Yocto Project Board Support Package (BSP) Developer's Guide



Tom Zanussi, Intel Corporation **<tom.zanussi@intel.com>**
Richard Purdie, Linux Foundation
<richard.purdie@linuxfoundation.org>

by Tom Zanussi and Richard Purdie
Copyright © 2010-2013 Linux Foundation

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Non-Commercial-Share Alike 2.0 UK: England & Wales [http://creativecommons.org/licenses/by-nc-sa/2.0/uk/] as published by Creative Commons.

Note

Due to production processes, there could be differences between the Yocto Project documentation bundled in the release tarball and the Yocto Project Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/1.3.1/bsp-guide/bsp-guide.html] on the Yocto Project [http://www.yoctoproject.org] website. For the latest version of this manual, see the manual on the website.

Table of Contents

1. Board Support Packages (BSP) - Developer's Guide 1
    1.1. BSP Layers 1
    1.2. Example Filesystem Layout 1
        1.2.1. License Files 3
        1.2.2. README File 3
        1.2.3. README.sources File 3
        1.2.4. Pre-built User Binaries 3
        1.2.5. Layer Configuration File 3
        1.2.6. Hardware Configuration Options 4
        1.2.7. Miscellaneous Recipe Files 5
        1.2.8. Display Support Files 5
        1.2.9. Linux Kernel Configuration 5
    1.3. Requirements and Recommendations for Released BSPs 8
        1.3.1. Released BSP Requirements 8
        1.3.2. Released BSP Recommendations 9
    1.4. Customizing a Recipe for a BSP 10
    1.5. BSP Licensing Considerations 10
    1.6. Using the Yocto Project's BSP Tools 11
        1.6.1. Common Features 11
        1.6.2. Creating a new BSP Layer Using the yocto-bsp Script 13
        1.6.3. Managing Kernel Patches and Config Items with yocto-kernel 15

Chapter 1. Board Support Packages (BSP) - Developer's Guide

A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required. The BSP also lists any additional software components required in addition to a generic Linux software stack for both essential and optional platform features.

This guide presents information about BSP Layers, defines a structure for components so that BSPs follow a commonly understood layout, discusses how to customize a recipe for a BSP, addresses BSP licensing, and provides information that shows you how to create and manage a BSP Layer using two Yocto Project BSP Tools.

1.1. BSP Layers

The BSP consists of a file structure inside a base directory. Collectively, you can think of the base directory and the file structure as a BSP Layer. BSP Layers use the following naming convention:

```
meta-<bsp_name>
```

"bsp_name" is a placeholder for the machine or platform name.

The layer's base directory (`meta-<bsp_name>`) is the root of the BSP Layer. This root is what you add to the BBLAYERS [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-BBLAYERS] variable in the `conf/bblayers.conf` file found in the Build Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#build-directory]. Adding the root allows the OpenEmbedded build system to recognize the BSP definition and from it build an image. Here is an example:

```
BBLAYERS = " \
   /usr/local/src/yocto/meta \
   /usr/local/src/yocto/meta-yocto \
   /usr/local/src/yocto/meta-yocto-bsp \
   /usr/local/src/yocto/meta-<bsp_name> \
   "
```

Some BSPs require additional layers on top of the BSP's root layer in order to be functional. For these cases, you also need to add those layers to the BBLAYERS variable in order to build the BSP. You must also specify in the "Dependencies" section of the BSP's `README` file any requirements for additional layers and, preferably, any build instructions that might be contained elsewhere in the `README` file.

Some layers function as a layer to hold other BSP layers. An example of this type of layer is the `meta-intel` layer. The `meta-intel` layer contains over 10 individual BSP layers.

For more detailed information on layers, see the "Understanding and Creating Layers [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#understanding-and-creating-layers]" section of the Yocto Project Development Manual.

1.2. Example Filesystem Layout

Providing a common form allows end-users to understand and become familiar with the layout. A common format also encourages standardization of software support of hardware.

The proposed form does have elements that are specific to the OpenEmbedded build system. It is intended that this information can be used by other build systems besides the OpenEmbedded build system and that it will be simple to extract information and convert it to other formats if required. The

OpenEmbedded build system, through its standard layers mechanism, can directly accept the format described as a layer. The BSP captures all the hardware-specific details in one place in a standard format, which is useful for any person wishing to use the hardware platform regardless of the build system they are using.

The BSP specification does not include a build system or other tools - it is concerned with the hardware-specific components only. At the end-distribution point, you can ship the BSP combined with a build system and other tools. However, it is important to maintain the distinction that these are separate components that happen to be combined in certain end products.

Before looking at the common form for the file structure inside a BSP Layer, you should be aware that some requirements do exist in order for a BSP to be considered compliant with the Yocto Project. For that list of requirements, see the "Released BSP Requirements" section.

Below is the common form for the file structure inside a BSP Layer. While you can use this basic form for the standard, realize that the actual structures for specific BSPs could differ.

```
meta-<bsp_name>/
meta-<bsp_name>/<bsp_license_file>
meta-<bsp_name>/README
meta-<bsp_name>/README.sources
meta-<bsp_name>/binary/<bootable_images>
meta-<bsp_name>/conf/layer.conf
meta-<bsp_name>/conf/machine/*.conf
meta-<bsp_name>/recipes-bsp/*
meta-<bsp_name>/recipes-core/*
meta-<bsp_name>/recipes-graphics/*
meta-<bsp_name>/recipes-kernel/linux/linux-yocto_<kernel_rev>.bbappend
```

Below is an example of the Crown Bay BSP:

```
meta-crownbay/COPYING.MIT
meta-crownbay/README
meta-crownbay/README.sources
meta-crownbay/binary/
meta-crownbay/conf/
meta-crownbay/conf/layer.conf
meta-crownbay/conf/machine/
meta-crownbay/conf/machine/crownbay.conf
meta-crownbay/conf/machine/crownbay-noemgd.conf
meta-crownbay/recipes-bsp/
meta-crownbay/recipes-bsp/formfactor/
meta-crownbay/recipes-bsp/formfactor/formfactor_0.0.bbappend
meta-crownbay/recipes-bsp/formfactor/formfactor/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/machconfig
meta-crownbay/recipes-graphics/
meta-crownbay/recipes-graphics/xorg-xserver/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/xorg.conf
meta-crownbay/recipes-kernel/
meta-crownbay/recipes-kernel/linux/
meta-crownbay/recipes-kernel/linux/linux-yocto-rt_3.2.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto-rt_3.4.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_3.2.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_3.4.bbappend
```

The following sections describe each part of the proposed BSP format.

1.2.1. License Files

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/<bsp_license_file>
```

These optional files satisfy licensing requirements for the BSP. The type or types of files here can vary depending on the licensing requirements. For example, in the Crown Bay BSP all licensing requirements are handled with the `COPYING.MIT` file.

Licensing files can be MIT, BSD, GPLv*, and so forth. These files are recommended for the BSP but are optional and totally up to the BSP developer.

1.2.2. README File

You can find this file in the BSP Layer at:

```
meta-<bsp_name>/README
```

This file provides information on how to boot the live images that are optionally included in the `binary/` directory. The `README` file also provides special information needed for building the image.

At a minimum, the `README` file must contain a list of dependencies, such as the names of any other layers on which the BSP depends and the name of the BSP maintainer with his or her contact information.

1.2.3. README.sources File

You can find this file in the BSP Layer at:

```
meta-<bsp_name>/README.sources
```

This file provides information on where to locate the BSP source files. For example, information provides where to find the sources that comprise the images shipped with the BSP. Information is also included to help you find the metadata used to generate the images that ship with the BSP.

1.2.4. Pre-built User Binaries

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/binary/<bootable_images>
```

This optional area contains useful pre-built kernels and user-space filesystem images appropriate to the target system. This directory typically contains graphical (e.g. sato) and minimal live images when the BSP tarball has been created and made available in the Yocto Project [http://www.yoctoproject.org] website. You can use these kernels and images to get a system running and quickly get started on development tasks.

The exact types of binaries present are highly hardware-dependent. However, a README file should be present in the BSP Layer that explains how to use the kernels and images with the target hardware. If pre-built binaries are present, source code to meet licensing requirements must also exist in some form.

1.2.5. Layer Configuration File

You can find this file in the BSP Layer at:

```
meta-<bsp_name>/conf/layer.conf
```

The `conf/layer.conf` file identifies the file structure as a layer, identifies the contents of the layer, and contains information about how the build system should use it. Generally, a standard boilerplate file such as the following works. In the following example, you would replace "bsp" and "_bsp" with the actual name of the BSP (i.e. <bsp_name> from the example template).

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

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

BBFILE_COLLECTIONS += "bsp"
BBFILE_PATTERN_bsp := "^${LAYERDIR}/"
BBFILE_PRIORITY_bsp = "6"
```

To illustrate the string substitutions, here are the last three statements from the Crown Bay `conf/layer.conf` file:

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

This file simply makes BitBake aware of the recipes and configuration directories. The file must exist so that the OpenEmbedded build system can recognize the BSP.

1.2.6. Hardware Configuration Options

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/conf/machine/*.conf
```

The machine files bind together all the information contained elsewhere in the BSP into a format that the build system can understand. If the BSP supports multiple machines, multiple machine configuration files can be present. These filenames correspond to the values to which users have set the `MACHINE` [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-MACHINE] variable.

These files define things such as the kernel package to use (`PREFERRED_PROVIDER` [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-PREFERRED_PROVIDER] of virtual/kernel), the hardware drivers to include in different types of images, any special software components that are needed, any bootloader information, and also any special image format requirements.

Each BSP Layer requires at least one machine file. However, you can supply more than one file. For example, in the Crown Bay BSP shown earlier in this section, the `conf/machine` directory contains two configuration files: `crownbay.conf` and `crownbay-noemgd.conf`. The `crownbay.conf` file is used for the Crown Bay BSP that supports the Intel® Embedded Media and Graphics Driver (Intel® EMGD), while the `crownbay-noemgd.conf` file is used for the Crown Bay BSP that does not support the Intel® EMGD.

This `crownbay.conf` file could also include a hardware "tuning" file that is commonly used to define the package architecture and specify optimization flags, which are carefully chosen to give best performance on a given processor.

Tuning files are found in the `meta/conf/machine/include` directory within the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory].

Tuning files can also reside in the BSP Layer itself. For example, the `ia32-base.inc` file resides in the `meta-intel` BSP Layer in `conf/machine/include`.

To use an include file, you simply include them in the machine configuration file. For example, the Crown Bay BSP `crownbay.conf` has the following statements:

```
require conf/machine/include/tune-atom.inc
require conf/machine/include/ia32-base.inc
```

1.2.7. Miscellaneous Recipe Files

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/recipes-bsp/*
```

This optional directory contains miscellaneous recipe files for the BSP. Most notably would be the formfactor files. For example, in the Crown Bay BSP there is the `formfactor_0.0.bbappend` file, which is an append file used to augment the recipe that starts the build. Furthermore, there are machine-specific settings used during the build that are defined by the `machconfig` files. In the Crown Bay example, two `machconfig` files exist: one that supports the Intel® Embedded Media and Graphics Driver (Intel® EMGD) and one that does not:

```
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor_0.0.bbappend
```

Note

If a BSP does not have a formfactor entry, defaults are established according to the formfactor configuration file that is installed by the main formfactor recipe `meta/recipes-bsp/formfactor/formfactor_0.0.bb`, which is found in the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory].

1.2.8. Display Support Files

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/recipes-graphics/*
```

This optional directory contains recipes for the BSP if it has special requirements for graphics support. All files that are needed for the BSP to support a display are kept here. For example, the Crown Bay BSP contains two versions of the `xorg.conf` file. The version in `crownbay` builds a BSP that supports the Intel® Embedded Media Graphics Driver (EMGD), while the version in `crownbay-noemgd` builds a BSP that supports Video Electronics Standards Association (VESA) graphics only:

```
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/xorg.conf
```

1.2.9. Linux Kernel Configuration

You can find these files in the BSP Layer at:

```
meta-<bsp_name>/recipes-kernel/linux/linux-yocto_*.bbappend
```

These files append your specific changes to the main kernel recipe you are using.

For your BSP, you typically want to use an existing Yocto Project kernel recipe found in the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory] at `meta/recipes-kernel/linux`. You can append your specific changes to the kernel recipe by using a similarly named append file, which is located in the BSP Layer (e.g. the `meta-<bsp_name>/recipes-kernel/linux` directory).

Suppose you are using the `linux-yocto_3.4.bb` recipe to build the kernel. In other words, you have selected the kernel in your `<bsp_name>.conf` file by adding these types of statements:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto = "3.4%"
```

Note

When the preferred provider is assumed by default, the PREFERRED_PROVIDER statement does not appear in the `<bsp_name>.conf` file.

You would use the `linux-yocto_3.4.bbappend` file to append specific BSP settings to the kernel, thus configuring the kernel for your particular BSP.

As an example, look at the existing Crown Bay BSP. The append file used is:

```
meta-crownbay/recipes-kernel/linux/linux-yocto_3.4.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-intel` Git source repository.

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

COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "crownbay"
KBRANCH_crownbay  = "standard/crownbay"

COMPATIBLE_MACHINE_crownbay-noemgd = "crownbay-noemgd"
KMACHINE_crownbay-noemgd  = "crownbay"
KBRANCH_crownbay-noemgd  = "standard/crownbay"

SRCREV_machine_pn-linux-yocto_crownbay ?= "449f7f520350700858f21a5554b81cc8ad23267d"
SRCREV_meta_pn-linux-yocto_crownbay ?= "9e3bdb7344054264b750e53fbbb6394cc1c942ac"
SRCREV_emgd_pn-linux-yocto_crownbay ?= "86643bdd8cbad616a161ab91f51108cf0da827bc"

SRCREV_machine_pn-linux-yocto_crownbay-noemgd ?= "449f7f520350700858f21a5554b81cc8ad23267d"
SRCREV_meta_pn-linux-yocto_crownbay-noemgd ?= "9e3bdb7344054264b750e53fbbb6394cc1c942ac"

KSRC_linux_yocto_3_4 ?= "git.yoctoproject.org/linux-yocto-3.4.git"
SRC_URI_crownbay = "git://git.yoctoproject.org/linux-yocto-3.4.git;protocol=git;nocheckout=1
SRC_URI_crownbay-noemgd = "git://git.yoctoproject.org/linux-yocto-3.4.git;protocol=git;noche
```

This append file contains statements used to support the Crown Bay BSP for both Intel® EMGD and the VESA graphics. The build process, in this case, recognizes and uses only the statements that apply to the defined machine name - crownbay in this case. So, the applicable statements in the `linux-yocto_3.4.bbappend` file are follows:

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

```
COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "crownbay"
KBRANCH_crownbay  = "standard/crownbay"

SRCREV_machine_pn-linux-yocto_crownbay ?= "449f7f520350700858f21a5554b81cc8ad23267d"
SRCREV_meta_pn-linux-yocto_crownbay ?= "9e3bdb7344054264b750e53fbbb6394cc1c942ac"
SRCREV_emgd_pn-linux-yocto_crownbay ?= "86643bdd8cbad616a161ab91f51108cf0da827bc"
```

The append file defines `crownbay` as the `COMPATIBLE_MACHINE` [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-COMPATIBLE_MACHINE] and uses the `KMACHINE` [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-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` [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-KBRANCH] variable to ensure the build process uses the `standard/default/crownbay` kernel branch. Finally, the append file points to specific commits in the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory] Git repository and the `meta` Git repository branches to identify the exact kernel needed to build the Crown Bay BSP.

Note

For `crownbay`, a specific commit is also needed to point to the branch that supports EMGD graphics. At a minimum, every BSP points to the `machine` and `meta` commits.

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 a `SRC_URI` statement in the append file.

For example, suppose you had a 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"
```

The `FILESEXTRAPATHS` variable is in boilerplate form in the previous example in order to make it easy to do that. This variable must be in your layer or BitBake will not find the patches or configurations even if you have them in your `SRC_URI`. The `FILESEXTRAPATHS` variable enables the build process to find those configuration files.

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.

1.3. Requirements and Recommendations for Released BSPs

Certain requirements exist for a released BSP to be considered compliant with the Yocto Project. Additionally, a single recommendation also exists. This section describes the requirements and recommendation for released BSPs.

1.3.1. Released BSP Requirements

Before looking at BSP requirements, you should consider the following:

- The requirements here assume the BSP layer is a well-formed, "legal" layer that can be added to the Yocto Project. For guidelines on creating a layer that meets these base requirements, see the "BSP Layers" and the "Understanding and Creating Layers" [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#understanding-and-creating-layers] in the Yocto Project Development Manual.

- The requirements in this section apply regardless of how you ultimately package a BSP. You should consult the packaging and distribution guidelines for your specific release process. For an example of packaging and distribution requirements, see the Third Party BSP Release Process [https://wiki.yoctoproject.org/wiki/Third_Party_BSP_Release_Process] wiki page.

- The requirements for the BSP as it is made available to a developer are completely independent of the released form of the BSP. For example, the BSP metadata can be contained within a Git repository and could have a directory structure completely different from what appears in the officially released BSP layer.

- It is not required that specific packages or package modifications exist in the BSP layer, beyond the requirements for general compliance with the Yocto Project. For example, no requirement exists dictating that a specific kernel or kernel version be used in a given BSP.

Following are the requirements for a released BSP that conforms to the Yocto Project:

- Layer Name: The BSP must have a layer name that follows the Yocto Project standards. For information on BSP layer names, see the "BSP Layers" section.

- File System Layout: When possible, use the same directory names in your BSP layer as listed in the recipes.txt file. In particular, you should place recipes (.bb files) and recipe modifications (.bbappend files) into recipes-* subdirectories by functional area as outlined in recipes.txt. If you cannot find a category in recipes.txt to fit a particular recipe, you can make up your own recipe-* subdirectory. You can find recipes.txt in the meta directory of the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory], or in the OpenEmbedded Core Layer (openembedded-core) found at http://git.openembedded.org/openembedded-core/tree/meta.

  Within any particular recipes-* category, the layout should match what is found in the OpenEmbedded Core Git repository (openembedded-core) or the Source Directory (poky). In other words, make sure you place related files in appropriately related recipes-* subdirectories specific to the recipe's function, or within a subdirectory containing a set of closely-related recipes. The recipes themselves should follow the general guidelines for recipes used in the Yocto Project found in the Yocto Recipe and Patch Style Guide [https://wiki.yoctoproject.org/wiki/Recipe_%26_Patch_Style_Guide].

- License File: You must include a license file in the meta-<bsp_name> directory. This license covers the BSP metadata as a whole. You must specify which license to use since there is no default license if one is not specified. See the COPYING.MIT [http://git.yoctoproject.org/cgit.cgi/meta-intel/tree/meta-fri2/COPYING.MIT] file for the Fish River Island 2 BSP in the meta-fri2 BSP layer as an example.

- README File: You must include a README file in the meta-<bsp_name> directory. See the README [http://git.yoctoproject.org/cgit.cgi/meta-intel/tree/meta-fri2/README] file for the Fish River Island 2 BSP in the meta-fri2 BSP layer as an example.

  At a minimum, the README file should contain the following:

  - A brief description about the hardware the BSP targets.
  - A list of all the dependencies on which a BSP layer depends. These dependencies are typically a list of required layers needed to build the BSP. However, the dependencies should also contain information regarding any other dependencies the BSP might have.
  - Any required special licensing information. For example, this information includes information on special variables needed to satisfy a EULA, or instructions on information needed to build or distribute binaries built from the BSP metadata.
  - The name and contact information for the BSP layer maintainer. This is the person to whom patches and questions should be sent.
  - Instructions on how to build the BSP using the BSP layer.
  - Instructions on how to boot the BSP build from the BSP layer.
  - Instructions on how to boot the binary images contained in the /binary directory, if present.
  - Information on any known bugs or issues that users should know about when either building or booting the BSP binaries.
- README.sources File: You must include a README.sources in the meta-<bsp_name> directory. This file specifies exactly where you can find the sources used to generate the binary images contained in the /binary directory, if present. See the README.sources [http://git.yoctoproject.org/cgit.cgi/meta-intel/tree/meta-fri2/README.sources] file for the Fish River Island 2 BSP in the meta-fri2 BSP layer as an example.
- Layer Configuration File: You must include a conf/layer.conf in the meta-<bsp_name> directory. This file identifies the meta-<bsp_name> BSP layer as a layer to the build system.
- Machine Configuration File: You must include a conf/machine/<bsp_name>.conf in the meta-<bsp_name> directory. This configuration file defines a machine target that can be built using the BSP layer. Multiple machine configuration files define variations of machine configurations that are supported by the BSP. If a BSP supports multiple machine variations, you need to adequately describe each variation in the BSP README file. Do not use multiple machine configuration files to describe disparate hardware. If you do have very different targets, you should create separate BSP layers for each target.

  > **Note**
  > It is completely possible for a developer to structure the working repository as a conglomeration of unrelated BSP files, and to possibly generate specifically targeted 'release' BSPs from that directory using scripts or some other mechanism. Such considerations are outside the scope of this document.

1.3.2. Released BSP Recommendations

Following are recommendations for a released BSP that conforms to the Yocto Project:

- Bootable Images: BSP releases can contain one or more bootable images. Including bootable images allows users to easily try out the BSP on their own hardware.

  In some cases, it might not be convenient to include a bootable image. In this case, you might want to make two versions of the BSP available: one that contains binary images, and one that does not. The version that does not contain bootable images avoids unnecessary download times for users not interested in the images.

  If you need to distribute a BSP and include bootable images or build kernel and filesystems meant to allow users to boot the BSP for evaluation purposes, you should put the images and artifacts within a binary/ subdirectory located in the meta-<bsp_name> directory.

Note

If you do include a bootable image as part of the BSP and the image was built by software covered by the GPL or other open source licenses, it is your responsibility to understand and meet all licensing requirements, which could include distribution of source files.

- Use a Yocto Linux Kernel: Kernel recipes in the BSP should be based on a Yocto Linux kernel. Basing your recipes on these kernels reduces the costs for maintaining the BSP and increases its scalability. See the `Yocto Linux Kernel` category in the Source Repositories [http://git.yoctoproject.org/cgit.cgi] for these kernels.

1.4. Customizing a Recipe for a BSP

If you plan on customizing a recipe for a particular BSP, you need to do the following:

- Include within the BSP layer a `.bbappend` file for the modified recipe.
- Place the BSP-specific file in the BSP's recipe `.bbappend` file path under a directory named after the machine.

To better understand this, consider an example that customizes a recipe by adding a BSP-specific configuration file named `interfaces` to the `netbase_5.0.bb` recipe for machine "xyz". Do the following:

1. Edit the `netbase_4.47.bbappend` file so that it contains the following:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
PRINC := "${@int(PRINC) + 2}"
```

2. Create and place the new `interfaces` configuration file in the BSP's layer here:

```
meta-xyz/recipes-core/netbase/files/xyz/interfaces
```

1.5. BSP Licensing Considerations

In some cases, a BSP contains separately licensed Intellectual Property (IP) for a component or components. For these cases, you are required to accept the terms of a commercial or other type of license that requires some kind of explicit End User License Agreement (EULA). Once the license is accepted, the OpenEmbedded build system can then build and include the corresponding component in the final BSP image. If the BSP is available as a pre-built image, you can download the image after agreeing to the license or EULA.

You could find that some separately licensed components that are essential for normal operation of the system might not have an unencumbered (or free) substitute. Without these essential components, the system would be non-functional. Then again, you might find that other licensed components that are simply 'good-to-have' or purely elective do have an unencumbered, free replacement component that you can use rather than agreeing to the separately licensed component. Even for components essential to the system, you might find an unencumbered component that is not identical but will work as a less-capable version of the licensed version in the BSP recipe.

For cases where you can substitute a free component and still maintain the system's functionality, the Yocto Project website's BSP Download Page [http://www.yoctoproject.org/download/all?keys=&download_type=1&download_version=] makes available de-featured BSPs that are completely free of any IP encumbrances. For these cases, you can use the substitution directly and without any further licensing requirements. If present, these fully de-featured BSPs are named appropriately different as compared to the names of the respective encumbered BSPs. If available, these substitutions are your simplest and most preferred options. Use of these substitutions of course assumes the resulting functionality meets system requirements.

If however, a non-encumbered version is unavailable or it provides unsuitable functionality or quality, you can use an encumbered version.

A couple different methods exist within the OpenEmbedded build system to satisfy the licensing requirements for an encumbered BSP. The following list describes them in order of preference:

1. Use the LICENSE_FLAGS variable to define the recipes that have commercial or other types of specially-licensed packages: For each of those recipes, you can specify a matching license string in a local.conf variable named LICENSE_FLAGS_WHITELIST. Specifying the matching license string signifies that you agree to the license. Thus, the build system can build the corresponding recipe and include the component in the image. See the "Enabling Commercially Licensed Recipes [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#enabling-commercially-licensed-recipes]" section in the Yocto Project Reference Manual for details on how to use these variables.

   If you build as you normally would, without specifying any recipes in the LICENSE_FLAGS_WHITELIST, the build stops and provides you with the list of recipes that you have tried to include in the image that need entries in the LICENSE_FLAGS_WHITELIST. Once you enter the appropriate license flags into the whitelist, restart the build to continue where it left off. During the build, the prompt will not appear again since you have satisfied the requirement.

   Once the appropriate license flags are on the white list in the LICENSE_FLAGS_WHITELIST variable, you can build the encumbered image with no change at all to the normal build process.

2. Get a pre-built version of the BSP: You can get this type of BSP by visiting the Yocto Project website's Download [http://www.yoctoproject.org/download] page and clicking on "BSP Downloads". You can download BSP tarballs that contain proprietary components after agreeing to the licensing requirements of each of the individually encumbered packages as part of the download process. Obtaining the BSP this way allows you to access an encumbered image immediately after agreeing to the click-through license agreements presented by the website. Note that if you want to build the image yourself using the recipes contained within the BSP tarball, you will still need to create an appropriate LICENSE_FLAGS_WHITELIST to match the encumbered recipes in the BSP.

   ### Note
   Pre-compiled images are bundled with a time-limited kernel that runs for a predetermined amount of time (10 days) before it forces the system to reboot. This limitation is meant to discourage direct redistribution of the image. You must eventually rebuild the image if you want to remove this restriction.

1.6. Using the Yocto Project's BSP Tools

The Yocto Project includes a couple of tools that enable you to create a BSP layer from scratch and do basic configuration and maintenance of the kernel without ever looking at a metadata file. These tools are yocto-bsp and yocto-kernel, respectively.

The following sections describe the common location and help features as well as provide details for the yocto-bsp and yocto-kernel tools.

1.6.1. Common Features

Designed to have a command interface somewhat like Git [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#git], each tool is structured as a set of sub-commands under a top-level command. The top-level command (yocto-bsp or yocto-kernel) itself does nothing but invoke or provide help on the sub-commands it supports.

Both tools reside in the scripts/ subdirectory of the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory]. Consequently, to use the scripts, you must source the environment just as you would when invoking a build:

```
$ source oe-init-build-env [build_dir]
```

The most immediately useful function is to get help on both tools. The built-in help system makes it easy to drill down at any time and view the syntax required for any specific command. Simply enter the name of the command with the help switch:

```
$ yocto-bsp help
Usage:

 Create a customized Yocto BSP layer.

 usage: yocto-bsp [--version] [--help] COMMAND [ARGS]

 Current 'yocto-bsp' commands are:
    create            Create a new Yocto BSP
    list              List available values for options and BSP properties

 See 'yocto-bsp help COMMAND' for more information on a specific command.


Options:
  --version    show program's version number and exit
  -h, --help   show this help message and exit
  -D, --debug  output debug information
```

Similarly, entering just the name of a sub-command shows the detailed usage for that sub-command:

```
$ yocto-bsp create

Usage:

 Create a new Yocto BSP

 usage: yocto-bsp create <bsp-name> <karch> [-o <DIRNAME> | --outdir <DIRNAME>]
        [-i <JSON PROPERTY FILE> | --infile <JSON PROPERTY_FILE>]

 This command creates a Yocto BSP based on the specified parameters.
 The new BSP will be a new Yocto BSP layer contained by default within
 the top-level directory specified as 'meta-bsp-name'.  The -o option
 can be used to place the BSP layer in a directory with a different
 name and location.

 ...
```

For any sub-command, you can also use the word 'help' just before the sub-command to get more extensive documentation:

```
$ yocto-bsp help create

NAME
    yocto-bsp create - Create a new Yocto BSP

SYNOPSIS
    yocto-bsp create <bsp-name> <karch> [-o <DIRNAME> | --outdir <DIRNAME>]
        [-i <JSON PROPERTY FILE> | --infile <JSON PROPERTY_FILE>]

DESCRIPTION
    This command creates a Yocto BSP based on the specified
    parameters.  The new BSP will be a new Yocto BSP layer contained
    by default within the top-level directory specified as
    'meta-bsp-name'.  The -o option can be used to place the BSP layer
    in a directory with a different name and location.

    The value of the 'karch' parameter determines the set of files
    that will be generated for the BSP, along with the specific set of
    'properties' that will be used to fill out the BSP-specific
    portions of the BSP.  The possible values for the 'karch' paramter
```

```
    can be listed via 'yocto-bsp list karch'.

    ...
```

Now that you know where these two commands reside and how to access information on them, you should find it relatively straightforward to discover the commands necessary to create a BSP and perform basic kernel maintenance on that BSP using the tools. The next sections provide a concrete starting point to expand on a few points that might not be immediately obvious or that could use further explanation.

1.6.2. Creating a new BSP Layer Using the yocto-bsp Script

The `yocto-bsp` script creates a new BSP layer for any architecture supported by the Yocto Project, as well as QEMU versions of the same. The default mode of the script's operation is to prompt you for information needed to generate the BSP layer. For the current set of BSPs, the script prompts you for various important parameters such as:

- The kernel to use
- The branch of that kernel to use (or re-use)
- Whether or not to use X, and if so, which drivers to use
- Whether to turn on SMP
- Whether the BSP has a keyboard
- Whether the BSP has a touchscreen
- Remaining configurable items associated with the BSP

You use the `yocto-bsp create` sub-command to create a new BSP layer. This command requires you to specify a particular kernel architecture (`karch`) on which to base the BSP. Assuming you have sourced the environment, you can use the `yocto-bsp list karch` sub-command to list the architectures available for BSP creation as follows:

```
$ yocto-bsp list karch
Architectures available:
    qemu
    x86_64
    i386
    powerpc
    arm
    mips
```

The remainder of this section presents an example that uses `myarm` as the machine name and `qemu` as the machine architecture. Of the available architectures, `qemu` is the only architecture that causes the script to prompt you further for an actual architecture. In every other way, this architecture is representative of how creating a BSP for a 'real' machine would work. The reason the example uses this architecture is because it is an emulated architecture and can easily be followed without requiring actual hardware.

As the `yocto-bsp create` command runs, default values for the prompts appear in brackets. Pressing enter without supplying anything on the command line or pressing enter and providing an invalid response causes the script to accept the default value. Once the script completes, the new `meta-myarm` BSP layer is created in the current working directory. This example assumes you have source the oe-init-build-env and are currently in the top-level folder of the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory].

Following is the complete example:

```
$ yocto-bsp create myarm qemu
```

```
Which qemu architecture would you like to use? [default: i386]
 1) i386    (32-bit)
 2) x86_64  (64-bit)
 3) ARM     (32-bit)
 4) PowerPC (32-bit)
 5) MIPS    (32-bit)
3
Would you like to use the default (3.4) kernel? (y/n) [default: y]
Do you need a new machine branch for this BSP (the alternative is to re-use an existing bran
Getting branches from remote repo git://git.yoctoproject.org/linux-yocto-3.4.git...
Please choose a machine branch to base your new BSP branch on: [default: standard/base]
        1) standard/arm-versatile-926ejs
        2) standard/base
        3) standard/beagleboard
        4) standard/cedartrail
        5) standard/crownbay
        6) standard/emenlow
        7) standard/fishriver
        8) standard/fri2
        9) standard/fsl-mpc8315e-rdb
        10) standard/mti-malta32
        11) standard/mti-malta64
        12) standard/qemuppc
        13) standard/routerstationpro
        14) standard/sys940x
1
Would you like SMP support? (y/n) [default: y]
Does your BSP have a touchscreen? (y/n) [default: n]
Does your BSP have a keyboard? (y/n) [default: y]
New qemu BSP created in meta-myarm
```

Let's take a closer look at the example now:

1. For the `qemu` architecture, the script first prompts you for which emulated architecture to use. In the example, we use the `arm` architecture.

2. The script then prompts you for the kernel. The default 3.4 kernel is acceptable. So, the example accepts the default. If you enter 'n', the script prompts you to further enter the kernel you do want to use (e.g. 3.0, 3.2_preempt-rt, and so forth.).

3. Next, the script asks whether you would like to have a new branch created especially for your BSP in the local Linux Yocto Kernel [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#local-kernel-files] Git repository . If not, then the script re-uses an existing branch.

   In this example, the default (or 'yes') is accepted. Thus, a new branch is created for the BSP rather than using a common, shared branch. The new branch is the branch committed to for any patches you might later add. The reason a new branch is the default is that typically new BSPs do require BSP-specific patches. The tool thus assumes that most of time a new branch is required.

4. Regardless of which choice you make in the previous step, you are now given the opportunity to select a particular machine branch on which to base your new BSP-specific machine branch (or to re-use if you had elected to not create a new branch). Because this example is generating an `arm` BSP, the example uses #1 at the prompt, which selects the arm-versatile branch.

5. The remainder of the prompts are routine. Defaults are accepted for each.

6. By default, the script creates the new BSP Layer in the current working directory of the Source Directory [http://www.yoctoproject.org/docs/1.3.1/dev-manual/dev-manual.html#source-directory], which is `poky` in this case.

Once the BSP Layer is created, you must add it to your `bblayers.conf` file. Here is an example:

```
BBLAYERS = " \
   /usr/local/src/yocto/meta \
```

```
    /usr/local/src/yocto/meta-yocto \
    /usr/local/src/yocto/meta-yocto-bsp \
    /usr/local/src/yocto/meta-myarm \
    "
```

Adding the layer to this file allows the build system to build the BSP and the `yocto-kernel` tool to be able to find the layer and other metadata it needs on which to operate.

1.6.3. Managing Kernel Patches and Config Items with yocto-kernel

Assuming you have created a BSP Layer using `yocto-bsp` and you added it to your BBLAYERS [http://www.yoctoproject.org/docs/1.3.1/poky-ref-manual/poky-ref-manual.html#var-BBLAYERS] variable in the `bblayers.conf` file, you can now use the `yocto-kernel` script to add patches and configuration items to the BSP's kernel.

The `yocto-kernel` script allows you to add, remove, and list patches and kernel config settings to a BSP's kernel `.bbappend` file. All you need to do is use the appropriate sub-command. Recall that the easiest way to see exactly what sub-commands are available is to use the `yocto-kernel` built-in help as follows:

```
$ yocto-kernel
Usage:

 Modify and list Yocto BSP kernel config items and patches.

 usage: yocto-kernel [--version] [--help] COMMAND [ARGS]

 Current 'yocto-kernel' commands are:
   config list       List the modifiable set of bare kernel config options for a BSP
   config add        Add or modify bare kernel config options for a BSP
   config rm         Remove bare kernel config options from a BSP
   patch list        List the patches associated with a BSP
   patch add         Patch the Yocto kernel for a BSP
   patch rm          Remove patches from a BSP

 See 'yocto-kernel help COMMAND' for more information on a specific command.



Options:
  --version    show program's version number and exit
  -h, --help   show this help message and exit
  -D, --debug  output debug information
```

The `yocto-kernel patch add` sub-command allows you to add a patch to a BSP. The following example adds two patches to the `myarm` BSP:

```
$ yocto-kernel patch add myarm ~/test.patch
Added patches:
        test.patch

$ yocto-kernel patch add myarm ~/yocto-testmod.patch
Added patches:
        yocto-testmod.patch
```

Note

Although the previous example adds patches one at a time, it is possible to add multiple patches at the same time.

You can verify patches have been added by using the `yocto-kernel patch list` sub-command. Here is an example:

```
$ yocto-kernel patch list myarm
The current set of machine-specific patches for myarm is:
        1) test.patch
        2) yocto-testmod.patch
```

You can also use the `yocto-kernel` script to remove a patch using the `yocto-kernel patch rm` sub-command. Here is an example:

```
$ yocto-kernel patch rm myarm
Specify the patches to remove:
        1) test.patch
        2) yocto-testmod.patch
1
Removed patches:
        test.patch
```

Again, using the `yocto-kernel patch list` sub-command, you can verify that the patch was in fact removed:

```
$ yocto-kernel patch list myarm
The current set of machine-specific patches for myarm is:
        1) yocto-testmod.patch
```

In a completely similar way, you can use the `yocto-kernel config add` sub-command to add one or more kernel config item settings to a BSP. The following commands add a couple of config items to the `myarm` BSP:

```
$ yocto-kernel config add myarm CONFIG_MISC_DEVICES=y
Added items:
        CONFIG_MISC_DEVICES=y

$ yocto-kernel config add myarm KCONFIG_YOCTO_TESTMOD=y
Added items:
        CONFIG_YOCTO_TESTMOD=y
```

Note

Although the previous example adds config items one at a time, it is possible to add multiple config items at the same time.

You can list the config items now associated with the BSP. Doing so shows you the config items you added as well as others associated with the BSP:

```
$ yocto-kernel config list myarm
The current set of machine-specific kernel config items for myarm is:
        1) CONFIG_MISC_DEVICES=y
        2) CONFIG_YOCTO_TESTMOD=y
```

Finally, you can remove one or more config items using the `yocto-kernel config rm` sub-command in a manner completely analogous to `yocto-kernel patch rm`.