1.Adding a New Machine in Yocto:

1). Adding the Machine Configuration File

To add a new machine, you need to add a new machine configuration file to the layer's `conf/machine` directory. This configuration file provides details about the device you are adding.

The OpenEmbedded build system uses the root name of the machine configuration file to reference the new machine. For example, given a machine configuration file named `crownbay.conf`, the build system recognizes the machine as "crownbay".

The most important variables you must set in your machine configuration file are as follows:

- `TARGET_ARCH` (e.g. "arm")
- `PREFERRED_PROVIDER_virtual/kernel`
- `MACHINE_FEATURES` (e.g. "apm screen wifi")

You might also need these variables:

- `SERIAL_CONSOLES` (e.g. "115200;ttyS0 115200;ttyS1")
- `KERNEL_IMAGETYPE` (e.g. "zImage")
- `IMAGE_FSTYPES` (e.g. "tar.gz jffs2")

PREFERRED_PROVIDER: If multiple recipes provide an item, this variable determines which recipe should be given preference. You should always suffix the variable with the name of the provided item, and you should set it to the `PN` of the recipe to which you want to give precedence. Some examples:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_PROVIDER_virtual/xserver = "xserver-xf86"
PREFERRED_PROVIDER_virtual/libgl ?= "mesa"
```

MACHINE_FEATURES: Specifies a list of hardware features the `MACHINE` is capable of supporting:

1). **DISTRO_FEATURES**: The software support you want in your distribution for various features. For example, specifying "x11" in `DISTRO_FEATURES`, causes every piece of software built for the target that can optionally support X11 to have its X11 support enabled.

This list only represents features as shipped with the Yocto Project metadata:

- ***alsa:*** Include ALSA support (OSS compatibility kernel modules installed if available).
- ***bluetooth:*** Include bluetooth support (integrated BT only).
- ***cramfs:*** Include CramFS support.
- ***directfb:*** Include DirectFB support.
- ***ext2:*** Include tools for supporting for devices with internal HDD/Microdrive for storing files (instead of Flash only devices).
- ***ipsec:*** Include IPSec support.
- ***ipv6:*** Include IPv6 support.
- ***irda:*** Include IrDA support.
- ***keyboard:*** Include keyboard support (e.g. keymaps will be loaded during boot).
- ***nfs:*** Include NFS client support (for mounting NFS exports on device).
- ***opengl:*** Include the Open Graphics Library, which is a cross-language, multi-platform application programming interface used for rendering two and three-dimensional graphics.
- ***pci:*** Include PCI bus support.
- ***pcmcia:*** Include PCMCIA/CompactFlash support.
- ***ppp:*** Include PPP dialup support.
- ***smbfs:*** Include SMB networks client support (for mounting Samba/Microsoft Windows shares on device).
- ***systemd:*** Include support for this `init` manager, which is a full replacement of for `init` with parallel starting of services, reduced shell overhead, and other features. This `init` manager is used by many distributions.
- ***usbgadget:*** Include USB Gadget Device support (for USB networking/serial/storage).
- ***usbhost:*** Include USB Host support (allows to connect external keyboard, mouse, storage, network etc).
- ***wayland:*** Include the Wayland display server protocol and the library that supports it.
- ***wifi:*** Include WiFi support (integrated only).
- ***x11:*** Include the X server and libraries.

2). **COMBINED_FEATURES**: Provides a list of hardware features that are enabled in both `MACHINE_FEATURES` and `DISTRO_FEATURES`. This list of features contains features that make sense to be controlled both at the machine and distribution configuration level. For example, the "bluetooth" feature requires hardware support but should also be optional at the distribution level, in case the hardware supports Bluetooth but you do not ever intend to use it.

3). **IMAGE_FEATURES**: The primary list of features to include in an image. Typically, you configure this variable in an image recipe. Although you can use this variable from your `local.conf` file, which is found in the Build Directory, best practices dictate that you do not.

So a good method for customizing your image is to enable or disable high-level image features by using the `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` variables. Although the functions for both variables are nearly equivalent, best practices dictate using `IMAGE_FEATURES` from within a recipe and using `EXTRA_IMAGE_FEATURES` from within your `local.conf` file, which is found in the Build Directory.

To understand how these features work, the best reference is `meta/classes/core-image.bbclass`. In summary, the file looks at the contents of the `IMAGE_FEATURES` variable and then maps those contents into a set of package groups. Based on this information, the build system automatically adds the appropriate packages to the `IMAGE_INSTALL` variable.

Use the `EXTRA_IMAGE_FEATURES` variable from within your local configuration file. Using a separate area from which to enable features with this variable helps you avoid overwriting the features in the image recipe that are enabled with `IMAGE_FEATURES`. The value of `EXTRA_IMAGE_FEATURES` is added to `IMAGE_FEATURES` within `meta/conf/bitbake.conf`.

Consider an example that selects the SSH server. The Yocto Project ships with two SSH servers you can use with your images: Dropbear and OpenSSH. Dropbear is a minimal SSH server appropriate for resource-constrained environments, while OpenSSH is a well-known standard SSH server implementation. By default, the `core-image-minimal` image does not contain an SSH server.

You can customize your image and change these defaults. Edit the `IMAGE_FEATURES` variable in your recipe or use the `EXTRA_IMAGE_FEATURES` in your `local.conf` file so that it configures the image you are working with to include `ssh-server-dropbear` or `ssh-server-openssh`.

Current list of `IMAGE_FEATURES` contains the following:

- ***dbg-pkgs:*** Installs debug symbol packages for all packages installed in a given image.

- ***dev-pkgs:*** Installs development packages (headers and extra library links) for all packages installed in a given image.
- ***doc-pkgs:*** Installs documentation packages for all packages installed in a given image.
- ***nfs-server:*** Installs an NFS server.
- ***read-only-rootfs:*** Creates an image whose root filesystem is read-only.
- ***splash:*** Enables showing a splash screen during boot. By default, this screen is provided by `psplash`, which does allow customization. If you prefer to use an alternative splash screen package, you can do so by setting the `SPLASH` variable to a different package name (or names) within the image recipe or at the distro configuration level.
- ***ssh-server-dropbear:*** Installs the Dropbear minimal SSH server.
- ***ssh-server-openssh:*** Installs the OpenSSH SSH server, which is more full-featured than Dropbear. Note that if both the OpenSSH SSH server and the Dropbear minimal SSH server are present in `IMAGE_FEATURES`, then OpenSSH will take precedence and Dropbear will not be installed.
- ***staticdev-pkgs:*** Installs static development packages (i.e. static libraries containing `*.a` files) for all packages installed in a given image.
- ***tools-debug:*** Installs debugging tools such as `strace` and `gdb`.
- ***tools-sdk:*** Installs a full SDK that runs on the device.
- ***tools-testapps:*** Installs device testing tools (e.g. touchscreen debugging).
- ***x11:*** Installs the X server
- ***x11-base:*** Installs the X server with a minimal environment.

SERIAL_CONSOLE: The speed and device for the serial port used to attach the serial console. This variable is given to the kernel as the "console" parameter and after booting occurs `getty` is started on that port so remote login is possible.

KERNEL_IMAGETYPE: The type of kernel to build for a device, usually set by the machine configuration files and defaults to "zImage". This variable is used when building the kernel and is passed to `make` as the target to build.

IMAGE_FSTYPES: Formats of root filesystem images that you want to have created.

2). Adding a Kernel for the Machine

The OpenEmbedded build system needs to be able to build a kernel for the machine. You need to either create a new kernel recipe for this machine, or extend an existing recipe. You can find several kernel examples in the Source Directory at `meta/recipes-kernel/linux` that you can use as references.

If you are creating a new recipe, normal recipe-writing rules apply for setting up a `SRC_URI`. Thus, you need to specify any necessary patches and set `S` to point at the source code. You need to create a `configure` task that configures the unpacked kernel with a defconfig. You can do this by using a `make defconfig` command or, more commonly, by copying in a suitable `defconfig` file and then running `make oldconfig`. By making use of `inherit kernel` and potentially some of the `linux-*.inc` files, most other functionality is centralized and the the defaults of the class normally work well.

If you are extending an existing kernel, it is usually a matter of adding a suitable defconfig file. The file needs to be added into a location similar to defconfig files used for other machines in a given kernel. A possible way to do this is by listing the file in the `SRC_URI` and adding the machine to the expression in `COMPATIBLE_MACHINE`:

```
COMPATIBLE_MACHINE = '(qemux86|qemumips)'
```

3). Adding a Formfactor Configuration File

A formfactor configuration file provides information about the target hardware for which the image is being built and information that the build system cannot obtain from other sources such as the kernel. Some examples of information contained in a formfactor configuration file include framebuffer orientation, whether or not the system has a keyboard, the positioning of the keyboard in relation to the screen, and the screen resolution.

The build system uses reasonable defaults in most cases. However, if customization is necessary, you need to create a `machconfig` file in the `meta/recipes-bsp/formfactor/files` directory. This directory contains directories for specific machines such as `qemuarm` and `qemux86`.

Following is an example for qemuarm:

```
HAVE_TOUCHSCREEN=1
```

```
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
#DISPLAY_WIDTH_PIXELS=640
#DISPLAY_HEIGHT_PIXELS=480
#DISPLAY_BPP=16
DISPLAY_DPI=150
DISPLAY_SUBPIXEL_ORDER=vrgb
```

2. Working with libraries:

1). Including Static Library Files

The `PACKAGES` and `FILES_*` variables in the `meta/conf/bitbake.conf` configuration file define how files installed by the `do_install` task are packaged. By default, the `PACKAGES` variable contains `${PN}-staticdev`, which includes all static library files.

Following is part of the BitBake configuration file. You can see where the static library files are defined:

```
PACKAGES = "${PN}-dbg ${PN} ${PN}-doc ${PN}-dev ${PN}-staticdev
${PN}-locale"
    PACKAGES_DYNAMIC = "${PN}-locale-*"
    FILES = ""

    FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS}
\
              ${sysconfdir} ${sharedstatedir} ${localstatedir} \
              ${base_bindir}/* ${base_sbindir}/* \
              ${base_libdir}/*${SOLIBS} \
              ${datadir}/${BPN} ${libdir}/${BPN}/* \
              ${datadir}/pixmaps ${datadir}/applications \
              ${datadir}/idl ${datadir}/omf ${datadir}/sounds \
              ${libdir}/bonobo/servers"

    FILES_${PN}-doc = "${docdir} ${mandir} ${infodir} ${datadir}/gtk-doc \
              ${datadir}/gnome/help"
    SECTION_${PN}-doc = "doc"

    FILES_${PN}-dev = "${includedir} ${libdir}/lib*${SOLIBSDEV} ${libdir}/*.la \
                ${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
                ${datadir}/aclocal ${base_libdir}/*.o"
    SECTION_${PN}-dev = "devel"
    ALLOW_EMPTY_${PN}-dev = "1"
    RDEPENDS_${PN}-dev = "${PN} (= ${EXTENDPKGV})"

    FILES_${PN}-staticdev = "${libdir}/*.a ${base_libdir}/*.a"
    SECTION_${PN}-staticdev = "devel"
    RDEPENDS_${PN}-staticdev = "${PN}-dev (= ${EXTENDPKGV})"
```

2). Installing Multiple Versions of the Same Library

Situations can exist where you need to install and use multiple versions of the same library on the same system at the same time. These situations almost always exist when a library API changes and you have multiple pieces of software that depend on the separate versions of the library. To accommodate these situations, you can install multiple versions of the same library in parallel on the same system.

The process is straightforward as long as the libraries use proper versioning. With properly versioned libraries, all you need to do to individually specify the libraries is create separate, appropriately named recipes where the `PN` part of the name includes a portion that differentiates each library version (e.g.the major part of the version number).

Thus, instead of having a single recipe that loads one version of a library (e.g. `clutter`), you provide multiple recipes that result in different versions of the libraries you want. As an example, the following two recipes would allow two separate versions of the `clutter` library to co-exist on the same system:

```
clutter-1.6_1.6.20.bb
clutter-1.8_1.8.4.bb
```

Additionally, if you have other recipes that depend on a given library, you need to use the `DEPENDS` variable to create the dependency. Continuing with the same example, if you want to have a recipe depend on the 1.8 version of the `clutter` library, use the following in your recipe:

```
DEPENDS = "clutter-1.8"
```

3. About KERNEL_FEATURES:

Includes additional metadata from the Yocto Project kernel Git repository. In the OpenEmbedded build system, the default Board Support Packages (BSPs)

Metadata is provided through the `KMACHINE` and `KBRANCH` variables. You can use the `KERNEL_FEATURES` variable to further add metadata for all BSPs.

The metadata you add through this variable includes config fragments and features descriptions, which usually includes patches as well as config fragments. You typically override the `KERNEL_FEATURES` variable for a specific machine. In this way, you can provide validated, but optional, sets of kernel configurations and features.

For example, the following adds `netfilter` to all the Yocto Project kernels and adds sound support to the `qemux86` machine:

```
# Add netfilter to all linux-yocto kernels
KERNEL_FEATURES="features/netfilter"

# Add sound support to the qemux86 machine
KERNEL_FEATURES_append_qemux86=" cfg/sound"
```

KERNEL_MODULE_AUTOLOAD:

Lists kernel modules that need to be auto-loaded during boot.

You can use the `KERNEL_MODULE_AUTOLOAD` variable anywhere that it can be recognized by the kernel recipe or by an out-of-tree kernel module recipe (e.g. a machine configuration file, a distribution configuration file, an append file for the recipe, or the recipe itself).

Specify it as follows:

```
KERNEL_MODULE_AUTOLOAD += "module_name1 module_name2 module_name3"
```

Including `KERNEL_MODULE_AUTOLOAD` causes the OpenEmbedded build system to populate the `/etc/modules-load.d/modname.conf` file with the list of modules to be auto-loaded on boot. The modules appear one-per-line in the file. Here is an example of the most common use case:

```
KERNEL_MODULE_AUTOLOAD += "module_name"
```

This is used when a custom module is compiled as loadable module, instead of building into the kernel.

4. About LINUX_VERSION and LINUX_VERSION_EXTENSION:\

1). LINUX_VERSION

The Linux version from `kernel.org` on which the Linux kernel image being built using the OpenEmbedded build system is based. You define this variable in the kernel recipe. For example, the `linux-yocto-3.4.bb` kernel recipe found in `meta/recipes-kernel/linux` defines the variables as follows:

```
LINUX_VERSION ?= "3.4.24"
```

The `LINUX_VERSION` variable is used to define `PV` for the recipe:

```
PV = "${LINUX_VERSION}+git${SRCPV}"
```

2). LINUX_VERSION_EXTENSION:

A string extension compiled into the version string of the Linux kernel built with the OpenEmbedded build system. You define this variable in the kernel recipe.

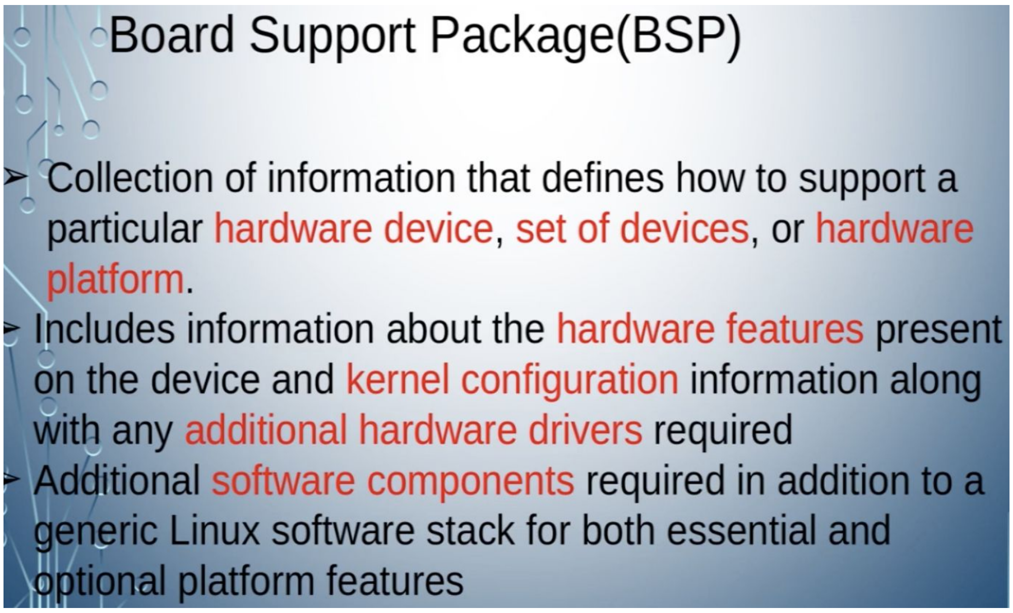
Defining this variable essentially sets `CONFIG_LOCALVERSION`, which is visible through the `uname` command. Here is an example that shows the extension assuming it was set as previously shown:

```
$ uname -r
3.7.0-rc8-custom
```

Note: when we are adding a new machine, we need to clean all previous states for the kernel built for another machine using ‘bitbake -c cleanall virtual/kernel’.

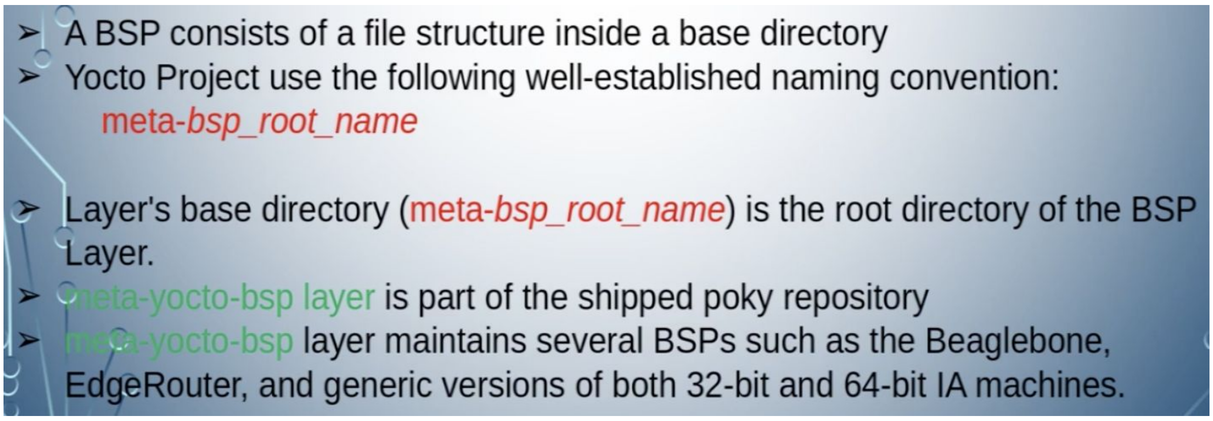
Section 6: BSP development

1. Introduction of BSP components



Board Support Package(BSP)

- ➢ Collection of information that defines how to support a particular hardware device, set of devices, or hardware platform.
- ➢ Includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required
- ➢ Additional software components required in addition to a generic Linux software stack for both essential and optional platform features

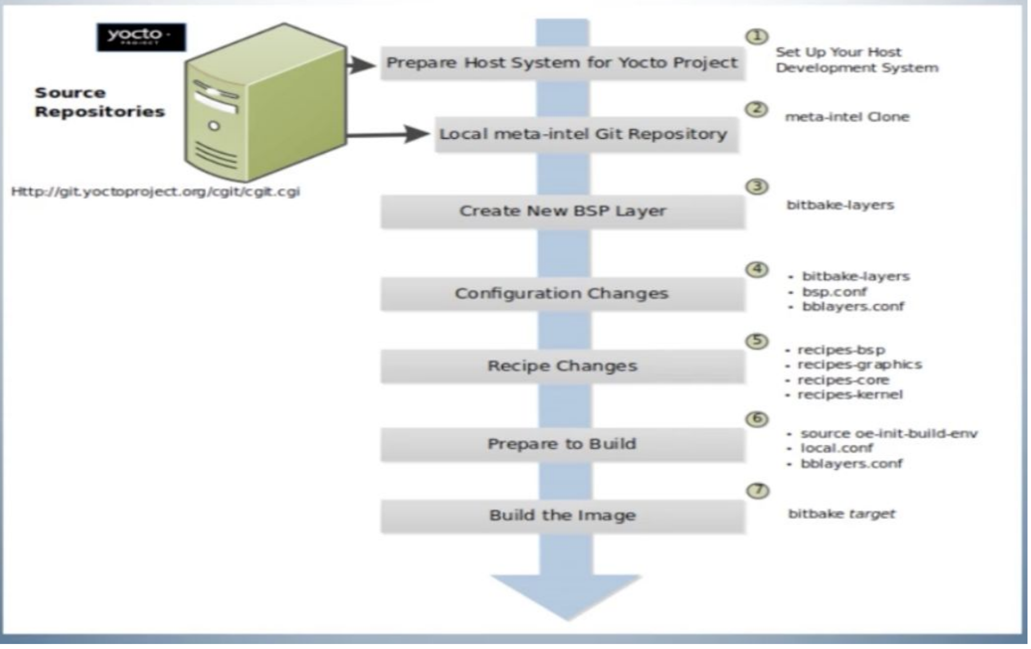


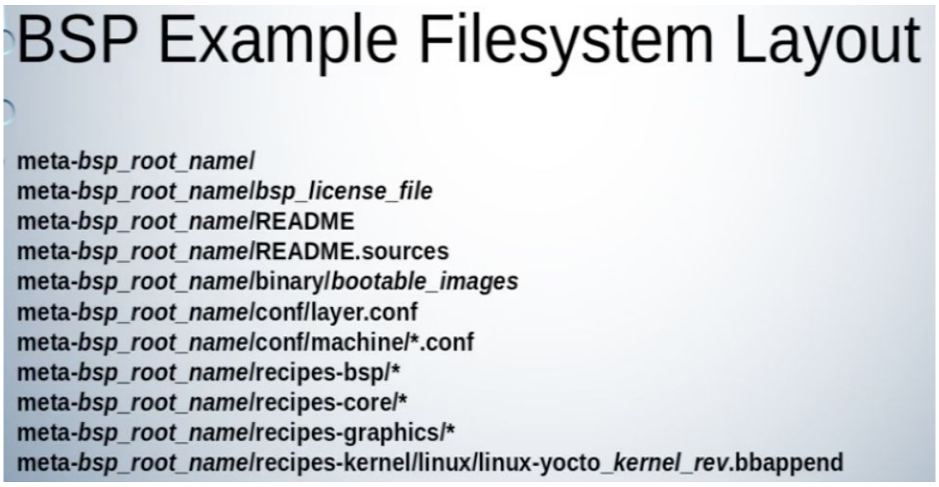
- ➢ A BSP consists of a file structure inside a base directory
- ➢ Yocto Project use the following well-established naming convention:

  meta-*bsp_root_name*

- ➢ Layer's base directory (meta-*bsp_root_name*) is the root directory of the BSP Layer.
- ➢ meta-yocto-bsp layer is part of the shipped poky repository
- ➢ meta-yocto-bsp layer maintains several BSPs such as the Beaglebone, EdgeRouter, and generic versions of both 32-bit and 64-bit IA machines.

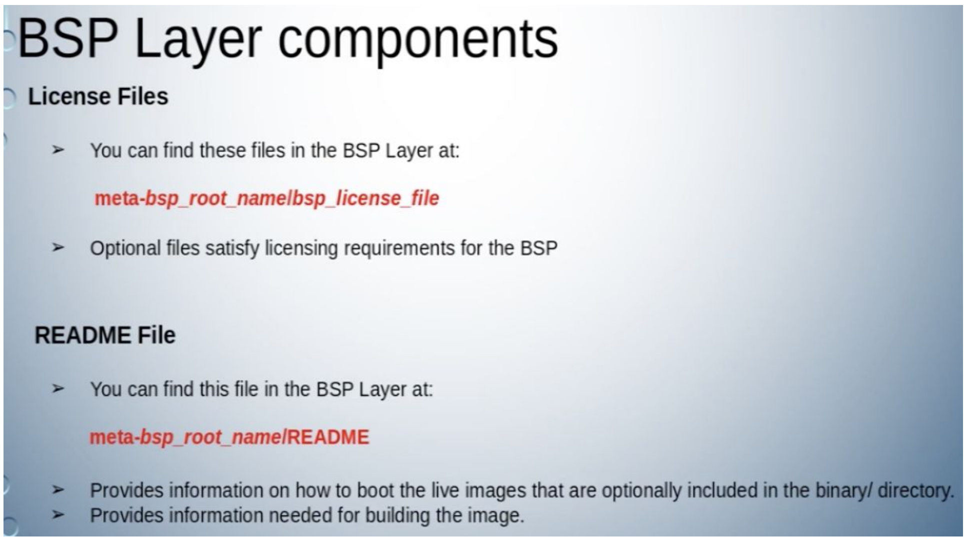
BSP Creation General Workflow


yocto
Source Repositories
Http://git.yoctoproject.org/cgit/cgit.cgi
Prepare Host System for Yocto Project
Local meta-intel Git Repository
Create New BSP Layer
Configuration Changes
Recipe Changes
Prepare to Build
Build the Image
① Set Up Your Host Development System
② meta-intel Clone
③ bitbake-layers
④ • bitbake-layers • bsp.conf • bblayers.conf
⑤ • recipes-bsp • recipes-graphics • recipes-core • recipes-kernel
⑥ • source oe-init-build-env • local.conf • bblayers.conf
⑦ bitbake target



BSP Example Filesystem Layout

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



BSP Layer components

License Files

➢ You can find these files in the BSP Layer at:

meta-*bsp_root_name*/*bsp_license_file*

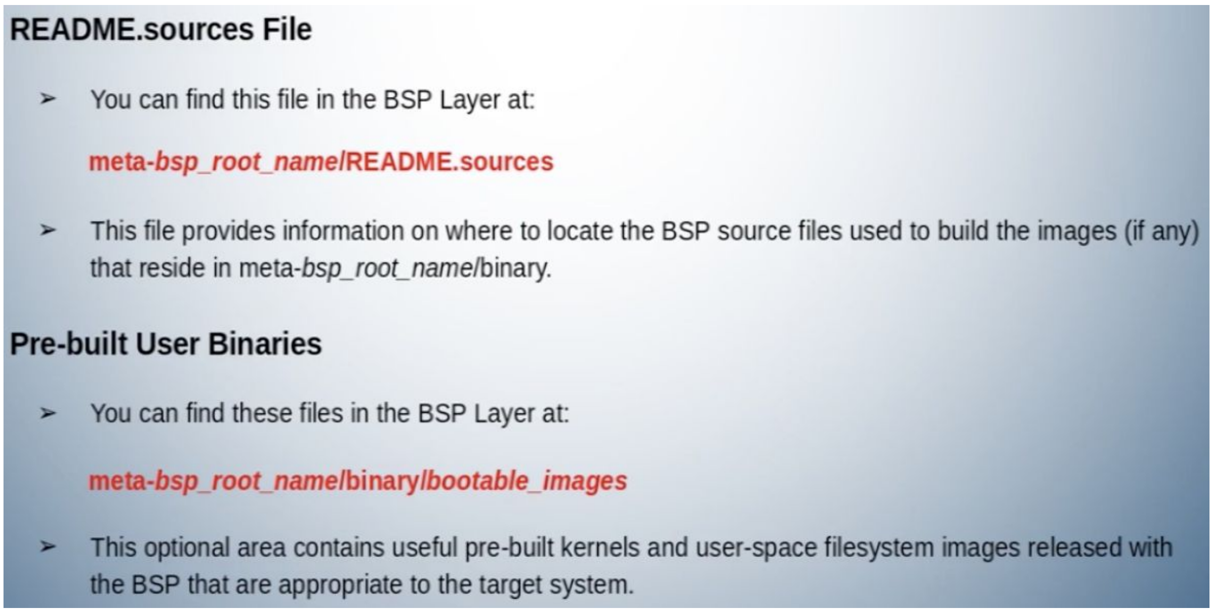
➢ Optional files satisfy licensing requirements for the BSP

README File

➢ You can find this file in the BSP Layer at:

meta-*bsp_root_name*/README

➢ Provides information on how to boot the live images that are optionally included in the binary/ directory.
➢ Provides information needed for building the image.



README.sources File

- You can find this file in the BSP Layer at:

  meta-*bsp_root_name*/README.sources

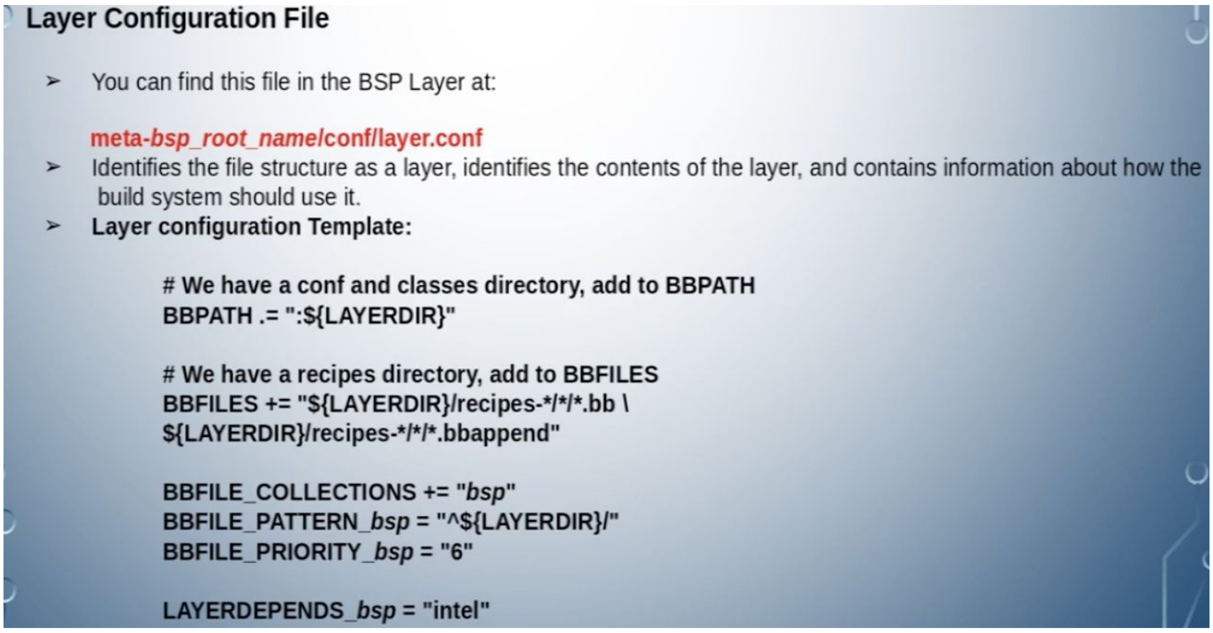
- This file provides information on where to locate the BSP source files used to build the images (if any) that reside in meta-*bsp_root_name*/binary.

Pre-built User Binaries

- You can find these files in the BSP Layer at:

  meta-*bsp_root_name*/binary/*bootable_images*

- This optional area contains useful pre-built kernels and user-space filesystem images released with the BSP that are appropriate to the target system.



Layer Configuration File

- You can find this file in the BSP Layer at:

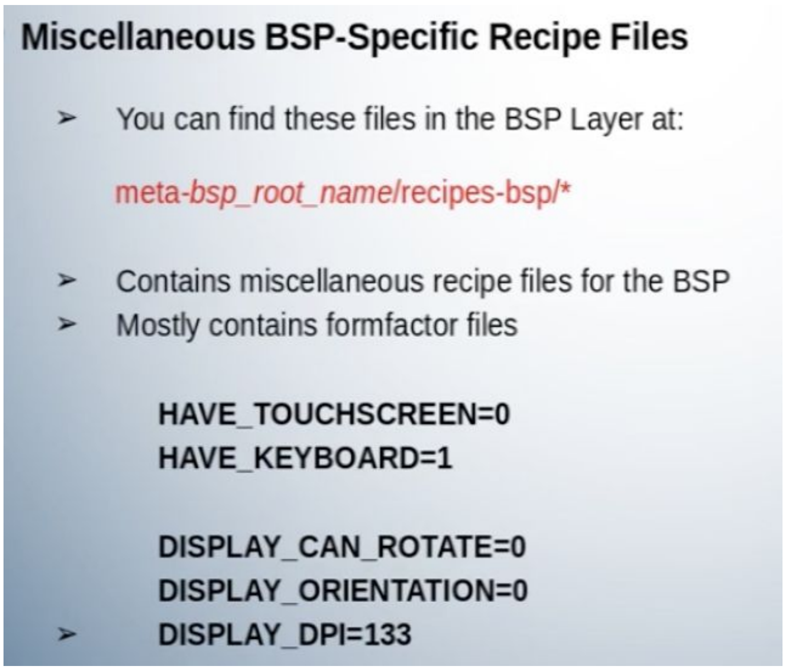
  meta-*bsp_root_name*/conf/layer.conf
- Identifies the file structure as a layer, identifies the contents of the layer, and contains information about how the build system should use it.
- **Layer configuration Template:**

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

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

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

LAYERDEPENDS_bsp = "intel"
```

BBFILES: List of recipe files used by BitBake to build software.
BBLAYERS: Lists the layers to enable during the build. This variable is defined in the `bblayers.conf` configuration file in the Build Directory. Here is an example:

```
BBLAYERS = " \
      /home/scottrif/poky/meta \
      /home/scottrif/poky/meta-yocto \
      /home/scottrif/poky/meta-yocto-bsp \
      /home/scottrif/poky/meta-mykernel \
      "
```

This example enables four layers, one of which is a custom, user-defined layer named `meta-mykernel`.



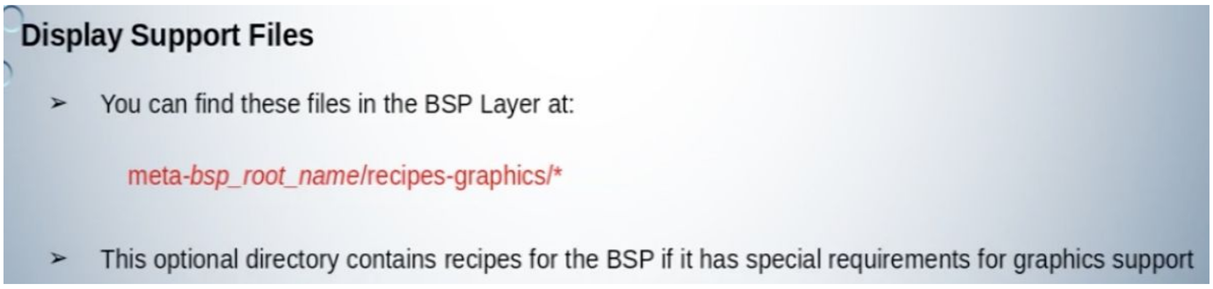
Miscellaneous BSP-Specific Recipe Files

- You can find these files in the BSP Layer at:

  meta-*bsp_root_name*/recipes-bsp/*

- Contains miscellaneous recipe files for the BSP
- Mostly contains formfactor files

```
HAVE_TOUCHSCREEN=0
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
DISPLAY_DPI=133
```

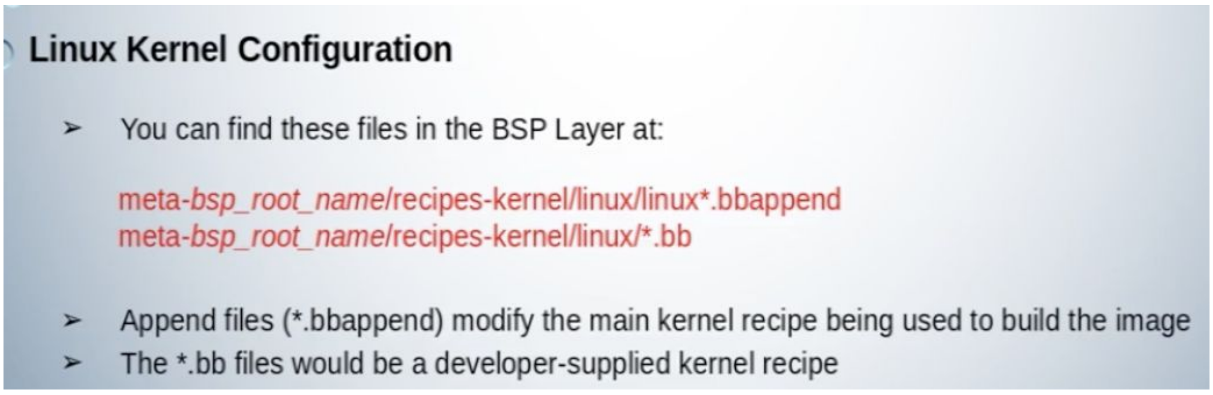


Display Support Files

- You can find these files in the BSP Layer at:

  meta-*bsp_root_name*/recipes-graphics/*

- This optional directory contains recipes for the BSP if it has special requirements for graphics support



Linux Kernel Configuration

- You can find these files in the BSP Layer at:

  meta-*bsp_root_name*/recipes-kernel/linux/linux*.bbappend
  meta-*bsp_root_name*/recipes-kernel/linux/*.bb

- Append files (*.bbappend) modify the main kernel recipe being used to build the image
- The *.bb files would be a developer-supplied kernel recipe

If we want to build an image for a particular hardware, we need to run 'source oe-init-build-env build-machine' command where build-machine directory is the build directory for the target machine. Inside the conf/ directory of this directory, we specify what layers we want to enable and what machines we want to build for. In case there is failure about connectivity, we also need to specify ‘CONNECTIVITY’ directive:

```
#check connectivity using google
CONNECTIVITY_CHECK_URIS = "https://www.google.com/"
#skip connectivity checks
CONNECTIVITY_CHECK_URIS = ""
```

https://stackoverflow.com/questions/52395512/why-does-bitbake-error-if-it-cant-find-www-example-com

LAYERSERIES_COMPAT:

Lists the versions of the OpenEmbedded-Core for which a layer is compatible. Using the `LAYERSERIES_COMPAT` variable allows the layer maintainer to indicate which combinations of the layer and OE-Core can be expected to work.

To specify the OE-Core versions for which a layer is compatible, use this variable in your layer's `conf/layer.conf` configuration file. For the list, use the Yocto Project Release Name (e.g. dunfell). To specify multiple OE-Core versions for the layer, use a space-separated list:

```
LAYERSERIES_COMPAT_layer_root_name = "dunfell zeus"
```

Setting LAYERSERIES_COMPAT is required by the Yocto Project Compatible version 2 standard. The OpenEmbedded build system produces a warning if the variable is not set for any given layer.

TUNE_FEATURES:

Features used to "tune" a compiler for optimal use given a specific processor. The features are defined within the tune files and allow arguments (i.e. `TUNE_*ARGS`) to be dynamically generated based on the features.

The OpenEmbedded build system verifies the features to be sure they are not conflicting and that they are supported.

The BitBake configuration file (`meta/conf/bitbake.conf`) defines `TUNE_FEATURES` as follows:

```
TUNE_FEATURES ??= "${TUNE_FEATURES_tune-${DEFAULTTUNE}}"
```

AVAILTUNES:

The list of defined CPU and Application Binary Interface (ABI) tunings (i.e. "tunes") available for use by the OpenEmbedded build system.

The list simply presents the tunes that are available. Not all tunes may be compatible with a particular machine configuration.

To add a tune to the list, be sure to append it with spaces using the "+=" BitBake operator. Do not simply replace the list by using the "=" operator.

DEFAULTTUNE:

The default CPU and Application Binary Interface (ABI) tunings (i.e. the "tune") used by the OpenEmbedded build system. The `DEFAULTTUNE` helps define `TUNE_FEATURES`.

The default tune is either implicitly or explicitly set by the machine (`MACHINE`). However, you can override the setting using available tunes as defined with `AVAILTUNES`.

KERNEL_DEVICETREE:

Specifies the name of the generated Linux kernel device tree (i.e. the `.dtb`) file. In order to use this variable, the `kernel-devicetree` class must be inherited.

IMAGE_BASENAME:

The base name of image output files. This variable defaults to the recipe name (`${PN}`).

IMAGE_BOOT_FILES:

A space-separated list of files installed into the boot partition when preparing an image using the Wic tool with the `bootimg-partition` source plugin. By default, the files are installed under the same name as the source files. To change the installed name, separate it from the original name with a semi-colon (;). Here are two examples:

```
IMAGE_BOOT_FILES = "u-boot.img uImage;kernel"
IMAGE_BOOT_FILES = "u-boot.${UBOOT_SUFFIX} ${KERNEL_IMAGETYPE}"
```

To install files into a directory within the target location, pass its name after a semi-colon (;). Here are two examples:

```
IMAGE_BOOT_FILES = "bcm2835-bootfiles/*"
IMAGE_BOOT_FILES = "bcm2835-bootfiles/*;boot/"
```

The first example installs all files from `${DEPLOY_DIR_IMAGE}/bcm2835-bootfiles` into the root of the target partition. The second example installs the same files into a `boot` directory within the target partition.

From wic directory:

```
part /boot --source bootimg-partition ...
```

It uses bootimg-partition.py wic plugin to generate `/boot` partition. It copies every files defined by `IMAGE_BOOT_FILES` variable to /boot.

`DEPLOY_DIR_IMAGE:`

Points to the area that the OpenEmbedded build system uses to place images and other associated output files that are ready to be deployed onto the target machine. The directory is machine-specific as it contains the `${MACHINE}` name. By default, this directory resides within the Build Directory as `${DEPLOY_DIR}/images/${MACHINE}/`.

UBOOT_ENTRYPOINT:

Specifies the entry point for the U-Boot image. During U-Boot image creation, the `UBOOT_ENTRYPOINT` variable is passed as a command-line parameter to the `uboot-mkimage` utility.

UBOOT_LOADADDRESS:

Specifies the load address for the U-Boot image. During U-Boot image creation, the `UBOOT_LOADADDRESS` variable is passed as a command-line parameter to the `uboot-mkimage` utility.

UBOOT_MACHINE:

Specifies the value passed on the `make` command line when building a U-Boot image. The value indicates the target platform configuration. You typically set this variable from the machine configuration file (i.e. `conf/machine/`*`machine_name`*`.conf`).

Load address vs entry point:

The load address is where in ram the image is uncompressed and stored and the entry point is the address where execution is transferred after the image has been written to ram.

mkimage -a <load address> -e <entry point>

So when the kernel gets loaded at <load address> which is in ram, it starts execution from <entry point> .

SPL_BINARY:

The file type for the Secondary Program Loader (SPL). Some devices use an SPL from which to boot (e.g. the BeagleBone development board). For such cases, you can declare the file type of the SPL binary in the `u-boot.inc` include file, which is used in the U-Boot recipe.

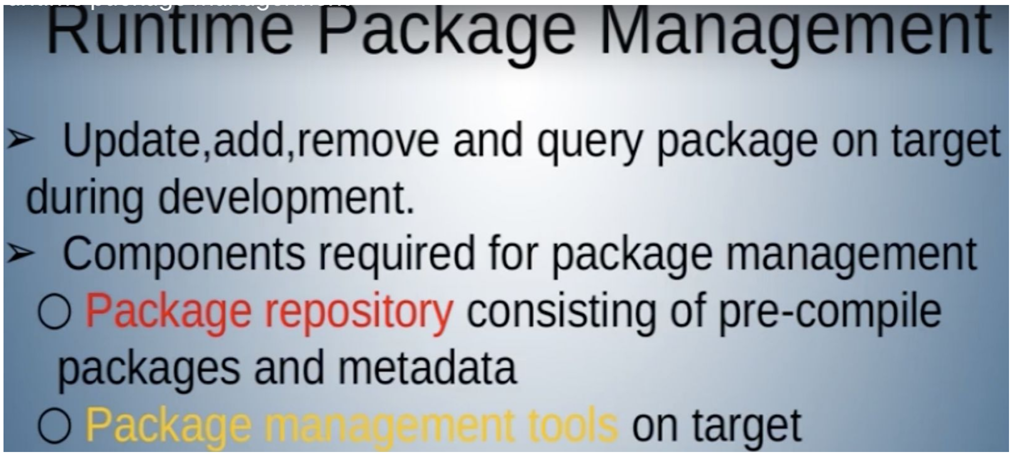
The SPL file type is set to "null" by default in the `u-boot.inc` file as follows:

```
# Some versions of u-boot build an SPL (Second Program Loader) image that
    # should be packaged along with the u-boot binary as well as placed in the
    # deploy directory.  For those versions they can set the following variables
    # to allow packaging the SPL.
    SPL_BINARY ?= ""
    SPL_BINARYNAME ?= "${@os.path.basename(d.getVar("SPL_BINARY"))}"
    SPL_IMAGE ?= "${SPL_BINARYNAME}-${MACHINE}-${PV}-${PR}"
    SPL_SYMLINK ?= "${SPL_BINARYNAME}-${MACHINE}"
```

Example machine conf file using all these directives:

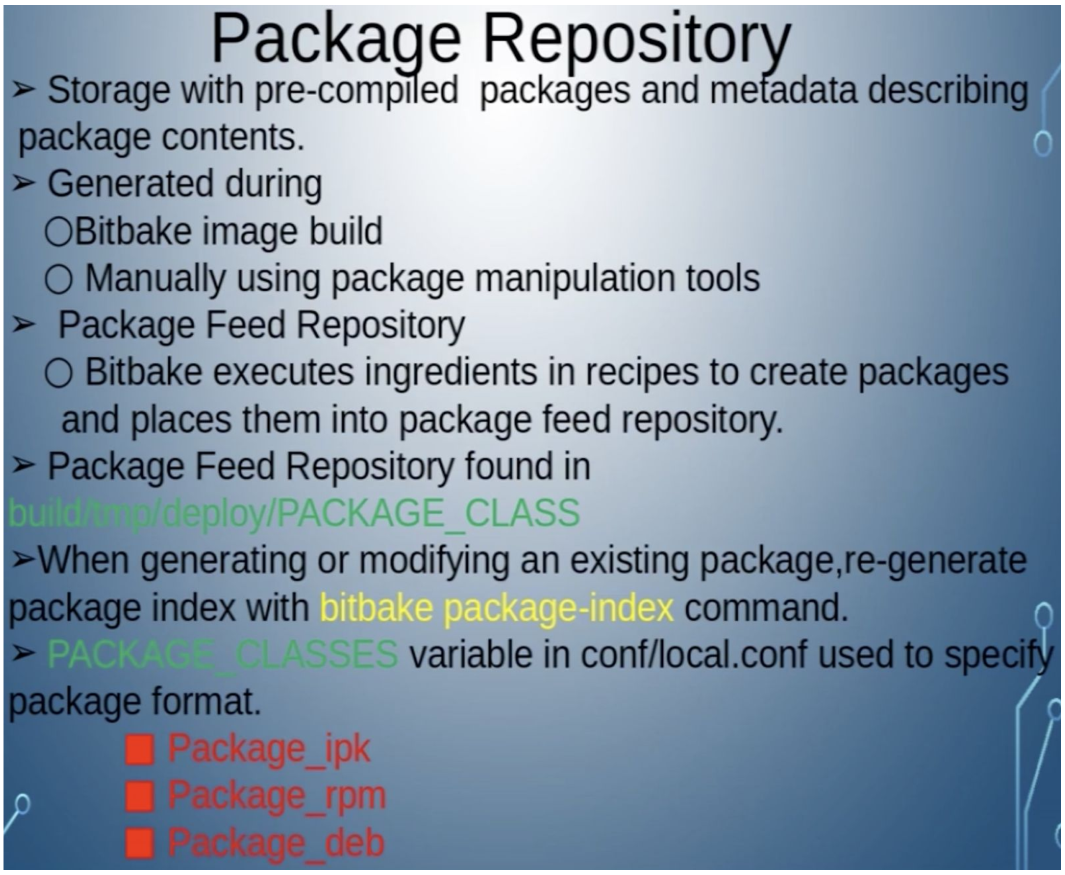
```
#@TYPE: Machine
#@NAME: Beaglebone-yocto machine
#@DESCRIPTION: Reference machine configuration for http://beagleboard.org/bone and
http://beagleboard.org/black boards

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
XSERVER ?= "xserver-xorg \
           xf86-video-modesetting \
           "

MACHINE_EXTRA_RRECOMMENDS = "kernel-modules kernel-devicetree"

EXTRA_IMAGEDEPENDS += "u-boot"

DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
EXTRA_IMAGECMD_jffs2 = "-lnp "
WKS_FILE ?= "beaglebone-yocto.wks"
IMAGE_INSTALL_append = " kernel-devicetree kernel-image-zimage"
do_image_wic[depends] += "mtools-native:do_populate_sysroot
dosfstools-native:do_populate_sysroot"

SERIAL_CONSOLE = "115200 ttyO0"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.15%"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "am335x-bone.dtb am335x-boneblack.dtb
am335x-bonegreen.dtb"
KERNEL_EXTRA_ARGS += "LOADADDR=${UBOOT_ENTRYPOINT}"

SPL_BINARY = "MLO"
UBOOT_SUFFIX = "img"
UBOOT_MACHINE = "am335x_boneblack_config"
UBOOT_ENTRYPOINT = "0x80008000"
UBOOT_LOADADDRESS = "0x80008000"

MACHINE_FEATURES = "usbgadget usbhost vfat alsa"

IMAGE_BOOT_FILES ?= "u-boot.${UBOOT_SUFFIX} MLO"
```

Section 7: Package Management



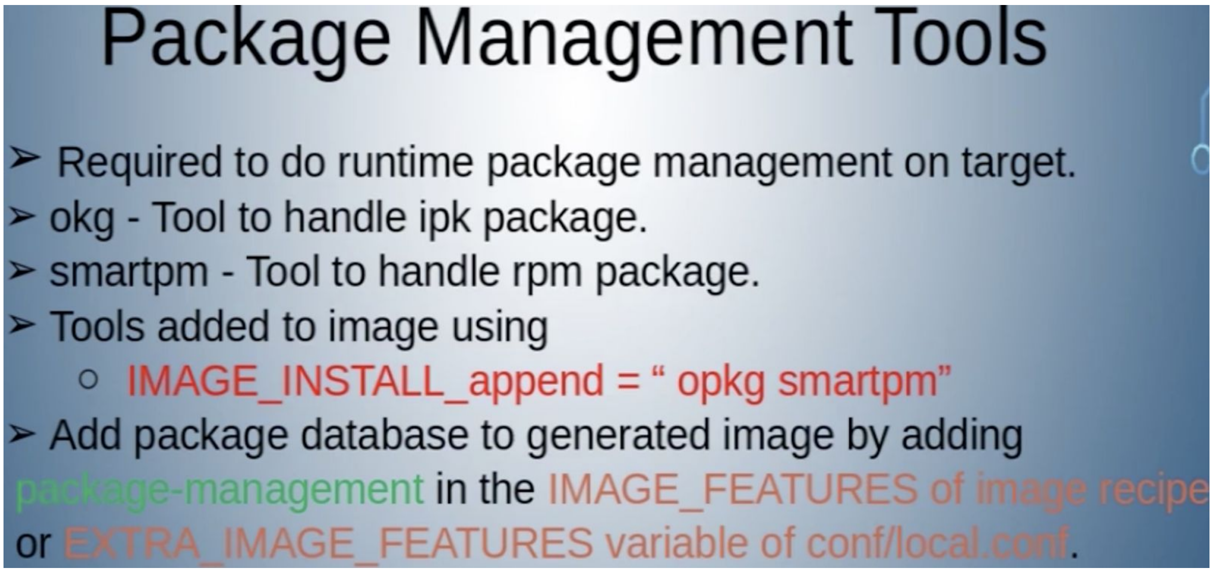
Runtime Package Management

- ➢ Update,add,remove and query package on target during development.
- ➢ Components required for package management
  - ○ Package repository consisting of pre-compile packages and metadata
  - ○ Package management tools on target



Package Repository

- ➢ Storage with pre-compiled packages and metadata describing package contents.
- ➢ Generated during
  - ○Bitbake image build
  - ○ Manually using package manipulation tools
- ➢ Package Feed Repository
  - ○ Bitbake executes ingredients in recipes to create packages and places them into package feed repository.
- ➢ Package Feed Repository found in build/tmp/deploy/PACKAGE_CLASS
- ➢When generating or modifying an existing package,re-generate package index with bitbake package-index command.
- ➢ PACKAGE_CLASSES variable in conf/local.conf used to specify package format.
  - 🟥 Package_ipk
  - 🟥 Package_rpm
  - 🟥 Package_deb


Package Management Tools

- Required to do runtime package management on target.
- okg - Tool to handle ipk package.
- smartpm - Tool to handle rpm package.
- Tools added to image using
  - IMAGE_INSTALL_append = “ opkg smartpm”
- Add package database to generated image by adding package-management in the IMAGE_FEATURES of image recipe or EXTRA_IMAGE_FEATURES variable of conf/local.conf.

1.Using OKPG package manager:

First we need to add the a couple of directives in conf/local.conf file:

CORE_IMAGE_EXTRA_INSTALL:

Specifies the list of packages to be added to the core image. You should only set this variable in the `local.conf` configuration file found in the Build Directory.

IMAGE_ROOTFS_EXTRA_SPACE:

Defines additional free disk space created in the image in Kbytes. By default, this variable is set to "0". This free disk space is added to the image after the build system determines the image size as described in `IMAGE_ROOTFS_SIZE`.

This variable is particularly useful when you want to ensure that a specific amount of free disk space is available on a device after an image is installed and running. For example, to be sure 5 Gbytes of free disk space is available, set the variable as follows:

```
IMAGE_ROOTFS_EXTRA_SPACE = "5242880"
```

For example, the Yocto Project Build Appliance specifically requests 40 Gbytes of extra space with the line:

```
IMAGE_ROOTFS_EXTRA_SPACE = "41943040"
```

IMAGE_ROOTFS_SIZE:

Defines the size in Kbytes for the generated image. The OpenEmbedded build system determines the final size for the generated image using an algorithm that takes into account the initial disk space used for the generated image, a requested size for the image, and requested additional free disk space to be added to the image. Programatically, the build system determines the final size of the generated image as follows:

```
if (image-du * overhead) < rootfs-size:
        internal-rootfs-size = rootfs-size + xspace
   else:
        internal-rootfs-size = (image-du * overhead) + xspace

   where:

     image-du = Returned value of the du command on
               the image.

     overhead = IMAGE_OVERHEAD_FACTOR

     rootfs-size = IMAGE_ROOTFS_SIZE

     internal-rootfs-size = Initial root filesystem
                          size before any modifications.

     xspace = IMAGE_ROOTFS_EXTRA_SPAC
```

IMAGE_OVERHEAD_FACTOR:

Defines a multiplier that the build system applies to the initial image size for cases when the multiplier times the returned disk usage value for the image is greater than the sum of `IMAGE_ROOTFS_SIZE` and `IMAGE_ROOTFS_EXTRA_SPACE`. The result of the multiplier applied to the initial image size creates free disk space in the image as overhead.

By default, the build process uses a multiplier of 1.3 for this variable. This default value results in 30% free disk space added to the image when this method is used to determine the final generated image size. You should be aware that post install scripts and the package management system uses disk space inside this overhead area. Consequently, the multiplier does not produce an image with all the theoretical free disk space.

The default 30% free disk space typically gives the image enough room to boot and allows for basic post installs while still leaving a small amount of free disk space. If 30% free space is inadequate, you can increase the default value. For example, the following setting gives you 50% free space added to the image:

```
IMAGE_OVERHEAD_FACTOR = "1.5"
```

Alternatively, you can ensure a specific amount of free disk space is added to the image by using the `IMAGE_ROOTFS_EXTRA_SPACE` variable.

IMAGE_ROOTFS:

The location of the root filesystem while it is under construction (i.e. during the `do_rootfs` task). This variable is not configurable. Do not change it.

So the additional configurations added to local.conf are:

```
CORE_IMAGE_EXTRA_INSTALL = "openssh"
PACKAGE_CLASSES = "package_ipk"
IMAGE_INSTALL_APPEND = "opkg"
IMAGE_ROOTFS_EXTRA_SPACE = "131072"
EXTRA_IMAGE_FEATURES = "package-management debug tweaks"
```

We then bitbake core-image-minimal or core-image-training and run qemu emulator.

Note: we need to ensure ssh server is installed on the host machine and we run the following at ~/yocto/poky/build-x86/tmp/deploy/ipk:

```
Python -m SimpleHttpServer
```

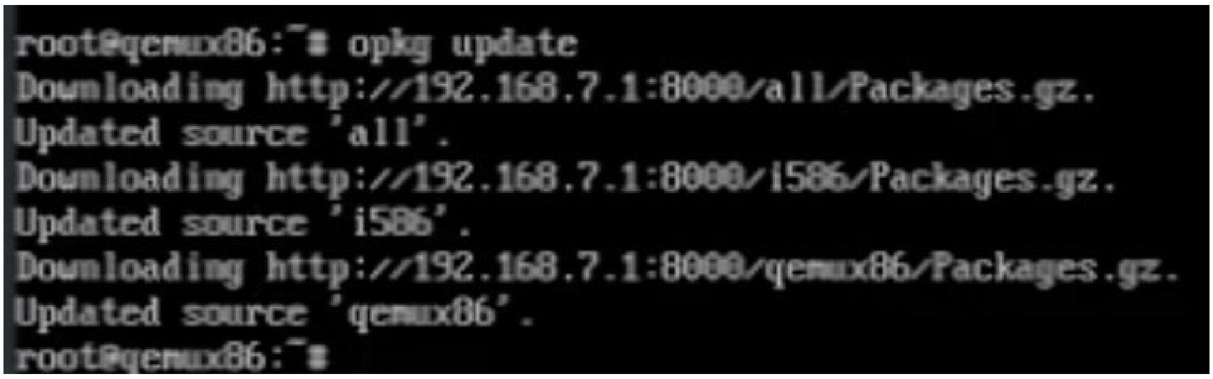
The default port is 8000, running this command can host a HTTP server that provides access to the ipk folder.

We log into the emulator and create /etc/opkg/base-feeds.conf and write Packages.gz for all architectures:

```
src/gz all http://192.168.7.1:8000 /all
src/gz i586 http://192.168.7.1:8000 /i586
src/gz qemux86 http://192.168.7.1:8000 /qemux86
```

We then run the following to grab these files from the host HTTP server:

```
Opkg update
```



```
root@qemux86:~# opkg update
Downloading http://192.168.7.1:8000/all/Packages.gz.
Updated source 'all'.
Downloading http://192.168.7.1:8000/i586/Packages.gz.
Updated source 'i586'.
Downloading http://192.168.7.1:8000/qemux86/Packages.gz.
Updated source 'qemux86'.
root@qemux86:~#
```

Now let's bitbake a new package called 'i2c-tools':

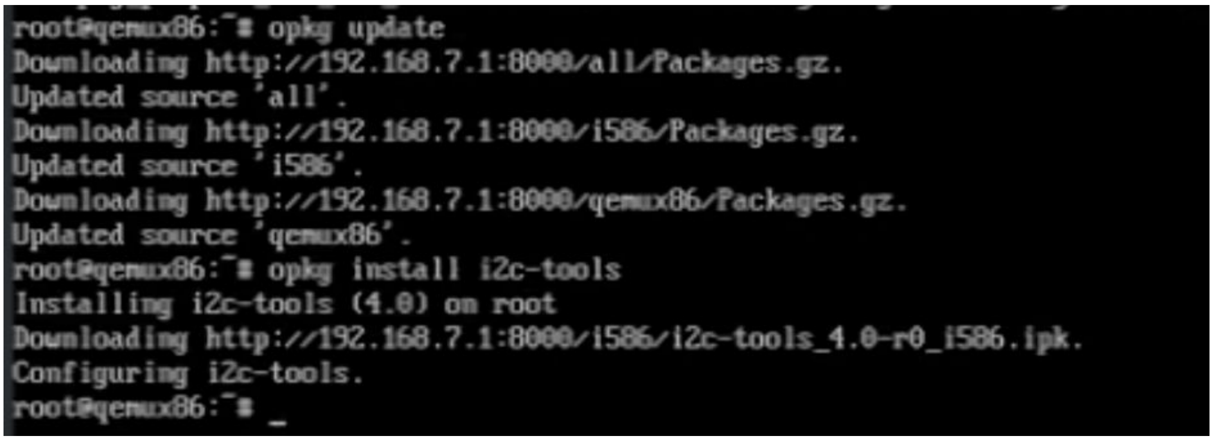
```
Bitbake i2c-tools
```

We need to run the following to update existing packages:

```
Bitbake package-index
```

And on the emulator side we first update opkg and then install i2c-tools:

```
Opkg update
Opkg install i2c-tools
```



```
root@qemux86:~# opkg update
Downloading http://192.168.7.1:8000/all/Packages.gz.
Updated source 'all'.
Downloading http://192.168.7.1:8000/i586/Packages.gz.
Updated source 'i586'.
Downloading http://192.168.7.1:8000/qemux86/Packages.gz.
Updated source 'qemux86'.
root@qemux86:~# opkg install i2c-tools
Installing i2c-tools (4.0) on root
Downloading http://192.168.7.1:8000/i586/i2c-tools_4.0-r0_i586.ipk.
Configuring i2c-tools.
root@qemux86:~# _
```

We can check the packages that can be upgraded using:

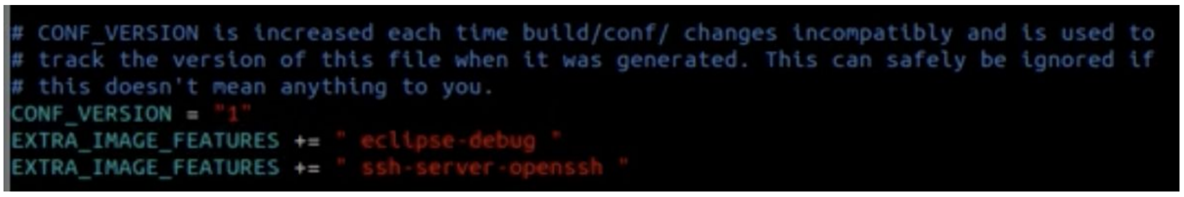
```
Opkg list-upgradable
```

And perform upgrade:

```
Opkg  upgrade
```

Section 8: Application Debugging

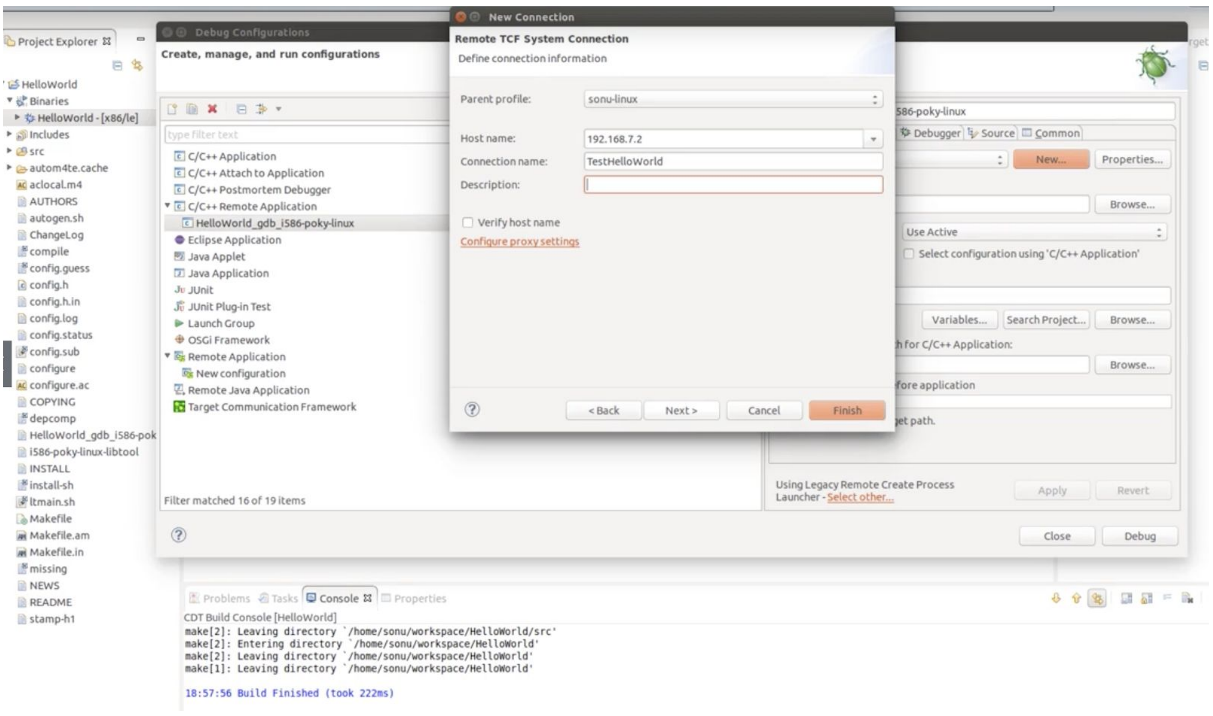
To use Eclipse IDE to do debugging, we first need to add the following to local.conf:



```
# CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
# track the version of this file when it was generated. This can safely be ignored if
# this doesn't mean anything to you.
CONF_VERSION = "1"
EXTRA_IMAGE_FEATURES += " eclipse-debug "
EXTRA_IMAGE_FEATURES += " ssh-server-openssh "
```

We then bitbake the target image with the updated local.conf.

Once the target is booted, we find its IP address, and open Eclipse on the host, build the application project for the target, and setup the debug configurations:



New Connection

Remote TCF System Connection
Define connection information

Parent profile: sonu-linux
Host name: 192.168.7.2
Connection name: TestHelloWorld
Description:

☐ Verify host name

Debug Configurations — C/C++ Remote Application: HelloWorld_gdb_i586-poky-linux

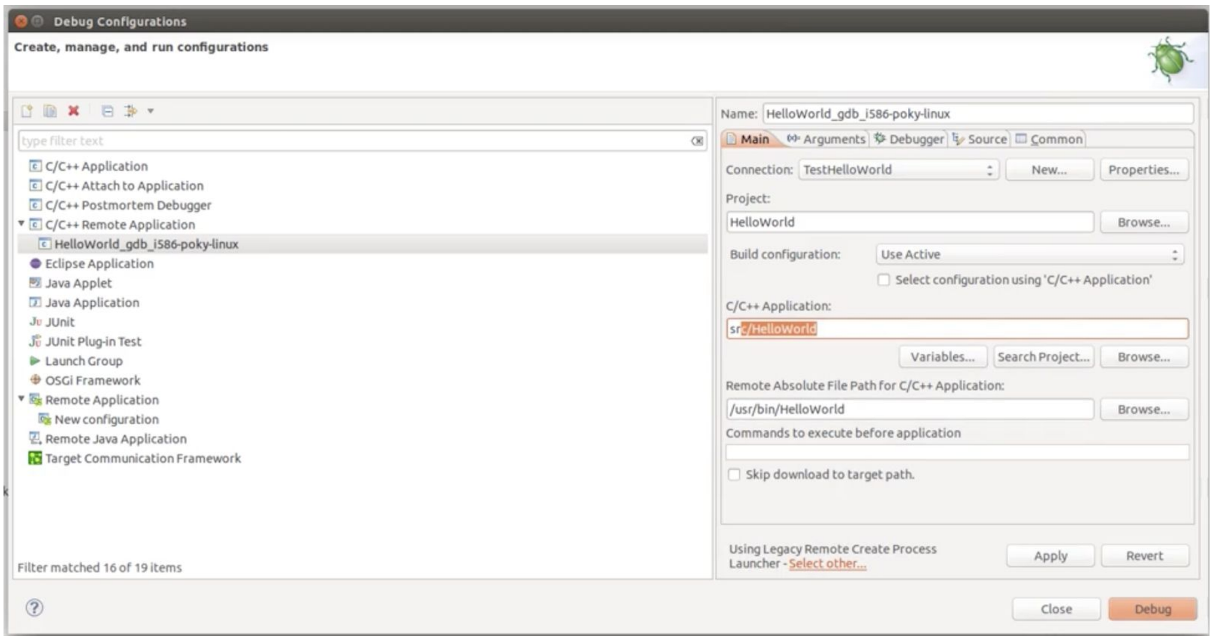
Use Active
☐ Select configuration using 'C/C++ Application'

Using Legacy Remote Create Process Launcher - Select other...

Filter matched 16 of 19 items

CDT Build Console [HelloWorld]

```
make[2]: Leaving directory `/home/sonu/workspace/HelloWorld/src'
make[2]: Entering directory `/home/sonu/workspace/HelloWorld'
make[2]: Leaving directory `/home/sonu/workspace/HelloWorld'
make[1]: Leaving directory `/home/sonu/workspace/HelloWorld'

18:57:56 Build Finished (took 222ms)
```


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
  - HelloWorld_gdb_i586-poky-linux
- Eclipse Application
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- OSGi Framework
- Remote Application
  - New configuration
- Remote Java Application
- Target Communication Framework

Filter matched 16 of 19 items

Name: HelloWorld_gdb_i586-poky-linux

Main | Arguments | Debugger | Source | Common

Connection: TestHelloWorld   New...   Properties...

Project: HelloWorld   Browse...

Build configuration: Use Active

☐ Select configuration using 'C/C++ Application'

C/C++ Application: src/HelloWorld

Variables...   Search Project...   Browse...

Remote Absolute File Path for C/C++ Application: /usr/bin/HelloWorld   Browse...

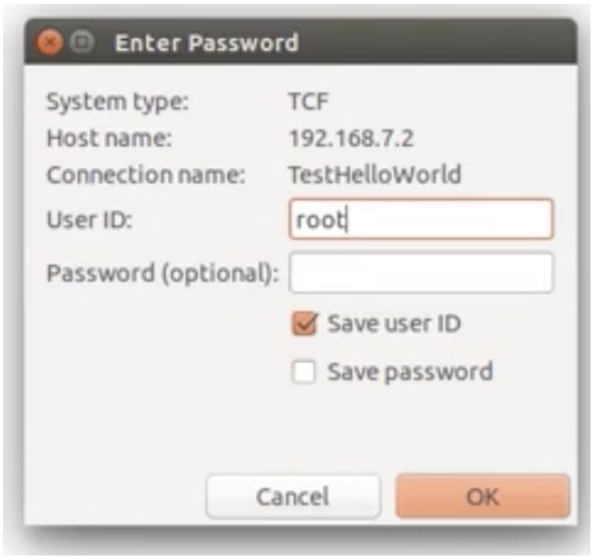
Commands to execute before application:

☐ Skip download to target path.

Using Legacy Remote Create Process Launcher - Select other...

Apply   Revert

Close   Debug


Enter Password

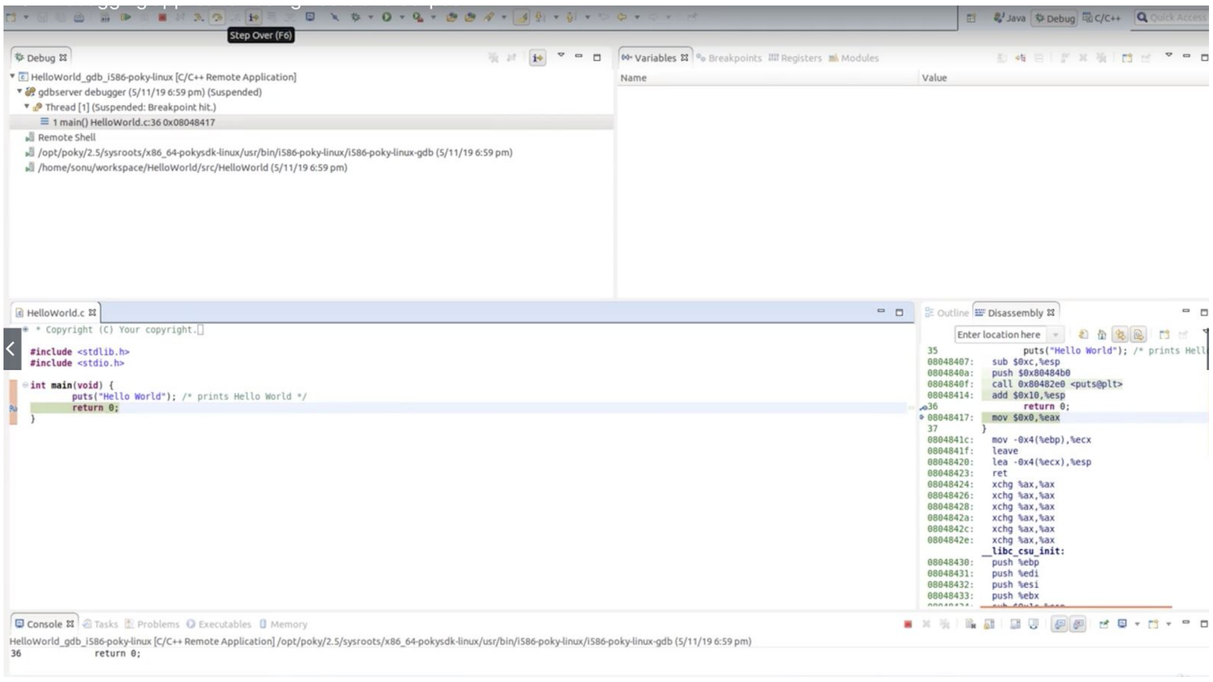
System type:	TCF
Host name:	192.168.7.2
Connection name:	TestHelloWorld
User ID:	root
Password (optional):	

☑ Save user ID

☐ Save password

Cancel   OK

Then we can debug the application as desktop application:



```
HelloWorld_gdb_i586-poky-linux [C/C++ Remote Application]
  gdbserver debugger (5/11/19 6:59 pm) (Suspended)
    Thread [1] (Suspended: Breakpoint hit.)
      1 main() HelloWorld.c:36 0x08048417
  Remote Shell
  /opt/poky/2.5/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb (5/11/19 6:59 pm)
  /home/sonu/workspace/HelloWorld/src/HelloWorld (5/11/19 6:59 pm)
```

```c
 * Copyright (C) Your copyright.

#include <stdlib.h>
#include <stdio.h>

int main(void) {
        puts("Hello World"); /* prints Hello World */
        return 0;
}
```

```
35                  puts("Hello World"); /* prints Hell
08048407:   sub $0xc,%esp
0804840a:   push $0x80484b0
0804840f:   call 0x80482e0 <puts@plt>
08048414:   add $0x10,%esp
36                  return 0;
08048417:   mov $0x0,%eax
37          }
0804841c:   mov -0x4(%ebp),%ecx
0804841f:   leave
08048420:   lea -0x4(%ecx),%esp
08048423:   ret
08048424:   xchg %ax,%ax
08048426:   xchg %ax,%ax
08048428:   xchg %ax,%ax
0804842a:   xchg %ax,%ax
0804842c:   xchg %ax,%ax
0804842e:   xchg %ax,%ax
          __libc_csu_init:
08048430:   push %ebp
08048431:   push %edi
08048432:   push %esi
08048433:   push %ebx
```

```
HelloWorld_gdb_i586-poky-linux [C/C++ Remote Application] /opt/poky/2.5/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb (5/11/19 6:59 pm)
36              return 0;
```