The 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

Note

Due to production processes, there could be differences between the Yocto Project documentation bundled in the release tarball and the Board Support Package (BSP) Developer's Guide [http://www.yoctoproject.org/docs/1.2/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

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 chapter (or document if you are reading the BSP Developer's Guide) talks 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/1.2/poky-ref-manual/poky-ref-manual.html#var-BBLAYERS] variable in the `conf/bblayers.conf` file found in the Yocto Project Build Directory [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-build-directory]. Adding the root allows the Yocto Project 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-<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 layers 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/1.2/dev-manual/dev-manual.html#understanding-and-creating-layers]" section of the Yocto Project Development Manual. You can also see the detailed examples in the appendices of The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html].

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 Yocto Project and OpenEmbedded build systems. It is intended that this information can be used by other systems besides Yocto Project and OpenEmbedded and that it will be simple to extract information and convert it to other formats

if required. Yocto Project, 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.

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-core/
meta-crownbay/recipes-core/tasks/
meta-crownbay/recipes-core/tasks/task-core-tools-profile.bbappend
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_3.0.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_2.6.37.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_3.0.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 Yocto Project layer, identifies the contents of the layer, and contains information about how Yocto Project 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 Yocto Project 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 Yocto Project 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/1.2/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/1.2/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 of the Yocto Project Files [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-files]. 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:

```
include conf/machine/include/tune-atom.inc
include 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 Yocto Project Files [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-files].

1.2.8. Core Recipe Files

You can find these files in the BSP Layer at:

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

This directory contains recipe files that are almost always necessary to build a useful, working Linux image. Thus, the term "core" is used to group these recipes. For example, in the Crown Bay BSP there is the `task-core-tools-profile.bbappend` file, which is an append file used to recommend that the SystemTap [http://sourceware.org/systemtap/wiki] package be included as a package when the image is built.

1.2.9. 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
```

1.2.10. 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 kernel you are using.

For your BSP, you typically want to use an existing Yocto Project kernel found in the Yocto Project Files [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-files] 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 the BSP uses the `linux-yocto_3.0.bb` kernel, which is the preferred kernel to use for developing a new BSP using the Yocto Project. In other words, you have selected the kernel in your `<bsp_name>.conf` file by adding the following statements:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto = "3.0%"
```

You would use the `linux-yocto_3.0.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_3.0.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  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay += " cfg/smp.scc"

COMPATIBLE_MACHINE_crownbay-noemgd = "crownbay-noemgd"
KMACHINE_crownbay-noemgd  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay-noemgd += " cfg/smp.scc"

SRCREV_machine_pn-linux-yocto_crownbay ?= "63c65842a3a74e4bd3128004ac29b5639f16433f"
SRCREV_meta_pn-linux-yocto_crownbay ?= "59314a3523e360796419d76d78c6f7d8c5ef2593"

SRCREV_machine_pn-linux-yocto_crownbay-noemgd ?= "63c65842a3a74e4bd3128004ac29b5639f16433f"
SRCREV_meta_pn-linux-yocto_crownbay-noemgd ?= "59314a3523e360796419d76d78c6f7d8c5ef2593"
```

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_3.0.bbappend` file are follows:

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

```
COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay += " cfg/smp.scc"

SRCREV_machine_pn-linux-yocto_crownbay ?= "63c65842a3a74e4bd3128004ac29b5639f16433f"
SRCREV_meta_pn-linux-yocto_crownbay ?= "59314a3523e360796419d76d78c6f7d8c5ef2593"
```

The append file defines `crownbay` as the compatible machine and defines the `KMACHINE`. The file also points to some configuration fragments to use by setting the `KERNEL_FEATURES` variable. The location for the configuration fragments is the kernel tree itself in the Yocto Project Build Directory [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-build-directory] under `linux/meta`. Finally, the append file points to the specific commits in the Yocto Project Files [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-files] Git repository and the `meta` Git repository branches to identify the exact kernel needed to build the Crown Bay 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 append file and having the same name as the kernel. With all these conditions met simply reference those files in a `SRC_URI` statement in the append file.

For example, suppose you had a set of configuration options in a file called `myconfig`. If you put that file inside a directory named `/linux-yocto` and then added a `SRC_URI` statement such as the following to the append file, those configuration options will be picked up and applied when the kernel is built.

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

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 \
       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 Yocto kernel `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. For an example showing how to change the BSP configuration, see the "Changing the BSP Configuration [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#changing-the-bsp-configuration]" section in the Yocto Project Development Manual. For a better understanding of working with a local clone of the kernel repository and a local bare clone of the kernel, see the "Modifying the Kernel Source Code [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#modifying-the-kernel-source-code]" section also in the Yocto Project Development Manual.

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. 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_4.47.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
```

1.4. 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 Yocto Project 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 Yocto Project 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 Yocto Project 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/1.2/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 whitelisted 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.

1.5. 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 Yocto Project metadata file. These tools are yocto-bsp and yocto-kernel, respectively.

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

1.5.1. Common Features

Designed to have a command interface somewhat like Git [http://www.yoctoproject.org/docs/1.2/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 Yocto Project Files [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-files]. 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, or the command along with help to display a list of the available sub-commands. Here is an example:

```
$ yocto-bsp
```

```
$ yocto-bsp help

Usage:

Create a customized Yocto BSP layer.

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

The most commonly used '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.
```

```
...

NOTE: Once created, you should add your new layer to your
bblayers.conf file in order for it to be subsequently seen and
modified by the yocto-kernel tool.

NOTE for x86- and x86_64-based BSPs: The generated BSP assumes the
presence of the of the meta-intel layer, so you should also have a
meta-intel layer present and added to your bblayers.conf as well.
```

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 maintainence 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.

1.5.2. 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:

- which kernel to use
- which 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
- any 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 architecture 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:
    arm
    powerpc
    i386
    mips
    x86_64
    qemu
```

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 requireing 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.

Following is the complete example:

```
$ yocto-bsp create myarm qemu
Which qemu architecture would you like to use? [default: x86]
        1) common 32-bit x86
        2) common 64-bit x86
        3) common 32-bit ARM
        4) common 32-bit PowerPC
        5) common 32-bit MIPS
3
Would you like to use the default (3.2) kernel? (Y/n)
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-3.2...
Please choose a machine branch to base this BSP on => [default: standard/default/common-pc]
        1) base
        2) standard/base
        3) standard/default/arm-versatile-926ejs
        4) standard/default/base
        5) standard/default/beagleboard
        6) standard/default/cedartrailbsp (copy).xml
        7) standard/default/common-pc-64/base
        8) standard/default/common-pc-64/jasperforest
        9) standard/default/common-pc-64/romley
        10) standard/default/common-pc-64/sugarbay
        11) standard/default/common-pc/atom-pc
        12) standard/default/common-pc/base
        13) standard/default/crownbay
        14) standard/default/emenlow
        15) standard/default/fishriver
        16) standard/default/fri2
        17) standard/default/fsl-mpc8315e-rdb
        18) standard/default/mti-malta32-be
        19) standard/default/mti-malta32-le
        20) standard/default/preempt-rt
        21) standard/default/qemu-ppc32
        22) standard/default/routerstationpro
        23) standard/preempt-rt/base
        24) standard/preempt-rt/qemu-ppc32
        25) standard/preempt-rt/routerstationpro
        26) standard/tiny
3
Do you need SMP support? (Y/n)
Does your BSP have a touchscreen? (y/N)
Does your BSP have a keyboard? (Y/n)
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 kernel is 3.2 and 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, etc.).

3. Next, the script asks whether you would like to have a new branch created especially for your BSPin the local Linux Yocto Kernel [http://www.yoctoproject.org/docs/1.2/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.

Note

In the current implementation, creation or re-use of a branch does not actually matter. The reason is because the generated BSPs assume that patches and configurations live in recipe-space, which is something that can be done with or without a dedicated branch. Generated BSPs, however, are different. This difference becomes significant once the tool's 'publish' functionality is implemented.

4. Regardless of which choice is made 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 on (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 #3 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 Yocto Project Build Directory [http://www.yoctoproject.org/docs/1.2/dev-manual/dev-manual.html#yocto-project-build-directory].

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-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.

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

Assuming you have created a Yocto Project BSP Layer using `yocto-bsp` and you added it to your BBLAYERS [http://www.yoctoproject.org/docs/1.2/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 Yocto Project 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]

The most commonly used '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.
```

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`.