The Yocto Project Kernel Architecture and Use Manual

Bruce Ashfield, Wind River Corporation
<bruce.ashfield@windriver.com>

Table of Contents

Chapter 1. Yocto Project Kernel Architecture and Use Manual

1.1. Introduction

The Yocto Project presents the kernel as a fully patched, history-clean git repository. The git tree represents the selected features, board support, and configurations extensively tested by Yocto Project. The Yocto Project kernel allows the end user to leverage community best practices to seamlessly manage the development, build and debug cycles.

This manual describes the Yocto Project kernel by providing information on its history, organization, benefits, and use. The manual consists of two sections:

- Concepts - Describes concepts behind the kernel. You will understand how the kernel is organized and why it is organized in the way it is. You will understand the benefits of the kernel's organization and the mechanisms used to work with the kernel and how to apply it in your design process.
- Using the Kernel - Describes best practices and "how-to" information that lets you put the kernel to practical use. Some examples are "How to Build a Project Specific Tree", "How to Examine Changes in a Branch", and "Saving Kernel Modifications."

For more information on the kernel, see the following links:

- http://ldn.linuxfoundation.org/book/1-a-guide-kernel-development-process
- http://userweb.kernel.org/~akpm/stuff/tpp.txt
- http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob_plain;f=Documentation/HOWTO;hb=HEAD

You can find more information on Yocto Project by visiting the website at http://www.yoctoproject.org.

Chapter 2. Yocto Project Kernel Concepts

2.1. Introduction

This chapter provides conceptual information about the Yocto Project kernel:

- Kernel Goals
- Yocto Project Kernel Development and Maintenance Overview
- Kernel Architecture
- Kernel Tools

2.2. Kernel Goals

The complexity of embedded kernel design has increased dramatically. Whether it is managing multiple implementations of a particular feature or tuning and optimizing board specific features, flexibility and maintainability are key concerns. The Yocto Project Linux kernel is presented with the embedded developer's needs in mind and has evolved to assist in these key concerns. For example, prior methods such as applying hundreds of patches to an extracted tarball have been replaced with proven techniques that allow easy inspection, bisection and analysis of changes. Application of these techniques also creates a platform for performing integration and collaboration with the thousands of upstream development projects.

With all these considerations in mind, the Yocto Project kernel and development team strives to attain these goals:

- Allow the end user to leverage community best practices to seamlessly manage the development, build and debug cycles.
- Create a platform for performing integration and collaboration with the thousands of upstream development projects that exist.
- Provide mechanisms that support many different work flows, front-ends and management techniques.
- Deliver the most up-to-date kernel possible while still ensuring that the baseline kernel is the most stable official release.
- Include major technological features as part of Yocto Project's up-rev strategy.
- Present a git tree, that just like the upstream kernel.org tree, has a clear and continuous history.
- Deliver a key set of supported kernel types, where each type is tailored to a specific use case (i.g. networking, consumer, devices, and so forth).
- Employ a git branching strategy that from a customer's point of view results in a linear path from the baseline kernel.org, through a select group of features and ends with their BSP-specific commits.

2.3. Yocto Project Kernel Development and Maintenance Overview

Yocto Project kernel, like other kernels, is based off the Linux kernel release from http://www.kernel.org. At the beginning of our major development cycle, we choose our Yocto Project kernel based on factors like release timing, the anticipated release timing of "final" (i.e. non "rc") upstream kernel.org versions, and Yocto Project feature requirements. Typically this will be a kernel that is in the final stages of development by the community (i.e. still in the release candidate or "rc" phase) and not yet a final release. But by being in the final stages of external development, we know that the kernel.org final release will clearly land within the early stages of the Yocto Project development window.

This balance allows us to deliver the most up-to-date kernel as possible, while still ensuring that we have a stable official release as our baseline kernel version.

The ultimate source for the Yocto Project kernel is a released kernel from kernel.org. In addition to a foundational kernel from kernel.org the released Yocto Project kernel contains a mix of important new mainline developments, non-mainline developments (when there is no alternative), Board Support Package (BSP) developments, and custom features. These additions result in a commercially released Yocto Project kernel that caters to specific embedded designer needs for targeted hardware.

Once a Yocto Project kernel is officially released the Yocto Project team goes into their next development cycle, or "uprev" cycle while continuing maintenance on the released kernel. It is important to note that the most sustainable and stable way to include feature development upstream is through a kernel uprev process. Back-porting of hundreds of individual fixes and minor features from various kernel versions is not sustainable and can easily compromise quality. During the uprev cycle, the Yocto Project team uses an ongoing analysis of kernel development, BSP support, and release timing to select the best possible kernel.org version. The team continually monitors community kernel development to look for significant features of interest. The team does consider back-porting large features if they have a significant advantage. User or community demand can also trigger a back-port or creation of new functionality in the Yocto Project baseline kernel during the uprev cycle.

Generally speaking, every new kernel both adds features and introduces new bugs. These consequences are the basic properties of upstream kernel development and are managed by the Yocto Project team's kernel strategy. It is the Yocto Project team's policy to not back-port minor features to the released kernel. They only consider back-porting significant technological jumps - and, that is done after a complete gap analysis. The reason for this policy is that simply back-porting any small to medium sized change from an evolving kernel can easily create mismatches, incompatibilities and very subtle errors.

These policies result in both a stable and a cutting edge kernel that mixes forward ports of existing features and significant and critical new functionality. Forward porting functionality in the Yocto Project kernel can be thought of as a "micro uprev." The many “micro uprevs” produce a kernel version with a mix of important new mainline, non-mainline, BSP developments and feature integrations. This kernel gives insight into new features and allows focused amounts of testing to be done on the kernel, which prevents surprises when selecting the next major uprev. The quality of these cutting edge kernels is evolving and the kernels are used in leading edge feature and BSP development.

2.4. Kernel Architecture

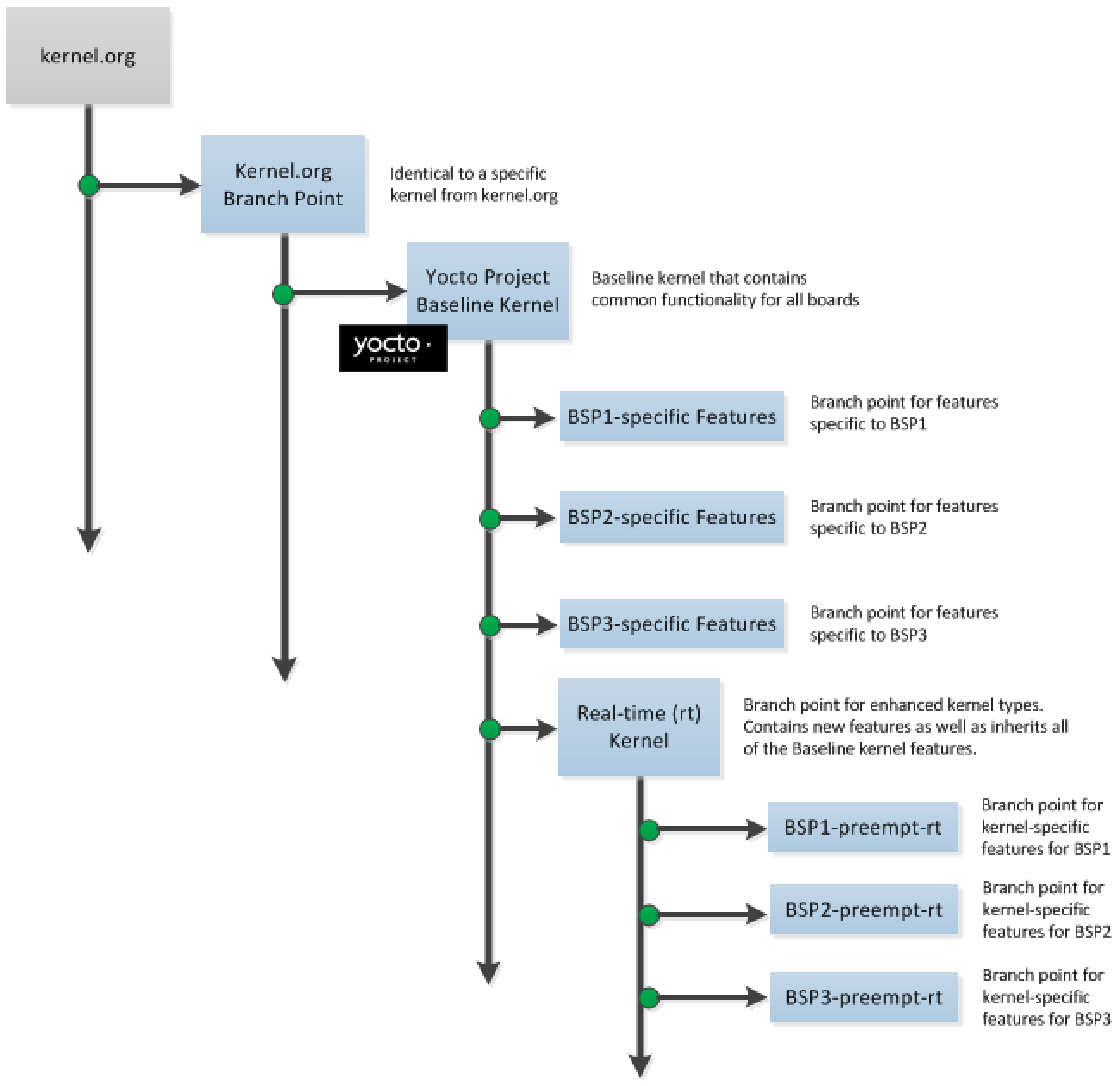
This section describes the architecture of the Yocto Project kernel and provides information on the mechanisms used to achieve that architecture.

2.4.1. Overview

As mentioned earlier, a key goal of Yocto Project is to present the developer with a kernel that has a clear and continuous history that is visible to the user. The architecture and mechanisms used achieve that goal in a manner similar to the upstream kernel.org.

You can think of the Yocto Project kernel as consisting of a baseline kernel with added features logically structured on top of the baseline. The features are tagged and organized by way of a branching strategy implemented by the source code manager (SCM) git. The result is that the user has the ability to see the added features and the commits that make up those features. In addition to being able to see added features, the user can also view the history of what made up the baseline kernel as well.

The following illustration shows the conceptual Yocto Project kernel.


kernel.org
Kernel.org Branch Point
Identical to a specific kernel from kernel.org
Yocto Project Baseline Kernel
yocto PROJECT
Baseline kernel that contains common functionality for all boards
BSP1-specific Features
Branch point for features specific to BSP1
BSP2-specific Features
Branch point for features specific to BSP2
BSP3-specific Features
Branch point for features specific to BSP3
Real-time (rt) Kernel
Branch point for enhanced kernel types. Contains new features as well as inherits all of the Baseline kernel features.
BSP1-preempt-rt
Branch point for kernel-specific features for BSP1
BSP2-preempt-rt
Branch point for kernel-specific features for BSP2
BSP3-preempt-rt
Branch point for kernel-specific features for BSP3

In the illustration, the "kernel.org Branch Point" marks the specific spot (or release) from which the Yocto Project kernel is created. From this point "up" in the tree features and differences are organized and tagged.

The "Yocto Project Baseline Kernel" contains functionality that is common to every kernel type and BSP that is organized further up the tree. Placing these common features in the tree this way means features don't have to be duplicated along individual branches of the structure.

From the Yocto Project Baseline Kernel branch points represent specific functionality for individual BSPs as well as real-time kernels. The illustration represents this through three BSP-specific branches and a real-time kernel branch. Each branch represents some unique functionality for the BSP or a real-time kernel.

In this example structure, the real-time kernel branch has common features for all real-time kernels and contains more branches for individual BSP-specific real-time kernels. The illustration shows three branches as an example. Each branch points the way to specific, unique features for a respective real-time kernel as they apply to a given BSP.

The resulting tree structure presents a clear path of markers (or branches) to the user that for all practical purposes is the kernel needed for any given set of requirements.

2.4.2. Branching Strategy and Workflow

The Yocto Project team creates kernel branches at points where functionality is no longer shared and thus, needs to be isolated. For example, board-specific incompatibilities would require different functionality and would require a branch to separate the features. Likewise, for specific kernel features the same branching strategy is used. This branching strategy results in a tree that has features organized to be specific for particular functionality, single kernel types, or a subset of kernel types. This strategy results in not having to store the same feature twice internally in the tree. Rather we store the unique differences required to apply the feature onto the kernel type in question.

Note

The Yocto Project team strives to place features in the tree such that they can be shared by all boards and kernel types where possible. However, during development cycles or when large features are merged this practice cannot always be followed. In those cases isolated branches are used for feature merging.

BSP-specific code additions are handled in a similar manner to kernel-specific additions. Some BSPs only make sense given certain kernel types. So, for these types, we create branches off the end of that kernel type for all of the BSPs that are supported on that kernel type. From the perspective of the tools that create the BSP branch, the BSP is really no different than a feature. Consequently, the same branching strategy applies to BSPs as it does to features. So again, rather than store the BSP twice, only the unique differences for the BSP across the supported multiple kernels are uniquely stored.

While this strategy can result in a tree with a significant number of branches, it is important to realize that from the user's point of view, there is a linear path that travels from the baseline kernel.org, through a select group of features and ends with their BSP-specific commits. In other words, the divisions of the kernel are transparent and are not relevant to the developer on a day-to-day basis. From the user's perspective, this is the "master" branch. They do not need not be aware of the existence of any other branches at all. Of course there is value in the existence of these branches in the tree, should a person decide to explore them. For example, a comparison between two BSPs at either the commit level or at the line-by-line code diff level is now a trivial operation.

Working with the kernel as a structured tree follows recognized community best practices. In particular, the kernel as shipped with the product should be considered an 'upstream source' and viewed as a series of historical and documented modifications (commits). These modifications represent the development and stabilization done by the Yocto Project kernel development team.

Because commits only change at significant release points in the product life cycle, developers can work on a branch created from the last relevant commit in the shipped Yocto Project kernel. As mentioned previously, the structure is transparent to the user because the kernel tree is left in this state after cloning and building the kernel.

2.4.3. Source Code Manager - git

The Source Code Manager (SCM) is git and it is the obvious mechanism for meeting the previously mentioned goals. Not only is it the SCM for kernel.org but git continues to grow in popularity and supports many different work flows, front-ends and management techniques.

Note

It should be noted that you can use as much, or as little, of what git has to offer as is appropriate to your project.

2.5. Kernel Tools

Since most standard workflows involve moving forward with an existing tree by continuing to add and alter the underlying baseline, the tools that manage Yocto Project's kernel construction are largely hidden from the developer to present a simplified view of the kernel for ease of use.

The fundamental properties of the tools that manage and construct the kernel are:

- the ability to group patches into named, reusable features
- to allow top down control of included features
- the binding of kernel configuration to kernel patches/features
- the presentation of a seamless git repository that blends Yocto Project value with the kernel.org history and development

Chapter 3. Working with the Yocto Project Kernel

3.1. Introduction

This chapter describes how to accomplish tasks involving the kernel's tree structure. The information covers the following:

- Tree construction
- Build strategies
- Workflow examples

3.2. Tree Construction

The Yocto Project kernel repository, as shipped with the product, is created by compiling and executing the set of feature descriptions for every BSP/feature in the product. Those feature descriptions list all necessary patches, configuration, branching, tagging and feature divisions found in the kernel.

You can find the files used to describe all the valid features and BSPs in the Yocto Project kernel in any clone of the kernel git tree. The directory `meta/cfg/kernel-cache/` is a snapshot of all the kernel configuration and feature descriptions (.scc) used to build the kernel repository. You should realize, however, that browsing the snapshot of feature descriptions and patches is not an effective way to determine what is in a particular kernel branch. Instead, you should use git directly to discover the changes in a branch. Using git is a efficient and flexible way to inspect changes to the kernel. For examples showing how to use git to inspect kernel commits, see the following sections in this chapter.

Note

Ground up reconstruction of the complete kernel tree is an action only taken by the Yocto Project team during an active development cycle. Creating a project simply clones this tree to make it efficiently available for building and development.

The general flow for constructing a project-specific kernel tree is as follows:

1. A top-level kernel feature is passed to the kernel build subsystem. Normally, this is a BSP for a particular kernel type.
2. The file that describes the top-level feature is located by searching these system directories:
   - The in-tree kernel-cache directories
   - Recipe SRC_URIs

   For a typical build a feature description of the format: <bsp name>-<kernel type>.scc is the target of the search.
3. Once located, the feature description is either compiled into a simple script of actions, or an existing equivalent script that was part of the shipped kernel is located.
4. Extra features are appended to the top-level feature description. These features can come from the KERNEL_FEATURES variable in recipes.
5. Each extra feature is located, compiled and appended to the script from step #3
6. The script is executed, and a meta-series is produced. The meta-series is a description of all the branches, tags, patches and configuration that needs to be applied to the base git repository to completely create the BSP source (build) branch.
7. The base repository is cloned, and the actions listed in the meta-series are applied to the tree.

8. The git repository is left with the desired branch checked out and any required branching, patching and tagging has been performed.

The tree is now ready for configuration and compilation.

Note

The end-user generated meta-series adds to the kernel as shipped with the Yocto Project release. Any add-ons and configuration data are applied to the end of an existing branch. The full repository generation that is found in the official Yocto Project kernel repositories is the combination of all supported boards and configurations.

This technique is flexible and allows the seamless blending of an immutable history with additional deployment specific patches. Any additions to the kernel become an integrated part of the branches.

3.3. Build Strategy

There are some prerequisites that must be met before starting the compilation phase of the kernel build system:

- There must be a kernel git repository indicated in the SRC_URI.
- There must be a BSP build branch - <bsp name>-<kernel type> in 0.9 or <kernel type>/<bsp name> in 1.0.

You can typically meet these prerequisites by running the tree construction/patching phase of the build system. However, other means do exist. For examples of alternate workflows such as bootstrapping a BSP, see the Workflow Examples section in this manual.

Before building a kernel it is configured by processing all of the configuration "fragments" specified by the scc feature descriptions. As the features are compiled, associated kernel configuration fragments are noted and recorded in the meta-series in their compilation order. The fragments are migrated, pre-processed and passed to the Linux Kernel Configuration subsystem (lkc) as raw input in the form of a `.config` file. The lkc uses its own internal dependency constraints to do the final processing of that information and generates the final `.config` file that is used during compilation.

Using the board's architecture and other relevant values from the board's template the Kernel compilation is started and a kernel image is produced.

The other thing that you will first see once you configure a kernel is that it will generate a build tree that is separate from your git source tree. This build tree has the name using the following form:

```
     linux-<BSPname>-<kerntype>-build
```

"kerntype" is one of the standard kernel types.

The existing support in the kernel.org tree achieves this default functionality.

What this means, is that all the generated files for a particular BSP are now in this directory. The files include the final `.config`, all the `.o` files, the `.a` files, and so forth. Since each BSP has its own separate build directory in its own separate branch of the git tree you can easily switch between different BSP builds.

3.4. Workflow Examples

As previously noted, the Yocto Project kernel has built in git integration. However, these utilities are not the only way to work with the kernel repository. Yocto Project has not made changes to git or to other tools that would invalidate alternate workflows. Additionally, the way the kernel repository is constructed results in using only core git functionality thus allowing any number of tools or front ends to use the resulting tree.

This section contains several workflow examples.

3.4.1. Change Inspection: Kernel Changes/Commits

A common question when working with a BSP or kernel is: "What changes have been applied to this tree?"

In projects that have a collection of directories that contain patches to the kernel it is possible to inspect or "grep" the contents of the directories to get a general feel for the changes. This sort of patch inspection is not an efficient way to determine what has been done to the kernel. The reason it is inefficient is because there are many optional patches that are selected based on the kernel type and the feature description. Additionally, patches could exist in directories that are not included in the search.

A more efficient way to determine what has changed in the kernel is to use git and inspect or search the kernel tree. This method gives you a full view of not only the source code modifications, but also provides the reasons for the changes.

3.4.1.1. What Changed in a BSP?

Following are a few examples that show how to use git to examine changes. Note that because the Yocto Project git repository does not break existing git functionality and because there exists many permutations of these types of commands there are many more methods to discover changes.

Note

Unless you provide a commit range (<kernel-type>..<bsp>-<kernel-type>), kernel.org history is blended with Yocto Project changes.

```
# full description of the changes
> git whatchanged <kernel type>..<kernel type>/<bsp>
   > eg: git whatchanged yocto/standard/base..yocto/standard/common-pc/base

# summary of the changes
> git log --pretty=oneline --abbrev-commit <kernel type>..<kernel type>/<bsp>

# source code changes (one combined diff)
> git diff <kernel type>..<kernel type>/<bsp>
> git show <kernel type>..<kernel type>/<bsp>

# dump individual patches per commit
> git format-patch -o <dir> <kernel type>..<kernel type>/<bsp>

# determine the change history of a particular file
> git whatchanged <path to file>

# determine the commits which touch each line in a file
> git blame <path to file>
```

3.4.1.2. Show a Particular Feature or Branch Change

Significant features or branches are tagged in the Yocto Project tree to divide changes. Remember to first determine (or add) the tag of interest.

Note

Because BSP branch, kernel.org, and feature tags are all present, there are many tags.

```
# show the changes tagged by a feature
> git show <tag>
   > eg: git show yaffs2
```

```
# determine which branches contain a feature
> git branch --contains <tag>

# show the changes in a kernel type
> git whatchanged yocto/base..<kernel type>
   > eg: git whatchanged yocto/base..yocto/standard/base
```

You can use many other comparisons to isolate BSP changes. For example, you can compare against kernel.org tags (e.g. v2.6.27.18, etc), or you can compare against subsystems (e.g. git whatchanged mm).

3.4.2. Development: Saving Kernel Modifications

Another common operation is to build a BSP supplied by Yocto Project, make some changes, rebuild and then test. Those local changes often need to be exported, shared or otherwise maintained.

Since the Yocto Project kernel source tree is backed by git, this activity is much easier as compared to with previous releases. Because git tracks file modifications, additions and deletions, it is easy to modify the code and later realize that the changes should be saved. It is also easy to determine what has changed. This method also provides many tools to commit, undo and export those modifications.

There are many ways to save kernel modifications. The technique employed depends on the destination for the patches:

- Bulk storage
- Internal sharing either through patches or by using git
- External submissions
- Exporting for integration into another SCM

Because of the following list of issues, the destination of the patches also influences the method for gathering them:

- Bisectability
- Commit headers
- Division of subsystems for separate submission or review

3.4.2.1. Bulk Export

This section describes how you can export in "bulk" changes that have not been separated or divided. This situation works well when you are simply storing patches outside of the kernel source repository, either permanently or temporarily, and you are not committing incremental changes during development.

> **Note**
>
> This technique is not appropriate for full integration of upstream submission because changes are not properly divided and do not provide an avenue for per-change commit messages. Therefore, this example assumes that changes have not been committed incrementally during development and that you simply must gather and export them.

```
# bulk export of ALL modifications without separation or division
# of the changes

> git add .
> git commit -s -a -m >commit message<
   or
> git commit -s -a # and interact with $EDITOR
```

The previous operations capture all the local changes in the project source tree in a single git commit. And, that commit is also stored in the project's source tree.

Once the changes are exported, you can restore them manually using a template or through integration with the `default_kernel`.

3.4.2.2. Incremental/Planned Sharing

This section describes how to save modifications when you are making incremental commits or practicing planned sharing. The examples in this section assume that changes have been incrementally committed to the tree during development and now need to be exported. The sections that follow describe how you can export your changes internally through either patches or by using git commands.

During development the following commands are of interest. For full git documentation, refer to the git man pages or to an online resource such as http://github.com.

```
# edit a file
> vi >path</file
# stage the change
> git add >path</file
# commit the change
> git commit -s
# remove a file
> git rm >path</file
# commit the change
> git commit -s

... etc.
```

Distributed development with git is possible when you use a universally agreed-upon unique commit identifier (set by the creator of the commit) that maps to a specific change set with a specific parent. This identifier is created for you when you create a commit, and is re-created when you amend, alter or re-apply a commit. As an individual in isolation, this is of no interest. However, if you intend to share your tree with normal git push and pull operations for distributed development, you should consider the ramifications of changing a commit that you have already shared with others.

Assuming that the changes have not been pushed upstream, or pulled into another repository, you can update both the commit content and commit messages associated with development by using the following commands:

```
> git add >path</file
> git commit --amend
> git rebase or git rebase -i
```

Again, assuming that the changes have not been pushed upstream, and that no pending works-in-progress exist (use "git status" to check) then you can revert (undo) commits by using the following commands:

```
# remove the commit, update working tree and remove all
# traces of the change
> git reset --hard HEAD^
# remove the commit, but leave the files changed and staged for re-commit
> git reset --soft HEAD^
# remove the commit, leave file change, but not staged for commit
> git reset --mixed HEAD^
```

You can create branches, "cherry-pick" changes or perform any number of git operations until the commits are in good order for pushing upstream or for pull requests. After a push or pull, commits are normally considered "permanent" and you should not modify them. If they need to be changed you can incrementally do so with new commits. These practices follow the standard "git" workflow and the kernel.org best practices, which Yocto Project recommends.

Note

It is recommended to tag or branch before adding changes to a Yocto Project BSP or before creating a new one. The reason for this recommendation is because the branch or tag provides a reference point to facilitate locating and exporting local changes.

3.4.2.2.1. Exporting Changes Internally by Using Patches

This section describes how you can extract committed changes from a working directory by exporting them as patches. Once extracted, you can use the patches for upstream submission, place them in a Yocto Project template for automatic kernel patching, or apply them in many other common uses.

This example shows how to create a directory with sequentially numbered patches. Once the directory is created, you can apply it to a repository using the `git am` command to reproduce the original commit and all the related information such as author, date, commit log, and so forth.

Note

The new commit identifiers (ID) will be generated upon re-application. This action reflects that the commit is now applied to an underlying commit with a different ID.

```
# <first-commit> can be a tag if one was created before development
# began. It can also be the parent branch if a branch was created
# before development began.

> git format-patch -o <dir> <first commit>..<last commit>
```

In other words:

```
# Identify commits of interest.

# If the tree was tagged before development
> git format-patch -o <save dir> <tag>

# If no tags are available
> git format-patch -o <save dir> HEAD^  # last commit
> git format-patch -o <save dir> HEAD^^ # last 2 commits
> git whatchanged # identify last commit
> git format-patch -o <save dir> <commit id>
> git format-patch -o <save dir> <rev-list>
```

3.4.2.2.2. Exporting Changes Internally by Using git

This section describes how you can export changes from a working directory by pushing the changes into a master repository or by making a pull request. Once you have pushed the changes in the master repository you can then pull those same changes into a new kernel build at a later time.

Use this command form to push the changes:

```
git push ssh://<master server>/<path to repo> <local branch>:<remote branch>
```

For example, the following command pushes the changes from your local branch `yocto/standard/common-pc/base` to the remote branch with the same name in the master repository `//git.mycompany.com/pub/git/kernel-2.6.37`.

```
     > push ssh://git.mycompany.com/pub/git/kernel-2.6.37 yocto/standard/common-pc/base:yocto/sta
```

A pull request entails using "git request-pull" to compose an email to the maintainer requesting that a branch be pulled into the master repository, see http://github.com/guides/pull-requests for an example.

Note

Other commands such as 'git stash' or branching can also be used to save changes, but are not covered in this document.

3.4.2.3. Exporting Changes for External (Upstream) Submission

This section describes how to export changes for external upstream submission. If the patch series is large or the maintainer prefers to pull changes, you can submit these changes by using a pull request. However, it is common to sent patches as an email series. This method allows easy review and integration of the changes.

Note

Before sending patches for review be sure you understand the community standards for submitting and documenting changes and follow their best practices. For example, kernel patches should follow standards such as:

- http://userweb.kernel.org/~akpm/stuff/tpp.txt
- http://linux.yyz.us/patch-format.html
- Documentation/SubmittingPatches (in any linux kernel source tree)

The messages used to commit changes are a large part of these standards. Consequently, be sure that the headers for each commit have the required information. If the initial commits were not properly documented or do not meet those standards, you can re-base by using the "git rebase -i" command to manipulate the commits and get them into the required format. Other techniques such as branching and cherry-picking commits are also viable options.

Once you complete the commits, you can generate the email that sends the patches to the maintainer(s) or lists that review and integrate changes. The command "git send-email" is commonly used to ensure that patches are properly formatted for easy application and avoid mailer-induced patch damage.

The following is an example of dumping patches for external submission:

```
     # dump the last 4 commits
     > git format-patch --thread -n -o ~/rr/ HEAD^^^^
     > git send-email --compose --subject '[RFC 0/N] <patch series summary>' \
      --to foo@yoctoproject.org --to bar@yoctoproject.org \
      --cc list@yoctoproject.org  ~/rr
     # the editor is invoked for the 0/N patch, and when complete the entire
     # series is sent via email for review
```

3.4.2.4. Exporting Changes for Import into Another SCM

When you want to export changes for import into another Source Code Manager (SCM) you can use any of the previously discussed techniques. However, if the patches are manually applied to a secondary tree and then that tree is checked into the SCM you can lose change information such as commit logs. Yocto Project does not recommend this process.

Many SCMs can directly import git commits, or can translate git patches so that information is not lost. Those facilities are SCM-dependent and you should use them whenever possible.

3.4.3. Working with the Yocto Project Kernel in Another SCM

This section describes kernel development in another SCM, which is not the same as exporting changes to another SCM. For this scenario you use the Yocto Project build system to develop the kernel in a different SCM. The following must be true for you to accomplish this:

- The delivered Yocto Project kernel must be exported into the second SCM.
- Development must be exported from that secondary SCM into a format that can be used by the Yocto Project build system.

3.4.3.1. Exporting the Delivered Kernel to the SCM

Depending on the SCM it might be possible to export the entire Yocto Project kernel git repository, branches and all, into a new environment. This method is preferred because it has the most flexibility and potential to maintain the meta data associated with each commit.

When a direct import mechanism is not available, it is still possible to export a branch (or series of branches) and check them into a new repository.

The following commands illustrate some of the steps you could use to import the yocto/standard/common-pc/base kernel into a secondary SCM:

```
     > git checkout yocto/standard/common-pc/base
     > cd .. ; echo linux/.git > .cvsignore
     > cvs import -m "initial import" linux MY_COMPANY start
```

You could now relocate the CVS repository and use it in a centralized manner.

The following commands illustrate how you can condense and merge two BSPs into a second SCM:

```
     > git checkout yocto/standard/common-pc/base
     > git merge yocto/standard/common-pc-64/base
     # resolve any conflicts and commit them
     > cd .. ; echo linux/.git > .cvsignore
     > cvs import -m "initial import" linux MY_COMPANY start
```

3.4.3.2. Importing Changes for the Build

Once development has reached a suitable point in the second development environment, you need to export the changes as patches. To export them place the changes in a recipe and automatically apply them to the kernel during patching.

3.4.4. Creating a BSP Based on an Existing Similar BSP

This section provides an example for creating a BSP that is based on an existing, and hopefully, similar one. It assumes you will be using a local kernel repository and will be pointing the kernel recipe at that. Follow these steps and keep in mind your particular situation and differences:

1. Identify a machine configuration file that matches your machine.

   You can start with something in `meta/conf/machine` - `meta/conf/machine/atom-pc.conf` for example. Or, you can start with a machine configuration from any of the BSP layers in the meta-intel repository at http://git.yoctoproject.org/cgit/cgit.cgi/meta-intel/, such as `meta-intel/meta-emenlow/conf/machine/emenlow.conf`.

   The main difference between the two is that "emenlow" is in its own layer. It is in its own layer because it needs extra machine-specific packages such as its own video driver and other supporting packages. The "atom-pc" is simpler and does not need any special packages - everything it needs can be specified in the configuration file. The "atom-pc" machine also supports all of Asus eee901, Acer Aspire One, Toshiba NB305, and the Intel® Embedded Development Board 1-N450 with no changes.

If you want to make minor changes to support a slightly different machine, you can create a new configuration file for it and add it alongside the others. You might consider keeping the common information separate and including it.

Similarly, you can also use multiple configuration files for different machines even if you do it as a separate layer like meta-emenlow.

As an example consider this:

- Copy meta-emenlow to meta-mymachine
- Fix or remove anything you do not need. For this example the only thing left was the kernel directory with a `linux-yocto_git.bbappend` file and `meta-mymachine/conf/machine/mymachine.conf` (linux-yocto is the kernel listed in `meta-emenlow/conf/machine/emenlow.conf`)
- Add a new entry in the `build/conf/bblayers.conf` so the new layer can be found by BitBake.

2. Create a machine branch for your machine.

   For the kernel to compile successfully, you need to create a branch in the git repository specifically named for your machine. To create this branch first create a bare clone of the Yocto Project git repository. Next, create a local clone of that:

```
$ git clone --bare git://git.yoctoproject.org/linux-yocto-2.6.37.git
linux-yocto-2.6.37.git
$ git clone linux-yocto-2.6.37.git linux-yocto-2.6.37
```

   Now create a branch in the local clone and push it to the bare clone:

```
$ git checkout -b yocto/standard/mymachine origin/yocto/standard/base
$ git push origin yocto/standard/mymachine:yocto/standard/mymachine
```

3. In a layer, create a `linux-yocto_git.bbappend` file with the following:

```
FILESEXTRAPATHS := "${THISDIR}/${PN}"
COMPATIBLE_MACHINE_mymachine = "mymachine"

# It is often nice to have a local clone of the kernel repository, to
# allow patches to be staged, branches created, and so forth.  Modify
# KSRC to point to your local clone as appropriate.

KSRC ?= /path/to/your/bare/clone/for/example/linux-yocto-2.6.37.git

# KMACHINE is the branch to be built, or alternatively
# KBRANCH can be directly set.
# KBRANCH is set to KMACHINE in the main linux-yocto_git.bb
# KBRANCH ?= "${LINUX_KERNEL_TYPE}/${KMACHINE}"

KMACHINE_mymachine  = "yocto/standard/mymachine"

SRC_URI = "git://${KSRC};nocheckout=1;branch=${KBRANCH},meta;name=machine,meta"
```

   After doing that, select the machine in `build/conf/local.conf`:

```
 #
```

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

You should now be able to build and boot an image with the new kernel:

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

4. Modify the kernel configuration for your machine.

   Of course, that will give you a kernel with the default configuration file, which is probably not what you want. If you just want to set some kernel configuration options, you can do that by putting them in a file. For example, inserting the following into some `.cfg` file:

   ```
        CONFIG_NETDEV_1000=y
        CONFIG_E1000E=y
   ```

   And, another `.cfg` file would contain:

   ```
        CONFIG_LOG_BUF_SHIFT=18
   ```

   These config fragments could then be picked up and applied to the kernel .config by appending them to the kernel SRC_URI:

   ```
        SRC_URI_append_mymachine = " file://some.cfg \
                                     file://other.cfg \
                                   "
   ```

   You could also add these directly to the git repository `meta` branch as well. However, the former method is a simple starting point.

5. If you're also adding patches to the kernel, you can do the same thing. Put your patches in the SRC_URI as well (plus `.cfg` for their kernel configuration options if needed).

   Practically speaking, to generate the patches, you'd go to the source in the build tree:

   ```
        build/tmp/work/mymachine-poky-linux/linux-yocto-2.6.37+git0+d1cd5c80ee97e81e130be8c3de3965
   0431115c9d720fee5bb105f6a7411efb4f851d26-r13/linux
   ```

   Then, modify the code there, using quilt to save the changes, and recompile until it works:

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

6. Once you have the final patch from quilt, copy it to the SRC_URI location. The patch is applied the next time you do a clean build. Of course, since you have a branch for the BSP in git, it would be better to put it there instead. For example, in this case, commit the patch to the "yocto/standard/mymachine" branch, and during the next build it is applied from there.

3.4.5. Creating a BSP Based on an Existing Similar BSP Without a Local Kernel Repository

If you are creating a BSP based on an existing similar BSP but you do not have a local kernel repository, the process is very similar to the process in the previous section (Section 3.4.4, “Creating a BSP Based on an Existing Similar BSP”).

Follow the exact same process as described in the previous section with these slight modifications:

1. Perform Step 1 as is from the previous section.
2. Perform Step 2 as is from the previous section.
3. Perform Step 3 but do not modify the KSRC line in the bbappend file.
4. Edit the `local.conf` so that it contains the following:

   ```
        YOCTO_KERNEL_EXTERNAL_BRANCH="<your-machine>-standard
   ```

   Adding this statement to the file triggers BSP bootstrapping to occur and the correct branches and base configuration to be used.
5. Perform Step 4 as is from the previous section.
6. Perform Step 5 as is from the previous section.

3.4.6. "-dirty" String

If kernel images are being built with "-dirty" on the end of the version string, this simply means that modifications in the source directory have not been committed.

```
     > git status
```

You can use the git command above to report modified, removed, or added files. You should commit those changes to the tree regardless of whether they will be saved, exported, or used. Once you commit the changes you need to rebuild the kernel.

To brute force pickup and commit all such pending changes enter the following:

```
     > git add .
     > git commit -s -a -m "getting rid of -dirty"
```

Next, rebuild the kernel.