the `KBRANCH_atom-pc` variables in that file. Here is the final `linux-yocto_3.2.bbappend` file after all the edits:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

COMPATIBLE_MACHINE_mymachine = "mymachine"
KMACHINE_mymachine  = "atom-pc"
KBRANCH_mymachine  = "standard/default/common-pc/atom-pc"

SRCREV_machine_pn-linux-yocto_mymachine ?= \
   "f29531a41df15d74be5ad47d958e4117ca9e489e"
SRCREV_meta_pn-linux-yocto_mymachine ?= \
   "b14a08f5c7b469a5077c10942f4e1aec171faa9d"
```

A.5.3. BSP Recipe Change Summary

In summary, the edits to the layer's recipe files result in removal of any files and statements that do not support your targeted hardware in addition to the inclusion of any new recipes you might need. In this example, it was simply a matter of ridding the new layer `meta-mymachine` of any code that supported the EMGD features and making sure we were identifying the kernel that supports our example, which is the `atom-pc-standard` kernel. We did not introduce any new recipes to the layer.

Finally, it is also important to update the layer's `README` file so that the information in it reflects your BSP.

A.6. Preparing for the Build

To get ready to build your image that uses the new layer you need to do the following:

1. Get the environment ready for the build by sourcing the environment script. The environment script is in the top-level of the local Yocto Project files directory structure. The script has the string `init-build-env` in the file's name. For this example, the following command gets the build environment ready:

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

   When you source the script a build directory is created in the current working directory. In our example we were in the poky directory. Thus, entering the previous command created the `yocto-build` directory. If you do not provide a name for the build directory it defaults to `build`. The `yocto-build` directory contains a `conf` directory that has two configuration files you will need to check: `bblayers.conf` and `local.conf`.

2. Check and edit the resulting `local.conf` file. This file minimally identifies the machine for which to build the image by configuring the MACHINE variable. For this example you must set the variable to mymachine as follows:

   ```
   MACHINE ??= "mymachine"
   ```

   You should also be sure any other variables in which you are interested are set. Some variables to consider are BB_NUMBER_THREADS and PARALLEL_MAKE, both of which can greatly reduce your build time if your development system supports multiple cores. For development systems that support multiple cores, a good rule of thumb is to set both the BB_NUMBER_THREADS and PARALLEL_MAKE variables to twice the number of cores your system supports.

3. Update the `bblayers.conf` file so that it includes both the path to your new BSP layer and the path to the `meta-intel` layer. In this example, you need to include both these paths as part of the BBLAYERS variable:

   ```
   $HOME/poky/meta-intel
   $HOME/poky/meta-intel/meta-mymachine
   ```

The appendix Reference: Variables Glossary [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#ref-variables-glos] in the Yocto Project Reference Manual has more information on configuration variables.

A.7. Building and Booting the Image

To build the image for our `meta-mymachine` BSP enter the following command from the same shell from which you ran the setup script. You should run the `bitbake` command without any intervening shell commands. For example, moving your working directory around could cause problems. Here is the command for this example:

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

This command specifies an image that has Sato support and that can be run from a USB device or from a CD without having to first install anything. The build process takes significant time and includes thousands of tasks, which are reported at the console. If the build results in any type of error you should check for misspellings in the files you changed or problems with your host development environment such as missing packages.

Finally, once you have an image, you can try booting it from a device (e.g. a USB device). To prepare a bootable USB device, insert a USB flash drive into your build system and copy the `.hddimg` file, located in the `poky/build/tmp/deploy/images` directory after a successful build to the flash drive. Assuming the USB flash drive takes device `/dev/sdf`, use dd to copy the live image to it. For example:

```
# dd if=core-image-sato-mymachine-20111101223904.hddimg of=/dev/sdf
```

```
# sync
# eject /dev/sdf
```

You should now have a bootable USB flash device.

Insert the device into a bootable USB socket on the target, and power it on. The system should boot to the Sato graphical desktop. [1]

For reference, the sato image produced by the previous steps for denzil should look like the following in terms of size. If your sato image is much different from this, you probably made a mistake in one of the above steps:

```
260538368 2012-04-27 01:44 core-image-sato-mymachine-20120427025051.hddimg
```

Note

The previous instructions are also present in the README that was copied from meta-crownbay, which should also be updated to reflect the specifics of your new BSP. That file and the `README.hardware` file in the top-level `poky` directory also provides some suggestions for things to try if booting fails and produces strange error messages.

[1]Because this new image is not in any way tailored to the system you're booting it on, which is assumed to be some sort of atom-pc (netbook) system for this example, it might not be completely functional though it should at least boot to a text prompt. Specifically, it might fail to boot into graphics without some tweaking. If this ends up being the case, a possible next step would be to replace the `mymachine.conf` contents with the contents of `atom-pc.conf` and replace `xorg.conf` with `atom-pc xorg.conf` in `meta-yocto` and see if it fares any better. In any case, following the previous steps will give you a buildable image that will probably boot on most systems. Getting things working like you want them to for your hardware will normally require some amount of experimentation with configuration settings.

Appendix B. Kernel Modification Example

Kernel modification involves changing or adding configurations to an existing kernel, changing or adding recipes to the kernel that are needed to support specific hardware features, or even altering the source code itself. This appendix presents simple examples that modify the kernel source code, change the kernel configuration, and add a kernel source recipe.

B.1. Modifying the Kernel Source Code

This example adds some simple QEMU emulator console output at boot time by adding `printk` statements to the kernel's `calibrate.c` source code file. Booting the modified image causes the added messages to appear on the emulator's console.

B.1.1. Understanding the Files You Need

Before you modify the kernel, you need to know what Git repositories and file structures you need. Briefly, you need the following:

- A local Yocto Project Files Git repository
- The `poky-extras` Git repository placed within the local Yocto Project files Git repository
- A bare clone of the Linux Yocto Kernel upstream Git repository to which you want to push your modifications.
- A copy of that bare clone in which you make your source modifcations

The following figure summarizes these four areas. Within each rectangular that represents a data structure, a host development directory pathname appears at the lower left-hand corner of the box. These pathnames are the locations used in this example. The figure also provides key statements and commands used during the kernel modification process:

Here is a brief description of the four areas:

- Local Yocto Project Files Git Repository: This area contains all the metadata that supports building images in the Yocto Project build environment - the local Yocto Project files. In this example, the local Yocto Project files Git repository also contains the build directory, which contains the configuration directory that lets you control the build. In this example, the repository also contains the `poky-extras` Git repository.

  See the bulleted item "Yocto Project Release" for information on how to get these files.

- `poky-extras` Git Repository: This area contains the `meta-kernel-dev` layer, which is where you make changes that append the kernel build recipes. You edit `.bbappend` files to locate your local kernel source files and to identify the kernel being built. This Git repository is a gathering place for extensions to the Linux Yocto (or really any) kernel recipes that faciliate the creation and development of kernel features, BSPs or configurations.

  See the bulleted item "The `poky-extras` Git Repository" for information on how to get these files.

- Bare Clone of the Linux Yocto kernel: This bare Git repository tracks the upstream Git repository of the Linux Yocto kernel source code you are changing. When you modify the kernel you must work through a bare clone. All source code changes you make to the kernel must be committed and pushed to the bare clone using Git commands. As mentioned, the `.bbappend` file in the `poky-extras` repository points to the bare clone so that the build process can locate the locally changed source files.

  See the bulleted item "Linux Yocto Kernel" for information on how to set up the bare clone.

- Copy of the Linux Yocto Kernel Bare Clone: This Git repository contains the actual source files that you modify. Any changes you make to files in this location need to ultimately be pushed to the bare clone using the `git push` command.

  See the bulleted item "Linux Yocto Kernel" for information on how to set up the bare clone.

Note

Typically, Git workflows follow a scheme where changes made to a local area are pulled into a Git repository. However, because the `git pull` command does not work with bare clones, this workflow pushes changes to the repository even though you could use other more complicated methods to get changes into the bare clone.

B.1.2. Setting Up the Local Yocto Project Files Git Repository

You can get the local Yocto Project files through tarball extraction or by cloning the poky Git repository. This example uses poky as the root directory of the local Yocto Project files Git repository. See the bulleted item "Yocto Project Release" for information on how to get these files.

Once you have the repository set up, you have many development branches from which you can work. From inside the repository you can see the branch names and the tag names used in the Git repository using either of the following two commands:

```
$ cd poky
$ git branch -a
$ git tag -l
```

This example uses the Yocto Project 1.2.2 Release code named "denzil", which maps to the `denzil` branch in the repository. The following commands create and checkout the local `denzil` branch:

```
$ git checkout -b denzil origin/denzil
Branch denzil set up to track remote branch denzil from origin.
Switched to a new branch 'denzil'
```

B.1.3. Setting Up the poky-extras Git Repository

This example places the `poky-extras` Git repository inside of poky. See the bulleted item "The `poky-extras` Git Repository" for information on how to get the `poky-extras` repository.

Because this example uses the Yocto Project 1.2.2 Release code named "denzil", which maps to the `denzil` branch in the repository, you need to be sure you are using that branch for `poky-extra`. The following commands create and checkout the local branch you are using for the `denzil` branch:

```
$ git checkout -b denzil origin/denzil
Branch denzil set up to track remote branch denzil from origin.
Switched to a new branch 'denzil'
```

B.1.4. Setting Up the Bare Clone and its Copy

This example modifies the `linux-yocto-3.2` kernel. Thus, you need to create a bare clone of that kernel and then make a copy of the bare clone. See the bulleted item "Linux Yocto Kernel" for information on how to do that.

The bare clone exists for the kernel build tools and simply as the receiving end of `git push` commands after you make edits and commits inside the copy of the clone. The copy (`my-linux-yocto-3.2-work` in this example) has to have a local branch created and checked out for your work. This example uses `common-pc-base` as the local branch. The following commands create and checkout the branch:

```
$ cd ~/my-linux-yocto-3.2-work
$ git checkout -b common-pc-base origin/standard/default/common-pc/base
Checking out files: 100% (532/532), done.
```

```
Branch common-pc-base set up to track remote branch
   standard/default/common-pc/base from origin.
Switched to a new branch 'common-pc-base'
```

B.1.5. Building and Booting the Default QEMU Kernel Image

Before we make changes to the kernel source files, this example first builds the default image and then boots it inside the QEMU emulator.

Note

Because a full build can take hours, you should check two variables in the build directory that is created after you source the oe-init-build-env script. You can find these variables BB_NUMBER_THREADS and PARALLEL_MAKE in the build/conf directory in the local.conf configuration file. By default, these variables are commented out. If your host development system supports multi-core and multi-thread capabilities, you can uncomment these statements and set the variables to significantly shorten the full build time. As a guideline, set both BB_NUMBER_THREADS and PARALLEL_MAKE to twice the number of cores your machine supports.

The following two commands `source` the build environment setup script and build the default qemux86 image. If necessary, the script creates the build directory:

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

     ### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-toolchain-sdk
    adt-installer
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
```

The following `bitbake` command starts the build:

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

Note

Be sure to check the settings in the `local.conf` before starting the build.

After the build completes, you can start the QEMU emulator using the resulting image qemux86 as follows:

```
$ runqemu qemux86
```

As the image boots in the emulator, console message and status output appears across the terminal window. Because the output scrolls by quickly, it is difficult to read. To examine the output, you log into the system using the login `root` with no password. Once you are logged in, issue the following command to scroll through the console output:

```
# dmesg | less
```

Take note of the output as you will want to look for your inserted print command output later in the example.

B.1.6. Changing the Source Code and Pushing it to the Bare Clone

The file you change in this example is named `calibrate.c` and is located in the `my-linux-yocto-3.2-work` Git repository (the copy of the bare clone) in `init`. This example simply inserts several `printk` statements at the beginning of the `calibrate_delay` function.

Here is the unaltered code at the start of this function:

```
void __cpuinit calibrate_delay(void)
{
 unsigned long lpj;
 static bool printed;
 int this_cpu = smp_processor_id();

 if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
          .
          .
          .
```

Here is the altered code showing five new `printk` statements near the top of the function:

```
void __cpuinit calibrate_delay(void)
{
    unsigned long lpj;
    static bool printed;
    int this_cpu = smp_processor_id();

    printk("*************************************\n");
    printk("*                                   *\n");
    printk("*        HELLO YOCTO KERNEL         *\n");
    printk("*                                   *\n");
    printk("*************************************\n");

 if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
          .
          .
          .
```

After making and saving your changes, you need to stage them for the push. The following Git commands are one method of staging and committing your changes:

```
$ git add calibrate.c
$ git commit --signoff
```

Once the source code has been modified, you need to use Git to push the changes to the bare clone. If you do not push the changes, then the Yocto Project build system will not pick up the changed source files.

The following command pushes the changes to the bare clone:

```
$ git push origin common-pc-base:standard/default/common-pc/base
```

B.1.7. Changing Build Parameters for Your Build

At this point, the source has been changed and pushed. The example now defines some variables used by the Yocto Project build system to locate your kernel source. You essentially need to identify where to find the kernel recipe and the changed source code. You also need to be sure some basic configurations are in place that identify the type of machine you are building and to help speed up the build should your host support multiple-core and thread capabilities.

Do the following to make sure the build parameters are set up for the example. Once you set up these build parameters, they do not have to change unless you change the target architecture of the machine you are building or you move the bare clone, copy of the clone, or the `poky-extras` repository:

- Build for the Correct Target Architecture: The `local.conf` file in the build directory defines the build's target architecture. By default, `MACHINE` is set to `qemux86`, which specifies a 32-bit Intel® Architecture target machine suitable for the QEMU emulator. In this example, `MACHINE` is correctly configured.
- Optimize Build Time: Also in the `local.conf` file are two variables that can speed your build time if your host supports multi-core and multi-thread capabilities: `BB_NUMBER_THREADS` and `PARALLEL_MAKE`. If the host system has multiple cores then you can optimize build time by setting both these variables to twice the number of cores.
- Identify Your `meta-kernel-dev` Layer: The `BBLAYERS` variable in the `bblayers.conf` file found in the `poky/build/conf` directory needs to have the path to your local `meta-kernel-dev` layer. By default, the `BBLAYERS` variable contains paths to `meta` and `meta-yocto` in the `poky` Git repository. Add the path to your `meta-kernel-dev` location. Be sure to substitute your user information in the statement. Here is an example:

  ```
  BBLAYERS = " \
    /home/scottrif/poky/meta \
    /home/scottrif/poky/meta-yocto \
    /home/scottrif/poky/poky-extras/meta-kernel-dev \
    "
  ```

- Identify Your Source Files: In the `linux-yocto_3.2.bbappend` file located in the `poky-extras/meta-kernel-dev/recipes-kernel/linux` directory, you need to identify the location of the local source code, which in this example is the bare clone named `linux-yocto-3.2.git`. To do this, set the `KSRC_linux_yocto` variable to point to your local `linux-yocto-3.2.git` Git repository by adding the following statement. Be sure to substitute your user information in the statement:

  ```
  KSRC_linux_yocto_3_2 ?= "/home/scottrif/linux-yocto-3.2.git"
  ```

  ### Note

  Before attempting to build the modified kernel, there is one more set of changes you need to make in the `meta-kernel-dev` layer. Because all the kernel `.bbappend` files are parsed during the build process regardless of whether you are using them or not, you should either comment out the `COMPATIBLE_MACHINE` statements in all unused `.bbappend` files. Alternatively, you can simply remove all the files except the one your are using for the build (i.e. `linux-yocto_3.2.bbappend` in this example).

B.1.8. Building and Booting the Modified QEMU Kernel Image

Next, you need to build the modified image. Do the following:

1. Your environment should be set up since you previously sourced the `oe-init-build-env` script. If it isn't, source the script again from `poky`.

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

2. Be sure old images are cleaned out by running the `cleanall` BitBake task as follows from your build directory:

```
    $ bitbake -c cleanall linux-yocto
```

Note

Never remove any files by hand from the `tmp/deploy` directory insided the local Yocto Project files build directory. Always use the BitBake `cleanall` task to clear out previous builds.

3. Next, build the kernel image using this command:

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

4. Finally, boot the modified image in the QEMU emulator using this command:

```
    $ runqemu qemux86
```

Log into the machine using `root` with no password and then use the following shell command to scroll through the console's boot output.

```
  # dmesg | less
```

You should see the results of your `printk` statements as part of the output.

B.2. Changing the Kernel Configuration

This example changes the default behavior, which is "on", of the Symmetric Multi-processing Support (`CONFIG_SMP`) to "off". It is a simple example that demonstrates how to reconfigure the kernel.

B.2.1. Getting Set Up to Run this Example

If you took the time to work through the example that modifies the kernel source code in "Modifying the Kernel Source Code" you should already have the Yocto Project files set up on your host machine. If this is the case, go to the next section, which is titled "Examining the Default `CONFIG_SMP` Behavior", and continue with the example.

If you don't have the Yocto Project files established on your system, you can get them through tarball extraction or by cloning the poky Git repository. This example uses `poky` as the root directory of the local Yocto Project Files Git repository. See the bulleted item "Yocto Project Release" for information on how to get these files.

Once you have the repository set up, you have many development branches from which you can work. From inside the repository you can see the branch names and the tag names used in the Git repository using either of the following two commands:

```
  $ cd poky
  $ git branch -a
  $ git tag -l
```

This example uses the Yocto Project 1.2.2 Release code named "denzil", which maps to the `denzil` branch in the repository. The following commands create and checkout the local `denzil` branch:

```
$ git checkout -b denzil origin/denzil
Branch denzil set up to track remote branch denzil from origin.
Switched to a new branch 'denzil'
```

Next, you need to build the default qemux86 image that you can boot using QEMU.

Note

Because a full build can take hours, you should check two variables in the `build` directory that is created after you source the `oe-init-build-env` script. You can find these variables `BB_NUMBER_THREADS` and `PARALLEL_MAKE` in the `build/conf` directory in the `local.conf` configuration file. By default, these variables are commented out. If your host development system supports multi-core and multi-thread capabilities, you can uncomment these statements and set the variables to significantly shorten the full build time. As a guideline, set `BB_NUMBER_THREADS` to twice the number of cores your machine supports and set `PARALLEL_MAKE` to one and a half times the number of cores your machine supports.

The following two commands `source` the build environment setup script and build the default qemux86 image. If necessary, the script creates the build directory:

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

     ### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-toolchain-sdk
    adt-installer
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
```

The following `bitbake` command starts the build:

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

Note

Be sure to check the settings in the `local.conf` before starting the build.

B.2.2. Examining the Default **`CONFIG_SMP`** Behavior

By default, `CONFIG_SMP` supports multiple processor machines. To see this default setting from within the QEMU emulator, boot your image using the emulator as follows:

```
$ runqemu qemux86 qemuparams="-smp 4"
```

Login to the machine using `root` with no password. After logging in, enter the following command to see how many processors are being supported in the emulator. The emulator reports support for the number of processors you specified using the `-smp` option, four in this case:

```
# cat /proc/cpuinfo | grep processor
processor       : 0
processor       : 1
processor       : 2
processor       : 3
#
```

To check the setting for CONFIG_SMP, you can use the following command:

```
zcat /proc/config.gz | grep CONFIG_SMP
```

The console returns the following showing that multi-processor machine support is set:

```
CONFIG_SMP=y
```

Logout of the emulator using the exit command and then close it down.

B.2.3. Changing the **CONFIG_SMP** Configuration Using **menuconfig**

The menuconfig tool provides an interactive method with which to set kernel configurations. You need to run menuconfig inside the Yocto BitBake environment. Thus, the environment must be set up using the oe-init-build-env script found in the Yocto Project files Git repository build directory. If you have not sourced this script do so with the following commands:

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

After setting up the environment to run menuconfig, you are ready to use the tool to interactively change the kernel configuration. In this example, we are basing our changes on the linux-yocto-3.2 kernel. The Yocto Project build environment recognizes this kernel as linux-yocto. Thus, the following commands from the shell in which you previously sourced the environment initialization script cleans the shared state memory and the WORKDIR [http://www.yoctoproject.org/docs/1.2.2/poky-ref-manual/poky-ref-manual.html#var-WORKDIR] directory and then builds and launches menuconfig:

```
$ bitbake linux-yocto -c cleansstate
$ bitbake linux-yocto -c menuconfig
```

Note

Due to a bug in the release, it is necessary to clean the shared state memory in order for configurations made using menuconfig to take effect. For information on the bug, see https://bugzilla.yoctoproject.org/show_bug.cgi?id=2256

Once menuconfig launches, navigate through the user interface to find the CONFIG_SMP configuration setting. You can find it at Processor Type and Features. The configuration selection is Symmetric Multi-processing Support. After using the arrow keys to highlight this selection, press "n" to turn it off. Then, exit out and save your selections.

Once you save the selection, the .config configuration file is updated. This is the file that the build system uses to configure the Linux Yocto kernel when it is built. You can find and examine this file in the Yocto Project Files Git repository in the build directory. This example uses the following:

```
~/poky/build/tmp/work/qemux86-poky-linux/linux-yocto-3.2.11+git1+84f...
   ...656ed30-r1/linux-qemux86-standard-build
```

> **Note**
> The previous example directory is artificially split and many of the characters in the actual filename are omitted in order to make it more readable. Also, depending on the kernel you are using, the exact pathname might differ slightly.

Within the `.config` file, you can see the following setting:

```
# CONFIG_SMP is not set
```

A good method to isolate changed configurations is to use a combination of the `menuconfig` tool and simple shell commands. Before changing configurations with `menuconfig`, copy the existing `.config` and rename it to something else, use `menuconfig` to make as many changes an you want and save them, then compare the renamed configuration file against the newly created file. You can use the resulting differences as your base to create configuration fragments to permanently save in your kernel layer.

> **Note**
> Be sure to make a copy of the `.config` and don't just rename it. The Yocto Project build system needs an existing `.config` from which to work.

B.2.4. Recompiling the Kernel and Testing the New Configuration

At this point, you are ready to recompile your kernel image with the new setting in effect using the BitBake commands below:

```
$ bitbake linux-yocto -c compile -f
$ bitbake linux-yocto
```

> **Note**
> Manually turning off a kernel configuration setting such as `CONFIG_SMP` can cause the kernel configuration audit to issue warnings during the build. In this example, warnings appear telling you that the expected value `CONFIG_SMP` does not appear in the `.config` file. Because in this example you specifically turned off `CONFIG_SMP`, you can safely ignore the apparent conflict.

Now run the QEMU emulator and pass it the same multi-processor option as before:

```
$ runqemu qemux86 qemuparams="-smp 4"
```

Login to the machine using `root` with no password and test for the number of processors the kernel supports:

```
# cat /proc/cpuinfo | grep processor
processor       : 0
#
```

From the output, you can see that the kernel no longer supports multi-processor systems. The output indicates support for a single processor. You can verify the `CONFIG_SMP` setting by using this command:

```
zcat /proc/config.gz | grep CONFIG_SMP
```

The console returns the following output:

```
     # CONFIG_SMP is not set
```

You have successfully reconfigured the kernel.

B.3. Adding Kernel Recipes

A future release of this manual will present an example that adds kernel recipes, which provide new functionality to the kernel.