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

Manual Notes

- This version of the ***Yocto Project Linux Kernel Development Manual*** is for the 2.5 release of the Yocto Project. To be sure you have the latest version of the manual for this release, go to the Yocto Project documentation page and select the manual from that site. Manuals from the site are more up-to-date than manuals derived from the Yocto Project released TAR files.
- If you located this manual through a web search, the version of the manual might not be the one you want (e.g. the search might have returned a manual much older than the Yocto Project version with which you are working). You can see all Yocto Project major releases by visiting the Releases page. If you need a version of this manual for a different Yocto Project release, visit the Yocto Project documentation page and select the manual set by using the "ACTIVE RELEASES DOCUMENTATION" or "DOCUMENTS ARCHIVE" pull-down menus.
- To report any inaccuracies or problems with this manual, send an email to the Yocto Project discussion group at `yocto@yoctoproject.com` or log into the freenode `#yocto` channel.

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 2.1	April 2016
Released with the Yocto Project 2.1 Release.	
Revision 2.2	October 2016
Released with the Yocto Project 2.2 Release.	
Revision 2.3	May 2017

Released with the Yocto Project 2.3 Release.	
Revision 2.4	October 2017
Released with the Yocto Project 2.4 Release.	
Revision 2.5	May 2018
Released with the Yocto Project 2.5 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 describes how to set up your build host to support kernel development, introduces the kernel development process, provides background information on the Yocto Linux kernel Metadata, describes common tasks you can perform using the kernel tools, shows you how to use the kernel Metadata needed to work with the kernel inside the Yocto Project, and provides insight into how the Yocto Project team develops and maintains Yocto Linux kernel Git repositories and Metadata.

Each Yocto Project release has a set of Yocto Linux kernel 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 Linux kernel upstream developments from http://www.kernel.org and introduce newly-supported platforms. Previous recipes in the release are refreshed and supported for at least one additional Yocto Project release. As they align, these previous releases are updated to include the latest from the Long Term Support Initiative (LTSI) project. You can learn more about Yocto Linux kernels and LTSI in the "Yocto Project Kernel Development and Maintenance" section.

Also included is a Yocto Linux kernel development recipe (`linux-yocto-dev.bb`) should you want to work with the very latest in upstream Yocto Linux kernel development and kernel Metadata development.

> **Note**
> For more on Yocto Linux kernels, see the "Yocto Project Kernel Development and Maintenance section.

The Yocto Project also provides a powerful set of kernel tools for managing Yocto 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 Yocto Linux kernel `menuconfig` system. Contrast this against a complete Yocto Linux kernel `.config` file, 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 Yocto Linux kernel recipes, an alternative exists by which you can use the Yocto Project Linux kernel tools with your own kernel sources.

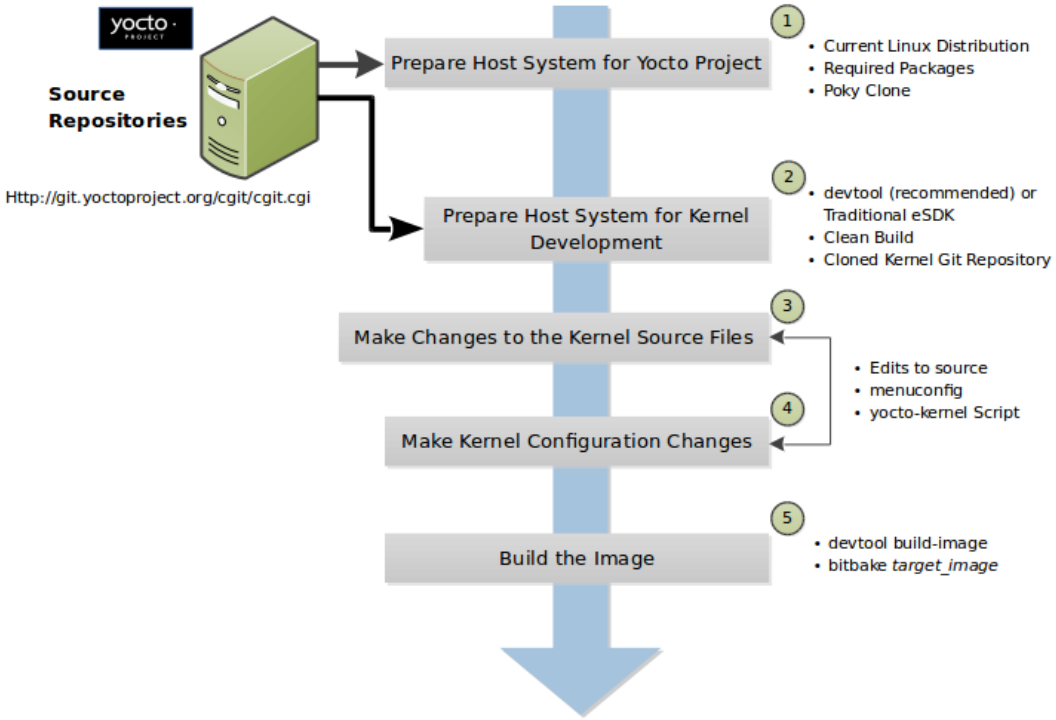
The remainder of this manual provides instructions for completing specific Linux kernel development tasks. These instructions assume you are comfortable working with BitBake recipes and basic open-source development tools. Understanding these concepts will facilitate the process of working with the kernel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

- Yocto Project Quick Build document.
- Yocto Project Overview and Concepts Manual.
- `devtool` workflow as described in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- The "Understanding and Creating Layers" section in the Yocto Project Development Tasks Manual.
- The "Kernel Modification Workflow" section.

1.2. Kernel Modification Workflow¶

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

This section presents a high-level overview of the Yocto Project kernel modification workflow. The illustration and accompanying list provide general information and references for further information.



1. ***Set up Your Host Development System to Support Development Using the Yocto Project***: See the "Setting Up the Development Host to Use the Yocto Project" section in the Yocto Project Development Tasks Manual for options on how to get a build host ready to use the Yocto Project.

2. ***Set Up Your Host Development System for Kernel Development:*** It is recommended that you use `devtool` and an extensible SDK for kernel development. Alternatively, you can use traditional kernel development methods with the Yocto Project. Either way, there are steps you need to take to get the development environment ready.

   Using `devtool` and the eSDK requires that you have a clean build of the image and that you are set up with the appropriate eSDK. For more information, see the "Getting Ready to Develop Using `devtool`" section.

   Using traditional kernel development requires that you have the kernel source available in an isolated local Git repository. For more information, see the "Getting Ready for Traditional Kernel Development" section.

3. ***Make Changes to the Kernel Source Code if applicable:*** Modifying the kernel does not always mean directly changing source files. However, if you have to do this, you make the changes to the files in the eSDK's Build Directory if you are using `devtool`. For more information, see the "Using `devtool` to Patch the Kernel" section.

   If you are using traditional kernel development, you edit the source files in the kernel's local Git repository. For more information, see the "Using Traditional Kernel Development to Patch the Kernel" section.

4. ***Make Kernel Configuration Changes if Applicable:*** If your situation calls for changing the kernel's configuration, you can use `menuconfig`, which allows you to interactively develop and test the configuration changes you are making to the kernel. Saving changes you make with `menuconfig` updates the kernel's `.config` file.

   > **Warning**
   > Try to resist the temptation to directly edit an existing `.config` file, which is found in the Build Directory among the source code used for the build. Doing so, can produce unexpected results when the OpenEmbedded build system regenerates the configuration file.

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

   Additionally, if you are working in a BSP layer and need to modify the BSP's kernel's configuration, you can use `menuconfig`.

5. ***Rebuild the Kernel Image With Your Changes:*** Rebuilding the kernel image applies your changes. Depending on your target hardware, you can verify your changes on actual hardware or perhaps QEMU.

The remainder of this developer's guide covers common tasks typically used during kernel development, advanced Metadata usage, and Yocto Linux kernel maintenance concepts.

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 your host development system for kernel development, preparing a layer, modifying an existing recipe, patching the kernel, configuring the kernel, 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 2.4 Release and forward.

2.1. Preparing the Build Host to Work on the Kernel¶

Before you can do any kernel development, you need to be sure your build host is set up to use the Yocto Project. For information on how to get set up, see the "Preparing the Build Host" section in the Yocto Project Development Tasks Manual. Part of preparing the system is creating a local Git repository of the Source Directory (poky) on your system. Follow the steps in the "Cloning the poky Repository" section in the Yocto Project Development Tasks Manual to set up your Source Directory.

> **Note**
> Be sure you check out the appropriate development branch or you create your local branch by checking out a specific tag to get the desired version of Yocto Project. See the "Checking Out by Branch in Poky" and "Checking Out by Tag in Poky" sections in the Yocto Project Development Tasks Manual for more information.

Kernel development is best accomplished using devtool and not through traditional kernel workflow methods. The remainder of this section provides information for both scenarios.

2.1.1. Getting Ready to Develop Using devtool¶

Follow these steps to prepare to update the kernel image using devtool. Completing this procedure leaves you with a clean kernel image and ready to make modifications as described in the "Using devtool to Patch the Kernel" section:

1. ***Initialize the BitBake Environment:*** Before building an extensible SDK, you need to initialize the BitBake build environment by sourcing the build environment script (i.e. oe-init-build-env):

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

> **Note**

The previous commands assume the Source Repositories (i.e. poky) have been cloned using Git and the local repository is named "poky".

2. ***Prepare Your local.conf File:*** By default, the MACHINE variable is set to "qemux86", which is fine if you are building for the QEMU emulator in 32-bit mode. However, if you are not, you need to set the MACHINE variable appropriately in your conf/local.conf file found in the Build Directory (i.e. ~/poky/build in this example).

   Also, since you are preparing to work on the kernel image, you need to set the MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS variable to include kernel modules.

   This example uses the default "qemux86" for the MACHINE variable but needs to add the "kernel-modules":

   ```
   MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules"
   ```

3. ***Create a Layer for Patches:*** You need to create a layer to hold patches created for the kernel image. You can use the bitbake-layers create-layer command as follows:

   ```
   $ cd ~/poky/build
   $ bitbake-layers create-layer ../../meta-mylayer
   NOTE: Starting bitbake server...
   Add your new layer with 'bitbake-layers add-layer ../../meta-mylayer'
   $
   ```

   Note

   For background information on working with common and BSP layers, see the "Understanding and Creating Layers" section in the Yocto Project Development Tasks Manual and the "BSP Layers" section in the Yocto Project Board Support (BSP) Developer's Guide, respectively. For information on how to use the bitbake-layers create-layer command to quickly set up a layer, see the "Creating a General Layer Using the bitbake-layers Script" section in the Yocto Project Development Tasks Manual.

4. ***Inform the BitBake Build Environment About Your Layer:*** As directed when you created your layer, you need to add the layer to the BBLAYERS variable in the bblayers.conf file as follows:

   ```
   $ cd ~/poky/build
   $ bitbake-layers add-layer ../../meta-mylayer
   NOTE: Starting bitbake server...
   $
   ```

5. ***Build the Extensible SDK:*** Use BitBake to build the extensible SDK specifically for use with images to be run using QEMU:

   ```
   $ cd ~/poky/build
   $ bitbake core-image-minimal -c populate_sdk_ext
   ```

   Once the build finishes, you can find the SDK installer file (i.e. *.sh file) in the following directory:

   ```
   ~/poky/build/tmp/deploy/sdk
   ```

   For this example, the installer file is named poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-2.5.sh

6. ***Install the Extensible SDK:*** Use the following command to install the SDK. For this example, install the SDK in the default ~/poky_sdk directory:

   ```
   $ cd ~/poky/build/tmp/deploy/sdk
   $ ./poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-2.5.sh
   Poky (Yocto Project Reference Distro) Extensible SDK installer version 2.5
   ==========================================================================
   Enter target directory for SDK (default: ~/poky_sdk):
   You are about to install the SDK to "/home/scottrif/poky_sdk". Proceed[Y/n]? Y
   Extracting SDK.....................................done
   Setting it up...
   Extracting buildtools...
   Preparing build system...
   Parsing recipes: 100% |#################################################################| Time: 0:00:52
   Initializing tasks: 100% |############## ##################################################| Time: 0:00:04
   Checking sstate mirror object availability: 100% |######################################| Time: 0:00:00
   Parsing recipes: 100% |#################################################################| Time: 0:00:33
   Initializing tasks: 100% |#################################################################| Time: 0:00:00
   done
   SDK has been successfully set up and is ready to be used.
   Each time you wish to use the SDK in a new shell session, you need to source the environment setup scri
    $ . /home/scottrif/poky_sdk/environment-setup-i586-poky-linux
   ```

7. ***Set Up a New Terminal to Work With the Extensible SDK:*** You must set up a new terminal to work with the SDK. You cannot use the same BitBake shell used to build the installer.

   After opening a new shell, run the SDK environment setup script as directed by the output from installing the SDK:

   ```
   $ source ~/poky_sdk/environment-setup-i586-poky-linux
   "SDK environment now set up; additionally you may now run devtool to perform development tasks.
   Run devtool --help for further details.
   ```

   Note

   If you get a warning about attempting to use the extensible SDK in an environment set up to run BitBake, you did not use a new shell.

8. ***Build the Clean Image:*** The final step in preparing to work on the kernel is to build an initial image using `devtool` in the new terminal you just set up and initialized for SDK work:

   ```
   $ devtool build-image
   Parsing recipes: 100% |##########################################| Time: 0:00:05
   Parsing of 830 .bb files complete (0 cached, 830 parsed). 1299 targets, 47 skipped, 0 masked, 0 errors.
   WARNING: No packages to add, building image core-image-minimal unmodified
   Loading cache: 100% |############################################| Time: 0:00:00
   Loaded 1299 entries from dependency cache.
   NOTE: Resolving any missing task queue dependencies
   Initializing tasks: 100% |#######################################| Time: 0:00:07
   Checking sstate mirror object availability: 100% |###############| Time: 0:00:00
   NOTE: Executing SetScene Tasks
   NOTE: Executing RunQueue Tasks
   NOTE: Tasks Summary: Attempted 2866 tasks of which 2604 didn't need to be rerun and all succeeded.
   NOTE: Successfully built core-image-minimal. You can find output files in /home/scottrif/poky_sdk/tmp/d
   ```

   If you were building for actual hardware and not for emulation, you could flash the image to a USB stick on `/dev/sdd` and boot your device. For an example that uses a Minnowboard, see the TipsAndTricks/KernelDevelopmentWithEsdk Wiki page.

At this point you have set up to start making modifications to the kernel by using the extensible SDK. For a continued example, see the "Using `devtool` to Patch the Kernel" section.

2.1.2. Getting Ready for Traditional Kernel Development¶

Getting ready for traditional kernel development using the Yocto Project involves many of the same steps as described in the previous section. However, you need to establish a local copy of the kernel source since you will be editing these files.

Follow these steps to prepare to update the kernel image using traditional kernel development flow with the Yocto Project. Completing this procedure leaves you ready to make modifications to the kernel source as described in the "Using Traditional Kernel Development to Patch the Kernel" section:

1. ***Initialize the BitBake Environment:*** Before you can do anything using BitBake, you need to initialize the BitBake build environment by sourcing the build environment script (i.e. `oe-init-build-env`). Also, for this example, be sure that the local branch you have checked out for `poky` is the Yocto Project Sumo branch. If you need to checkout out the Sumo branch, see the "Checking out by Branch in Poky" section in the Yocto Project Development Tasks Manual.

   ```
   $ cd ~/poky
   $ git branch
   master
   * Sumo
   $ source oe-init-build-env
   ```

   Note

   The previous commands assume the Source Repositories (i.e. `poky`) have been cloned using Git and the local repository is named "poky".

2. ***Prepare Your `local.conf` File:*** By default, the `MACHINE` variable is set to "qemux86", which is fine if you are building for the QEMU emulator in 32-bit mode. However, if you are not, you need to set the `MACHINE` variable appropriately in your `conf/local.conf` file found in the Build Directory (i.e. `~/poky/build` in this example).

   Also, since you are preparing to work on the kernel image, you need to set the `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS` variable to include kernel modules.

   This example uses the default "qemux86" for the `MACHINE` variable but needs to add the "kernel-modules":

   ```
   MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules"
   ```

3. ***Create a Layer for Patches:*** You need to create a layer to hold patches created for the kernel image. You can use the `bitbake-layers create-layer` command as follows:

```
$ cd ~/poky/build
$ bitbake-layers create-layer ../../meta-mylayer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../../meta-mylayer'
```

> **Note**
> For background information on working with common and BSP layers, see the "Understanding and Creating Layers" section in the Yocto Project Development Tasks Manual and the "BSP Layers" section in the Yocto Project Board Support (BSP) Developer's Guide, respectively. For information on how to use the `bitbake-layers create-layer` command to quickly set up a layer, see the "Creating a General Layer Using the `bitbake-layers` Script" section in the Yocto Project Development Tasks Manual.

4. ***Inform the BitBake Build Environment About Your Layer:*** As directed when you created your layer, you need to add the layer to the `BBLAYERS` variable in the `bblayers.conf` file as follows:

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../../meta-mylayer
NOTE: Starting bitbake server ...
$
```

5. ***Create a Local Copy of the Kernel Git Repository:*** You can find Git repositories of supported Yocto Project kernels organized under "Yocto Linux Kernel" in the Yocto Project Source Repositories at http://git.yoctoproject.org.

   For simplicity, it is recommended that you create your copy of the kernel Git repository outside of the Source Directory, which is usually named `poky`. Also, be sure you are in the `standard/base` branch.

   The following commands show how to create a local copy of the `linux-yocto-4.12` kernel and be in the `standard/base` branch.

> **Note**
> The `linux-yocto-4.12` kernel can be used with the Yocto Project 2.4 release and forward. You cannot use the `linux-yocto-4.12` kernel with releases prior to Yocto Project 2.4:

```
$ cd ~
$ git clone git://git.yoctoproject.org/linux-yocto-4.12 --branch standard/base
Cloning into 'linux-yocto-4.12'...
remote: Counting objects: 6097195, done.
remote: Compressing objects: 100% (901026/901026), done.
remote: Total 6097195 (delta 5152604), reused 6096847 (delta 5152256)
Receiving objects: 100% (6097195/6097195), 1.24 GiB | 7.81 MiB/s, done.
Resolving deltas: 100% (5152604/5152604), done.
Checking connectivity... done.
Checking out files: 100% (59846/59846), done.
```

6. ***Create a Local Copy of the Kernel Cache Git Repository:*** For simplicity, it is recommended that you create your copy of the kernel cache Git repository outside of the Source Directory, which is usually named `poky`. Also, for this example, be sure you are in the `yocto-4.12` branch.

   The following commands show how to create a local copy of the `yocto-kernel-cache` and be in the `yocto-4.12` branch:

```
$ cd ~
$ git clone git://git.yoctoproject.org/yocto-kernel-cache --branch yocto-4.12
Cloning into 'yocto-kernel-cache'...
remote: Counting objects: 22639, done.
remote: Compressing objects: 100% (9761/9761), done.
remote: Total 22639 (delta 12400), reused 22586 (delta 12347)
Receiving objects: 100% (22639/22639), 22.34 MiB | 6.27 MiB/s, done.
Resolving deltas: 100% (12400/12400), done.
Checking connectivity... done.
```

At this point, you are ready to start making modifications to the kernel using traditional kernel development steps. For a continued example, see the "Using Traditional Kernel Development to Patch the Kernel" section.

2.2. 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`) as well as store and use kernel patch files. For background information on working with layers, see the "Understanding and Creating Layers" section in the Yocto Project Development Tasks Manual.

> **Tip**
> The Yocto Project comes with many tools that simplify tasks you need to perform. One such tool is the `bitbake-layers create-layer` command, which simplifies creating a new layer. See the "Creating a General Layer Using the `bitbake-layers` Script" section in the Yocto Project Development Tasks Manual for information on how to use this script to quick set up a new layer.

To better understand the layer you create for kernel development, the following section describes how to create a layer without the aid of tools. These steps assume creation of a layer named `mylayer` in your home directory:

1. ***Create Structure***: Create the layer's structure:

```
$ cd $HOME
$ mkdir meta-mylayer
$ mkdir meta-mylayer/conf
$ mkdir meta-mylayer/recipes-kernel
$ mkdir meta-mylayer/recipes-kernel/linux
$ mkdir meta-mylayer/recipes-kernel/linux/linux-yocto
```

   The `conf` directory holds your configuration files, while the `recipes-kernel` directory holds your append file and eventual patch files.

2. ***Create the Layer Configuration File***: Move to the `meta-mylayer/conf` directory and create the `layer.conf` file as follows:

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

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "mylayer"
BBFILE_PATTERN_mylayer = "^${LAYERDIR}/"
BBFILE_PRIORITY_mylayer = "5"
```

   Notice `mylayer` as part of the last three statements.

3. ***Create the Kernel Recipe Append File***: Move to the `meta-mylayer/recipes-kernel/linux` directory and create the kernel's append file. This example uses the `linux-yocto-4.12` kernel. Thus, the name of the append file is `linux-yocto_4.12.bbappend`:

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

SRC_URI_append += "file://patch-file-one"
SRC_URI_append += "file://patch-file-two"
SRC_URI_append += "file://patch-file-three"
```

   The `FILESEXTRAPATHS` and `SRC_URI` statements enable the OpenEmbedded build system to find patch files. For more information on using append files, see the "Using .bbappend Files in Your Layer" section in the Yocto Project Development Tasks Manual.

2.3. 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 information.

2.3.1. 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_4.12.bb` recipe,

the append file will typically be located as follows within your custom layer:

```
your-layer/recipes-kernel/linux/linux-yocto_4.12.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.

As an example, consider the following append file used by the BSPs in meta-yocto-bsp:

```
meta-yocto-bsp/recipes-kernel/linux/linux-yocto_4.12.bbappend
```

The following listing shows the file. Be aware that the actual commit ID strings in this example listing might be different than the actual strings in the file from the meta-yocto-bsp layer upstream.

```
KBRANCH_genericx86  = "standard/base"
KBRANCH_genericx86-64  = "standard/base"

KMACHINE_genericx86 ?= "common-pc"
KMACHINE_genericx86-64 ?= "common-pc-64"
KBRANCH_edgerouter = "standard/edgerouter"
KBRANCH_beaglebone = "standard/beaglebone"
KBRANCH_mpc8315e-rdb = "standard/fsl-mpc8315e-rdb"

SRCREV_machine_genericx86    ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_genericx86-64 ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_edgerouter ?= "b5c8cfda2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_beaglebone ?= "b5c8cfda2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_mpc8315e-rdb ?= "2d1d010240846d7bff15d1fcc0cb6eb8a22fc78a"


COMPATIBLE_MACHINE_genericx86 = "genericx86"
COMPATIBLE_MACHINE_genericx86-64 = "genericx86-64"
COMPATIBLE_MACHINE_edgerouter = "edgerouter"
COMPATIBLE_MACHINE_beaglebone = "beaglebone"
COMPATIBLE_MACHINE_mpc8315e-rdb = "mpc8315e-rdb"

LINUX_VERSION_genericx86 = "4.12.7"
LINUX_VERSION_genericx86-64 = "4.12.7"
LINUX_VERSION_edgerouter = "4.12.10"
LINUX_VERSION_beaglebone = "4.12.10"
LINUX_VERSION_mpc8315e-rdb = "4.12.10"
```

This append file contains statements used to support several BSPs that ship with the Yocto Project. The file defines machines using the COMPATIBLE_MACHINE variable and uses the KMACHINE variable to ensure the machine name used by the OpenEmbedded build system maps to the machine name used by the Linux Yocto kernel. The file also uses the optional KBRANCH variable to ensure the build process uses the appropriate kernel branch.

Although this particular example does not use it, the KERNEL_FEATURES variable could be used to enable features specific to the kernel. The append file points to specific commits in the Source Directory Git repository and the meta Git repository branches to identify the exact kernel needed to build the BSP.

One thing missing in this particular BSP, which you will typically need when developing a BSP, is the kernel configuration file (.config) for your BSP. When developing a BSP, you probably have a kernel configuration file or a set of kernel configuration files that, when taken together, define the kernel configuration for your BSP. You can accomplish this definition by putting the configurations in a file or a set of files inside a directory located at the same level as your kernel's append file and having the same name as the kernel's main recipe file. With all these conditions met, simply reference those files in the SRC_URI statement in the append file.

For example, suppose you had some configuration options in a file called network_configs.cfg. You can place that file inside a directory named linux-yocto and then add a SRC_URI statement such as the following to the append file. When the OpenEmbedded build system builds the kernel, the configuration options are picked up and applied.

```
SRC_URI += "file://network_configs.cfg"
```

To group related configurations into multiple files, you perform a similar procedure. Here is an example that groups separate configurations specifically for Ethernet and graphics into their own files and adds the configurations by using a SRC_URI statement like the following in your append file:

```
SRC_URI += "file://myconfig.cfg \
            file://eth.cfg \
            file://gfx.cfg"
```

Another variable you can use in your kernel recipe append file is the FILESEXTRAPATHS variable. When you use this statement, you are extending the locations used by the OpenEmbedded system to look for files and patches as the recipe is processed.

> **Note**
>
> Other methods exist to accomplish grouping and defining configuration options. For example, if you are working with a local clone of the kernel repository, you could checkout the kernel's meta branch, make your changes, and then push the changes to the local bare clone of the kernel. The result is that you directly add configuration options to the meta branch for your BSP. The configuration options will likely end up in that location anyway if the BSP gets added to the Yocto Project.
>
> In general, however, the Yocto Project maintainers take care of moving the SRC_URI-specified configuration options to the kernel's meta branch. Not only is it easier for BSP developers to not have to worry about putting those configurations in the branch, but having the maintainers do it allows them to apply 'global' knowledge about the kinds of common configuration options multiple BSPs in the tree are typically using. This allows for promotion of common configurations into common features.

2.3.2. 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 using devtool, see the "Using devtool to Patch the Kernel" and "Using Traditional Kernel Development to Patch the Kernel" sections.

2.3.3. 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.

2.3.4. 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, use the following statement form:

```
KBUILD_DEFCONFIG_KMACHINE ?= defconfig_file
```

Here is an example that appends the KBUILD_DEFCONFIG variable with "common-pc" and provides the path to the "in-tree" defconfig file:

```
KBUILD_DEFCONFIG_common-pc ?= "/home/scottrif/configfiles/my_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.

2.4. Using devtool to Patch the Kernel¶

The steps in this procedure show you how you can patch the kernel using the extensible SDK and devtool.

> **Note**
> Before attempting this procedure, be sure you have performed the steps to get ready for updating the kernel as described in the "Getting Ready to Develop Using devtool" section.

Patching the kernel involves changing or adding configurations to an existing kernel, changing or adding recipes to the kernel that are needed to support specific hardware features, or even altering the source code itself.

This example creates a simple patch by adding some QEMU emulator console output at boot time through printk statements in the kernel's calibrate.c source code file. Applying the patch and booting the modified image causes the added messages to appear on the emulator's console. The example is a continuation of the setup procedure found in the "Getting Ready to Develop Using devtool" Section.

1. ***Check Out the Kernel Source Files:*** First you must use devtool to checkout the kernel source code in its workspace. Be sure you are in the terminal set up to do work with the extensible SDK.

   > **Note**
   > See this step in the "Getting Ready to Develop Using devtool" section for more information.

   Use the following devtool command to check out the code:

   ```
   $ devtool modify linux-yocto
   ```

Note

During the checkout operation, a bug exists that could cause errors such as the following to appear:

```
ERROR: Taskhash mismatch 2c793438c2d9f8c3681fd5f7bc819efa versus
       be3a89ce7c47178880ba7bf6293d7404 for
       /path/to/esdk/layers/poky/meta/recipes-kernel/linux/linux-yocto_4.10.bb.d
```

You can safely ignore these messages. The source code is correctly checked out.

2. ***Edit the Source Files*** Follow these steps to make some simple changes to the source files:

   a. ***Change the working directory***: In the previous step, the output noted where you can find the source files (e.g. `~/poky_sdk/workspace/sources/linux-yocto`). Change to where the kernel source code is before making your edits to the `calibrate.c` file:

   ```
   $ cd ~/poky_sdk/workspace/sources/linux-yocto
   ```

   b. ***Edit the source file***: Edit the `init/calibrate.c` file to have the following changes:

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

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

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

3. ***Build the Updated Kernel Source:*** To build the updated kernel source, use `devtool`:

   ```
   $ devtool build linux-yocto
   ```

4. ***Create the Image With the New Kernel:*** Use the `devtool build-image` command to create a new image that has the new kernel.

Note

If the image you originally created resulted in a Wic file, you can use an alternate method to create the new image with the updated kernel. For an example, see the steps in the TipsAndTricks/KernelDevelopmentWithEsdk Wiki Page.

```
$ cd ~
$ devtool build-image core-image-minimal
```

5. ***Test the New Image:*** For this example, you can run the new image using QEMU to verify your changes:

   a. ***Boot the image***: Boot the modified image in the QEMU emulator using this command:

   ```
   $ runqemu qemux86
   ```

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

   ```
   # dmesg | less
   ```

   You should see the results of your `printk` statements as part of the output when you scroll down the console window.

6. ***Stage and commit your changes***: Within your eSDK terminal, change your working directory to where you modified the `calibrate.c` file and use these Git commands to stage and commit your changes:

   ```
   $ cd ~/poky_sdk/workspace/sources/linux-yocto
   $ git status
   $ git add init/calibrate.c
   $ git commit -m "calibrate: Add printk example"
   ```

7. ***Export the Patches and Create an Append File:*** To export your commits as patches and create a `.bbappend` file, use the following command in the terminal used to work with the extensible SDK. This example uses the previously established layer named `meta-mylayer`.

   Note
   See Step 3 of the "Getting Ready to Develop Using devtool" section for information on setting up this layer.

   ```
   $ devtool finish linux-yocto ~/meta-mylayer
   ```

   Once the command finishes, the patches and the `.bbappend` file are located in the `~/meta-mylayer/recipes-kernel/linux` directory.

8. ***Build the Image With Your Modified Kernel:*** You can now build an image that includes your kernel patches. Execute the following command from your Build Directory in the terminal set up to run BitBake:

   ```
   $ cd ~/poky/build
   $ bitbake core-image-minimal
   ```

2.5. Using Traditional Kernel Development to Patch the Kernel¶

The steps in this procedure show you how you can patch the kernel using traditional kernel development (i.e. not using `devtool` and the extensible SDK as described in the "Using `devtool` to Patch the Kernel" section).

Note
Before attempting this procedure, be sure you have performed the steps to get ready for updating the kernel as described in the "Getting Ready for Traditional Kernel Development" section.

Patching the kernel involves changing or adding configurations to an existing kernel, changing or adding recipes to the kernel that are needed to support specific hardware features, or even altering the source code itself.

The example in this section creates a simple patch by adding some QEMU emulator console output at boot time through `printk` statements in the kernel's `calibrate.c` source code file. Applying the patch and booting the modified image causes the added messages to appear on the emulator's console. The example is a continuation of the setup procedure found in the "Getting Ready for Traditional Kernel Development" Section.

1. ***Edit the Source Files*** Prior to this step, you should have used Git to create a local copy of the repository for your kernel. Assuming you created the repository as directed in the "Getting Ready for Traditional Kernel Development" section, use the following commands to edit the `calibrate.c` file:

   a. ***Change the working directory***: You need to locate the source files in the local copy of the kernel Git repository: Change to where the kernel source code is before making your edits to the `calibrate.c` file:

      ```
      $ cd ~/linux-yocto-4.12/init
      ```

   b. ***Edit the source file***: Edit the `calibrate.c` file to have the following changes:

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

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

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

2. ***Stage and Commit Your Changes:*** Use standard Git commands to stage and commit the changes you just made:

   ```
   $ git add calibrate.c
   $ git commit -m "calibrate.c - Added some printk statements"
   ```

   If you do not stage and commit your changes, the OpenEmbedded Build System will not pick up the changes.

3. ***Update Your `local.conf` File to Point to Your Source Files:*** In addition to your `local.conf` file specifying to use "kernel-modules" and the "qemux86" machine, it must also point to the updated kernel source files. Add `SRC_URI` and `SRCREV` statements similar to the following to your `local.conf`:

```
$ cd ~/poky/build/conf
```

Add the following to the `local.conf`:

```
SRC_URI_pn-linux-yocto = "git:///path-to/linux-yocto-4.12;protocol=file;name=machine;branch=standard/ba
                          git:///path-to/yocto-kernel-cache;protocol=file;type=kmeta;name=meta;branch=y
SRCREV_meta_qemux86 = "${AUTOREV}"
SRCREV_machine_qemux86 = "${AUTOREV}"
```

> **Note**
> Be sure to replace *path-to* with the pathname to your local Git repositories. Also, you must be sure to specify the correct branch and machine types. For this example, the branch is `standard/base` and the machine is "qemux86".

4. ***Build the Image:*** With the source modified, your changes staged and committed, and the `local.conf` file pointing to the kernel files, you can now use BitBake to build the image:

```
$ cd ~/poky/build
$ bitbake core-image-minimal
```

5. ***Boot the image***: Boot the modified image in the QEMU emulator using this command. When prompted to login to the QEMU console, use "root" with no password:

```
$ cd ~/poky/build
$ runqemu qemux86
```

6. ***Look for Your Changes:*** As QEMU booted, you might have seen your changes rapidly scroll by. If not, use these commands to see your changes:

```
# dmesg | less
```

You should see the results of your `printk` statements as part of the output when you scroll down the console window.

7. ***Generate the Patch File:*** Once you are sure that your patch works correctly, you can generate a `*.patch` file in the kernel source repository:

```
$ cd ~/linux-yocto-4.12/init
$ git format-patch -1
0001-calibrate.c-Added-some-printk-statements.patch
```

8. ***Move the Patch File to Your Layer:*** In order for subsequent builds to pick up patches, you need to move the patch file you created in the previous step to your layer `meta-mylayer`. For this example, the layer created earlier is located in your home directory as `meta-mylayer`. When the layer was created using the `yocto-create` script, no additional hierarchy was created to support patches. Before moving the patch file, you need to add additional structure to your layer using the following commands:

```
$ cd ~/meta-mylayer
$ mkdir recipes-kernel
$ mkdir recipes-kernel/linux
$ mkdir recipes-kernel/linux/linux-yocto
```

Once you have created this hierarchy in your layer, you can move the patch file using the following command:

```
$ mv ~/linux-yocto-4.12/init/0001-calibrate.c-Added-some-printk-statements.patch ~/meta-mylayer/recipes
```

9. ***Create the Append File:*** Finally, you need to create the `linux-yocto_4.12.bbappend` file and insert statements that allow the OpenEmbedded build system to find the patch. The append file needs to be in your layer's `recipes-kernel/linux` directory and it must be named `linux-yocto_4.12.bbappend` and have the following contents:

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

SRC_URI_append = " file://0001-calibrate.c-Added-some-printk-statements.patch"
```

The `FILESEXTRAPATHS` and `SRC_URI` statements enable the OpenEmbedded build system to find the patch file.

For more information on append files and patches, see the "Creating the Append File" and "Applying Patches" sections. You can also see the "Using .bbappend Files in Your Layer"" section in the Yocto Project Development Tasks

Manual.

> **Note**
> To build `core-image-minimal` again and see the effects of your patch, you can essentially eliminate the temporary source files saved in `poky/build/tmp/work/...` and residual effects of the build by entering the following sequence of commands:
>
> ```
> $ cd ~/poky/build
> $ bitbake -c cleanall yocto-linux
> $ bitbake core-image-minimal -c cleanall
> $ bitbake core-image-minimal
> $ runqemu qemux86
> ```

2.6. Configuring the Kernel¶

Configuring the Yocto Project kernel consists of making sure the `.config` file has all the right information in it for the image you are building. You can use the `menuconfig` tool and configuration fragments to make sure your `.config` file is just how you need it. You can also save known configurations in a `defconfig` file that the build system can use for kernel configuration.

This section describes how to use `menuconfig`, create and use configuration fragments, and how to interactively modify your `.config` file to create the leanest kernel configuration file possible.

For more information on kernel configuration, see the "Changing the Configuration" section.

2.6.1. Using `menuconfig`¶

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

To use the `menuconfig` tool in the Yocto Project development environment, you must launch it using BitBake. Thus, the environment must be set up using the `oe-init-build-env` script found in the Build Directory. You must also be sure of the state of your build's configuration in the Source Directory. The following commands initialize the BitBake environment, run the `do_kernel_configme` task, and launch `menuconfig`. These commands assume the Source Directory's top-level folder is `~/poky`:

```
$ cd poky
$ source oe-init-build-env
$ bitbake linux-yocto -c kernel_configme -f
$ bitbake linux-yocto -c menuconfig
```

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

> **Note**
> You can use the entire `.config` file as the `defconfig` file. For information on `defconfig` files, see the "Changing the Configuration", "Using an In-Tree `defconfig` File, and "Creating a `defconfig` File" sections.

Consider an example that configures the "CONFIG_SMP" setting for the `linux-yocto-4.12` kernel.

> **Note**
> The OpenEmbedded build system recognizes this kernel as `linux-yocto` through Metadata (e.g. `PREFERRED_VERSION_linux-yocto ?= "12.4%"`).

Once `menuconfig` launches, use the interface to navigate through the selections to find the configuration settings in which you are interested. For this example, you deselect "CONFIG_SMP" by clearing the "Symmetric Multi-Processing Support" option. Using the interface, you can find the option under "Processor Type and Features". To deselect "CONFIG_SMP", use the arrow keys to highlight "Symmetric Multi-Processing Support" and enter "N" to clear the asterisk. When you are finished, exit out and save the change.

Saving the selections updates the `.config` configuration file. This is the file that the OpenEmbedded build system uses to configure the kernel during the build. You can find and examine this file in the Build Directory in `tmp/work/`. The

actual `.config` is located in the area where the specific kernel is built. For example, if you were building a Linux Yocto kernel based on the `linux-yocto-4.12` kernel and you were building a QEMU image targeted for `x86` architecture, the `.config` file would be:

```
poky/build/tmp/work/qemux86-poky-linux/linux-yocto/4.12.12+gitAUTOINC+eda4d18...
...967-r0/linux-qemux86-standard-build/.config
```

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

Within the `.config` file, you can see the kernel settings. For example, the following entry shows that symmetric multi-processor support is not set:

```
# CONFIG_SMP is not set
```

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

> **Note**
> Be sure to make a copy of the `.config` file and do not just rename it. The build system needs an existing `.config` file from which to work.

2.6.2. Creating a `defconfig` File¶

A `defconfig` file is simply a `.config` renamed to "defconfig". You can use a `defconfig` file to retain a known set of kernel configurations from which the OpenEmbedded build system can draw to create the final `.config` file.

> **Note**
> Out-of-the-box, the Yocto Project never ships a `defconfig` or `.config` file. The OpenEmbedded build system creates the final `.config` file used to configure the kernel.

To create a `defconfig`, start with a complete, working Linux kernel `.config` file. Copy that file to the appropriate `${PN}` directory in your layer's `recipes-kernel/linux` directory, and rename the copied file to "defconfig" (e.g. `~/meta-mylayer/recipes-kernel/linux/linux-yocto/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.

For more information on configuring the kernel, see the "Changing the Configuration" section.

2.6.3. Creating Configuration Fragments¶

Configuration fragments are simply kernel options that appear in a file placed where the OpenEmbedded build system can find and apply them. The build system applies configuration fragments after applying configurations from a `defconfig` file. Thus, the final kernel configuration is a combination of the configurations in the `defconfig` file and then any

configuration fragments you provide. The build system applies fragments on top of and after applying the existing defconfig file configurations.

Syntactically, the configuration statement is identical to what would appear in the `.config` file, which is in the Build Directory.

> **Note**
> For more information about where the `.config` file is located, see the example in the "Using `menuconfig`" section.

It is simple to create a configuration fragment. One method is to use shell commands. For example, issuing the following from the shell creates a configuration fragment file named `my_smp.cfg` that enables multi-processor support within the kernel:

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

> **Note**
> All configuration fragment files must use the `.cfg` extension in order for the OpenEmbedded build system to recognize them as a configuration fragment.

Another 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 Through Kernel Configuration:*** Complete a build at least through the kernel configuration task as follows:

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

   This step ensures that you create a `.config` file from a known state. Because situations exist where your build state might become unknown, it is best to run this task prior to starting `menuconfig`.

2. ***Launch `menuconfig`:*** Run the `menuconfig` command:

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

3. ***Create the Configuration Fragment:*** Run the `diffconfig` command to prepare a configuration fragment. The resulting file `fragment.cfg` is 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 additional 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.

Where do you put your configuration fragment files? You can place these files in an area pointed to by `SRC_URI` as directed by your `bblayers.conf` file, which is located in your layer. The OpenEmbedded build system picks up the configuration and adds it to the kernel's configuration. For example, suppose you had a set of configuration options in a file called `myconfig.cfg`. If you put that file inside a directory named `linux-yocto` that resides in the same directory as the kernel's append file within your layer and then add the following statements to the kernel's append file, those configuration options will be picked up and applied when the kernel is built:

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

As mentioned earlier, you can group related configurations into multiple files and name them all in the `SRC_URI` statement as well. For example, you could group separate configurations specifically for Ethernet and graphics into their own files and add those by using a `SRC_URI` statement like the following in your append file:

```
SRC_URI += "file://myconfig.cfg \
       file://eth.cfg \
       file://gfx.cfg"
```

2.6.4. Validating Configuration¶

You can use the do_kernel_configcheck task to provide configuration validation:

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

Running this task produces 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.

In order to run this task, you must have an existing .config file. See the "Using menuconfig" section for information on how to create a configuration file.

Following is sample output from the do_kernel_configcheck task:

```
Loading cache: 100% |#####################################################| Time: 0:00:00
Loaded 1275 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
    .
    .
    .

NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: linux-yocto-4.12.12+gitAUTOINC+eda4d18ce4_16de014967-r0 do_kernel_configcheck:
    [kernel config]: specified values did not make it into the kernel's final configuration:

---------- CONFIG_X86_TSC -----------------
Config: CONFIG_X86_TSC
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/bsp/common
Requested value:  CONFIG_X86_TSC=y
Actual value:


---------- CONFIG_X86_BIGSMP -----------------
Config: CONFIG_X86_BIGSMP
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cf
      /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value:  # CONFIG_X86_BIGSMP is not set
Actual value:


---------- CONFIG_NR_CPUS -----------------
Config: CONFIG_NR_CPUS
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cf
      /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/bsp/common
      /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value:  CONFIG_NR_CPUS=8
Actual value:     CONFIG_NR_CPUS=1


---------- CONFIG_SCHED_SMT -----------------
Config: CONFIG_SCHED_SMT
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cf
      /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value:  CONFIG_SCHED_SMT=y
Actual value:



NOTE: Tasks Summary: Attempted 288 tasks of which 285 didn't need to be rerun and all succeeded.

Summary: There were 3 WARNING messages shown.
```

> **Note**
> The previous output example has artificial line breaks to make it more readable.

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 do_kernel_configme and do_kernel_configcheck tasks until they produce no warnings.

For more information on how to use the menuconfig tool, see the "Using menuconfig" section.

2.6.5. Fine-Tuning the Kernel Configuration File¶

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

As part of the kernel build process, the do_kernel_configcheck task runs. This task validates the kernel configuration by checking the final .config file against the input files. During the check, the task produces warning

messages for the following issues:

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

> **Note**
> The `do_kernel_configcheck` task can also optionally report if an option is overridden during processing.

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

To streamline the configuration, do the following:

1. ***Use a Working Configuration:*** Start with a full configuration that you know works. Be sure the configuration builds and boots successfully. Use this configuration file as your baseline.
2. ***Run Configure and Check Tasks:*** Separately run the `do_kernel_configme` and `do_kernel_configcheck` tasks:

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

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

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

2.7. Expanding Variables¶

Sometimes it is helpful to determine what a variable expands to during a build. You can do examine the values of variables by examining the output of the `bitbake -e` command. The output is long and is more easily managed in a text file, which allows for easy searches:

```
$ bitbake -e virtual/kernel > some_text_file
```

Within the text file, you can see exactly how each variable is expanded and used by the OpenEmbedded build system.

2.8. Working with a "Dirty" Kernel Version String¶

If you build a kernel image and the version string has a "+" or a "-dirty" at the end, uncommitted modifications exist in the kernel's source directory. Follow these steps to clean up the version string:

1. ***Discover the Uncommitted Changes:*** Go to the kernel's locally cloned Git repository (source directory) and use the following Git command to list the files that have been changed, added, or removed:

   ```
   $ git status
   ```

2. ***Commit the Changes:*** You should commit those changes to the kernel source tree regardless of whether or not you will save, export, or use the changes:

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

3. ***Rebuild the Kernel Image:*** Once you commit the changes, rebuild the kernel.

Depending on your particular kernel development workflow, the commands you use to rebuild the kernel might differ. For information on building the kernel image when using `devtool`, see the "Using `devtool` to Patch the Kernel" section. For information on building the kernel image when using Bitbake, see the "Using Traditional Kernel Development to Patch the Kernel" section.

2.9. 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. ***Create a Copy of the Kernel Recipe:*** Copy the `linux-yocto-custom.bb` recipe to your layer and give it a meaningful name. The name should include the version of the Yocto Linux kernel you are using (e.g. `linux-yocto-myproject_4.12.bb`, where "4.12" is the base version of the Linux kernel with which you would be working).

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

3. ***Ensure You Have Configurations:*** Make sure you have either a `defconfig` file or configuration fragment files in your layer. 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` file to the `files` directory in your layer 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 Recipe:*** 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. "4.12").
   - `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+218bd8d2022b9852c60d32f0d770931e3cf343e2
     ```

     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. ***Customize Your Recipe as Needed:*** 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.

2.10. 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.

2.10.1. 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.

2.10.2. 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.

2.11. 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.

2.11.1. 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.

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
```

2.11.2. 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
```

2.12. Adding Recipe-Space Kernel Features¶

You can add kernel features in the recipe-space by using the `KERNEL_FEATURES` variable and by specifying the feature's `.scc` file path in the `SRC_URI` statement. When you add features using this method, the OpenEmbedded

build system checks to be sure the features are present. If the features are not present, the build stops. Kernel features are the last elements processed for configuring and patching the kernel. Therefore, adding features in this manner is a way to enforce specific features are present and enabled without needing to do a full audit of any other layer's additions to the SRC_URI statement.

You add a kernel feature by providing the feature as part of the KERNEL_FEATURES variable and by providing the path to the feature's .scc file, which is relative to the root of the kernel Metadata. The OpenEmbedded build system searches all forms of kernel Metadata on the SRC_URI statement regardless of whether the Metadata is in the "kernel-cache", system kernel Metadata, or a recipe-space Metadata (i.e. part of the kernel recipe). See the "Kernel Metadata Location" section for additional information.

When you specify the feature's .scc file on the SRC_URI statement, the OpenEmbedded build system adds the directory of that .scc file along with all its subdirectories to the kernel feature search path. Because subdirectories are searched, you can reference a single .scc file in the SRC_URI statement to reference multiple kernel features.

Consider the following example that adds the "test.scc" feature to the build.

1. ***Create the Feature File:*** Create a .scc file and locate it just as you would any other patch file, .cfg file, or fetcher item you specify in the SRC_URI statement.

   ## Notes

   - You must add the directory of the .scc file to the fetcher's search path in the same manner as you would add a .patch file.
   - You can create additional .scc files beneath the directory that contains the file you are adding. All subdirectories are searched during the build as potential feature directories.

   Continuing with the example, suppose the "test.scc" feature you are adding has a test.scc file in the following directory:

   ```
   my_recipe
       |
       +-linux-yocto
          |
          +-test.cfg
          +-test.scc
   ```

   In this example, the linux-yocto directory has both the feature test.scc file and a similarly named configuration fragment file test.cfg.

2. ***Add the Feature File to SRC_URI:*** Add the .scc file to the recipe's SRC_URI statement:

   ```
   SRC_URI_append = " file://test.scc"
   ```

   The leading space before the path is important as the path is appended to the existing path.

3. ***Specify the Feature as a Kernel Feature:*** Use the KERNEL_FEATURES statement to specify the feature as a kernel feature:

   ```
   KERNEL_FEATURES_append = " test.scc"
   ```

   The OpenEmbedded build system processes the kernel feature when it builds the kernel.

   ## Note

   If other features are contained below "test.scc", then their directories are relative to the directory containing the test.scc file.

Chapter 3. Working with Advanced Metadata (yocto-kernel-cache)¶

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 is to help you manage the complexity of the configuration and sources used to support multiple BSPs and Linux kernel types.

Kernel Metadata exists in many places. One area in the Yocto Project Source Repositories is the `yocto-kernel-cache` Git repository. You can find this repository grouped under the "Yocto Linux Kernel" heading in the Yocto Project Source Repositories.

Kernel development tools ("kern-tools") exist also in the Yocto Project Source Repositories under the "Yocto Linux Kernel" heading in the `yocto-kernel-tools` Git repository. The recipe that builds these tools is `meta/recipes-kernel/kern-tools/kern-tools-native_git.bb` in the Source Directory (e.g. `poky`).

3.2. Using Kernel Metadata in a Recipe¶

As mentioned in the introduction, the Yocto Project contains kernel Metadata, which is located in the `yocto-kernel-cache` Git repository. This Metadata defines Board Support Packages (BSPs) that correspond to definitions in linux-yocto recipes for corresponding BSPs. A BSP consists of an aggregation of kernel policy and enabled hardware-specific features. The BSP can be influenced from within the linux-yocto recipe.

Note

A Linux kernel recipe that contains kernel Metadata (e.g. inherits from the `linux-yocto.inc` 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. Multiple Corei7-based BSPs could share the same "intel-corei7-64" value for `KMACHINE`. It is important to realize that `KMACHINE` is just for kernel mapping, while `MACHINE` is the machine type within a BSP Layer. Even with this distinction, however, these two variables can hold the same value. 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-yocto-bsp` layer:

```
KBRANCH_edgerouter = "standard/edgerouter"
```

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.

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 yocto-kernel-cache repository. Each branch of this repository contains "features" and "cfg" subdirectories at the top-level. For more information, see the "Kernel Metadata Syntax" section.

3.3. 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 the top level of yocto-kernel-cache if you are creating Metadata outside of the recipe-space.

3.3.1. 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.

As an example, consider the Symmetric Multi-Processing (SMP) fragment used with the linux-yocto-4.12 kernel as defined outside of the recipe space (i.e. yocto-kernel-cache). This Metadata consists of two files:

smp.scc and smp.cfg. You can find these files in the cfg directory of the yocto-4.12 branch in the yocto-kernel-cache Git repository:

```
cfg/smp.scc:
   define KFEATURE_DESCRIPTION "Enable SMP for 32 bit builds"
   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
   # The following is needed when setting NR_CPUS to something
   # greater than 8 on x86 architectures, it should be automatically
   # disregarded by Kconfig when using a different arch
   CONFIG_X86_BIGSMP=y
```

You can find general information on configuration fragment files in the "Creating Configuration Fragments" section.

Within the smp.scc file, the KFEATURE_DESCRIPTION statement provides a short description of the fragment. Higher level kernel tools use this description.

Also within the smp.scc file, the kconf command includes 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 "Validating Configuration" section, you can use the following BitBake command to audit your configuration:

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

3.3.2. 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 (i.e. .patch files).

A typical patch includes a description file and the patch itself. As an example, consider the build patches used with the linux-yocto-4.12 kernel as defined outside of the recipe space (i.e. yocto-kernel-cache). This Metadata consists of several files: build.scc and a set of *.patch files. You can find these files in the patches/build directory of the yocto-4.12 branch in the yocto-kernel-cache Git repository.

The following listings show the build.scc file and part of the modpost-mask-trivial-warnings.patch file:

```
patches/build/build.scc:
   patch arm-serialize-build-targets.patch
   patch powerpc-serialize-image-targets.patch
   patch kbuild-exclude-meta-directory-from-distclean-processi.patch

   # applied by kgit
   # patch kbuild-add-meta-files-to-the-ignore-li.patch

   patch modpost-mask-trivial-warnings.patch
   patch menuconfig-check-lxdiaglog.sh-Allow-specification-of.patch

patches/build/modpost-mask-trivial-warnings.patch:
   From bd48931bc142bdd104668f3a062a1f22600aae61 Mon Sep 17 00:00:00 2001
   From: Paul Gortmaker <paul.gortmaker@windriver.com>
   Date: Sun, 25 Jan 2009 17:58:09 -0500
   Subject: [PATCH] modpost: mask trivial warnings

   Newer HOSTCC will complain about various stdio fcns because
                     .
                     .
                     .
           char *dump_write = NULL, *files_source = NULL;
           int opt;
   --
   2.10.1

   generated by cgit v0.10.2 at 2017-09-28 15:23:23 (GMT)
```

The description file can include multiple patch statements where each statement handles a single patch. In the example build.scc file, five patch statements exist for the five patches in the directory.

You can create a typical `.patch` file using `diff -Nurp` or `git format-patch` commands. For information on how to create patches, see the "Using `devtool` to Patch the Kernel" and "Using Traditional Kernel Development to Patch the Kernel" sections.

3.3.3. Features¶

Features are complex kernel Metadata types that consist of configuration fragments, patches, and possibly other feature description files. As an example, consider the following generic listing:

```
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 with the `include` command.

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.

3.3.4. 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 kernel of a specific type (e.g. a real-time kernel). 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. For example, in the `linux-yocto_4.12.bb` kernel recipe found in `poky/meta/recipes-kernel/linux`, a `require` directive includes the `poky/meta/recipes-kernel/linux/linux-yocto.inc` file, which has the following statement that defines the default kernel type:

```
LINUX_KERNEL_TYPE ??= "standard"
```

Another example would be the real-time kernel (i.e. `linux-yocto-rt_4.12.bb`). This kernel recipe directly sets the kernel type as follows:

```
LINUX_KERNEL_TYPE = "preempt-rt"
```

> **Note**
> You can find kernel recipes in the `meta/recipes-kernel/linux` directory of the Source Directory (e.g. `poky/meta/recipes-kernel/linux/linux-yocto_4.12.bb`). See the "Using Kernel Metadata in a Recipe" section for more information.

Three kernel types ("standard", "tiny", and "preempt-rt") are supported for Linux Yocto kernels:

- "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.

For any given kernel type, the Metadata is defined by the `.scc` (e.g. `standard.scc`). Here is a partial listing for the `standard.scc` file, which is found in the `ktypes/standard` directory of the `yocto-kernel-cache` Git repository:

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

# Note: if only the features are desired, but not the configuration
#       then this should be included as:
#             include ktypes/standard/standard.scc nocfg
#       if no chained configuration is desired, include it as:
#             include ktypes/standard/standard.scc nocfg inherit



include ktypes/base/base.scc
branch standard
```

```
kconf non-hardware standard.cfg

include features/kgdb/kgdb.scc
           .
           .
           .

include cfg/net/ip6_nf.scc
include cfg/net/bridge.scc

include cfg/systemd.scc

include features/rfkill/rfkill.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 myktype` line. See the "BSP Descriptions" section for more information.

3.3.5. BSP Descriptions¶

BSP descriptions (i.e. `*.scc` files) combine kernel types with hardware-specific features. The hardware-specific Metadata is typically defined independently in the BSP layer, and then aggregated with each supported kernel type.

> **Note**
> For BSPs supported by the Yocto Project, the BSP description files are located in the `bsp` directory of the `yocto-kernel-cache` repository organized under the "Yocto Linux Kernel" heading in the Yocto Project Source Repositories.

This section overviews the BSP description structure, the aggregation concepts, and presents a detailed example using a BSP supported by the Yocto Project (i.e. BeagleBone Board). For complete information on BSP layer file hierarchy, see the Yocto Project Board Support Package (BSP) Developer's Guide.

3.3.5.1. Overview¶

For simplicity, consider the following root BSP layer description files for the BeagleBone board. These files employ both a structure and naming convention for consistency. The naming convention for the file is as follows:

```
bsp_root_name-kernel_type.scc
```

Here are some example root layer BSP filenames for the BeagleBone Board BSP, which is supported by the Yocto Project:

```
beaglebone-standard.scc
beaglebone-preempt-rt.scc
```

Each file uses the root name (i.e "beaglebone") BSP name followed by the kernel type.

Examine the `beaglebone-standard.scc` file:

```
define KMACHINE beaglebone
define KTYPE standard
define KARCH arm

include ktypes/standard/standard.scc
branch beaglebone

include beaglebone.scc

# default policy for standard kernels
include features/latencytop/latencytop.scc
include features/profiling/profiling.scc
```

Every top-level BSP description file 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 example supports the "beaglebone" machine for the "standard" kernel and the "arm" 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 the kernel type defined in your kernel Metadata as it is here, you only need to ensure that the `LINUX_KERNEL_TYPE` variable in the kernel recipe and the `KTYPE` variable in the BSP description file match.

To separate your kernel policy from your hardware configuration, you include a kernel type (`ktype`), such as "standard". In the previous example, this is done using the following:

```
include ktypes/standard/standard.scc
```

This file aggregates all the configuration fragments, patches, and features that make up your standard kernel policy. See the "Kernel Types" section for more information.

To aggregate common configurations and features specific to the kernel for *mybsp*, use the following:

```
include mybsp.scc
```

You can see that in the BeagleBone example with the following:

```
include beaglebone.scc
```

For information on how to break a complete `.config` file into the various configuration fragments, see the "Creating Configuration Fragments" section.

Finally, if you have any configurations specific to the hardware that are not in a `*.scc` file, you can include them as follows:

```
kconf hardware mybsp-extra.cfg
```

The BeagleBone example does not include these types of configurations. However, the Malta 32-bit board does ("mti-malta32"). Here is the `mti-malta32-le-standard.scc` file:

```
define KMACHINE mti-malta32-le
define KMACHINE qemumipsel
define KTYPE standard
define KARCH mips

include ktypes/standard/standard.scc
branch mti-malta32

include mti-malta32.scc
kconf hardware mti-malta32-le.cfg
```

3.3.5.2. Example¶

Many real-world examples are more complex. Like any other `.scc` file, BSP descriptions can aggregate features. Consider the Minnow BSP definition given the `linux-yocto-4.4` branch of the `yocto-kernel-cache` (i.e. `yocto-kernel-cache/bsp/minnow/minnow.scc`):

> **Note**
> Although the Minnow Board BSP is unused, the Metadata remains and is being used here just as an example.

```
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 features/leds/leds.scc
include features/spi/spidev.scc
include features/i2c/i2cdev.scc
include features/mei/mei-txe.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 `minnow.scc` 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 (i.e. `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 enabled hardware 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 (i.e. `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 KTYPE has changed to specify the "tiny" kernel type.

3.4. Kernel Metadata Location¶

Kernel Metadata always exists outside of the kernel tree either defined in a kernel recipe (recipe-space) or outside of the recipe. 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 kernel Metadata kept outside the recipe-space. Working with Metadata in this area can make iterative development of the Linux kernel more efficient outside of the BitBake environment.

3.4.1. 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.

If the BSP description is in recipe space, you cannot simply list the `*.scc` in the SRC_URI statement. You need to use the following form from your kernel append file:

```
SRC_URI_append_myplatform = " \
   file://myplatform;type=kmeta;destsuffix=myplatform \
   "
```

3.4.2. Metadata Outside the Recipe-Space¶

When stored outside of the recipe-space, the kernel Metadata files reside in a separate repository. The OpenEmbedded build system adds the Metadata to the build as a "type=kmeta" repository through the SRC_URI variable. As an example, consider the following SRC_URI statement from the linux-yocto_4.12.bb kernel recipe:

```
SRC_URI = "git://git.yoctoproject.org/linux-yocto-4.12.git;name=machine;branch=${KBRANCH}; \
           git://git.yoctoproject.org/yocto-kernel-cache;type=kmeta;name=meta;branch=yocto-4.12;destsuffix=${KM
```

${KMETA}, in this context, is simply used to name the directory into which the Git fetcher places the Metadata. This behavior is no different than any multi-repository SRC_URI statement used in a recipe (e.g. see the previous section).

You can keep kernel Metadata in a "kernel-cache", which is a directory containing configuration fragments. As with any Metadata kept outside the recipe-space, you simply need to use the SRC_URI statement with the "type=kmeta" attribute. Doing so makes the kernel Metadata available during the configuration phase.

If you modify the Metadata, 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.

3.5. 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 Yocto Linux Kernel Git 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 (Yocto Linux kernels), like other kernels, are based off the Linux kernel releases from http://www.kernel.org. At the beginning of a major Linux kernel development cycle, the Yocto Project team chooses a Linux 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 Linux kernel chosen is in the final stages of development by the Linux community. In other words, the Linux kernel is in the release candidate or "rc" phase and has yet to reach 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 Yocto Project team to deliver the most up-to-date Yocto Linux kernel possible, while still ensuring that the team has a stable official release for the baseline Linux kernel version.

As implied earlier, the ultimate source for Yocto Linux kernels are released kernels from `kernel.org`. In addition to a foundational kernel from `kernel.org`, the available Yocto Linux kernels contain a mix of important new mainline developments, non-mainline developments (when no alternative exists), 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.

You can find a web interface to the Yocto Linux kernels in the Source Repositories at http://git.yoctoproject.org. If you look at the interface, you will see to the left a grouping of Git repositories titled "Yocto Linux Kernel". Within this group, you will find several Linux Yocto kernels developed and included with Yocto Project releases:

- ***linux-yocto-4.1:*** The stable Yocto Project kernel to use with the Yocto Project Release 2.0. This kernel is based on the Linux 4.1 released kernel.
- ***linux-yocto-4.4:*** The stable Yocto Project kernel to use with the Yocto Project Release 2.1. This kernel is based on the Linux 4.4 released kernel.
- ***linux-yocto-4.6:*** A temporary kernel that is not tied to any Yocto Project release.
- ***linux-yocto-4.8:*** The stable yocto Project kernel to use with the Yocto Project Release 2.2.
- ***linux-yocto-4.9:*** The stable Yocto Project kernel to use with the Yocto Project Release 2.3. This kernel is based on the Linux 4.9 released kernel.
- ***linux-yocto-4.10:*** The default stable Yocto Project kernel to use with the Yocto Project Release 2.3. This kernel is based on the Linux 4.10 released kernel.
- ***linux-yocto-4.12:*** The default stable Yocto Project kernel to use with the Yocto Project Release 2.4. This kernel is based on the Linux 4.12 released kernel.
- ***yocto-kernel-cache:*** The `linux-yocto-cache` contains patches and configurations for the linux-yocto kernel tree. This repository is useful when working on the linux-yocto kernel. For more information on this "Advanced Kernel Metadata", see the "Working With Advanced Metadata (`yocto-kernel-cache`)" Chapter.
- ***linux-yocto-dev:*** A development kernel based on the latest upstream release candidate available.

Notes

Long Term Support Initiative (LTSI) for Yocto Linux kernels is as follows:

- For Yocto Project releases 1.7, 1.8, and 2.0, the LTSI kernel is `linux-yocto-3.14`.
- For Yocto Project releases 2.1, 2.2, and 2.3, the LTSI kernel is `linux-yocto-4.1`.
- For Yocto Project release 2.4, the LTSI kernel is `linux-yocto-4.9`
- `linux-yocto-4.4` is an LTS kernel.

Once a Yocto Linux 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 Linux kernel development, BSP support, and release timing to select the best possible `kernel.org` Linux kernel version on which to base subsequent Yocto Linux kernel development. The team continually monitors Linux 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 Linux kernel both adds features and introduces new bugs. These consequences are the basic properties of upstream Linux kernel development and are managed by the Yocto Project team's Yocto Linux kernel development strategy. It is the Yocto Project team's policy to not back-port minor features to the released Yocto Linux 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 Linux kernel can easily create mismatches, incompatibilities and very subtle errors.

The policies described in this section result in both a stable and a cutting edge Yocto Linux kernel that mixes forward ports of existing Linux kernel features and significant and critical new functionality. Forward porting Linux kernel functionality into the Yocto Linux kernels available through the Yocto Project can be thought of as a "micro uprev." The many “micro uprevs” produce a Yocto Linux kernel version with a mix of important new mainline, non-mainline, BSP developments and feature integrations. This Yocto Linux 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 Yocto Linux kernels is evolving and the kernels are used in leading edge feature and BSP development.

A.2. Yocto Linux Kernel Architecture and Branching Strategies¶

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, in particular the branching strategies, used achieve that goal in a manner similar to upstream Linux kernel development in `kernel.org`.

You can think of a Yocto Linux 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 Yocto Project team using the Source Code Manager (SCM) Git.

Notes

- Git is the obvious SCM for meeting the Yocto Linux kernel organizational and structural goals described in this section. Not only is Git the SCM for Linux kernel development in `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 Overview and Concepts Manual. The latter reference provides an overview of Git and presents a minimal set of Git commands that allows you to be functional using Git. 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 Expert" in order to use it with the Yocto Project.

Using Git's tagging and branching features, 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 "tree-like" architecture results in a structure that has features organized to be specific for particular functionality, single kernel types, or a subset of kernel types. Thus, 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 Linux kernel.

Another consequence of this strategy 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

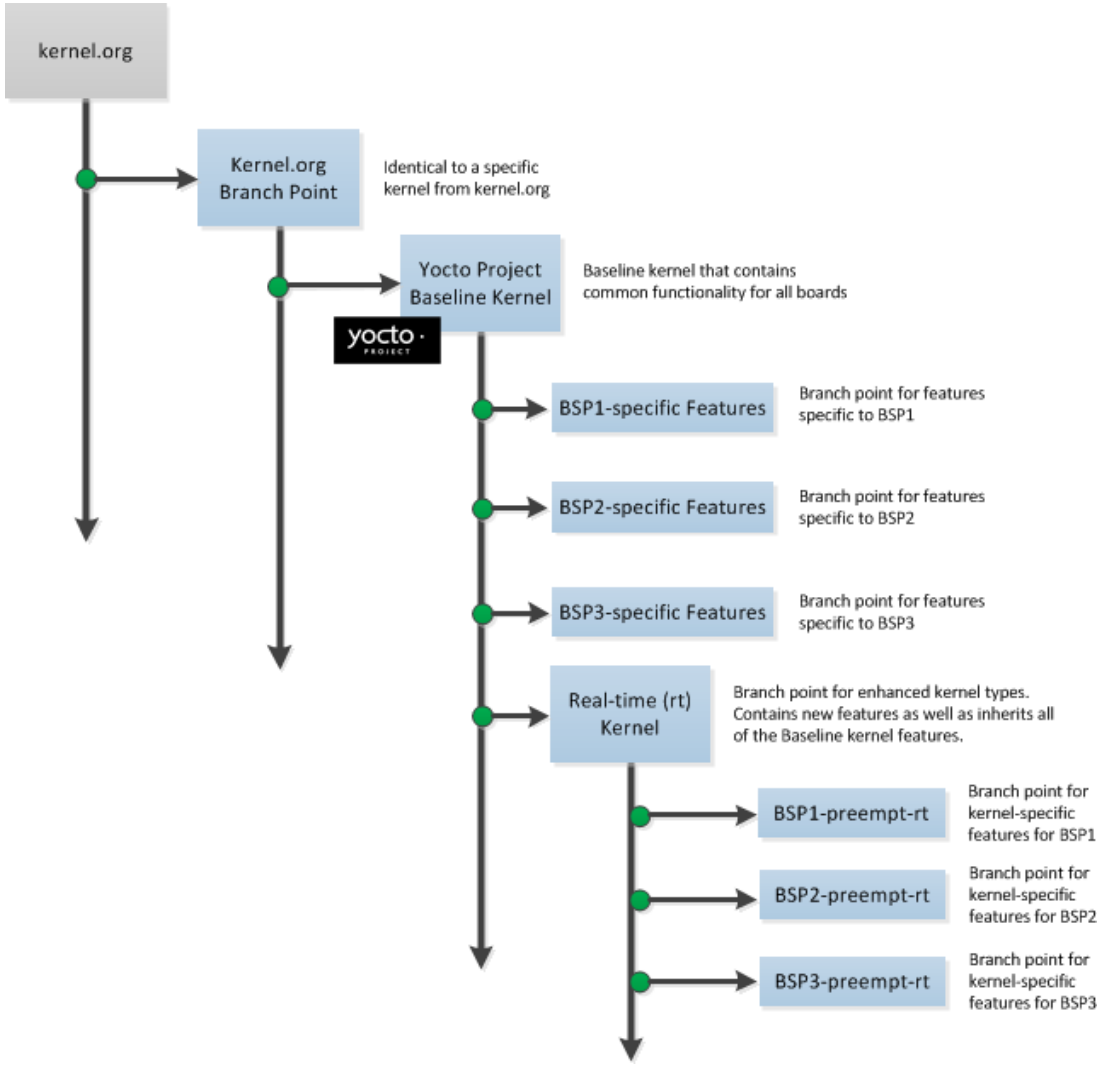
The Yocto Project team strives to place features in the tree such that features can be shared by all boards and kernel types where possible. However, during development cycles or when large features are merged, the team cannot always follow this practice. In those cases, the team uses isolated branches 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 kernel 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 in Git terms. The developer does not need to be aware of the existence of any other branches at all. Of course, value exists in the

having 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.

The following illustration shows the conceptual Yocto Linux kernel.



In the illustration, the "Kernel.org Branch Point" marks the specific spot (or Linux kernel release) from which the Yocto Linux kernel is created. From this point forward 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 along in the tree. Placing these common features in the tree this way means features do not have to be duplicated along individual branches of the tree structure.

From the "Yocto Project Baseline Kernel", branch points represent specific functionality for individual Board Support Packages (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 for a real-time Yocto Linux kernel.

In this example structure, the "Real-time (rt) Kernel" branch has common features for all real-time Yocto Linux 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 Yocto Linux kernel needed for any given set of requirements.

Note

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

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 Linux 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.3. Kernel Build File Hierarchy¶

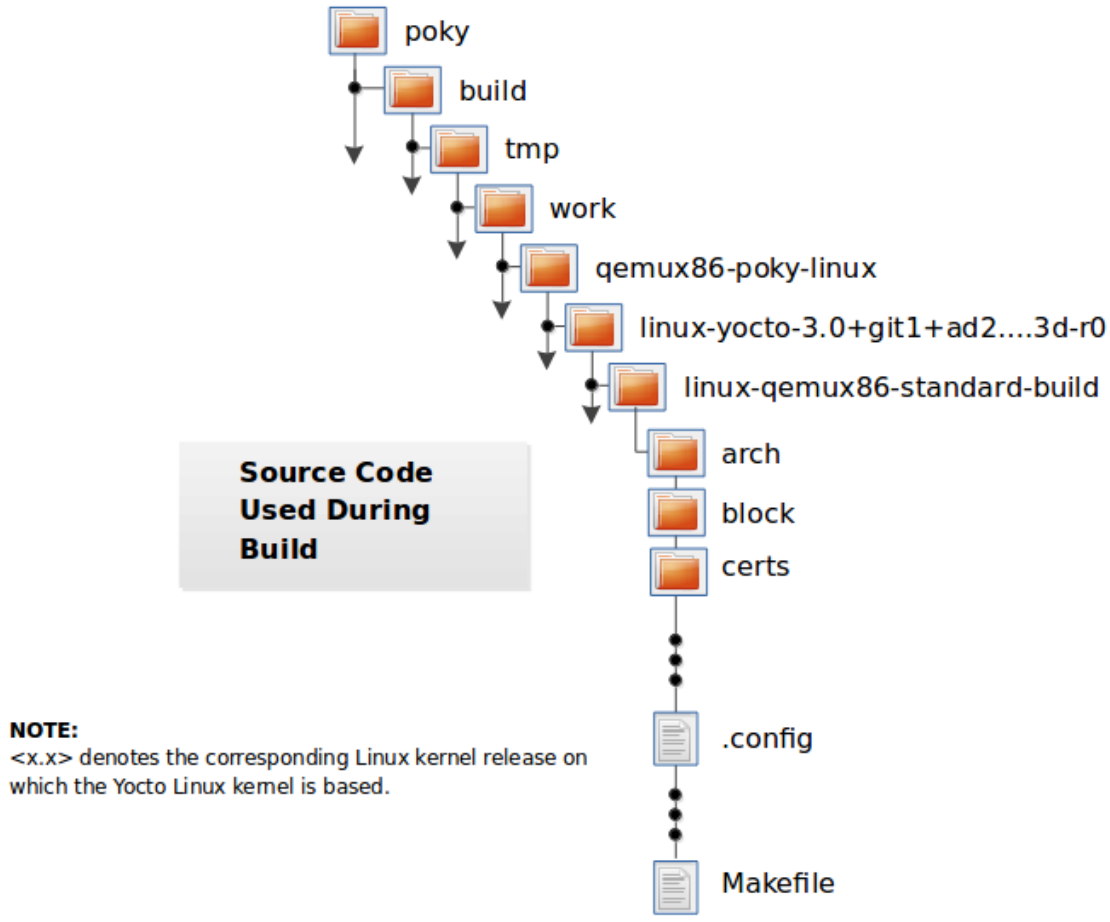
Upstream storage of all the available kernel source code is one thing, while representing and using the code on your host development system is another. Conceptually, you can think of the kernel source repositories as all the source files necessary for all the supported Yocto Linux kernels. As a developer,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

Kernel source code is available on your host system several different ways:

- ***Files Accessed While using `devtool`:*** `devtool`, which is available with the Yocto Project, is the preferred method by which to modify the kernel. See the "Kernel Modification Workflow" section.
- ***Cloned Repository:*** If you are working in the kernel all the time, you probably would want to set up your own local Git repository of the Yocto Linux kernel tree. For information on how to clone a Yocto Linux kernel Git repository, see the "Preparing the Build Host to Work on the Kernel" section.
- ***Temporary Source Files from a Build:*** If you just need to make some patches to the kernel using a traditional BitBake workflow (i.e. not using the `devtool`), you can access temporary kernel source files that were extracted and used during a kernel build.

The temporary kernel source files resulting from a build using BitBake have a particular hierarchy. When you build the kernel on your development system, all files needed for the build are taken from the source repositories pointed to by the `SRC_URI` variable and gathered in a temporary work area where they are subsequently used to create the unique kernel. Thus, in a sense, the process constructs a local source tree specific to your kernel from which to generate the new kernel image.

The following figure shows the temporary file structure created on your host system when you build the kernel using Bitbake. This Build Directory contains all the source files used during the build.



Again, for additional information on the Yocto Project kernel's architecture and its branching strategy, see the "Yocto Linux Kernel Architecture and Branching Strategies" section. You can also reference the "Using `devtool` to Patch the Kernel" and "Using Traditional Kernel Development to Patch the Kernel" sections for detailed example that modifies the kernel.

A.4. Determining Hardware and Non-Hardware Features for the Kernel Configuration Audit Phase¶

This section describes part of the kernel configuration audit phase that most developers can ignore. For general information on kernel configuration including `menuconfig`, `defconfig` files, and configuration fragments, see the "Configuring the Kernel" section.

During this part of the audit phase, the contents of the final `.config` file are compared against the fragments specified by the system. These fragments can be system fragments, distro fragments, or user-specified configuration elements. Regardless of their origin, the OpenEmbedded build system warns the user if a specific option is not included in the final kernel configuration.

By default, in order to not overwhelm the user with configuration warnings, the system only reports missing "hardware" options as they could result in a boot failure or indicate that important hardware is not available.

To determine whether or not a given option is "hardware" or "non-hardware", the kernel Metadata in `yocto-kernel-cache` contains files that classify individual or groups of options as either hardware or non-hardware. To better show this, consider a situation where the `yocto-kernel-cache` contains the following files:

```
yocto-kernel-cache/features/drm-psb/hardware.cfg
yocto-kernel-cache/features/kgdb/hardware.cfg
yocto-kernel-cache/ktypes/base/hardware.cfg
yocto-kernel-cache/bsp/mti-malta32/hardware.cfg
yocto-kernel-cache/bsp/fsl-mpc8315e-rdb/hardware.cfg
yocto-kernel-cache/bsp/qemu-ppc32/hardware.cfg
yocto-kernel-cache/bsp/qemuarma9/hardware.cfg
yocto-kernel-cache/bsp/mti-malta64/hardware.cfg
yocto-kernel-cache/bsp/arm-versatile-926ejs/hardware.cfg
yocto-kernel-cache/bsp/common-pc/hardware.cfg
yocto-kernel-cache/bsp/common-pc-64/hardware.cfg
yocto-kernel-cache/features/rfkill/non-hardware.cfg
yocto-kernel-cache/ktypes/base/non-hardware.cfg
yocto-kernel-cache/features/aufs/non-hardware.kcf
yocto-kernel-cache/features/ocf/non-hardware.kcf
yocto-kernel-cache/ktypes/base/non-hardware.kcf
yocto-kernel-cache/ktypes/base/hardware.kcf
yocto-kernel-cache/bsp/qemu-ppc32/hardware.kcf
```

The following list provides explanations for the various files:

- `hardware.kcf`: Specifies a list of kernel Kconfig files that contain hardware options only.
- `non-hardware.kcf`: Specifies a list of kernel Kconfig files that contain non-hardware options only.
- `hardware.cfg`: Specifies a list of kernel `CONFIG_` options that are hardware, regardless of whether or not they are within a Kconfig file specified by a hardware or non-hardware Kconfig file (i.e. `hardware.kcf` or `non-hardware.kcf`).
- `non-hardware.cfg`: Specifies a list of kernel `CONFIG_` options that are not hardware, regardless of whether or not they are within a Kconfig file specified by a hardware or non-hardware Kconfig file (i.e. `hardware.kcf` or `non-hardware.kcf`).

Here is a specific example using the `kernel-cache/bsp/mti-malta32/hardware.cfg`:

```
CONFIG_SERIAL_8250
CONFIG_SERIAL_8250_CONSOLE
CONFIG_SERIAL_8250_NR_UARTS
CONFIG_SERIAL_8250_PCI
CONFIG_SERIAL_CORE
CONFIG_SERIAL_CORE_CONSOLE
CONFIG_VGA_ARB
```

The kernel configuration audit automatically detects these files (hence the names must be exactly the ones discussed here), and uses them as inputs when generating warnings about the final `.config` file.

A user-specified kernel Metadata repository, or recipe space feature, can use these same files to classify options that are found within its `.cfg` files as hardware or non-hardware, to prevent the OpenEmbedded build system from producing an error or warning when an option is not in the final `.config` file.

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 Yocto Linux kernel repositories. These kernel repositories are found under the heading "Yocto Linux Kernel" at http://git.yoctoproject.org and are 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, configurations, branches, tags, and feature divisions found in a Yocto Linux kernel. Thus, the Yocto Project Linux kernel repository (or tree) and accompanying Metadata in the `yocto-kernel-cache` are built.

The existence of these repositories allow you to access and clone a particular Yocto Project Linux 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 Linux kernel in any clone of the Yocto Project Linux kernel source repository and `yocto-kernel-cache` Git trees. For example, the following commands clone the Yocto Project baseline Linux kernel that branches off `linux.org` version 4.12 and the `yocto-kernel-cache`, which contains stores of kernel Metadata:

```
$ git clone git://git.yoctoproject.org/linux-yocto-4.12
$ git clone git://git.yoctoproject.org/linux-kernel-cache
```

For more information on how to set up a local Git repository of the Yocto Project Linux kernel files, see the "Preparing the Build Host to Work on the Kernel" section.

Once you have cloned the kernel Git repository and the cache of Metadata on your local machine, you can discover the branches that are available in the repository using the following Git command:

```
$ git branch -a
```

Checking out a branch allows you to work with a particular Yocto Linux kernel. For example, the following commands check out the "standard/beagleboard" branch of the Yocto Linux kernel repository and the "yocto-4.12" branch of the `yocto-kernel-cache` repository:

```
$ cd ~/linux-yocto-4.12
$ git checkout -b my-kernel-4.12 remotes/origin/standard/beagleboard
$ cd ~/linux-kernel-cache
$ git checkout -b my-4.12-metadata remotes/origin/yocto-4.12
```

> **Note**
> Branches in the `yocto-kernel-cache` repository correspond to Yocto Linux kernel versions (e.g. "yocto-4.12", "yocto-4.10", "yocto-4.9", and so forth).

Once you have checked out and switched to appropriate branches, you can see a snapshot of all the kernel source files used to used to build that particular Yocto Linux kernel for a particular board.

To see the features and configurations for a particular Yocto Linux kernel, you need to examine the `yocto-kernel-cache` Git repository. As mentioned, branches in the `yocto-kernel-cache` repository correspond to Yocto Linux kernel versions (e.g. `yocto-4.12`). Branches contain descriptions in the form of `.scc` and `.cfg` files.

You should realize, however, that browsing your local `yocto-kernel-cache` 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 given the introduction of a new top-level kernel feature or BSP. The following actions effectively provide the Metadata and create the tree that includes the new feature, patch, or BSP:

1. ***Pass Feature to the OpenEmbedded Build System:*** A top-level kernel feature is passed to the kernel build subsystem. Normally, this feature is a BSP for a particular kernel type.

2. ***Locate Feature:*** 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 the `yocto-kernel-cache` repository organized under the "Yocto Linux Kernel" heading in the Yocto Project Source Repositories.
   - Areas pointed to by `SRC_URI` statements found in kernel recipes

   For a typical build, the target of the search is a feature description in an `.scc` file whose name follows this format (e.g. `beaglebone-standard.scc` and `beaglebone-preempt-rt.scc`):

   bsp_root_name-kernel_type.scc

3. ***Expand Feature:*** Once located, the feature description is either expanded into a simple script of actions, or into an existing equivalent script that is already part of the shipped kernel.
4. ***Append Extra Features:*** Extra features are appended to the top-level feature description. These features can come from the KERNEL_FEATURES variable in recipes.
5. ***Locate, Expand, and Append Each Feature:*** Each extra feature is located, expanded and appended to the script as described in step three.
6. ***Execute the Script:*** The script is executed to produce files `.scc` and `.cfg` files in appropriate directories of the `yocto-kernel-cache` repository. These files 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. ***Clone Base Repository:*** The base repository is cloned, and the actions listed in the `yocto-kernel-cache` directories are applied to the tree.
8. ***Perform Cleanup:*** The Git repositories are left with the desired branches checked out and any required branching, patching and tagging has been performed.

The kernel tree and cache are ready for developer consumption to be locally cloned, configured, and built into a Yocto Project kernel specific to some target hardware.

Notes

- The generated `yocto-kernel-cache` repository 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 at http://git.yoctoproject.org 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.
- The full kernel tree that you see on http://git.yoctoproject.org is generated through repeating the above steps for all valid BSPs. The end result is a branched, clean history tree that makes up the kernel for a given release. You can see the script (`kgit-scc`) responsible for this in the `yocto-kernel-tools` repository.
- The steps used to construct the full kernel tree are the same steps that BitBake uses when it builds a kernel image.

B.2. Build Strategy¶

Once you have cloned a Yocto Linux kernel repository and the cache repository (`yocto-kernel-cache`) onto your development system, you can consider the compilation phase of kernel development, which is 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 with Metadata exists in the `yocto-kernel-cache` repository. The branch is based on the Yocto Linux kernel version and has configurations and features grouped under the `yocto-kernel-cache/bsp` directory. For example, features and configurations for the BeagleBone Board assuming a `linux-yocto_4.12` kernel reside in the following area of the `yocto-kernel-cache` repository:

```
yocto-kernel-cache/bsp/beaglebone
```

Note

In the previous example, the "yocto-4.12" branch is checked out in the `yocto-kernel-cache` repository.

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 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?

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

C.1.2. How do I create configuration fragments?

Refer to the "Creating Configuration Fragments" 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 in Your Layer" section in the Yocto Project Development Tasks 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"
```