Yocto Project Linux Kernel Development Manual

Darren Hart

Intel Corporation
<darren.hart@intel.com>

Note
For the latest version of this manual associated with this Yocto Project release, see the Yocto Project Linux Kernel Development Manual from the Yocto Project website.

Revision History	
Revision 1.4	April 2013
Released with the Yocto Project 1.4 Release.	
Revision 1.5	October 2013
Released with the Yocto Project 1.5 Release.	
Revision 1.5.1	January 2014
Released with the Yocto Project 1.5.1 Release.	
Revision 1.6	April 2014
Released with the Yocto Project 1.6 Release.	
Revision 1.7	October 2014
Released with the Yocto Project 1.7 Release.	
Revision 1.8	April 2015
Released with the Yocto Project 1.8 Release.	
Revision 2.0	October 2015
Released with the Yocto Project 2.0 Release.	
Revision 2.0.1	March 2016
Released with the Yocto Project 2.0.1 Release.	

Table of Contents

Chapter 1. Introduction¶

Table of Contents

1.1. Overview¶

Regardless of how you intend to make use of the Yocto Project, chances are you will work with the Linux kernel. This manual provides background information on the Yocto Linux kernel Metadata, describes common tasks you can perform using the kernel tools, and shows you how to use the kernel Metadata needed to work with the kernel inside the Yocto Project.

Each Yocto Project release has a set of linux-yocto recipes, whose Git repositories you can view in the Yocto Source Repositories under the "Yocto Linux Kernel" heading. New recipes for the release track the latest upstream developments and introduce newly-supported platforms. Previous recipes in the release are refreshed and supported for at least one additional release. As they align, these previous releases are updated to include the latest from the Long Term Support Initiative (LTSI) project. Also included is a linux-yocto development recipe (`linux-yocto-dev.bb`) should you want to work with the very latest in upstream Linux kernel development and kernel Metadata development.

The Yocto Project also provides a powerful set of kernel tools for managing Linux kernel sources and configuration data. You can use these tools to make a single configuration change, apply multiple patches, or work with your own kernel sources.

In particular, the kernel tools allow you to generate configuration fragments that specify only what you must, and nothing more. Configuration fragments only need to contain the highest level visible `CONFIG` options as presented by the Linux kernel `menuconfig` system. Contrast this against a complete Linux kernel `.config`, which includes all the automatically selected `CONFIG` options. This efficiency reduces your maintenance effort and allows you to further separate your configuration in ways that make sense for your project. A common split separates policy and hardware. For example, all your kernels might support the `proc` and `sys` filesystems, but only specific boards require sound, USB, or specific drivers. Specifying these configurations individually allows you to aggregate them together as needed, but maintains them in only one place. Similar logic applies to separating source changes.

If you do not maintain your own kernel sources and need to make only minimal changes to the sources, the released recipes provide a vetted base upon which to layer your changes. Doing so allows you to benefit from the continual kernel integration and testing performed during development of the Yocto Project.

If, instead, you have a very specific Linux kernel source tree and are unable to align with one of the official linux-yocto recipes, an alternative exists by which you can use the Yocto Project Linux kernel tools with your own kernel sources.

1.2. Other Resources¶

The sections that follow provide instructions for completing specific Linux kernel development tasks. These instructions assume you are comfortable working with BitBake recipes and basic open-source development tools. Understanding these concepts will facilitate the process of working with the kernel recipes. If you find you need some additional background, please be sure to review and understand the following documentation:

- Yocto Project Quick Start
- The "Modifying Source Code" section in the Yocto Project Development Manual
- The "Understanding and Creating Layers" section in the Yocto Project Development Manual
- The "Modifying the Kernel" section in the Yocto Project Development Manual.

Finally, while this document focuses on the manual creation of recipes, patches, and configuration files, the Yocto Project Board Support Package (BSP) tools are available to automate this process with existing content and work well to create the initial framework and boilerplate code. For details on these tools, see the "Using the Yocto Project's BSP Tools" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

Chapter 2. Common Tasks¶

Table of Contents

This chapter presents several common tasks you perform when you work with the Yocto Project Linux kernel. These tasks include preparing a layer, modifying an existing recipe, iterative development, working with your own sources, and incorporating out-of-tree modules.

Note
The examples presented in this chapter work with the Yocto Project 1.2.2 Release and forward.

2.1. Creating and Preparing a Layer¶

If you are going to be modifying kernel recipes, it is recommended that you create and prepare your own layer in which to do your work. Your layer contains its own BitBake append files (`.bbappend`) and provides a convenient mechanism to create your own recipe files (`.bb`). For details on how to create and work with layers, see the following sections in the Yocto Project Development Manual:

- "Understanding and Creating Layers" for general information on layers and how to create layers.
- "Set Up Your Layer for the Build" for specific instructions on setting up a layer for kernel development.

2.2. Modifying an Existing Recipe¶

In many cases, you can customize an existing linux-yocto recipe to meet the needs of your project. Each release of the Yocto Project provides a few Linux kernel recipes from which you can choose. These are located in the Source Directory in `meta/recipes-kernel/linux`.

Modifying an existing recipe can consist of the following:

- Creating the append file
- Applying patches
- Changing the configuration

Before modifying an existing recipe, be sure that you have created a minimal, custom layer from which you can work. See the "Creating and Preparing a Layer" section for some general resources. You can also see the "Set Up Your Layer for the Build" section of the Yocto Project Development Manual for a detailed example.

2.2.1. Creating the Append File¶

You create this file in your custom layer. You also name it accordingly based on the linux-yocto recipe you are using. For example, if you are modifying the `meta/recipes-kernel/linux/linux-yocto_3.19.bb` recipe, the append file will typically be located as follows within your custom layer:

```
     your-layer/recipes-kernel/linux/linux-yocto_3.19.bbappend
```

The append file should initially extend the `FILESPATH` search path by prepending the directory that contains your files to the `FILESEXTRAPATHS` variable as follows:

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

The path `${THISDIR}/${PN}` expands to "linux-yocto" in the current directory for this example. If you add any new files that modify the kernel recipe and you have extended `FILESPATH` as described above, you must place the files in your layer in the following area:

```
your-layer/recipes-kernel/linux/linux-yocto/
```

> **Note**
> If you are working on a new machine Board Support Package (BSP), be sure to refer to the Yocto Project Board Support Package (BSP) Developer's Guide.

2.2.2. Applying Patches¶

If you have a single patch or a small series of patches that you want to apply to the Linux kernel source, you can do so just as you would with any other recipe. You first copy the patches to the path added to `FILESEXTRAPATHS` in your `.bbappend` file as described in the previous section, and then reference them in `SRC_URI` statements.

For example, you can apply a three-patch series by adding the following lines to your linux-yocto `.bbappend` file in your layer:

```
SRC_URI += "file://0001-first-change.patch"
SRC_URI += "file://0002-second-change.patch"
SRC_URI += "file://0003-third-change.patch"
```

The next time you run BitBake to build the Linux kernel, BitBake detects the change in the recipe and fetches and applies the patches before building the kernel.

For a detailed example showing how to patch the kernel, see the "Patching the Kernel" section in the Yocto Project Development Manual.

2.2.3. Changing the Configuration¶

You can make wholesale or incremental changes to the final `.config` file used for the eventual Linux kernel configuration by including a `defconfig` file and by specifying configuration fragments in the `SRC_URI` to be applied to that file.

If you have a complete, working Linux kernel `.config` file you want to use for the configuration, as before, copy that file to the appropriate `${PN}` directory in your layer's `recipes-kernel/linux` directory, and rename the copied file to "defconfig". Then, add the following lines to the linux-yocto `.bbappend` file in your layer:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
```

The `SRC_URI` tells the build system how to search for the file, while the `FILESEXTRAPATHS` extends the `FILESPATH` variable (search directories) to include the `${PN}` directory you created to hold the configuration changes.

> **Note**
> The build system applies the configurations from the `defconfig` file before applying any subsequent configuration fragments. The final kernel configuration is a combination of the configurations in the `defconfig` file and any configuration fragments you provide. You need to realize that if you have any configuration fragments, the build system applies these on top of and after applying the existing `defconfig` file configurations.

Generally speaking, the preferred approach is to determine the incremental change you want to make and add that as a configuration fragment. For example, if you want to add support for a basic serial console, create a file named `8250.cfg` in the `${PN}` directory with the following content (without indentation):

```
     CONFIG_SERIAL_8250=y
     CONFIG_SERIAL_8250_CONSOLE=y
     CONFIG_SERIAL_8250_PCI=y
     CONFIG_SERIAL_8250_NR_UARTS=4
     CONFIG_SERIAL_8250_RUNTIME_UARTS=4
     CONFIG_SERIAL_CORE=y
     CONFIG_SERIAL_CORE_CONSOLE=y
```

Next, include this configuration fragment and extend the `FILESPATH` variable in your `.bbappend` file:

```
     FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
     SRC_URI += "file://8250.cfg"
```

The next time you run BitBake to build the Linux kernel, BitBake detects the change in the recipe and fetches and applies the new configuration before building the kernel.

For a detailed example showing how to configure the kernel, see the "Configuring the Kernel" section in the Yocto Project Development Manual.

2.2.4. Using an "In-Tree" `defconfig` File¶

It might be desirable to have kernel configuration fragment support through a `defconfig` file that is pulled from the kernel source tree for the configured machine. By default, the OpenEmbedded build system looks for `defconfig` files in the layer used for Metadata, which is "out-of-tree", and then configures them using the following:

```
     SRC_URI += "file://defconfig"
```

If you do not want to maintain copies of `defconfig` files in your layer but would rather allow users to use the default configuration from the kernel tree and still be able to add configuration fragments to the `SRC_URI` through, for example, append files, you can direct the OpenEmbedded build system to use a `defconfig` file that is "in-tree".

To specify an "in-tree" `defconfig` file, edit the recipe that builds your kernel so that it has the following command form:

```
     KBUILD_DEFCONFIG_KMACHINE ?= defconfig_file
```

You need to append the variable with `KMACHINE` and then supply the path to your "in-tree" `defconfig` file.

Aside from modifying your kernel recipe and providing your own `defconfig` file, you need to be sure no files or statements set `SRC_URI` to use a `defconfig` other than your "in-tree" file (e.g. a kernel's `linux-`*`machine`*`.inc` file). In other words, if the build system detects a statement that identifies an "out-of-tree" `defconfig` file, that statement will override your `KBUILD_DEFCONFIG` variable.

See the `KBUILD_DEFCONFIG` variable description for more information.

2.3. Using an Iterative Development Process¶

If you do not have existing patches or configuration files, you can iteratively generate them from within the BitBake build environment as described within this section. During an iterative workflow, running a previously completed BitBake task causes BitBake to invalidate the tasks that follow the completed task in the build sequence. Invalidated tasks rebuild the next time you run the build using BitBake.

As you read this section, be sure to substitute the name of your Linux kernel recipe for the term "linux-yocto".

2.3.1. "-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 above Git command 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 force a pickup and commit of all such pending changes, enter the following:

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

Next, rebuild the kernel.

2.3.2. Generating Configuration Files¶

You can manipulate the `.config` file used to build a linux-yocto recipe with the `menuconfig` command as follows:

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

This command starts the Linux kernel configuration tool, which allows you to prepare a new `.config` file for the build. When you exit the tool, be sure to save your changes at the prompt.

The resulting `.config` file is located in `${WORKDIR}` under the `linux-${PACKAGE_ARCH}-${LINUX_KERNEL_TYPE}-build` directory. You can use the entire `.config` file as the `defconfig` file as described in the "Changing the Configuration" section. For more information on the `.config` file, see the "Using `menuconfig`" section in the Yocto Project Development Manual.

A better method is to create a configuration fragment using the differences between two configuration files: one previously created and saved, and one freshly created using the `menuconfig` tool.

To create a configuration fragment using this method, follow these steps:

1. Complete a build at least through the kernel configuration task as follows:

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

This step ensures that you will be creating a `.config` file from a known state. Because situations exist where your build state might become unknown, it is best to run the previous command prior to starting up `menuconfig`.

2. Run the `menuconfig` command:

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

3. Run the `diffconfig` command to prepare a configuration fragment. The resulting file `fragment.cfg` will be placed in the `${WORKDIR}` directory:

   ```
   $ bitbake linux-yocto -c diffconfig
   ```

The `diffconfig` command creates a file that is a list of Linux kernel `CONFIG_` assignments. See the "Changing the Configuration" section for information on how to use the output as a configuration fragment.

Note
You can also use this method to create configuration fragments for a BSP. See the "BSP Descriptions" section for more information.

The kernel tools also provide configuration validation. You can use these tools to produce warnings for when a requested configuration does not appear in the final `.config` file or when you override a policy configuration in a hardware configuration fragment. Here is an example with some sample output of the command that runs these tools:

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

...

NOTE: validating kernel configuration
This BSP sets 3 invalid/obsolete kernel options.
These config options are not offered anywhere within this kernel.
The full list can be found in your kernel src dir at:
meta/cfg/standard/mybsp/invalid.cfg

This BSP sets 21 kernel options that are possibly non-hardware related.
The full list can be found in your kernel src dir at:
meta/cfg/standard/mybsp/specified_non_hdw.cfg

WARNING: There were 2 hardware options requested that do not
         have a corresponding value present in the final ".config" file.
         This probably means you are not getting the config you wanted.
         The full list can be found in your kernel src dir at:
         meta/cfg/standard/mybsp/mismatch.cfg
```

The output describes the various problems that you can encounter along with where to find the offending configuration items. You can use the information in the logs to adjust your configuration files and then repeat the `kernel_configme` and `kernel_configcheck` commands until they produce no warnings.

For more information on how to use the `menuconfig` tool, see the "Using `menuconfig`" section in the Yocto Project Development Manual.

2.3.3. Modifying Source Code¶

You can experiment with source code changes and create a simple patch without leaving the BitBake environment. To get started, be sure to complete a build at least through the kernel configuration task:

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

Taking this step ensures you have the sources prepared and the configuration completed. You can find the sources in the `${WORKDIR}/linux` directory.

You can edit the sources as you would any other Linux source tree. However, keep in mind that you will lose changes if you trigger the `do_fetch` task for the recipe. You can avoid triggering this task by not using BitBake to run the `cleanall`, `cleansstate`, or forced `fetch` commands. Also, do not modify the recipe itself while working with temporary changes or BitBake might run the `fetch` command depending on the changes to the recipe.

To test your temporary changes, instruct BitBake to run the `compile` again. The `-f` option forces the command to run even though BitBake might think it has already done so:

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

If the compile fails, you can update the sources and repeat the `compile`. Once compilation is successful, you can inspect and test the resulting build (i.e. kernel, modules, and so forth) from the following build directory:

```
     ${WORKDIR}/linux-${PACKAGE_ARCH}-${LINUX_KERNEL_TYPE}-build
```

Alternatively, you can run the `deploy` command to place the kernel image in the `tmp/deploy/images` directory:

```
 $ bitbake linux-yocto -c deploy
```

And, of course, you can perform the remaining installation and packaging steps by issuing:

```
 $ bitbake linux-yocto
```

For rapid iterative development, the edit-compile-repeat loop described in this section is preferable to rebuilding the entire recipe because the installation and packaging tasks are very time consuming.

Once you are satisfied with your source code modifications, you can make them permanent by generating patches and applying them to the `SRC_URI` statement as described in the "Applying Patches" section. If you are not familiar with generating patches, refer to the "Creating the Patch" section in the Yocto Project Development Manual.

2.4. Working With Your Own Sources¶

If you cannot work with one of the Linux kernel versions supported by existing linux-yocto recipes, you can still make use of the Yocto Project Linux kernel tooling by working with your own sources. When you use your own sources, you will not be able to leverage the existing kernel Metadata and stabilization work of the linux-yocto sources. However, you will be able to manage your own Metadata in the same format as the linux-yocto sources. Maintaining format compatibility facilitates converging with linux-yocto on a future, mutually-supported kernel version.

To help you use your own sources, the Yocto Project provides a linux-yocto custom recipe (`linux-yocto-custom.bb`) that uses `kernel.org` sources and the Yocto Project Linux kernel tools for managing kernel Metadata. You can find this recipe in the `poky` Git repository of the Yocto Project Source Repository at:

```
poky/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb
```

Here are some basic steps you can use to work with your own sources:

1. Copy the `linux-yocto-custom.bb` recipe to your layer and give it a meaningful name. The name should include the version of the Linux kernel you are using (e.g. `linux-yocto-myproject_3.19.bb`, where "3.19" is the base version of the Linux kernel with which you would be working).

2. In the same directory inside your layer, create a matching directory to store your patches and configuration files (e.g. `linux-yocto-myproject`).

3. Make sure you have either a `defconfig` file or configuration fragment files. When you use the `linux-yocto-custom.bb` recipe, you must specify a configuration. If you do not have a `defconfig` file, you can run the following:

   ```
   $ make defconfig
   ```

   After running the command, copy the resulting `.config` to the `files` directory as "defconfig" and then add it to the `SRC_URI` variable in the recipe.

   Running the `make defconfig` command results in the default configuration for your architecture as defined by your kernel. However, no guarantee exists that this configuration is valid for your use case, or that your board will even boot. This is particularly true for non-x86 architectures. To use non-x86 `defconfig` files, you need to be more specific and find one that matches your board (i.e. for arm, you look in `arch/arm/configs` and use the one that is the best starting point for your board).

4. Edit the following variables in your recipe as appropriate for your project:

   - `SRC_URI`: The `SRC_URI` should specify a Git repository that uses one of the supported Git fetcher protocols (i.e. `file`, `git`, `http`, and so forth). The `SRC_URI` variable should also specify either a `defconfig` file or some configuration fragment files. The skeleton recipe provides an example `SRC_URI` as a syntax reference.

   - `LINUX_VERSION`: The Linux kernel version you are using (e.g. "3.19").

   - `LINUX_VERSION_EXTENSION`: The Linux kernel `CONFIG_LOCALVERSION` that is compiled into the resulting kernel and visible through the `uname` command.

   - `SRCREV`: The commit ID from which you want to build.

   - `PR`: Treat this variable the same as you would in any other recipe. Increment the variable to indicate to the OpenEmbedded build system that the recipe has changed.

   - `PV`: The default `PV` assignment is typically adequate. It combines the `LINUX_VERSION` with the Source Control Manager (SCM) revision as derived from the SRCPV variable. The combined results are a string with the following form:

```
3.19.11+git1+68a635bf8dfb64b02263c1ac80c948647cc76d5f_1+218bd8d2022b9852c60d
```

While lengthy, the extra verbosity in `PV` helps ensure you are using the exact sources from which you intend to build.

- COMPATIBLE_MACHINE: A list of the machines supported by your new recipe. This variable in the example recipe is set by default to a regular expression that matches only the empty string, "(^$)". This default setting triggers an explicit build failure. You must change it to match a list of the machines that your new recipe supports. For example, to support the `qemux86` and `qemux86-64` machines, use the following form:

```
COMPATIBLE_MACHINE = "qemux86|qemux86-64"
```

5. Provide further customizations to your recipe as needed just as you would customize an existing linux-yocto recipe. See the "Modifying an Existing Recipe" section for information.

2.5. Working with Out-of-Tree Modules¶

This section describes steps to build out-of-tree modules on your target and describes how to incorporate out-of-tree modules in the build.

2.5.1. Building Out-of-Tree Modules on the Target¶

While the traditional Yocto Project development model would be to include kernel modules as part of the normal build process, you might find it useful to build modules on the target. This could be the case if your target system is capable and powerful enough to handle the necessary compilation. Before deciding to build on your target, however, you should consider the benefits of using a proper cross-development environment from your build host.

If you want to be able to build out-of-tree modules on the target, there are some steps you need to take on the target that is running your SDK image. Briefly, the `kernel-dev` package is installed by default on all `*.sdk` images and the `kernel-devsrc` package is installed on many of the `*.sdk` images. However, you need to create some scripts prior to attempting to build the out-of-tree modules on the target that is running that image.

Prior to attempting to build the out-of-tree modules, you need to be on the target as root and you need to change to the `/usr/src/kernel` directory. Next, `make` the scripts:

```
# cd /usr/src/kernel
# make scripts
```

Because all SDK image recipes include `dev-pkgs`, the `kernel-dev` packages will be installed as part of the SDK image and the `kernel-devsrc` packages will be installed as part of applicable SDK images. The SDK uses the scripts when building out-of-tree modules. Once you have switched to that directory and created the scripts, you should be able to build your out-of-tree modules on the target.

2.5.2. Incorporating Out-of-Tree Modules

While it is always preferable to work with sources integrated into the Linux kernel sources, if you need an external kernel module, the `hello-mod.bb` recipe is available as a template from which you can create your own out-of-tree Linux kernel module recipe.

This template recipe is located in the `poky` Git repository of the Yocto Project Source Repository at:

```
poky/meta-skeleton/recipes-kernel/hello-mod/hello-mod_0.1.bb
```

To get started, copy this recipe to your layer and give it a meaningful name (e.g. `mymodule_1.0.bb`). In the same directory, create a new directory named `files` where you can store any source files, patches, or other files necessary for building the module that do not come with the sources. Finally, update the recipe as needed for the module. Typically, you will need to set the following variables:

- `DESCRIPTION`
- `LICENSE*`
- `SRC_URI`
- `PV`

Depending on the build system used by the module sources, you might need to make some adjustments. For example, a typical module `Makefile` looks much like the one provided with the `hello-mod` template:

```
obj-m := hello.o

SRC := $(shell pwd)

all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
...
```

The important point to note here is the `KERNEL_SRC` variable. The `module` class sets this variable and the `KERNEL_PATH` variable to `${STAGING_KERNEL_DIR}` with the necessary Linux kernel build information to build modules. If your module `Makefile` uses a different variable, you might want to override the `do_compile()` step, or create a patch to the `Makefile` to work with the more typical `KERNEL_SRC` or `KERNEL_PATH` variables.

After you have prepared your recipe, you will likely want to include the module in your images. To do this, see the documentation for the following variables in the Yocto Project Reference Manual and set one of them appropriately for your machine configuration file:

- `MACHINE_ESSENTIAL_EXTRA_RDEPENDS`
- `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS`

- MACHINE_EXTRA_RDEPENDS
- MACHINE_EXTRA_RRECOMMENDS

Modules are often not required for boot and can be excluded from certain build configurations. The following allows for the most flexibility:

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-mymodule"
```

The value is derived by appending the module filename without the `.ko` extension to the string "kernel-module-".

Because the variable is `RRECOMMENDS` and not a `RDEPENDS` variable, the build will not fail if this module is not available to include in the image.

2.6. Inspecting Changes and Commits¶

A common question when working with a kernel is: "What changes have been applied to this tree?" Rather than using "grep" across directories to see what has changed, you can use Git to inspect or search the kernel tree. Using Git is an efficient way to see what has changed in the tree.

2.6.1. What Changed in a Kernel?¶

Following are a few examples that show how to use Git commands to examine changes. These examples are by no means the only way to see changes.

Note
In the following examples, unless you provide a commit range, `kernel.org` history is blended with Yocto Project kernel changes. You can form ranges by using branch names from the kernel tree as the upper and lower commit markers with the Git commands. You can see the branch names through the web interface to the Yocto Project source repositories at http://git.yoctoproject.org/cgit.cgi.

To see a full range of the changes, use the `git whatchanged` command and specify a commit range for the branch (*commit*..*commit*).

Here is an example that looks at what has changed in the `emenlow` branch of the `linux-yocto-3.19` kernel. The lower commit range is the commit associated with the `standard/base` branch, while the upper commit range is the commit associated with the `standard/emenlow` branch.

```
$ git whatchanged origin/standard/base..origin/standard/emenlow
```

To see short, one line summaries of changes use the `git log` command:

```
$ git log --oneline origin/standard/base..origin/standard/emenlow
```

Use this command to see code differences for the changes:

```
$ git diff origin/standard/base..origin/standard/emenlow
```

Use this command to see the commit log messages and the text differences:

```
$ git show origin/standard/base..origin/standard/emenlow
```

Use this command to create individual patches for each change. Here is an example that that creates patch files for each commit and places them in your `Documents` directory:

```
$ git format-patch -o $HOME/Documents origin/standard/base..origin/standard/emenlow
```

2.6.2. Showing a Particular Feature or Branch Change¶

Tags in the Yocto Project kernel tree divide changes for significant features or branches. The `git show` *`tag`* command shows changes based on a tag. Here is an example that shows `systemtap` changes:

```
$ git show systemtap
```

You can use the `git branch --contains` *`tag`* command to show the branches that contain a particular feature. This command shows the branches that contain the `systemtap` feature:

```
$ git branch --contains systemtap
```

Chapter 3. Working with Advanced Metadata¶

Table of Contents

3.1. Overview¶

In addition to supporting configuration fragments and patches, the Yocto Project kernel tools also support rich Metadata that you can use to define complex policies and Board Support Package (BSP) support. The purpose of the Metadata and the tools that manage it, known as the kern-tools (`kern-tools-native_git.bb`), is to help you manage the complexity of the configuration and sources used to support multiple BSPs and Linux kernel types.

3.2. Using Kernel Metadata in a Recipe¶

The kernel sources in the Yocto Project contain kernel Metadata, which is located in the `meta` branches of the kernel source Git repositories. This Metadata defines Board Support Packages (BSPs) that correspond to definitions in linux-yocto recipes for the same BSPs. A BSP consists of an aggregation of kernel policy and hardware-specific feature enablements. The BSP can be influenced from within the linux-yocto recipe.

> **Note**
> Linux kernel source that contains kernel Metadata is said to be "linux-yocto style" kernel source. A Linux kernel recipe that inherits from the `linux-yocto.inc` include file is said to be a "linux-yocto style" recipe.

Every linux-yocto style recipe must define the `KMACHINE` variable. This variable is typically set to the same value as the `MACHINE` variable, which is used by BitBake. However, in some cases, the variable might instead refer to the underlying platform of the `MACHINE`.

Multiple BSPs can reuse the same `KMACHINE` name if they are built using the same BSP description. The "ep108-zynqmp" and "qemuzynqmp" BSP combination in the `meta-xilinx` layer is a good example of two BSPs using the same `KMACHINE` value (i.e. "zynqmp"). See the BSP Descriptions section for more information.

Every linux-yocto style recipe must also indicate the Linux kernel source repository branch used to build the Linux kernel. The `KBRANCH` variable must be set to indicate the branch.

> **Note**
> You can use the `KBRANCH` value to define an alternate branch typically with a machine override as shown here from the `meta-emenlow` layer:
>
> ```
> KBRANCH_emenlow-noemgd = "standard/base"
> ```

The linux-yocto style recipes can optionally define the following variables:

```
KERNEL_FEATURES
LINUX_KERNEL_TYPE
```

`LINUX_KERNEL_TYPE` defines the kernel type to be used in assembling the configuration. If you do not specify a `LINUX_KERNEL_TYPE`, it defaults to "standard". Together with `KMACHINE`, `LINUX_KERNEL_TYPE` defines the search arguments used by the kernel tools to find the appropriate description within the kernel Metadata with which to build out the sources and configuration. The linux-yocto recipes define "standard", "tiny", and "preempt-rt" kernel types. See the "Kernel Types" section for more information on kernel types.

During the build, the kern-tools search for the BSP description file that most closely matches the KMACHINE and LINUX_KERNEL_TYPE variables passed in from the recipe. The tools use the first BSP description it finds that match both variables. If the tools cannot find a match, they issue a warning such as the following:

```
WARNING: Can't find any BSP hardware or required configuration fragments.
WARNING: Looked at meta/cfg/broken/emenlow-broken/hdw_frags.txt and
         meta/cfg/broken/emenlow-broken/required_frags.txt in directory:
         meta/cfg/broken/emenlow-broken
```

In this example, KMACHINE was set to "emenlow-broken" and LINUX_KERNEL_TYPE was set to "broken".

The tools first search for the KMACHINE and then for the LINUX_KERNEL_TYPE. If the tools cannot find a partial match, they will use the sources from the KBRANCH and any configuration specified in the SRC_URI.

You can use the KERNEL_FEATURES variable to include features (configuration fragments, patches, or both) that are not already included by the KMACHINE and LINUX_KERNEL_TYPE variable combination. For example, to include a feature specified as "features/netfilter/netfilter.scc", specify:

```
KERNEL_FEATURES += "features/netfilter/netfilter.scc"
```

To include a feature called "cfg/sound.scc" just for the qemux86 machine, specify:

```
KERNEL_FEATURES_append_qemux86 = " cfg/sound.scc"
```

The value of the entries in KERNEL_FEATURES are dependent on their location within the kernel Metadata itself. The examples here are taken from the meta branch of the linux-yocto-3.19 repository. Within that branch, "features" and "cfg" are subdirectories of the meta/cfg/kernel-cache directory. For more information, see the "Kernel Metadata Syntax" section.

Note
The processing of the these variables has evolved some between the 0.9 and 1.3 releases of the Yocto Project and associated kern-tools sources. The descriptions in this section are accurate for 1.3 and later releases of the Yocto Project.

3.3. Kernel Metadata Location¶

Kernel Metadata can be defined in either the kernel recipe (recipe-space) or in the kernel tree (in-tree). Where you choose to define the Metadata depends on what you want to do and how you intend to work. Regardless of where you define the kernel Metadata, the syntax used applies equally.

If you are unfamiliar with the Linux kernel and only wish to apply a configuration and possibly a couple of patches provided to you by others, the recipe-space method is recommended. This method is also a good approach if you are working with Linux kernel sources you do not control or if you just do not want to maintain a Linux kernel Git repository on your own. For partial information on how you can define kernel Metadata in the recipe-space, see the "Modifying an Existing Recipe" section.

Conversely, if you are actively developing a kernel and are already maintaining a

Linux kernel Git repository of your own, you might find it more convenient to work with the kernel Metadata in the same repository as the Linux kernel sources. This method can make iterative development of the Linux kernel more efficient outside of the BitBake environment.

3.3.1. Recipe-Space Metadata¶

When stored in recipe-space, the kernel Metadata files reside in a directory hierarchy below `FILESEXTRAPATHS`. For a linux-yocto recipe or for a Linux kernel recipe derived by copying and modifying `oe-core/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb` to a recipe in your layer, `FILESEXTRAPATHS` is typically set to `${THISDIR}/${PN}`. See the "Modifying an Existing Recipe" section for more information.

Here is an example that shows a trivial tree of kernel Metadata stored in recipe-space within a BSP layer:

```
meta-my_bsp_layer/
`-- recipes-kernel
    `-- linux
        `-- linux-yocto
            |-- bsp-standard.scc
            |-- bsp.cfg
            `-- standard.cfg
```

When the Metadata is stored in recipe-space, you must take steps to ensure BitBake has the necessary information to decide what files to fetch and when they need to be fetched again. It is only necessary to specify the `.scc` files on the `SRC_URI`. BitBake parses them and fetches any files referenced in the `.scc` files by the `include`, `patch`, or `kconf` commands. Because of this, it is necessary to bump the recipe `PR` value when changing the content of files not explicitly listed in the `SRC_URI`.

3.3.2. In-Tree Metadata¶

When stored in-tree, the kernel Metadata files reside in the `meta` directory of the Linux kernel sources. The `meta` directory can be present in the same repository branch as the sources, such as "master", or `meta` can be its own orphan branch.

Note
An orphan branch in Git is a branch with unique history and content to the other branches in the repository. Orphan branches are useful to track Metadata changes independently from the sources of the Linux kernel, while still keeping them together in the same repository.

For the purposes of this document, we will discuss all in-tree Metadata as residing below the `meta/cfg/kernel-cache` directory.

Following is an example that shows how a trivial tree of Metadata is stored in a custom Linux kernel Git repository:

```
meta/
`-- cfg
    `-- kernel-cache
        |-- bsp-standard.scc
        |-- bsp.cfg
        `-- standard.cfg
```

To use a branch different from where the sources reside, specify the branch in the `KMETA` variable in your Linux kernel recipe. Here is an example:

```
KMETA = "meta"
```

To use the same branch as the sources, set `KMETA` to an empty string:

```
KMETA = ""
```

If you are working with your own sources and want to create an orphan `meta` branch, use these commands from within your Linux kernel Git repository:

```
$ git checkout --orphan meta
$ git rm -rf .
$ git commit --allow-empty -m "Create orphan meta branch"
```

If you modify the Metadata in the linux-yocto `meta` branch, you must not forget to update the `SRCREV` statements in the kernel's recipe. In particular, you need to update the `SRCREV_meta` variable to match the commit in the `KMETA` branch you wish to use. Changing the data in these branches and not updating the `SRCREV` statements to match will cause the build to fetch an older commit.

3.4. Kernel Metadata Syntax¶

The kernel Metadata consists of three primary types of files: `scc` [1] description files, configuration fragments, and patches. The `scc` files define variables and include or otherwise reference any of the three file types. The description files are used to aggregate all types of kernel Metadata into what ultimately describes the sources and the configuration required to build a Linux kernel tailored to a specific machine.

The `scc` description files are used to define two fundamental types of kernel Metadata:

- Features
- Board Support Packages (BSPs)

Features aggregate sources in the form of patches and configuration fragments into a modular reusable unit. You can use features to implement conceptually separate kernel Metadata descriptions such as pure configuration fragments, simple patches, complex features, and kernel types. Kernel types define general kernel features and policy to be reused in the BSPs.

BSPs define hardware-specific features and aggregate them with kernel types to form the final description of what will be assembled and built.

While the kernel Metadata syntax does not enforce any logical separation of configuration fragments, patches, features or kernel types, best practices dictate a logical separation of these types of Metadata. The following Metadata file hierarchy is recommended:

```
base/
   bsp/
   cfg/
   features/
   ktypes/
   patches/
```

The bsp directory contains the BSP descriptions. The remaining directories all contain "features". Separating bsp from the rest of the structure aids conceptualizing intended usage.

Use these guidelines to help place your scc description files within the structure:

- If your file contains only configuration fragments, place the file in the cfg directory.
- If your file contains only source-code fixes, place the file in the patches directory.
- If your file encapsulates a major feature, often combining sources and configurations, place the file in features directory.
- If your file aggregates non-hardware configuration and patches in order to define a base kernel policy or major kernel type to be reused across multiple BSPs, place the file in ktypes directory.

These distinctions can easily become blurred - especially as out-of-tree features slowly merge upstream over time. Also, remember that how the description files are placed is a purely logical organization and has no impact on the functionality of the kernel Metadata. There is no impact because all of cfg, features, patches, and ktypes, contain "features" as far as the kernel tools are concerned.

Paths used in kernel Metadata files are relative to <base>, which is either FILESEXTRAPATHS if you are creating Metadata in recipe-space, or meta/cfg/kernel-cache/ if you are creating Metadata in-tree.

3.4.1. Configuration¶

The simplest unit of kernel Metadata is the configuration-only feature. This feature consists of one or more Linux kernel configuration parameters in a configuration fragment file (.cfg) and a .scc file that describes the fragment.

The Symmetric Multi-Processing (SMP) fragment included in the linux-yocto-3.19 Git repository consists of the following two files:

```
     cfg/smp.scc:
        define KFEATURE_DESCRIPTION "Enable SMP"
        define KFEATURE_COMPATIBILITY all

        kconf hardware smp.cfg

     cfg/smp.cfg:
        CONFIG_SMP=y
        CONFIG_SCHED_SMT=y
        # Increase default NR_CPUS from 8 to 64 so that platform with
        # more than 8 processors can be all activated at boot time
        CONFIG_NR_CPUS=64
```

You can find information on configuration fragment files in the "Creating Configuration Fragments" section of the Yocto Project Development Manual and in the "Generating Configuration Files" section earlier in this manual.

KFEATURE_DESCRIPTION provides a short description of the fragment. Higher level kernel tools use this description.

The kconf command is used to include the actual configuration fragment in an .scc file, and the "hardware" keyword identifies the fragment as being hardware enabling, as opposed to general policy, which would use the "non-hardware" keyword. The distinction is made for the benefit of the configuration validation tools, which warn you if a hardware fragment overrides a policy set by a non-hardware fragment.

> **Note**
> The description file can include multiple `kconf` statements, one per fragment.

As described in the "Generating Configuration Files" section, you can use the following BitBake command to audit your configuration:

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

3.4.2. Patches¶

Patch descriptions are very similar to configuration fragment descriptions, which are described in the previous section. However, instead of a `.cfg` file, these descriptions work with source patches.

A typical patch includes a description file and the patch itself:

```
patches/mypatch.scc:
   patch mypatch.patch

patches/mypatch.patch:
   typical-patch
```

You can create the typical `.patch` file using `diff -Nurp` or `git format-patch`.

The description file can include multiple patch statements, one per patch.

3.4.3. Features¶

Features are complex kernel Metadata types that consist of configuration fragments (`kconf`), patches (`patch`), and possibly other feature description files (`include`).

Here is an example that shows a feature description file:

```
features/myfeature.scc
   define KFEATURE_DESCRIPTION "Enable myfeature"

   patch 0001-myfeature-core.patch
   patch 0002-myfeature-interface.patch

   include cfg/myfeature_dependency.scc
   kconf non-hardware myfeature.cfg
```

This example shows how the `patch` and `kconf` commands are used as well as how an additional feature description file is included.

Typically, features are less granular than configuration fragments and are more likely than configuration fragments and patches to be the types of things you want to specify in the `KERNEL_FEATURES` variable of the Linux kernel recipe. See the "Using Kernel Metadata in a Recipe" section earlier in the manual.

3.4.4. Kernel Types¶

A kernel type defines a high-level kernel policy by aggregating non-hardware configuration fragments with patches you want to use when building a Linux kernels of a specific type. Syntactically, kernel types are no different than features as described in the "Features" section. The `LINUX_KERNEL_TYPE` variable in the kernel recipe selects the kernel type. See the "Using Kernel Metadata in a Recipe" section for more information.

As an example, the `linux-yocto-3.19` tree defines three kernel types: "standard", "tiny", and "preempt-rt":

- "standard": Includes the generic Linux kernel policy of the Yocto Project linux-yocto kernel recipes. This policy includes, among other things, which file systems, networking options, core kernel features, and debugging and tracing options are supported.
- "preempt-rt": Applies the `PREEMPT_RT` patches and the configuration options required to build a real-time Linux kernel. This kernel type inherits from the "standard" kernel type.
- "tiny": Defines a bare minimum configuration meant to serve as a base for very small Linux kernels. The "tiny" kernel type is independent from the "standard" configuration. Although the "tiny" kernel type does not currently include any source changes, it might in the future.

The "standard" kernel type is defined by `standard.scc`:

```
# Include this kernel type fragment to get the standard features and
# configuration values.

# Include all standard features
include standard-nocfg.scc

kconf non-hardware standard.cfg

# individual cfg block section
include cfg/fs/devtmpfs.scc
include cfg/fs/debugfs.scc
include cfg/fs/btrfs.scc
include cfg/fs/ext2.scc
include cfg/fs/ext3.scc
include cfg/fs/ext4.scc

include cfg/net/ipv6.scc
include cfg/net/ip_nf.scc
include cfg/net/ip6_nf.scc
include cfg/net/bridge.scc
```

As with any `.scc` file, a kernel type definition can aggregate other `.scc` files with `include` commands. These definitions can also directly pull in configuration fragments and patches with the `kconf` and `patch` commands, respectively.

> **Note**
> It is not strictly necessary to create a kernel type `.scc` file. The Board Support Package (BSP) file can implicitly define the kernel type using a `define KTYPE mykype` line. See the "BSP Descriptions" section for more information.

3.4.5. BSP Descriptions¶

BSP descriptions combine kernel types with hardware-specific features. The hardware-specific portion is typically defined independently, and then aggregated with each supported kernel type. Consider this simple BSP description that supports the *`mybsp`* machine:

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386

   kconf mybsp.cfg
```

Every BSP description should define the `KMACHINE`, `KTYPE`, and `KARCH` variables. These variables allow the OpenEmbedded build system to identify the description as meeting the criteria set by the recipe being built. This simple example supports the "mybsp" machine for the "standard" kernel and the "i386" architecture.

Be aware that a hard link between the `KTYPE` variable and a kernel type description file does not exist. Thus, if you do not have kernel types defined in your kernel Metadata, you only need to ensure that the kernel recipe's `LINUX_KERNEL_TYPE` variable and the `KTYPE` variable in the BSP description file match.

> **Note**
> Future versions of the tooling make the specification of `KTYPE` in the BSP optional.

If you did want to separate your kernel policy from your hardware configuration, you could do so by specifying a kernel type, such as "standard" and including that description file in the BSP description file. See the "Kernel Types" section for more information.

You might also have multiple hardware configurations that you aggregate into a single hardware description file that you could include in the BSP description file, rather than referencing a single `.cfg` file. Consider the following:

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386

   include standard.scc
   include mybsp-hw.scc
```

In the above example, `standard.scc` aggregates all the configuration fragments, patches, and features that make up your standard kernel policy whereas *`mybsp`*`-hw.scc` aggregates all those necessary to support the hardware available on the *`mybsp`* machine. For information on how to break a complete `.config` file into the various configuration fragments, see the "Generating Configuration Files" section.

Many real-world examples are more complex. Like any other `.scc` file, BSP descriptions can aggregate features. Consider the Minnow BSP definition from the `linux-yocto-3.19` Git repository:

```
minnow.scc:
   include cfg/x86.scc
   include features/eg20t/eg20t.scc
   include cfg/dmaengine.scc
   include features/power/intel.scc
   include cfg/efi.scc
   include features/usb/ehci-hcd.scc
   include features/usb/ohci-hcd.scc
   include features/usb/usb-gadgets.scc
   include features/usb/touchscreen-composite.scc
   include cfg/timer/hpet.scc
   include cfg/timer/rtc.scc
   include features/leds/leds.scc
   include features/spi/spidev.scc
   include features/i2c/i2cdev.scc

   # Earlyprintk and port debug requires 8250
   kconf hardware cfg/8250.cfg

   kconf hardware minnow.cfg
   kconf hardware minnow-dev.cfg
```

The `minnow.scc` description file includes a hardware configuration fragment (`minnow.cfg`) specific to the Minnow BSP as well as several more general configuration fragments and features enabling hardware found on the machine. This description file is then included in each of the three "minnow" description files for the supported kernel types (i.e. "standard", "preempt-rt", and "tiny"). Consider the "minnow" description for the "standard" kernel type:

```
minnow-standard.scc:
   define KMACHINE minnow
   define KTYPE standard
   define KARCH i386

   include ktypes/standard

   include minnow.scc

   # Extra minnow configs above the minimal defined in minnow.scc
   include cfg/efi-ext.scc
   include features/media/media-all.scc
   include features/sound/snd_hda_intel.scc

   # The following should really be in standard.scc
   # USB live-image support
   include cfg/usb-mass-storage.scc
   include cfg/boot-live.scc

   # Basic profiling
   include features/latencytop/latencytop.scc
   include features/profiling/profiling.scc

   # Requested drivers that don't have an existing scc
   kconf hardware minnow-drivers-extra.cfg
```

The `include` command midway through the file includes the `minnow.scc` description that defines all hardware enablements for the BSP that is common to all kernel types. Using this command significantly reduces duplication.

Now consider the "minnow" description for the "tiny" kernel type:

```
minnow-tiny.scc:
   define KMACHINE minnow
   define KTYPE tiny
   define KARCH i386

   include ktypes/tiny

   include minnow.scc
```

As you might expect, the "tiny" description includes quite a bit less. In fact, it includes only the minimal policy defined by the "tiny" kernel type and the hardware-specific configuration required for booting the machine along with the most basic functionality of the system as defined in the base "minnow" description file.

Notice again the three critical variables: `KMACHINE`, `KTYPE`, and `KARCH`. Of these variables, only the `KTYPE` has changed. It is now set to "tiny".

3.5. Organizing Your Source¶

Many recipes based on the `linux-yocto-custom.bb` recipe use Linux kernel sources that have only a single branch - "master". This type of repository structure is fine for linear development supporting a single machine and architecture. However, if you work with multiple boards and architectures, a kernel source repository with multiple branches is more efficient. For example, suppose you need a series of patches for one board to boot. Sometimes, these patches are works-in-progress or fundamentally wrong, yet they are still necessary for specific boards. In these situations, you most likely do not want to include these patches in every kernel you build (i.e. have the patches as part of the lone "master" branch). It is situations like these that give rise to multiple branches used within a Linux kernel sources Git repository.

Repository organization strategies exist that maximize source reuse, remove redundancy, and logically order your changes. This section presents strategies for the following cases:

- Encapsulating patches in a feature description and only including the patches in the BSP descriptions of the applicable boards.
- Creating a machine branch in your kernel source repository and applying the patches on that branch only.
- Creating a feature branch in your kernel source repository and merging that branch into your BSP when needed.

The approach you take is entirely up to you and depends on what works best for your development model.

3.5.1. Encapsulating Patches¶

if you are reusing patches from an external tree and are not working on the patches, you might find the encapsulated feature to be appropriate. Given this scenario, you do not need to create any branches in the source repository. Rather, you just take the static patches you need and encapsulate them within a feature description. Once you have the feature description, you simply include that into the BSP description as described in the "BSP Descriptions" section.

You can find information on how to create patches and BSP descriptions in the "Patches" and "BSP Descriptions" sections.

3.5.2. Machine Branches¶

When you have multiple machines and architectures to support, or you are actively working on board support, it is more efficient to create branches in the repository based on individual machines. Having machine branches allows common source to remain in the "master" branch with any features specific to a machine stored in the appropriate machine branch. This organization method frees you from continually reintegrating your patches into a feature.

Once you have a new branch, you can set up your kernel Metadata to use the branch a couple different ways. In the recipe, you can specify the new branch as the `KBRANCH` to use for the board as follows:

```
KBRANCH = "mynewbranch"
```

Another method is to use the `branch` command in the BSP description:

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386
   include standard.scc

   branch mynewbranch

   include mybsp-hw.scc
```

If you find yourself with numerous branches, you might consider using a hierarchical branching system similar to what the linux-yocto Linux kernel repositories use:

```
common/kernel_type/machine
```

If you had two kernel types, "standard" and "small" for instance, three machines, and *`common`* as `mydir`, the branches in your Git repository might look like this:

```
mydir/base
mydir/standard/base
mydir/standard/machine_a
mydir/standard/machine_b
mydir/standard/machine_c
mydir/small/base
mydir/small/machine_a
```

This organization can help clarify the branch relationships. In this case, `mydir/standard/machine_a` includes everything in `mydir/base` and `mydir/standard/base`. The "standard" and "small" branches add sources specific to those kernel types that for whatever reason are not appropriate for the other branches.

> **Note**
> The "base" branches are an artifact of the way Git manages its data internally on the filesystem: Git will not allow you to use `mydir/standard` and `mydir/standard/machine_a` because it would have to create a file and a directory named "standard".

3.5.3. Feature Branches¶

When you are actively developing new features, it can be more efficient to work with that feature as a branch, rather than as a set of patches that have to be regularly updated. The Yocto Project Linux kernel tools provide for this with the `git merge` command.

To merge a feature branch into a BSP, insert the `git merge` command after any `branch` commands:

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386
   include standard.scc

   branch mynewbranch
   git merge myfeature

   include mybsp-hw.scc
```

3.6. SCC Description File Reference¶

This section provides a brief reference for the commands you can use within an SCC description file (`.scc`):

- `branch [ref]`: Creates a new branch relative to the current branch (typically `${KTYPE}`) using the currently checked-out branch, or "ref" if specified.
- `define`: Defines variables, such as `KMACHINE`, `KTYPE`, `KARCH`, and `KFEATURE_DESCRIPTION`.
- `include SCC_FILE`: Includes an SCC file in the current file. The file is parsed as if you had inserted it inline.
- `kconf [hardware|non-hardware] CFG_FILE`: Queues a configuration fragment for merging into the final Linux `.config` file.
- `git merge GIT_BRANCH`: Merges the feature branch into the current branch.
- `patch PATCH_FILE`: Applies the patch to the current Git branch.

[1] `scc` stands for Series Configuration Control, but the naming has less significance in the current implementation of the tooling than it had in the past. Consider `scc` files to be description files.

Appendix A. Advanced Kernel Concepts¶

Table of Contents

A.1. Yocto Project Kernel Development and Maintenance¶

Kernels available through the Yocto Project, like other kernels, are based off the Linux kernel releases from http://www.kernel.org. At the beginning of a major development cycle, the Yocto Project team chooses its kernel based on factors such as release timing, the anticipated release timing of final upstream `kernel.org` versions, and Yocto Project feature requirements. Typically, the kernel chosen is in the final stages of development by the community. In other words, the kernel is in the release candidate or "rc" phase and not yet a final release. But, by being in the final stages of external development, the team knows that the `kernel.org` final release will clearly be within the early stages of the Yocto Project development window.

This balance allows the team to deliver the most up-to-date kernel possible, while still ensuring that the team has a stable official release for the baseline Linux kernel version.

The ultimate source for kernels available through the Yocto Project are released kernels from `kernel.org`. In addition to a foundational kernel from `kernel.org`, the kernels available contain 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 Linux kernel that caters to specific embedded designer needs for targeted hardware.

Once a kernel is officially released, the Yocto Project team goes into their next development cycle, or upward revision (uprev) cycle, while still 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 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 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 kernels available through 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.

A.2. Kernel Architecture¶

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

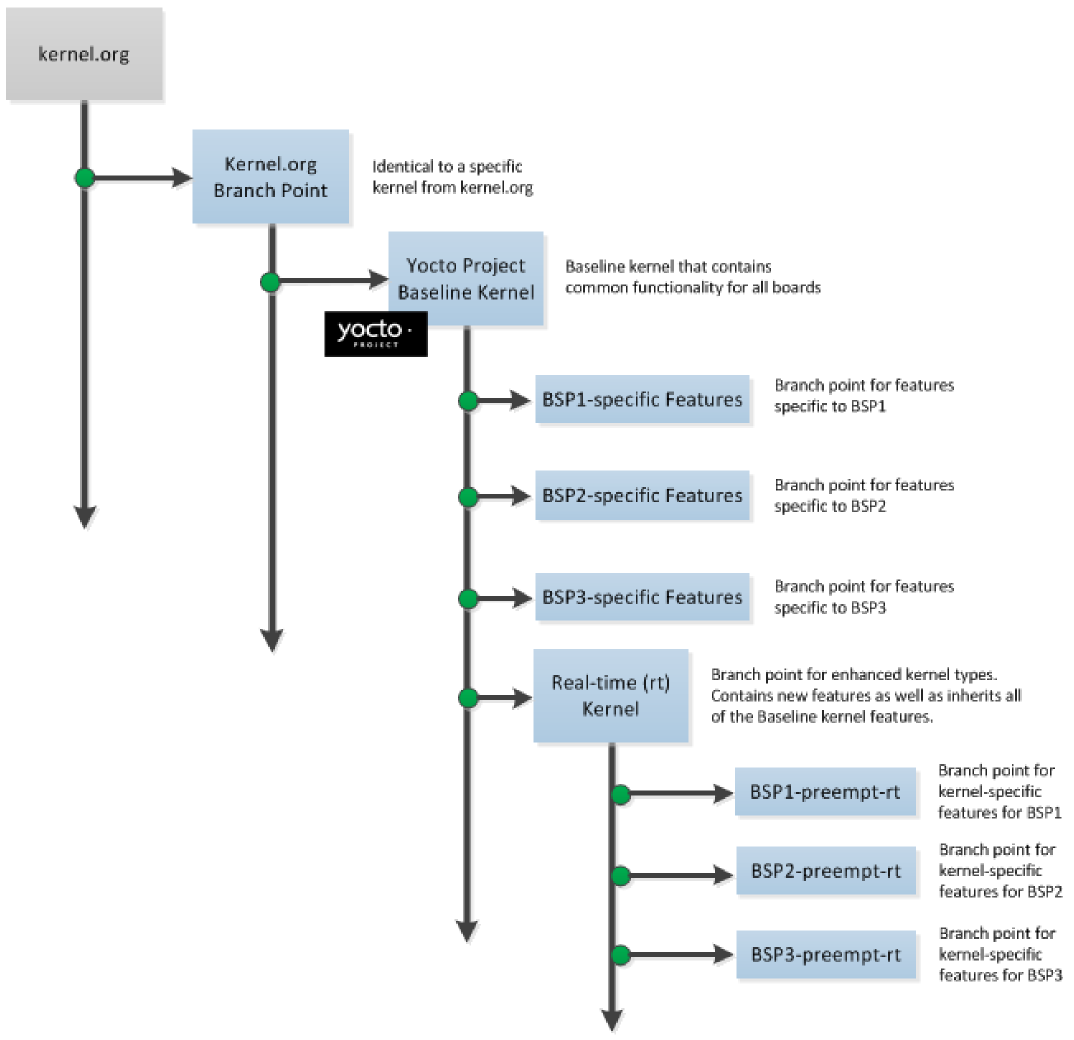
A.2.1. Overview¶

As mentioned earlier, a key goal of the 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 a Yocto Project kernel as consisting of a baseline Linux 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. For information on Git as applied to the Yocto Project, see the "Git" section in the Yocto Project Development Manual.

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.

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
Baseline kernel that contains common functionality for all boards
yocto PROJECT
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 do not 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 developer that, for all practical purposes, is the kernel needed for any given set of requirements.

A.2.2. 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 also results in not having to store the same feature twice internally in the tree. Rather, the kernel team stores 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, the team cannot always follow this practice. In those cases, the team uses isolated branches to merge features.

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, the team creates 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, the team only stores the unique differences for the BSP across the supported multiple kernels.

While this strategy can result in a tree with a significant number of branches, it is important to realize that from the developer'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 developer's perspective, this path is the "master" branch. The developer does not need to 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 developer because the kernel tree is left in this state after cloning and building the kernel.

A.2.3. Source Code Manager - Git¶

The Source Code Manager (SCM) is Git. This SCM 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.

You can find documentation on Git at http://git-scm.com/documentation. You can also get an introduction to Git as it applies to the Yocto Project in the "Git" section in the Yocto Project Development Manual. These referenced sections overview Git and describe a minimal set of commands that allows you to be functional using Git.

> **Note**
> You can use as much, or as little, of what Git has to offer to accomplish what you need for your project. You do not have to be a "Git Master" in order to use it with the Yocto Project.

Appendix B. Kernel Maintenance¶

Table of Contents

B.1. Tree Construction¶

This section describes construction of the Yocto Project kernel source repositories as accomplished by the Yocto Project team to create kernel repositories. These kernel repositories are found under the heading "Yocto Linux Kernel" at http://git.yoctoproject.org/cgit.cgi and can be shipped as part of a Yocto Project release. The team creates these repositories by compiling and executing the set of feature descriptions for every BSP and feature in the product. Those feature descriptions list all necessary patches, configuration, branching, tagging and feature divisions found in a kernel. Thus, the Yocto Project kernel repository (or tree) is built.

The existence of this tree allows you to access and clone a particular Yocto Project kernel repository and use it to build images based on their configurations and features.

You can find the files used to describe all the valid features and BSPs in the Yocto Project kernel in any clone of the Yocto Project kernel source repository Git tree. For example, the following command clones the Yocto Project baseline kernel that branched off of `linux.org` version 3.19:

```
     $ git clone git://git.yoctoproject.org/linux-yocto-3.19
```

For another example of how to set up a local Git repository of the Yocto Project kernel files, see the "Yocto Project Kernel" bulleted item in the Yocto Project Development Manual.

Once you have cloned the kernel Git repository on your local machine, you can switch to the `meta` branch within the repository. Here is an example that assumes the local Git repository for the kernel is in a top-level directory named `linux-yocto-3.19`:

```
     $ cd linux-yocto-3.19
     $ git checkout -b meta origin/meta
```

Once you have checked out and switched to the `meta` branch, you can see a snapshot of all the kernel configuration and feature descriptions that are used to build that particular kernel repository. These descriptions are in the form of `.scc` files.

You should realize, however, that browsing your local kernel repository for 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 an efficient and flexible way to inspect changes to the kernel.

> **Note**
> Ground up reconstruction of the complete kernel tree is an action only taken by the Yocto Project team during an active development cycle. When you create a clone of the kernel Git repository, you are simply making it efficiently available for building and development.

The following steps describe what happens when the Yocto Project Team constructs the Yocto Project kernel source Git repository (or tree) found at http://git.yoctoproject.org/cgit.cgi given the introduction of a new top-level kernel feature or BSP. These are the actions that effectively create the tree that includes the new feature, patch or BSP:

1. A top-level kernel feature is passed to the kernel build subsystem. Normally, this feature 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, which are located in `meta/cfg/kernel-cache`

   - Areas pointed to by `SRC_URI` statements found in recipes

   For a typical build, the target of the search is a feature description in an `.scc` file whose name follows this format:

```
bsp_name-kernel_type.scc
```

3. Once located, the feature description is either compiled into a simple script of actions, or into an existing equivalent script that is already part of the shipped kernel.

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 as described in step three.

6. The script is executed to produce a series of `meta-*` directories. These directories are descriptions of all the branches, tags, patches and configurations that need to be applied to the base Git repository to completely create the source (build) branch for the new BSP or feature.

7. The base repository is cloned, and the actions listed in the `meta-*` directories 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 kernel tree is now ready for developer consumption to be locally cloned, configured, and built into a Yocto Project kernel specific to some target hardware.

Note

The generated `meta-*` directories add 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 at http://git.yoctoproject.org/cgit.cgi is the combination of all supported boards and configurations.

The technique the Yocto Project team uses is flexible and allows for seamless blending of an immutable history with additional patches specific to a deployment. Any additions to the kernel become an integrated part of the branches.

B.2. Build Strategy¶

Once a local Git repository of the Yocto Project kernel exists on a development system, you can consider the compilation phase of kernel development - building a kernel image. Some prerequisites exist that are validated by the build process before compilation starts:

- The `SRC_URI` points to the kernel Git repository.

- A BSP build branch exists. This branch has the following form:

```
kernel_type/bsp_name
```

The OpenEmbedded build system makes sure these conditions exist before attempting compilation. Other means, however, do exist, such as as bootstrapping a BSP.

Before building a kernel, the build process verifies the tree and configures the kernel by processing all of the configuration "fragments" specified by feature descriptions in the `.scc` files. As the features are compiled, associated kernel configuration fragments are noted and recorded in the `meta-*` series of directories 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, kernel compilation is started and a kernel image is produced.

The other thing that you notice once you configure a kernel is that the build process generates a build tree that is separate from your kernel's local Git source repository tree. This build tree has a name that uses the following form, where `${MACHINE}` is the metadata name of the machine (BSP) and "kernel_type" is one of the Yocto Project supported kernel types (e.g. "standard"):

```
     linux-${MACHINE}-kernel_type-build
```

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

This behavior means that all the generated files for a particular machine or BSP are now in the build tree directory. The files include the final `.config` file, all the `.o` files, the `.a` files, and so forth. Since each machine or BSP has its own separate Build Directory in its own separate branch of the Git repository, you can easily switch between different builds.

Appendix C. Kernel Development FAQ¶

Table of Contents

C.1. Common Questions and Solutions¶

The following lists some solutions for common questions.

C.1.1. How do I use my own Linux kernel .config file?
C.1.2. How do I create configuration fragments?
C.1.3. How do I use my own Linux kernel sources?
C.1.4. How do I install/not-install the kernel image on the rootfs?
C.1.5. How do I install a specific kernel module?
C.1.6. How do I change the Linux kernel command line?

C.1.1. How do I use my own Linux kernel `.config` file?

Refer to the "Changing the Configuration" section for information.

C.1.2. How do I create configuration fragments?

Refer to the "Generating Configuration Files" section for information.

C.1.3. How do I use my own Linux kernel sources?

Refer to the "Working With Your Own Sources" section for information.

C.1.4. How do I install/not-install the kernel image on the rootfs?

The kernel image (e.g. `vmlinuz`) is provided by the `kernel-image` package. Image recipes depend on `kernel-base`. To specify whether or not the kernel image is installed in the generated root filesystem, override `RDEPENDS_kernel-base` to include or not include "kernel-image".

See the "Using .bbappend Files" section in the Yocto Project Development Manual for information on how to use an append file to override metadata.

C.1.5. How do I install a specific kernel module?

Linux kernel modules are packaged individually. To ensure a specific kernel module is included in an image, include it in the appropriate machine `RRECOMMENDS` variable.

These other variables are useful for installing specific modules:

```
MACHINE_ESSENTIAL_EXTRA_RDEPENDS
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS
MACHINE_EXTRA_RDEPENDS
MACHINE_EXTRA_RRECOMMENDS
```

For example, set the following in the `qemux86.conf` file to include the `ab123` kernel modules with images built for the `qemux86` machine:

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-ab123"
```

For more information, see the "Incorporating Out-of-Tree Modules" section.

C.1.6. How do I change the Linux kernel command line?

The Linux kernel command line is typically specified in the machine config using the `APPEND` variable. For example, you can add some helpful debug information doing the following:

```
APPEND += "printk.time=y initcall_debug debug"
```