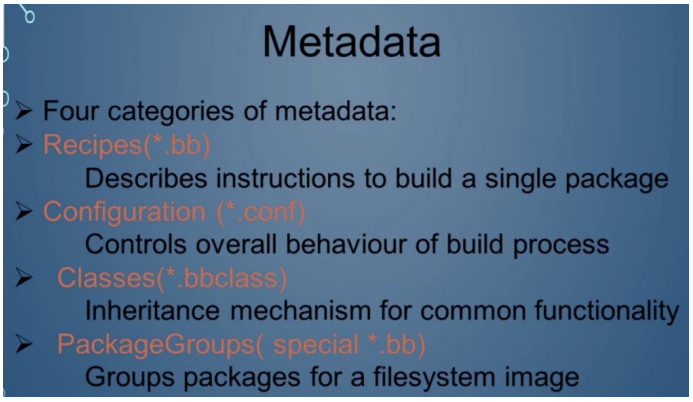
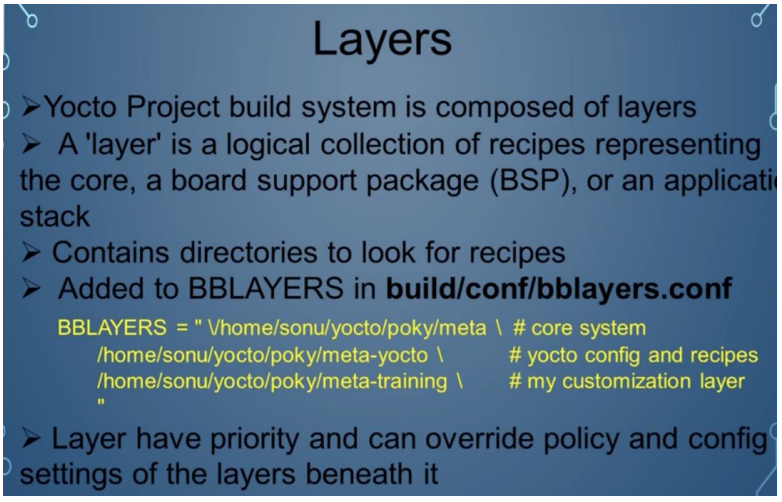
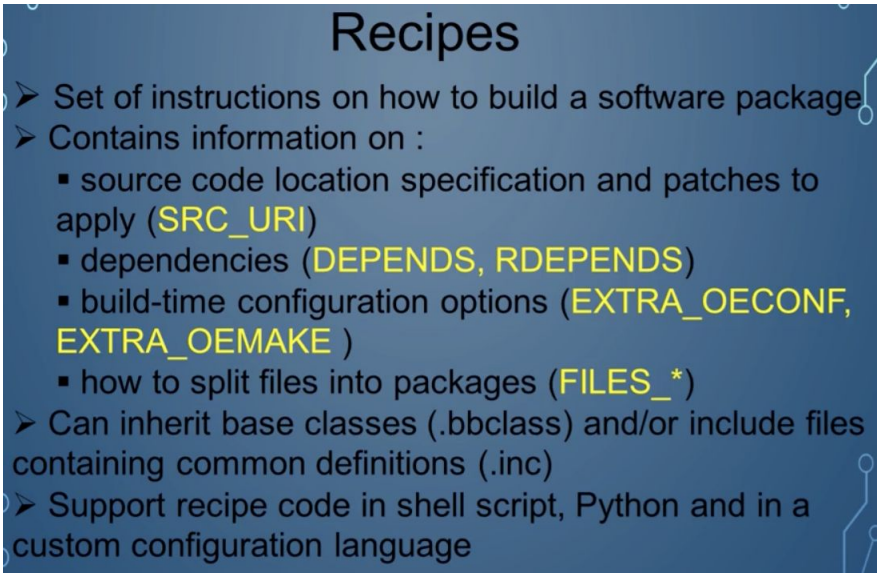
Section 2: Understand Yocto Build system components

1.Basic components







Recipe – Default Tasks

`do_fetch`	Locate and download source code
`do_unpack`	Unpack source into working directory
`do_patch`	Apply any patches
`do_configure`	Perform any necessary pre-build configuration
`do_compile`	Compile the source code
`do_install`	Installation of resulting build artifacts in WORKDIR
`do_populate_sysroot`	Copy artifacts to sysroot
`do_package_*`	Create binary package(s)

Recipe – Variables used (1/2)

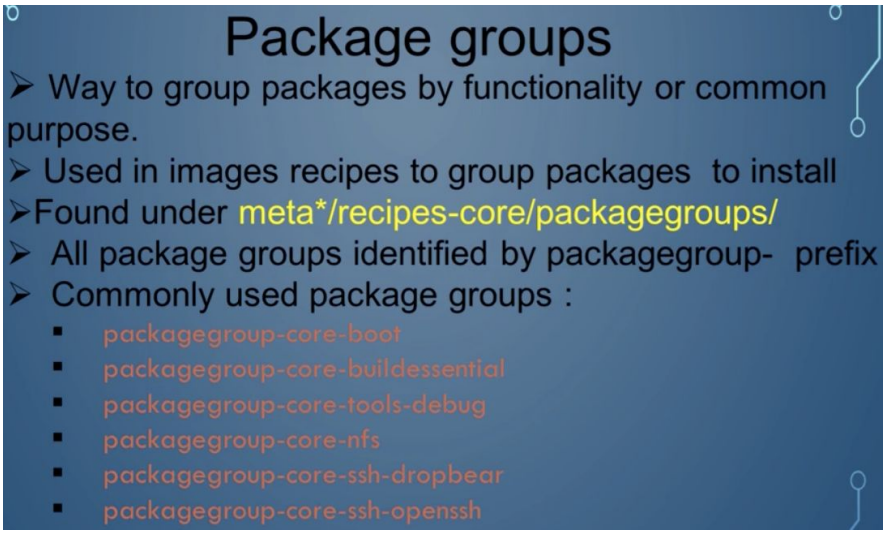
Variable Name	Description
PN	Package Name (ex: busybox , connman)
PV	Package Version (ex : 1.2.1 , 2.0)
PR	Package Revision (ex: ro, r1, r2)
WORKDIR	Top of the build directory tmp/work/$ARCH/${PN}-${PV}-${PR}
S	Source code directory path used by do_unpack and do_patch
B	Build directory path used by do_configure and do_compile
D	Destination directory path used by do_install

Variable Name	Description
${includedir}	/usr/include
${libdir}	/usr/lib
${sbindir}	/usr/bin
${sysconfdir}	/etc
${datadir}	/usr/share
${base_libdir}	/lib
${base_sbindir}	/sbin
${base_bindir}	/bin
${prefix}	/usr

Classes

- Provide encapsulation and inheritance logic
- Abstracts common functionality and share it amongst several recipes
- Few important classes are:
  - base.bbclass
  - autotools.bbclass
  - pkgconfig.bbclass
  - kernel.bbclas
  - image.bbclass
  - rootfs*.bbclass
  - sanity.bbclass
- Every recipe inherits the base.bbclass automatically.

- base.bbclass contains definitions of basic tasks as fetching, unpacking, configuring, compiling, installing and packaging
- base.bbclass usually overridden by other classes as autotool.bbclass or package.bbclass



2. Create a new layer

Create a directory called ‘meta-test’ under yocto/, and create a directory ‘conf’ under ‘meta-test’, and need to create layer.conf and fill it out:

BBPATH: Used by BitBake to locate class (`.bbclass`) and configuration (`.conf`) files. This variable is analogous to the `PATH` variable.
If you run BitBake from a directory outside of the build directory, you must be sure to set `BBPATH` to point to the build directory. Set the variable as you would any environment variable and then run BitBake:

```
$ BBPATH="<build_directory>"
$ export BBPATH
$ bitbake <target>
```

BBFILES: List of recipe files BitBake uses to build software.

BBFILE_COLLECTIONS: this variable lists the names of configured layers. These names are used to find the other `BBFILE_*` variables.

BBFILE_PATTERN: The variable that expands to match files from BBFILES in a particular layer. This variable is used in the conf/layer.conf file and must be suffixed with the name of the specific layer (e.g. BBFILE_PATTERN_emenlow).

LAYERDIR: When used inside the `layer.conf` configuration file, this variable provides the path of the current layer. This variable is not available outside of `layer.conf` and references are expanded immediately when parsing of the file completes.

BBFILE_PRIORITY: Assigns the priority for recipe files in each layer.
This variable is useful in situations where the same recipe appears in more than one layer. Setting this variable allows you to prioritize a layer against other layers that contain the same recipe - effectively letting you control the precedence for the multiple layers.

A larger value for the BBFILE_PRIORITY variable results in a higher precedence. For example, the value 6 has a higher precedence than the value 5. The default priority, if unspecified for a layer with no dependencies, is the lowest defined priority + 1 (or 1 if no priorities are defined).

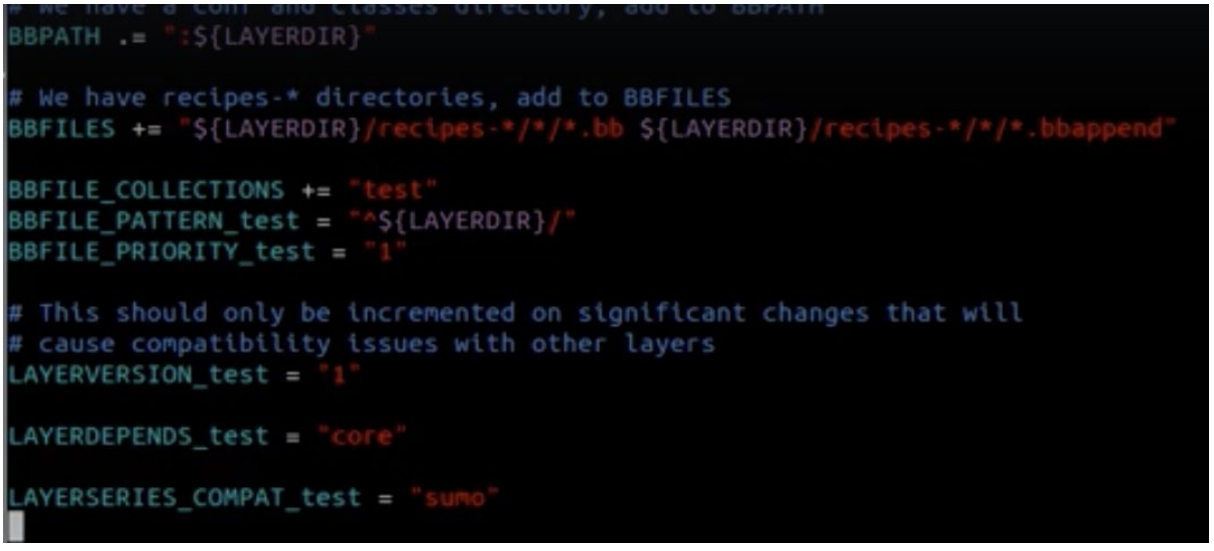
LAYERDEPENDS: Lists the layers, separated by spaces, upon which this recipe depends. Optionally, you can specify a specific layer version for a dependency by adding it to the end of the layer name with a colon, (e.g. "anotherlayer:3" to be compared against LAYERVERSION_anotherlayer in this case). BitBake produces an error if any dependency is missing or the version numbers do not match exactly (if specified).

You use this variable in the conf/layer.conf file. You must also use the specific layer name as a suffix to the variable (e.g. LAYERDEPENDS_mylayer).

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 multiple OE-Core versions for the layer, use a space-separated list:

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



Then we need to modify bblayers.conf in the build/conf directory, and then use 'bitbake-layers show-layers' to check if the new layer is added.

Alternatively, we can use bitbake command to create a new layer:

```
Bitbake-layers create-layer layer-name
```

To add this layer to the existing layers, need to run:

```
Bitbake-layers add-layer layer-name
```

Note: the above command line has to be run under build/, so we need to specify the path of the new layer as '../layer-name'.

3. Build a new recipe

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

This variable replaces POKY_EXTRA_INSTALL, which is no longer supported.

To build a recipe like 'core-image-training', create directory recipes-core/images and the file 'recipes-core/image/core-image-training_0.1.bb'. And fill in basic information: inherit core class, install packagegroup-core-boot.

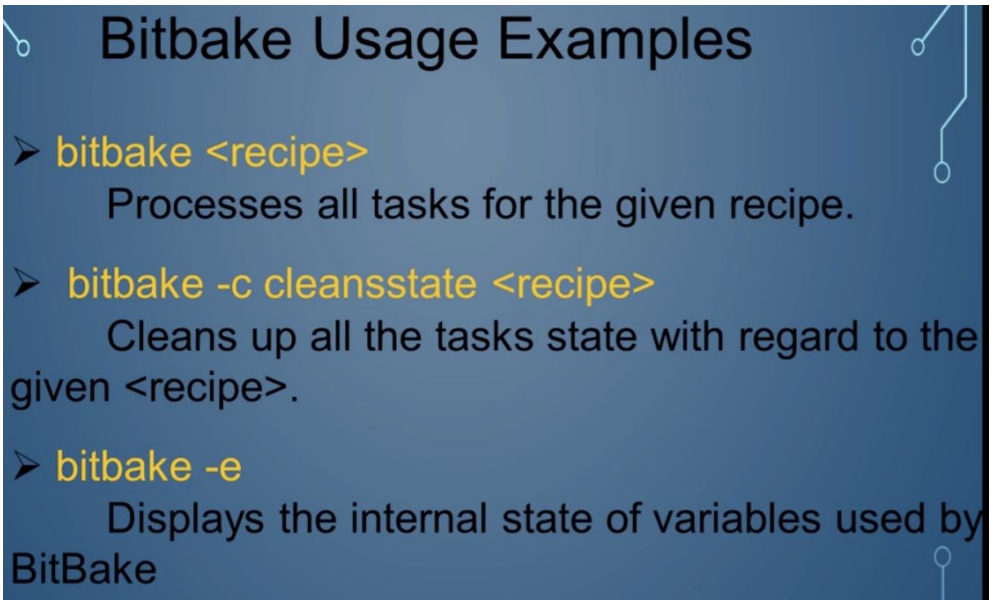
```
SUMMARY = "A small image capable of allowing a device to boot"

IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"

IMAGE_LINGUAS = ''

LICENSE = "MIT"

inherit core-image
```

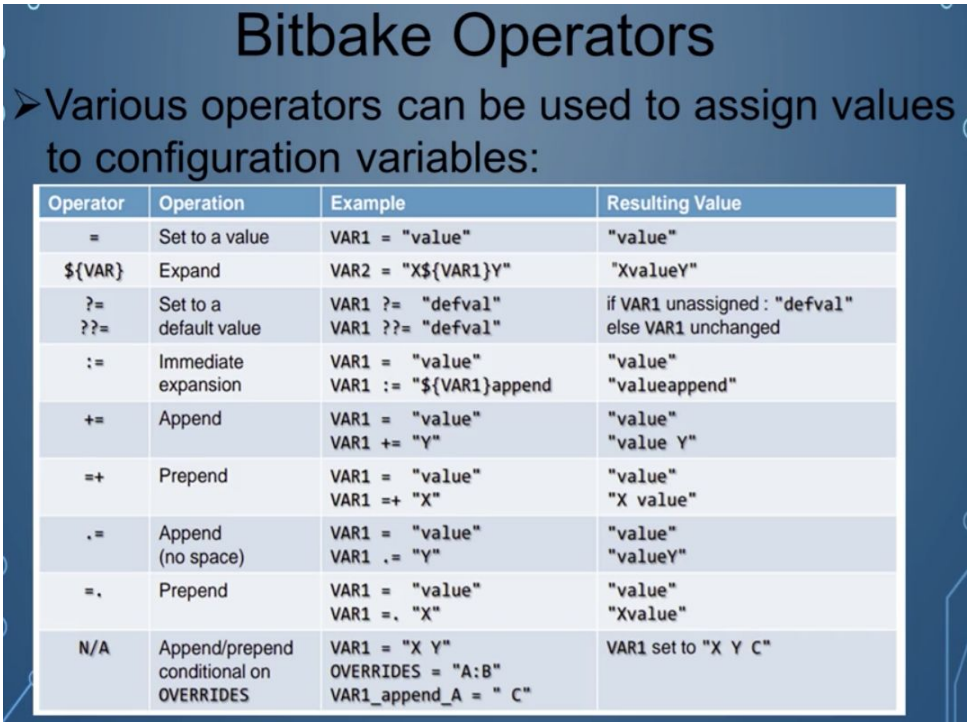
And then go to build directory and do 'bitbake core-image-training', and finally we can test the image using qemu:

```
Runqemu qemux86  bzImage-qemux86.bin core-image-training-qemux86.ext4
```

Section 3: Understanding bitbake tool

1. Bitbake command tutorial





OVERRIDES: BitBake uses OVERRIDES to control what variables are overridden after BitBake parses recipes and configuration files. The OVERRIDES variable is a colon-character-separated list that contains items for which you want to satisfy conditions.

BitBake supports append and prepend operations to variable values based on whether a specific item is listed in OVERRIDES. Here is an example:

```
DEPENDS = "glibc ncurses"
OVERRIDES = "machine:local"
DEPENDS_append_machine = "libmad"
```

In this example, DEPENDS becomes "glibc ncurses libmad".

Bitbake Commands

Bitbake Command	Description
bitbake <pkgname> -c listtasks	List tasks available for a package/recipe
bitbake <pkgname> -c rebuild -f	Clean and build again a package
bitbake <pkgname> -c fetch -f	Download again the source program
bitbake <pkgname> -c devshell	Expand a gnome xterm ready to raise commands
bitbake <pkgname> -c clean/compile	Clean or compile a package
bitbake <pkgname>	Build a package
bitbake package-index	Make index files for package feeds
bitbake <target_image>	Build image containing task-base packages
bitbake <target_image> -c buildall -f	Build pending packages for target_image
bitbake -e <pkgname> | grep ^S=	Finds the source code location of a package
bitbake <target_image> -c rootfs -f	Populate rootfs again for target_image
bitbake -e <target> | grep ^WORKDIR=	Finds working directory of a package
bitbake -e <image-target> | grep ^IMAGE_FSTYPES=	Finds the image types being build
bitbake -g -u depexp <target_image>	Dependency explorer UI

Bitbake -c devshell image/recipe allows us to edit the source code of the recipe, and then we need to run ‘bitbake -c compile image/recipe’ to save the changes.

To list task and recipe dependencies, we can use ‘bitbake -g image/recipe’, it will generate 3 files:

```
NOTE: PN build list saved to 'pn-buildlist'
NOTE: Task dependencies saved to 'task-depends.dot'
NOTE: Flattened recipe dependencies saved to 'recipe-depends.dot'
```

Section 3: Application Development

1.Add applications using autotools

LIC_FILES_CHKSUM: Checksums of the license text in the recipe source code.

This variable tracks changes in license text of the source code files. If the license text is changed, it will trigger a build failure, which gives the developer an opportunity to review any license change.

This variable must be defined for all recipes (unless LICENSE is set to "CLOSED")

If your source files have a configure.ac file, then your software is built using Autotools. If this is the case, you just need to worry about tweaking the configuration.

When using Autotools, your recipe needs to inherit the autotools class and your recipe does not have to contain a do_configure task. However, you might still want to make some adjustments. For example, you can set EXTRA_OECONF to pass any needed configure options that are specific to the recipe.

Applications that use Autotools such as autoconf and automake require a recipe that has a source archive listed in SRC_URI and also inherit the autotools class, which contains the definitions of all the steps needed to build an Autotool-based application. The result of the build is automatically packaged. Following is one example: (hello_2.3.bb)

```
SUMMARY = "GNU Helloworld application"
    SECTION = "examples"
    LICENSE = "GPLv2+"
    LIC_FILES_CHKSUM =
"file://COPYING;md5=751419260aa954499f7abaabaa882bbe"

    SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"

    inherit autotools gettext
```

We just need to run ‘bitbake helloworld’ to build the recipe.

GNU_MIRROR is an open-source website containing free software:

https://www.gnu.org/prep/ftp.en.html

We can verify the image by appending it to IMAGE_INSTALL in the core image recipe, and either load the core image to sdcard or run runqemu to use the emulator to check the application. The application will be stored at /usr/bin by default.

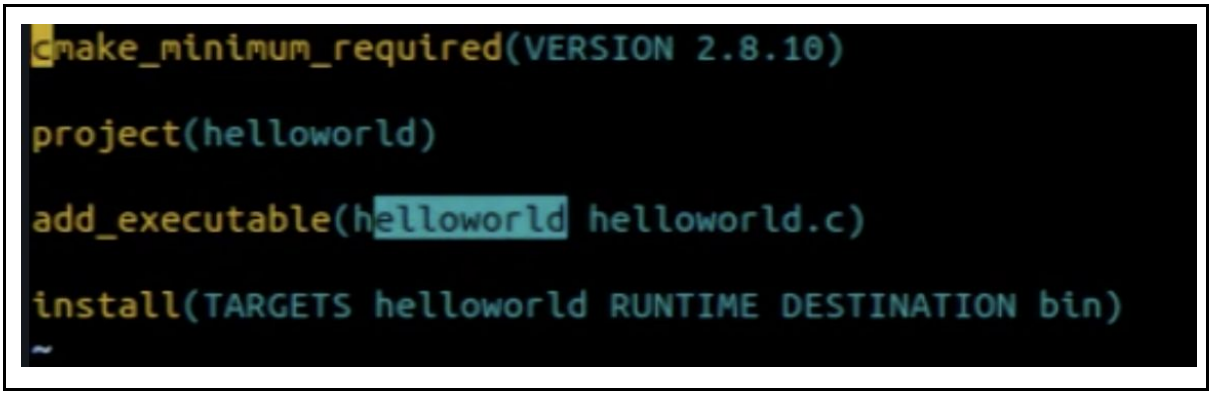
2. Add applications using CMAKE

If your source files have a CMakeLists.txt file, then your software is built using CMake. If this is the case, you just need to worry about tweaking the configuration.

When you use CMake, your recipe needs to inherit the cmake class and your recipe does not have to contain a do_configure task. You can make some adjustments by setting EXTRA_OECMAKE to pass any needed configure options that are specific to the recipe.

If the software your recipe is building uses Autotools or CMake, the OpenEmbedded build system understands how to install the software. Consequently, you do not have to have a do_install task as part of your recipe. You just need to make sure the install portion of the build completes with no issues. However, if you wish to install additional files not already being installed by make install, you should do this using a do_install_append function using the install command.

A sample CMakeLists.txt is as follows:



```
cmake_minimum_required(VERSION 2.8.10)

project(helloworld)

add_executable(helloworld helloworld.c)

install(TARGETS helloworld RUNTIME DESTINATION bin)
```

In the bitbake recipe, we don't need to specify checksum, but we have to inherit cmake and specify CMakeLists.txt as part of SRC_URI.

```
DESCRIPTION = "Simple helloworld application built by Cmake"
LICENSE = "CLOSED"

SRC_URI = "file://CMakeLists.txt \
           file://helloworld.c"

S = "${WORKDIR}"

inherit cmake

EXTRA_OECMAKE = " "
```

We can use the same way to test the helloworld-cmake, just like helloworld built by autotools.

3. Add applications using Makefile

Applications that use GNU make also require a recipe that has the source archive listed in SRC_URI. You do not need to add a do_compile step since by default BitBake starts the make command to compile the application,but you can implement do_compile( ) with the following:

```
do_compile(){
 Oe_runmake -f Makefile
}
```

If you need additional make options, you should store them in the EXTRA_OEMAKE variable. BitBake passes these options into the make GNU invocation. Note that a do_install task is still required. Otherwise, BitBake runs an empty do_install task by default.

Some applications might require extra parameters to be passed to the compiler. For example, the application might need an additional header path. You can accomplish this by adding to the CFLAGS variable. The following example shows this:

```
CFLAGS_prepend = "-I ${S}/include "
```

How to fix : ERROR: do_package_qa: QA Issue: No GNU_HASH in the elf binary

If you get the error as below, at the end of the compilation, check the solution to fix this:

```
ERROR: helloworld-0.1-r0 do_package_qa: QA Issue: No GNU_HASH in the elf binary:
'/home/myuser/poky/build/tmp/work/cortexa7hf-neon-vfpv4-poky-linux-gnueabi/helloworld/0.1-r0/pa
ckages-split/helloworld/usr/bin/helloworld' [ldflags]
ERROR: helloworld-0.1-r0 do_package_qa: QA run found fatal errors. Please consider fixing them.
ERROR: helloworld-0.1-r0 do_package_qa: Function failed: do_package_qa
ERROR: Logfile of failure stored in:
/home/myuser/poky/build/tmp/work/cortexa7hf-neon-vfpv4-poky-linux-gnueabi/helloworld/0.1-r0/tem
p/log.do_package_qa.7193
ERROR: Task
(/home/myuser/meta-lynxbee/recipes-example/example/helloworld_0.1.bb:do_package_qa) failed
with exit code '1'
NOTE: Tasks Summary: Attempted 469 tasks of which 456 didn't need to be rerun and 1 failed.

Summary: 1 task failed:
  /home/myuser/poky/meta-lynxbee/recipes-example/example/helloworld_0.1.bb:do_package_qa
Summary: There were 3 ERROR messages shown, returning a non-zero exit code.
```

Solution :

```
$ vim helloworld_0.1.bb
```

and add following line and save file:

```
TARGET_CC_ARCH += "${LDFLAGS}"
```

Now, save and compile the program and you will not see that error. This above fix works only when you have your recipe does some compilation, but in certain scenarios where you want to integrate already compiled binaries (precompiled) then the above fix doesn't work. In that case, use the following lines into your recipe:

```
INSANE_SKIP_${PN} = "ldflags"
INSANE_SKIP_${PN}-dev = "ldflags"
```

If you use this INSANE_SKIP , it works for both cases and you may not need TARGET_CC_ARCH in the first case.

INSANE_SKIP: Specifies the QA checks to skip for a specific package within a recipe. For example, to skip the check for symbolic link .so files in the main package of a recipe, add the following to the recipe. The package name override must be used, which in this example is ${PN}:

```
INSANE_SKIP_${PN} += "dev-so"
```

https://www.lynxbee.com/how-to-fix-error-do_package_qa-qa-issue-no-gnu_hash-in-the-elf-binary/

The example Makefile and .bb recipe can be as follows:

helloworld-make.bb:

```
SUMMARY = "Helloworld application using GNU Makefile"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://${COREBASE}/meta/files/common-licenses/GPL-2.0;md5=801f80980d
171dd6425610833a22dbe6"

SRC_URI = "file://helloworld-make.c \
           file://Makefile"

S = "${WORKDIR}"
EXTRA_OEMAKE = " 'CC=${CC}' 'LINK=${CC}'"
```

```
do_compile(){
   oe_runmake -f Makefile
}

do_install(){
   install -d ${D}${bindir}
   install -c -m 0755 ${S}/helloworld-make ${D}${bindir}
}

INSANE_SKIP_${PN} = "ldflags"
INSANE_SKIP_${PN}-dev = "ldflags"
```

Makefile:

```
#
# what to call the final executable
TARGET=helloworld-make

# which object files that the executable consists of
OBJS=helloworld-make.o

# what compile to use
CC=gcc

# compile flags
CFLAGS=-g

# how to remove files
DEL_FILE=rm -f

%.o : %.c
   $(CC) -c -o $@ $< $(CFLAGS)

# link the target with all objects and libraries
${TARGET}: ${OBJS}
   ${CC} -o helloworld-make ${OBJS}

clean:
   ${DEL_FILE} helloworld-make.o
```

4. Application development using static libraries

If you are building a library and the library offers static linking, you can control which static library files (*.a files) get included in the built library.

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.

ALLOW_EMPTY: Specifies if an output package should still be produced if it is empty. By default, BitBake does not produce empty packages. This default behavior can cause issues when there is an RDEPENDS or some other hard runtime requirement on the existence of the package.

Like all package-controlling variables, you must always use them in conjunction with a package name override, as in:

```
ALLOW_EMPTY_${PN} = "1"
    ALLOW_EMPTY_${PN}-dev = "1"
    ALLOW_EMPTY_${PN}-staticdev = "1"
```

How to make a static library (.a) on Linux:

Compile C programs in files lib1.c and lib2.c:

```
$ gcc -c lib1.c lib2.c
```

Create a library “libmy.a” using ar:

```
$ ar -cvq libmy.a lib1.o lib2.o
```

You can also list the files in a library with ar:

```
$ ar -t libmy.a
```

You can link the library in your program (e.g. p.c):

```
$ gcc -o p p.c libmy.a
```

https://www.systutorials.com/how-to-make-a-static-library-a-on-linux/

When building .bb file for the target static library, we need to ensure certain packages can still be generated even if they are empty, and we need to skip QA check for libraries. In addition, do_compile( ) and do_install( ) have to be implemented. In do_compile( ), we compile source files and generate .a file using ${AR}; in do_install( ), we deploy the static library to ${D}${libdir}.

Check the following libtraining-static-1.0.bb:

```
DESCRIPTION = "Simple static library"
SECTION = "libs"
LICENSE = "CLOSED"

SRC_URI = "file://add.c \
           file://multiply.c"

S = "${WORKDIR}"

ALLOW_EMPTY_${PN} = "1"
INSANE_SKIP_${PN} = "ldflags"

do_compile(){

   ${CC} -c add.c multiply.c
   ${AR} -cvq libtraining.a add.o multiply.o
}

do_install(){

   install -d ${D}${libdir}
   install -m 0644 libtraining.a ${D}${libdir}
}
```

And we need to explicitly bitbake the recipe:

```
Bitbake libtraining-static-1.0
```

Upon bitbaking the library, we need to create a application recipe to use it.

STAGING_LIBDIR: Specifies the path to the /usr/lib subdirectory of the sysroot directory for the target for which the current recipe is being builtSpecifies the path to the /usr/lib subdirectory of the sysroot directory for the target for which the current recipe is being built.

The application recipe also needs to have user-defined do_compile( ) and do_install( ). In do_compile( ), the target application needs to be compiled with libtraining.a linked. Additionally, the recipe must explicitly add DEPENDS directive to include ‘libtraining-static’ package.

Test-libtraining-static_1.0.bb:

```
DESCRIPTION = "Application that uses libtraining-static library"
SECTION = "examples"
LICENSE = "CLOSED"

DEPENDS = "libtraining-static"
SRC_URI = "file://test-libtraining-static.c"

S = "${WORKDIR}"

INSANE_SKIP_${PN} = "ldflags"

do_compile(){
   ${CC} -o test-libtraining-static test-libtraining-static.c
${STAGING_LIBDIR}/libtraining.a
}

do_install(){
   install -d ${D}${bindir}
   install -m 0755 test-libtraining-static ${D}${bindir}
}
```

5. Application development using dynamic libraries

Build a shared library using GCC:

1). Compile the source code using the -fPIC option

Object code intended for use in a shared library must be ‘position-independent’, meaning that it can execute without first being modified to account for where it has been loaded in memory.

GCC can be instructed to generate position independent code using the -fPIC option:

```
gcc -c -fPIC -o foo.o foo.c
gcc -c -fPIC -o bar.o bar.c
gcc -c -fPIC -o baz.o baz.c
```

This option is not enabled by default because it tends to cause some loss of performance, and for purposes other than building shared libraries it is often not necessary.

2). Link the object code using the -fPIC and -shared options

The default behaviour of the gcc and g++ commands when linking is to produce an executable program. They can be instructed to produce a shared library instead by means of -shared option:

```
gcc -shared -fPIC -Wl,-soname,libqux.so.1 -o libqux.so.1.5.0 foo.o bar.o baz.o -lc
```

The -Wl option passes a comma-separated list of arguments to the linker. As its name suggests, -soname specifies the required soname. If these options are omitted then the library will not have a soname.

The ldconfig manpage recommends linking against libc, which has been done above using the -l option (-lc).

3). Testing

One way to test the library is to install it in a directory on the library search path. /usr/local/lib is usually the most appropriate choice. You will need to create softlinks corresponding to the soname of the library, and the name used to refer to the library when building the executable, if these are different from the filename:

```
ln -s libqux.so.1.5.0 libqux.so.1
ln -s libqux.so.1.5.0 libqux.so
```

If you cannot or do not want to move the library to /usr/local/lib then it is possible to link against the library at build time. This can be done by listing the pathname of the library as an argument to gcc without use of the -l option:

```
gcc main.c libqux.so.1.5.0
```

At load time you will need to add the relevant directory to the library search path. This can be done by setting the environment variable LD_LIBRARY_PATH, for example:

```
export LD_LIBRARY_PATH=`pwd`
```

As above, you will need to create a softlink corresponding to the soname of the library. If there is a need to search multiple directories then they should be specified as a colon-separated list in LD_LIBRARY_PATH.

http://www.microhowto.info/howto/build_a_shared_library_using_gcc.html

To build a dynamic library recipe, we need to implement do_compile( ) and do_install( ). And in do_compile( ), we need to compile source files into build artifacts that are position-

independent. And then we need to build a dynamic library out of them with linker options configured.

In do_install( ), we need to not only move the dynamic library artifact to /usr/local/lib, but also create soft links for both soname and shortcut names used by applications.

libtraining-dyn_1.0.bb:

```
DESCRIPTION = "Simple Dynamic library"
SECTION = "libs"
LICENSE = "CLOSED"

SRC_URI = "file://subtract.c \
           file://divide.c"

S = "${WORKDIR}"

do_compile(){
   ${CC} -fPIC -g -c subtract.c divide.c
   ${CC} -shared -Wl,-soname,libtraining.so.1 -o libtraining.so.1.0
subtract.o divide.o ${LDFLAGS}
}

do_install(){
   install -d ${D}${libdir}
   install -m 0755 libtraining.so.1.0 ${D}${libdir}
   ln -s libtraining.so.1.0 ${D}/${libdir}/libtraining.so.1
   ln -s libtraining.so.1 ${D}/${libdir}/libtraining.so
}
```

We then bitbake the recipe:

```
Bitbake libtraining-dyn
```

We then need to create a application recipe to test the dynamic library. We need to ensure that we explicitly add build dependency to libtraining-dyn module.

test-libtraining-dyn_1.0.bb:

```
DESCRIPTION = "Simple program to test dynamic library"
```

```
SECTION = "examples"
LICENSE = "CLOSED"
DEPENDS = "libtraining-dyn"

SRC_URI = "file://test-libtraining-dyn.c"
S = "${WORKDIR}"

INSANE_SKIP_${PN} = "ldflags"

do_compile(){
   ${CC} -o test-libtraining-dyn test-libtraining-dyn.c -ltraining
}

do_install(){
   install -d ${D}${bindir}
   install -m 0755 test-libtraining-dyn ${D}${bindir}
}
```

6. Using standard SDK for application development

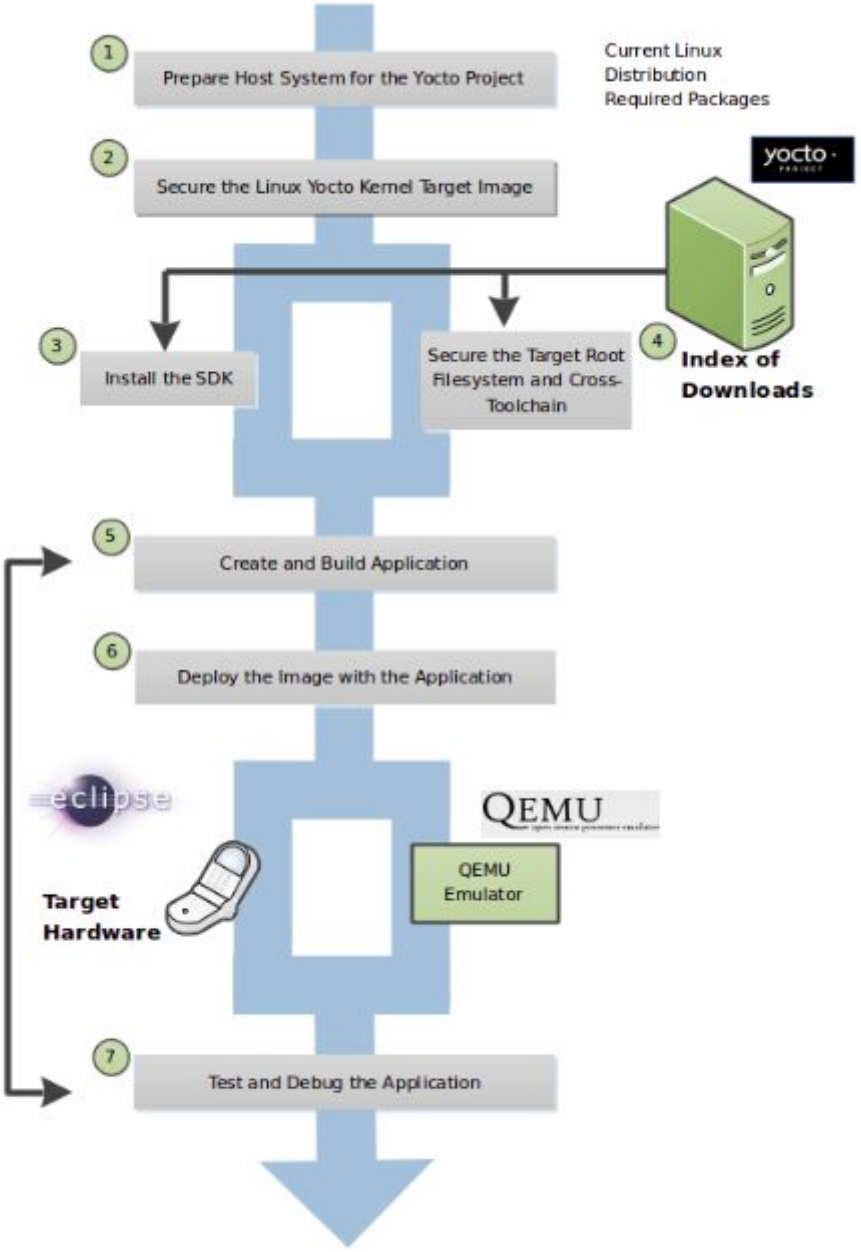
We first bitbake SDK tool on core-image-minimal image:

```
Bitbake -c populate_sdk core-image-minimal
```

We then run the generated .sh file to install the SDK.

7. Application development using Eclipse IDE

Workflow Using Eclipse:

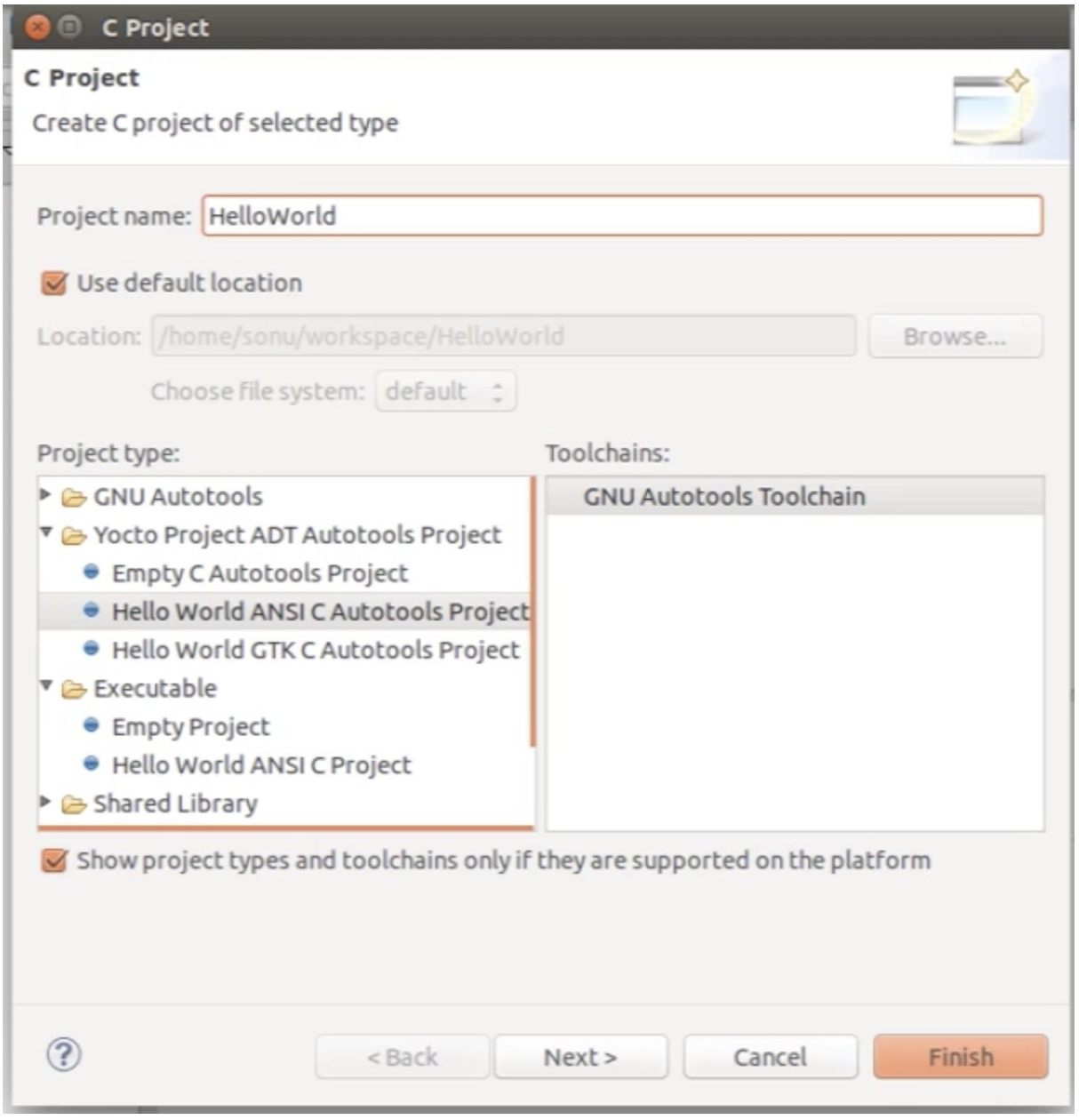


1. ***Create and build your application***: At this point, you need to have source files for your application. Once you have the files, you can use the Eclipse IDE to import them and build the project. If you are not using Eclipse, you need to use the cross-development tools you have installed to create the image.
2. ***Deploy the image with the application***: If you are using the Eclipse IDE, you can deploy your image to the hardware or to QEMU through the project's preferences. If you are not using the Eclipse IDE, then you need to deploy the application to the hardware using other methods. Or, if you are using QEMU, you need to use that tool and load your image in for testing.

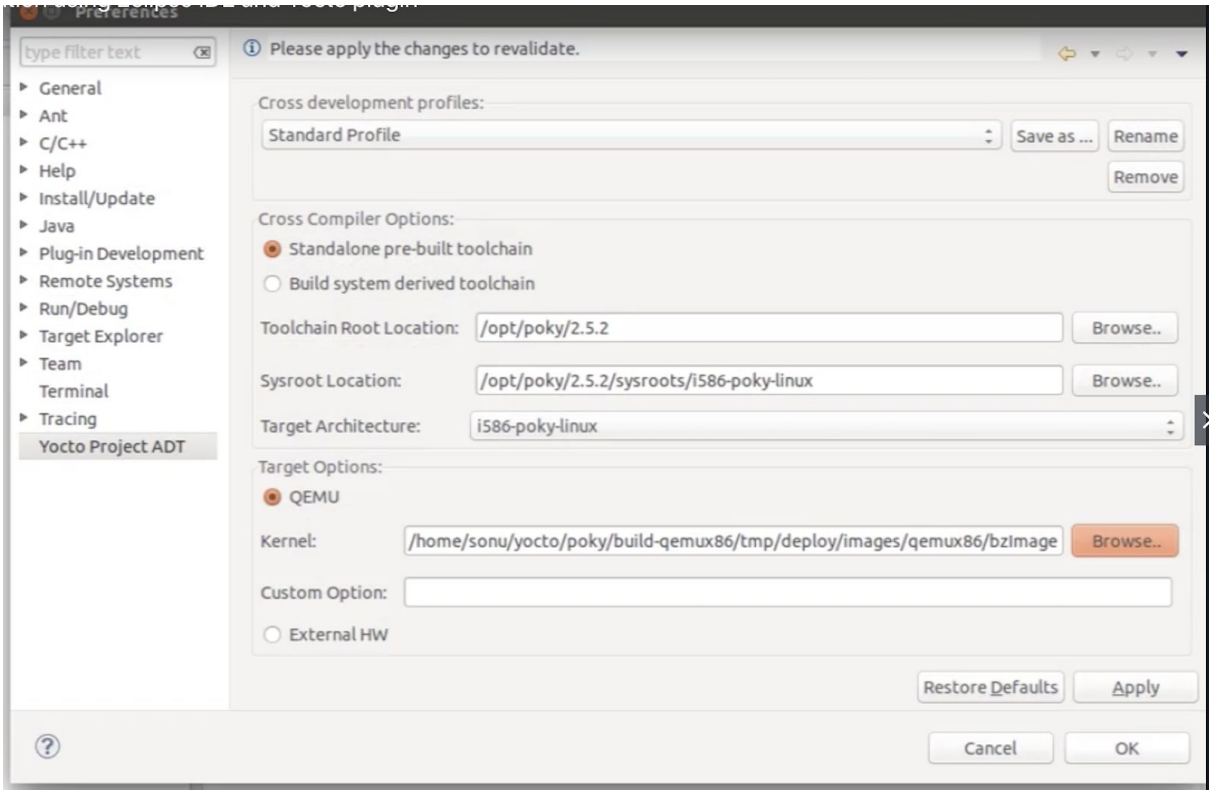
3. ***Test and debug the application***: Once your application is deployed, you need to test it. Within the Eclipse IDE, you can use the debugging environment along with the set of installed user-space tools to debug your application. Of course, the same user-space tools are available separately if you choose not to use the Eclipse IDE.

https://examples.javacodegeeks.com/desktop-java/ide/eclipse/eclipse-ide-yocto-plugin-tutorial/

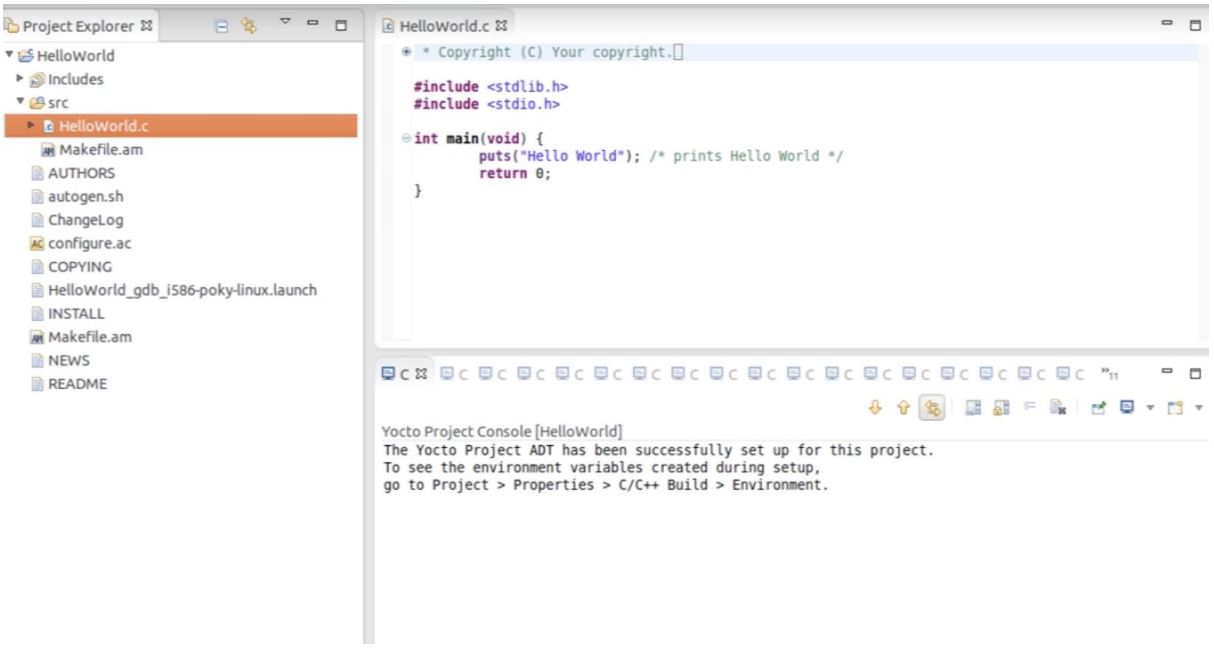
To build applications using Eclipse IDE:



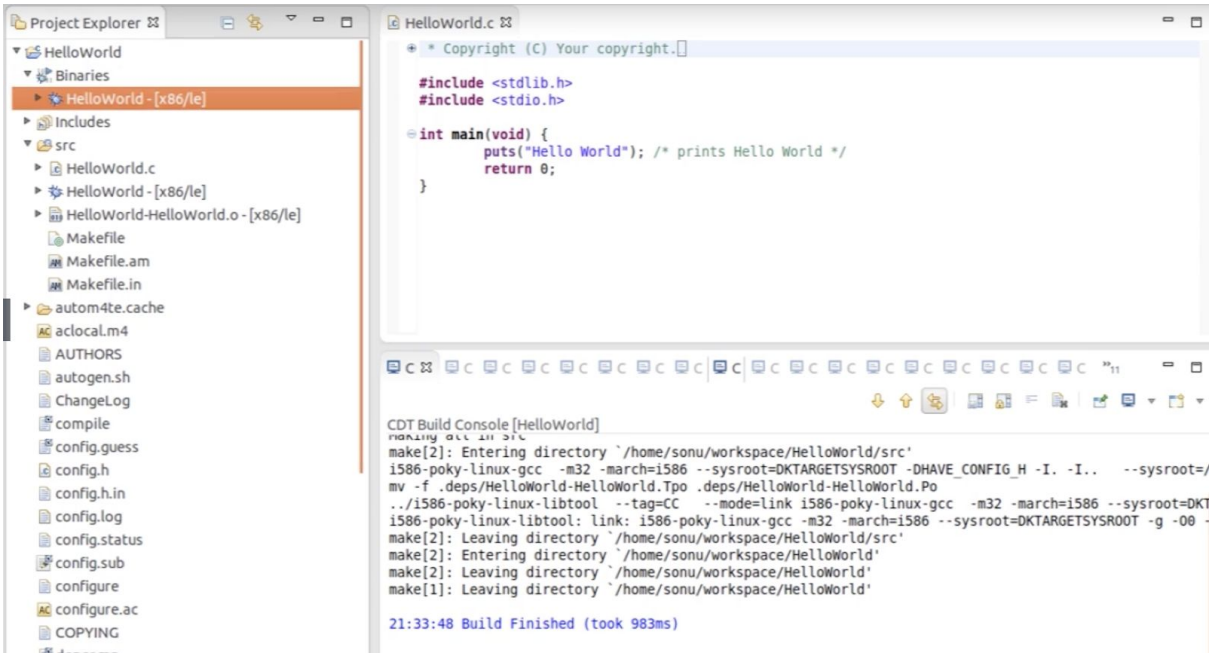
Then configure project ADT:



Then we can develop the project as normal desktop project:



Once we compile the project we will create an application binary in Binaries folder:



8. Using Quilt in Your Workflow

Quilt is a powerful tool that allows you to capture source code changes without having a clean source tree.

Follow these general steps:

1. ***Find the Source Code:*** Temporary source code used by the OpenEmbedded build system is kept in the Build Directory.
2. ***Change Your Working Directory:*** You need to be in the directory that has the temporary source code. That directory is defined by the `S` variable.
3. ***Create a New Patch:*** Before modifying source code, you need to create a new patch. To create a new patch file, use `quilt new` as below:

```
$ quilt new my_changes.patch
```

4. ***Notify Quilt and Add Files:*** After creating the patch, you need to notify Quilt about the files you plan to edit. You notify Quilt by adding the files to the patch you just created:

```
$ quilt add file1.c file2.c file3.c
```

5. ***Edit the Files:*** Make your changes in the source code to the files you added to the patch.

6. ***Test Your Changes:*** Once you have modified the source code, the easiest way to your changes is by calling the `do_compile` task as shown in the following example:

```
$ bitbake -c compile -f package
```

   The `-f` or `--force` option forces the specified task to execute. If you find problems with your code, you can just keep editing and re-testing iteratively until things work as expected.

7. ***Generate the Patch:*** Once your changes work as expected, you need to use Quilt to generate the final patch that contains all your modifications:

```
$ quilt refresh
```

   At this point, the `my_changes.patch` file has all your edits made to the `file1.c`, `file2.c`, and `file3.c` files.
   You can find the resulting patch file in the `patches/` subdirectory of the source (`S`) directory.

8. ***Copy the Patch File:*** For simplicity, copy the patch file into a directory named `files`, which you can create in the same directory that holds the recipe (`.bb`) file or the append (`.bbappend`) file. Placing the patch here guarantees that the OpenEmbedded build system will find the patch. Next, add the patch into the `SRC_URI` of the recipe. Here is an example:

```
SRC_URI += "file://my_changes.patch"
```

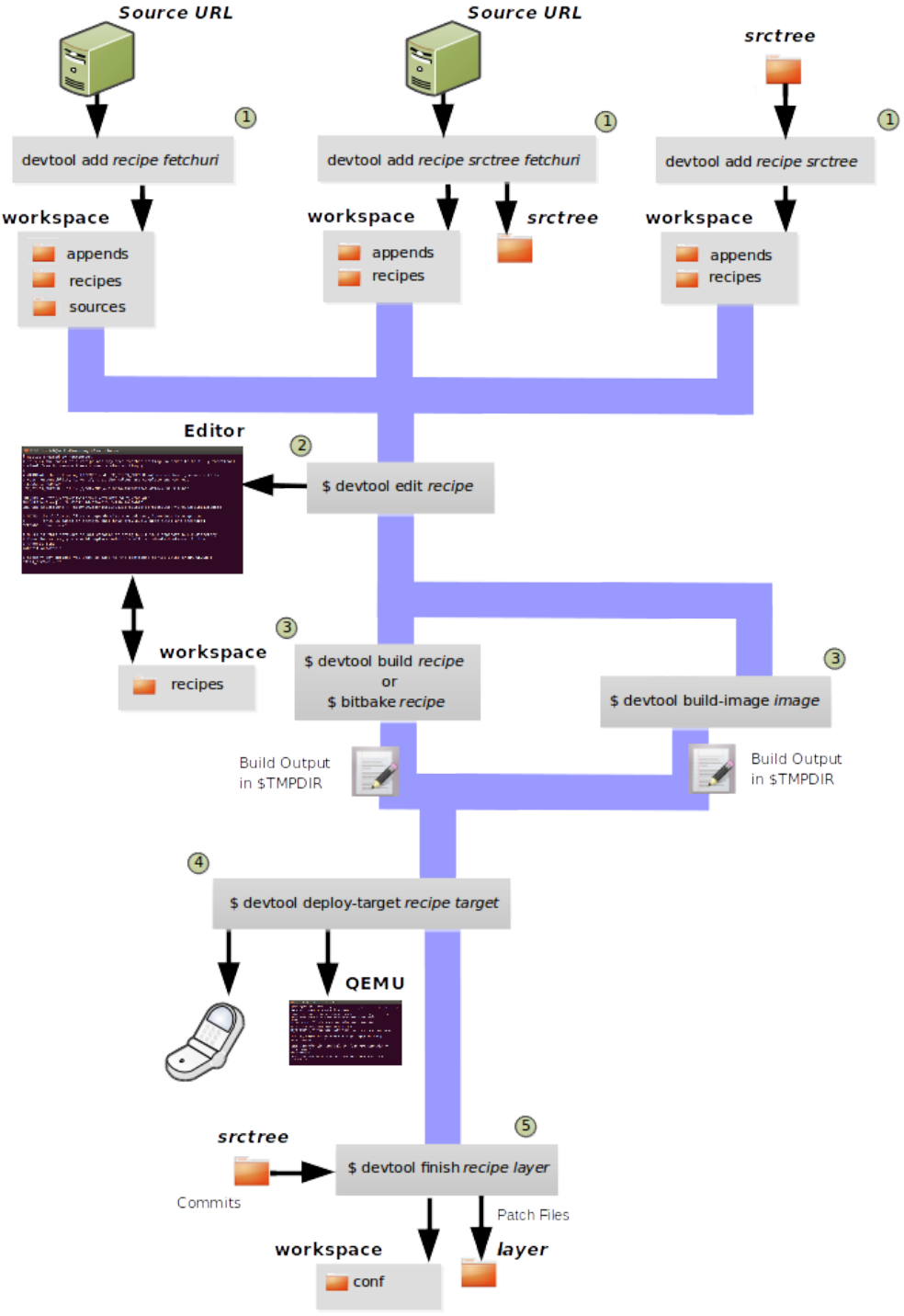
9. Using devtool in Your Workflow

Three entry points exist that allow you to develop using `devtool`:

- ***`devtool add`***
- ***`devtool modify`***
- ***`devtool upgrade`***

1). Use devtool add to Add an Application

The devtool add command generates a new recipe based on existing source code. This command takes advantage of the workspace layer that many devtool commands use. The command is flexible enough to allow you to extract source code into both the workspace or a separate local Git repository and to use existing code that does not need to be extracted.



Generating the New Recipe: The top part of the flow shows three scenarios by which you could use `devtool add` to generate a recipe based on existing source code.

In a shared development environment, it is typical where other developers are responsible for various areas of source code. As a developer, you are probably interested in using that source code as part of your development using the Yocto Project. All you need is access to the code, a recipe, and a controlled area in which to do your work.

Within the diagram, three possible scenarios feed into the `devtool add` workflow:

Left: The left scenario represents a common situation where the source code does not exist locally and needs to be extracted. In this situation, you just let it get extracted to the default workspace - you do not want it in some specific location outside of the workspace. Thus, everything you need will be located in the workspace:

```
$ devtool add recipe fetchuri
```

With this command, `devtool` creates a recipe and an append file in the workspace as well as extracts the upstream source files into a local Git repository also within the `sources` folder.

Middle: The middle scenario also represents a situation where the source code does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area - this time outside of the default workspace.

If required, `devtool` always creates a Git repository locally during the extraction. Furthermore, the first positional argument *srctree* in this case identifies where the `devtool add` command will locate the extracted code outside of the workspace:

```
$ devtool add recipe srctree fetchuri
```

In summary, the source code is pulled from fetchuri and extracted into the location defined by srctree as a local Git repository.

Within the workspace, devtool creates both the recipe and an append file for the recipe.

Right: The right scenario represents a situation where the source tree (srctree) has been previously prepared outside of the `devtool` workspace.

The following command names the recipe and identifies where the existing source tree is located:

```
$ devtool add recipe srctree
```

The command examines the source code and creates a recipe for it placing the recipe into the workspace.

Because the extracted source code already exists, `devtool` does not try to relocate it into the workspace - just the new recipe is placed in the workspace. Aside from a recipe folder, the command also creates an append folder and places an initial `*.bbappend` within.

Edit the Recipe: At this point, you can use `devtool edit-recipe` to open up the editor as defined by the `$EDITOR` environment variable and modify the file:

```
$ devtool edit-recipe recipe
```

From within the editor, you can make modifications to the recipe that take effect when you build it later.

Build the Recipe or Rebuild the Image: At this point in the flow, the next step you take depends on what you are going to do with the new code.

If you need to take the build output and eventually move it to the target hardware, you would use `devtool build`:

```
$ devtool build recipe
```

On the other hand, if you want an image to contain the recipe's packages for immediate deployment onto a device (e.g. for testing purposes), you can use the `devtool build-image` command:

```
$ devtool build-image image
```

Deploy the Build Output: When you use the `devtool build` command to build out your recipe, you probably want to see if the resulting build output works as expected on target hardware.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The *target* is a live target machine running as an SSH server.

You can, of course, also deploy the image you build using the `devtool build-image` command to actual hardware. However, `devtool` does not provide a specific command that allows you to do this.

Finish Your Work With the Recipe: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace.

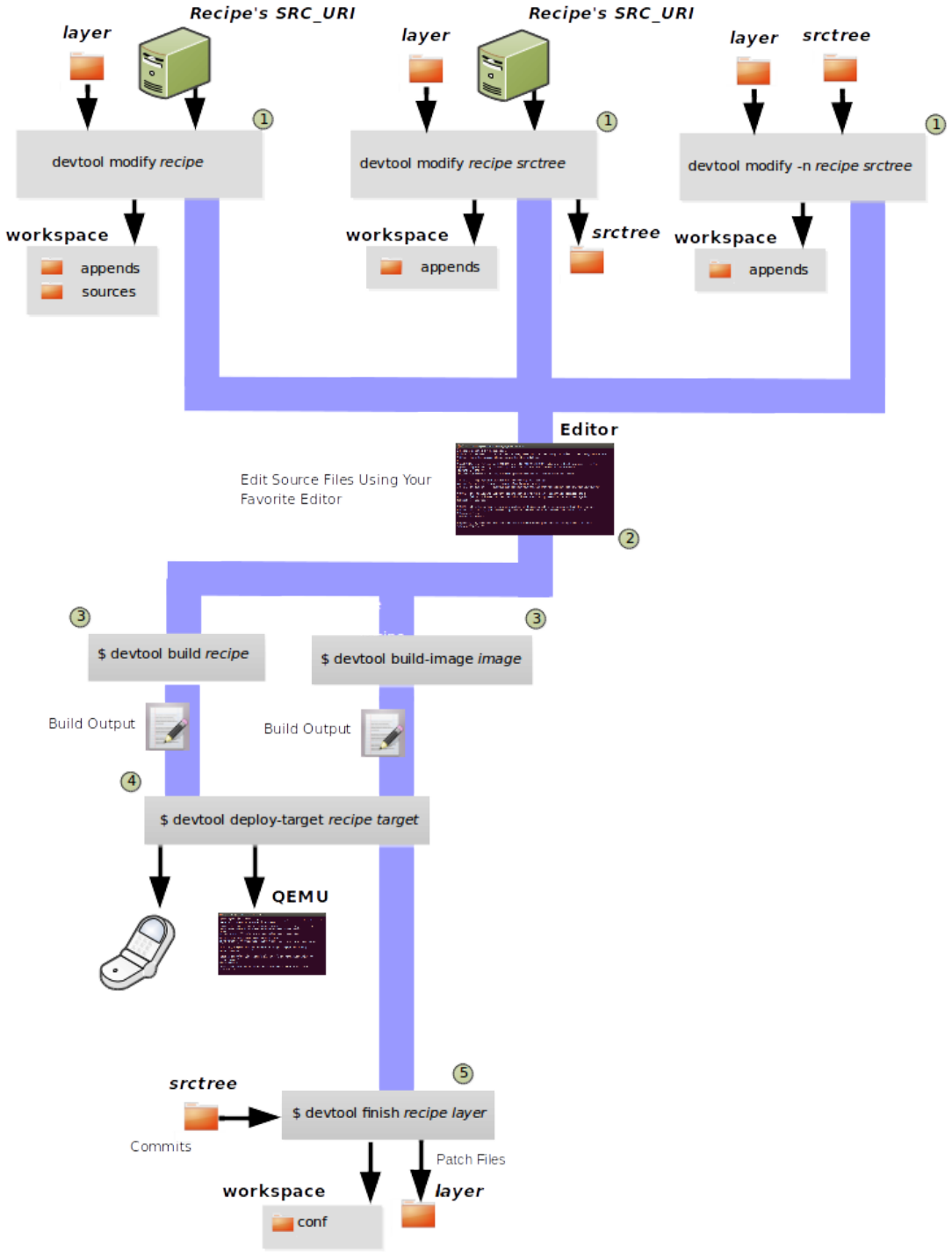
```
$ devtool finish recipe layer
```

Note: Any changes you want to turn into patches must be committed to the Git repository in the source tree.

As mentioned, the `devtool finish` command moves the final recipe to its permanent layer. As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

2). Use devtool modify to Modify the Source of an Existing Component

The `devtool modify` command prepares the way to work on existing code that already has a recipe in place. The command is flexible enough to allow you to extract code, specify the existing recipe, and keep track of and gather any patch files from other developers that are associated with the code.



Preparing to Modify the Code: The top part of the flow shows three scenarios by which you could use `devtool modify` to prepare to work on source files. Each scenario assumes the following:

- The recipe exists in some layer external to the `devtool` workspace.

- The source files exist upstream in an un-extracted state or locally in a previously extracted state.

The typical situation is where another developer has created some layer for use with the Yocto Project and their recipe already resides in that layer. Furthermore, their source code is readily available either upstream or locally.

Left: The left scenario represents a common situation where the source code does not exist locally and needs to be extracted. In this situation, the source is extracted into the default workspace location. The recipe, in this scenario, is in its own layer outside the workspace (i.e. `meta-`*layername*).

The following command identifies the recipe and by default extracts the source files:

```
$ devtool modify recipe
```

Once `devtool` locates the recipe, it uses the `SRC_URI` variable to locate the source code and any local patch files from other developers are located.

Note: You cannot provide an URL for srctree when using the devtool modify command.

With this scenario, however, since no *srctree* argument exists, the `devtool modify` command by default extracts the source files to a Git structure. Furthermore, the location for the extracted source is the default area within the workspace. The result is that the command sets up both the source code and an append file within the workspace with the recipe remaining in its original location.

Middle: The middle scenario represents a situation where the source code also does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area as a Git repository. The recipe, in this scenario, is again in its own layer outside the workspace.

The following command tells `devtool` what recipe with which to work and, in this case, identifies a local area for the extracted source files that is outside of the default workspace:

```
$ devtool modify recipe srctree
```

As with all extractions, the command uses the recipe's `SRC_URI` to locate the source files. Once the files are located, the command by default extracts them.

Providing the *srctree* argument instructs `devtool` where to place the extracted source.

Within the workspace, `devtool` creates an append file for the recipe. The recipe remains in its original location but the source files are extracted to the location you provided with *srctree*.

Right: The right scenario represents a situation where the source tree (*srctree*) exists as a previously extracted Git structure outside of the `devtool` workspace. In this example, the recipe also exists elsewhere in its own layer.

The following command tells `devtool` the recipe with which to work, uses the "-n" option to indicate source does not need to be extracted, and uses *srctree* to point to the previously extracted source files:

```
$ devtool modify -n recipe srctree
```

Once the command finishes, it creates only an append file for the recipe in the workspace. The recipe and the source code remain in their original locations.

Edit the Source: Once you have used the `devtool modify` command, you are free to make changes to the source files. You can use any editor you like to make and save your source code modifications.

Build the Recipe: Once you have updated the source files, you can build the recipe.

Build the Recipe: Once you have updated the source files, you can build the recipe.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The *target* is a live target machine running as an SSH server.

Finish Your Work With the Recipe: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, updates the recipe to point to them (or creates a `.bbappend` file to do so, depending on the specified destination layer), and then resets the recipe so that the recipe is built normally rather than from the workspace.

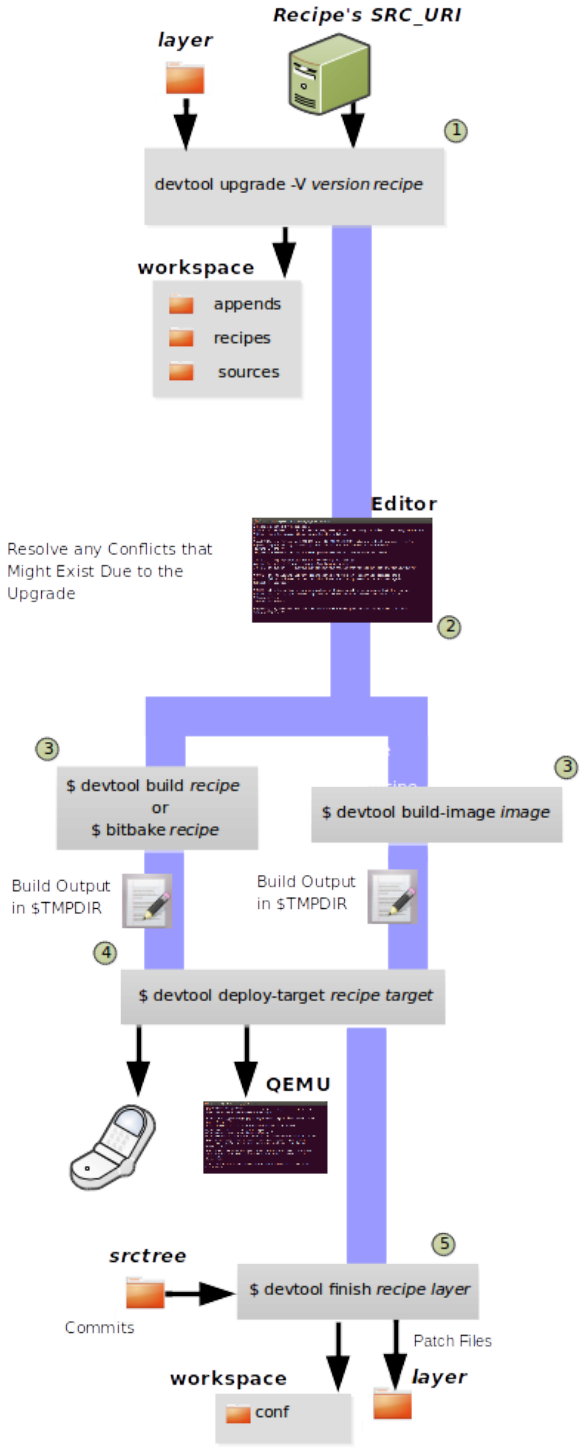
```
$ devtool finish recipe layer
```

Because there is no need to move the recipe, `devtool finish` either updates the original recipe in the original layer or the command creates a `.bbappend` in a different layer as provided by *layer*.

As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

3). Use devtool upgrade to Create a Version of the Recipe that Supports a Newer Version of the Software

The `devtool upgrade` command updates an existing recipe so that you can build it for an updated set of source files. The command is flexible enough to allow you to specify source code revision and versioning schemes, extract code into or out of the `devtool` workspace, and work with any source file forms that the fetchers support.



Initiate the Upgrade: The top part of the flow shows a typical scenario by which you could use `devtool upgrade`. The following conditions exist:

- The recipe exists in some layer external to the `devtool` workspace.

- The source files for the new release exist adjacent to the same location pointed to by SRC_URI in the recipe (e.g. a tarball with the new version number in the name, or as a different revision in the upstream Git repository).

A common situation is where third-party software has undergone a revision so that it has been upgraded. The recipe you have access to is likely in your own layer. Thus, you need to upgrade the recipe to use the newer version of the software:

```
$ devtool upgrade -V version recipe
```

By default, the devtool upgrade command extracts source code into the sources directory in the workspace. If you want the code extracted to any other location, you need to provide the *srctree* positional argument with the command as follows:

```
$ devtool upgrade -V version recipe srctree
```

Also, in this example, the "-V" option is used to specify the new version. If the source files pointed to by the SRC_URI statement in the recipe are in a Git repository, you must provide the "-S" option and specify a revision for the software.

Once devtool locates the recipe, it uses the SRC_URI variable to locate the source code and any local patch files from other developers are located. The result is that the command sets up the source code, the new version of the recipe, and an append file all within the workspace.

Resolve any Conflicts created by the Upgrade: At this point, there could be some conflicts due to the software being upgraded to a new version. This would occur if your recipe specifies some patch files in SRC_URI that conflict with changes made in the new version of the software. If this is the case, you need to resolve the conflicts by editing the source and following the normal git rebase conflict resolution process.

Build the Recipe: Once you have your recipe in order, you can build it. You can either use devtool build or bitbake. Either method produces build output that is stored in TMPDIR.

Deploy the Build Output: When you use the `devtool build` command or `bitbake` to build out your recipe, you probably want to see if the resulting build output works as expected on target hardware.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The *target* is a live target machine running as an SSH server.

Finish Your Work With the Recipe: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace. If you specify a destination layer that is the same as the original source, then the old version of the recipe and associated files will be removed prior to adding the new version.

```
$ devtool finish recipe layer
```

As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

Note:

1).before running devtool deploy-target command, we need to make sure the ssh is live on the target machine. If not, we might need to add openssh package to the core image recipe, rebuild it and either run it in a QEMU emulator or on the real hardware.

We can test if sshd is running by using 'ps aux | grep sshd' command. Once the sshd is up running, we need to find the IP address of the target hardware and run the deploy command:

```
Devtool deploy-target -s application root@192.168.7.2
```

The default path is /sbin/, so the application will be installed at /sbin/mdam on the target platform.

2). Instead of using the ‘devtool finish layer’ command to update the recipe, we can use ‘devtool update-recipe recipe’ to explicitly update the recipe. This will typically generate a new patch that matches the local git commits.

10. Using Toaster

Toaster is a web interface to the Yocto Project's OpenEmbedded build system. You can initiate builds using Toaster as well as examine the results and statistics of builds.

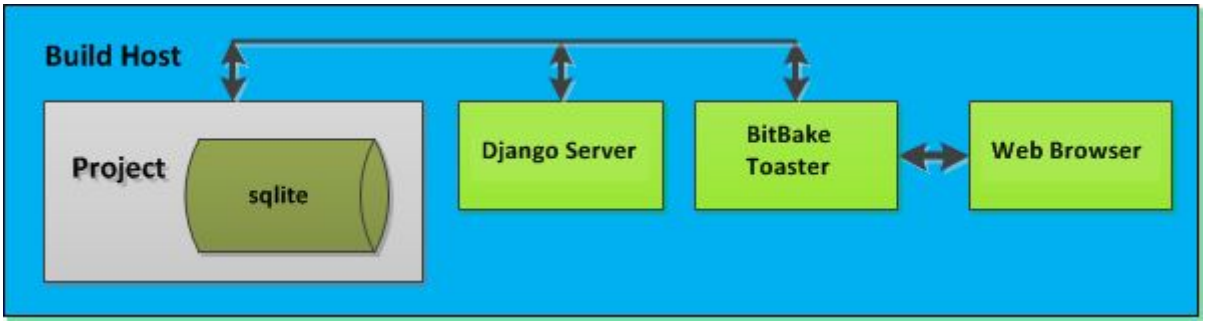
With the Toaster web interface, you can:

- Browse layers listed in the various layer sources that are available in your project (e.g. the OpenEmbedded Metadata Index at http://layers.openembedded.org/layerindex/).
- Browse images, recipes, and machines provided by those layers.
- Import your own layers for building.
- Add and remove layers from your configuration.
- Set configuration variables.
- Select a target or multiple targets to build.
- Start your builds.

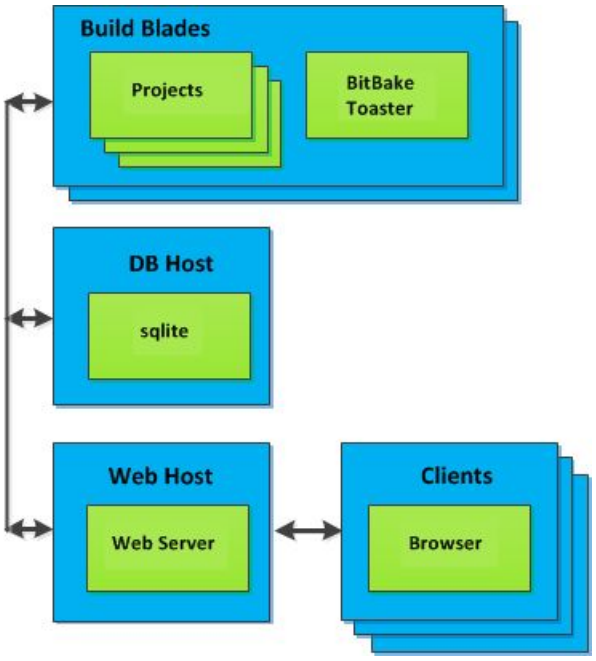
Toaster also allows you to configure and run your builds from the command line, and switch between the command line and the web interface at any time. Builds started from the command line appear within a special Toaster project called "Command line builds".

1). Installation options

You can set Toaster up to run as a local instance or as a shared hosted service. When Toaster is set up as a local instance, all the components reside on a single build host. Fundamentally, a local instance of Toaster is suited for a single user developing on a single build host.



Toaster as a hosted service is suited for multiple users developing across several build hosts. When Toaster is set up as a hosted service, its components can be spread across several machines:



You need to install the packages that Toaster requires. Use this command:

```
$ $ pip3 install --user -r bitbake/toaster-requirements.txt
```

The previous command installs the necessary Toaster modules into a local python 3 cache in your `$HOME` directory. The caches are actually located in `$HOME/.local`. To see what packages have been installed into your `$HOME` directory, do the following:

```
$ pip3 list installed --local
```

If you need to remove something, the following works:

```
$ pip3 uninstall PackageNameToUninstall
```

2). Starting Toaster for Local Development

Once you have set up the Yocto Project and installed the Toaster system dependencies, you are ready to start Toaster.Navigate to the root of your Source Directory (e.g. poky):

```
$ cd poky
```

Once in that directory, source the build environment script:

```
$ source oe-init-build-env
```

Next, from the build directory (e.g. poky/build), start Toaster using this command:

```
$ source toaster start
```

You can now run your builds from the command line, or with Toaster. To access the Toaster web interface, open your favorite browser and enter the following:

```
http://127.0.0.1:8000
```

3). Setting a Different Port

By default, Toaster starts on port 8000. You can use the WEBPORT parameter to set a different port. For example, the following command sets the port to "8400":

```
$ source toaster start webport=8400
```

4). Setting a Different Address

By default, Toaster binds to the loopback address (i.e. localhost). You can use the WEBPORT parameter to set a different host. For example, the following command sets the host and port to "0.0.0.0:8400":

```
$ source toaster start webport=0.0.0.0:8400
```

5). The Directory for Cloning Layers

Toaster creates a _toaster_clones directory inside your Source Directory (i.e. poky) to clone any layers needed for your builds.

Alternatively, if you would like all of your Toaster related files and directories to be in a particular location other than the default, you can set the TOASTER_DIR environment variable, which takes precedence over your current working directory. Setting this environment variable causes Toaster to create and use $TOASTER_DIR./_toaster_clones.

6). The Build Directory

Toaster creates a build directory within your Source Directory (e.g. poky) to execute the builds.

Alternatively, if you would like all of your Toaster related files and directories to be in a particular location, you can set the TOASTER_DIR environment variable, which takes precedence over your current working directory. Setting this environment variable causes Toaster to use $TOASTER_DIR/build as the build directory.

Section 5: Linux Kernel Development

1.From linux-korg_4.4.bb:

FILESEXTRAPATHS: Extends the search path the OpenEmbedded build system uses when looking for files and patches as it processes recipes and append files. The default directories BitBake uses when it processes recipes are initially defined by the FILESPATH variable. You can extend FILESPATH variable by using FILESEXTRAPATHS.

Best practices dictate that you accomplish this by using FILESEXTRAPATHS from within a .bbappend file and that you prepend paths as follows:

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

In the above example, the build system first looks for files in a directory that has the same name as the corresponding append file.

```
##################################################################
# Download locations and utilities for Yocto
##################################################################

GNU_MIRROR = "ftp://ftp.gnu.org/gnu"
DEBIAN_MIRROR = "ftp://ftp.debian.org/debian/pool"
SOURCEFORGE_MIRROR = "http://heanet.dl.sourceforge.net/sourceforge"
GPE_MIRROR = "http://gpe.linuxtogo.org/download/source"
GPE_SVN = "svn://projects.linuxtogo.org/svn/gpe/trunk/base;module=${PN}"
XLIBS_MIRROR = "http://xlibs.freedesktop.org/release"
XORG_MIRROR = "http://xorg.freedesktop.org/releases"
GNOME_MIRROR = "http://ftp.gnome.org/pub/GNOME/sources"
FREEBSD_MIRROR = "ftp://ftp.freebsd.org/pub/FreeBSD/"
HANDHELDS_CVS = "cvs://anoncvs:anoncvs@anoncvs.handhelds.org/cvs"
E_CVS = "cvs://anonymous@anoncvs.enlightenment.org/var/cvs/e"
E_URI = "http://enlightenment.freedesktop.org/files"
FREEDESKTOP_CVS = "cvs://anoncvs:anoncvs@anoncvs.freedesktop.org/cvs"
GENTOO_MIRROR = "http://distro.ibiblio.org/pub/linux/distributions/gentoo/distfiles"
APACHE_MIRROR = "http://www.apache.org/dist"
KERNELORG_MIRROR = "http://kernel.org/"
```

COMPATIBLE_MACHINE: A regular expression that resolves to one or more target machines with which a recipe is compatible. You can use the variable to stop recipes from being built for machines with which the recipes are not compatible. Stopping these builds is particularly useful with kernels. The variable also helps to increase parsing speed since the build system skips parsing recipes not compatible with the current machine.

To configure kernel using bitbake:

```
Bitbake -c menuconfig virtual/kernel
```

virtual/kernel refers to the current kernel.

And then we force to compile and deploy the re-configured kernel:

```
Bitbake -c compile -f virtual/kernel
Bitbake -c deploy virtual/kernel
```

Upon verifying the kernel is working as expected, we can copy the resulting config file to the directory of the target kernel recipe as defconfig. For example:

```
cp tmp/work/kernel_lab1_qemux86-poky-linux/linux-korg/4.4-r0/build/.config
~/Desktop/yocto/poky/meta-kernel-lab1-qemux86/recipes-kernel/linux/linux-korg/defconfig
```

KMACHINE:

The machine as known by the kernel. Sometimes the machine name used by the kernel does not match the machine name used by the OpenEmbedded build system. For example, the machine name that the OpenEmbedded build system understands as `qemuarm` goes by a different name in the Linux Yocto kernel. The kernel understands that machine as `arm_versatile926ejs`. For cases like these, the `KMACHINE` variable maps the kernel machine name to the OpenEmbedded build system machine name.

Kernel machine names are initially defined in the Yocto Linux Kernel's `meta` branch. From the `meta` branch, look in the `meta/cfg/kernel-cache/bsp/<bsp_name>/<bsp-name>-<kernel-type>.scc` file. For example, from the `meta` branch in the `linux-yocto-3.0` kernel, the `meta/cfg/kernel-cache/bsp/cedartrail/cedartrail-standard.scc` file has the following:

```
define KMACHINE cedartrail
    define KTYPE standard
    define KARCH i386

    include ktypes/standard
    branch cedartrail
```

```
include cedartrail.scc
```

You can see that the kernel understands the machine name for the Cedar Trail Board Support Package (BSP) as `cedartrail`.

If you look in the Cedar Trail BSP layer in the `meta-intel` Source Repositories at `meta-cedartrail/recipes-kernel/linux/linux-yocto_3.0.bbappend`, you will find the following statements among others:

```
COMPATIBLE_MACHINE_cedartrail = "cedartrail"
    KMACHINE_cedartrail  = "cedartrail"
    KBRANCH_cedartrail  = "yocto/standard/cedartrail"
    KERNEL_FEATURES_append_cedartrail += "bsp/cedartrail/cedartrail-pvr-merge.scc"
    KERNEL_FEATURES_append_cedartrail += "cfg/efi-ext.scc"

    COMPATIBLE_MACHINE_cedartrail-nopvr = "cedartrail"
    KMACHINE_cedartrail-nopvr  = "cedartrail"
    KBRANCH_cedartrail-nopvr  = "yocto/standard/cedartrail"
    KERNEL_FEATURES_append_cedartrail-nopvr += " cfg/smp.scc"
```

The `KMACHINE` statements in the kernel's append file make sure that the OpenEmbedded build system and the Yocto Linux kernel understand the same machine names.

This append file uses two `KMACHINE` statements. The first is not really necessary but does ensure that the machine known to the OpenEmbedded build system as `cedartrail` maps to the machine in the kernel also known as `cedartrail`:

```
KMACHINE_cedartrail  = "cedartrail"
```

The second statement is a good example of why the `KMACHINE` variable is needed. In this example, the OpenEmbedded build system uses the `cedartrail-nopvr` machine name to refer to the Cedar Trail BSP that does not support the proprietary PowerVR driver. The kernel, however, uses the machine name `cedartrail`. Thus, the append file must map the `cedartrail-nopvr` machine name to the kernel's `cedartrail` name:

```
KMACHINE_cedartrail-nopvr  = "cedartrail"
```

BSPs that ship with the Yocto Project release provide all mappings between the Yocto Project kernel machine names and the OpenEmbedded machine names.

2. Kernel Types

The `LINUX_KERNEL_TYPE` variable in the kernel recipe selects the kernel type. As an example, the `linux-yocto-3.4` tree defines three kernel types:

"standard", "tiny", and "preempt-rt":

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

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

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

    # Include all standard features
    include standard-nocfg.scc

    kconf non-hardware standard.cfg

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

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

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

3. BSP description

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

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

    kconf mybsp.cfg
```

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

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

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

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

In the above example, `standard.scc` aggregates all the configuration fragments, patches, and features that make up your standard kernel policy whereas `mybsp-hw.scc` aggregates all those necessary to support the hardware available on the "mybsp" machine.

Many real-world examples are more complex. Like any other `.scc` file, BSP descriptions can aggregate features. Consider the Fish River Island 2 (fri2) BSP definition from the `linux-yocto-3.4` Git repository:

```
fri2.scc:
```

```
    kconf hardware fri2.cfg

    include cfg/x86.scc
    include features/eg20t/eg20t.scc
    include cfg/dmaengine.scc
    include features/ericsson-3g/f5521gw.scc
    include features/power/intel.scc
    include cfg/efi.scc
    include features/usb/ehci-hcd.scc
    include features/usb/ohci-hcd.scc
    include features/iwlwifi/iwlwifi.scc
```

The `fri2.scc` description file includes a hardware configuration fragment (`fri2.cfg`) specific to the Fish River Island 2 BSP as well as several more general configuration fragments and features enabling hardware found on the machine.

This description file is then included in each of the three "fri2" description files for the supported kernel types (i.e. "standard", "preempt-rt", and "tiny"). Consider the "fri2" description for the "standard" kernel type:

```
fri2-standard.scc:
    define KMACHINE fri2
    define KTYPE standard
    define KARCH i386

    include ktypes/standard/standard.scc
    branch fri2

    git merge emgd-1.14

    include fri2.scc

    # Extra fri2 configs above the minimal defined in fri2.scc
    include cfg/efi-ext.scc
    include features/drm-emgd/drm-emgd.scc
    include cfg/vesafb.scc

    # default policy for standard kernels
    include cfg/usb-mass-storage.scc
```

The `include` command midway through the file includes the `fri2.scc` description that defines all hardware enablements for the BSP that is common to all kernel types. Using this command significantly reduces duplication.

`branch fri2` command creates a machine-specific branch into which source changes are applied. With this branch set up, the `git merge` command uses Git to merge in a feature branch named "emgd-1.14".

Now consider the "fri2" description for the "tiny" kernel type:

```
fri2-tiny.scc:
        define KMACHINE fri2
        define KTYPE tiny
        define KARCH i386

        include ktypes/tiny/tiny.scc
        branch fri2

        include fri2.scc
```

It includes only the minimal policy defined by the "tiny" kernel type and the hardware-specific configuration required for booting the machine along with the most basic functionality of the system as defined in the base "fri2" description file.

4. Reorganizing source

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

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

```
KBRANCH = "mynewbranch"
```

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

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

        branch mynewbranch

        include mybsp-hw.scc
```

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

```
<common>/<kernel_type>/<machine>
```

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

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

In this case, `common/standard/machine_a` includes everything in `common/base` and `common/standard/base`. The "standard" and "small" branches add sources specific to those kernel types that for whatever reason are not appropriate for the other branches.

5. Using feature branches

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

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

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

      branch mynewbranch
      git merge myfeature

      include mybsp-hw.scc
```

6. SCC Description File Reference

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

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

7. Features

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

Here is an example that shows a feature description file:

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

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

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

8. Kernel Metadata Syntax

Kernel Metadata can be defined in either the kernel recipe (recipe-space) or in the kernel tree (in-tree).

If you are unfamiliar with the Linux kernel and only wish to apply a configuration and possibly a couple of patches provided to you by others, the recipe-space method is recommended. This method is also a good approach if you are working with Linux kernel sources you do not control or if you just do not want to maintain a Linux kernel Git repository on your own.

Conversely, if you are actively developing a kernel and are already maintaining a Linux kernel Git repository of your own, you might find it more convenient to work with the kernel Metadata in the same repository as the Linux kernel sources. This method can make iterative development of the Linux kernel more efficient outside of the BitBake environment.

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

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

- Features
- Board Support Packages (BSPs)

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

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

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

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

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

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

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

9. Recipe-Space Metadata

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

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

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

It is only necessary to specify the `.scc` files on the `SRC_URI`. BitBake parses them and fetches any files referenced in the `.scc` files by the `include`, `patch`, or `kconf` commands. Because of this, it is necessary to bump the recipe `PR` value when changing the content of files not explicitly listed in the `SRC_URI`.

10. In-Tree Metadata

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

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

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

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

```
KMETA = "meta"
```

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

```
KMETA = ""
```

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

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

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

11. Configurations

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

The Symmetric Multi-Processing (SMP) fragment included in the `linux-yocto-3.4` Git repository consists of the following two files:

```
cfg/smp.scc:
        define KFEATURE_DESCRIPTION "Enable SMP"
        kconf hardware smp.cfg

cfg/smp.cfg:
        CONFIG_SMP=y
        CONFIG_SCHED_SMT=y
```

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

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

You can use the following BitBake command to audit your configuration:

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

12. Patches

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

```
patches/mypatch.scc:
        patch mypatch.patch
patches/mypatch.patch:
        <typical-patch>
```

You can create the typical `.patch` file using `diff -Nurp` or `git format-patch`. The description file can include multiple patch statements, one per patch.

13. How to examine the changes in the kernel source

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

Here is an example that looks at what has changed in the `emenlow` branch of the `linux-yocto-3.4` kernel. The lower commit range is the commit associated with the `standard/base` branch, while the upper commit range is the commit associated with the `standard/emenlow` branch:

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

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

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

3). Use git diff command to see code differences for the changes:

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

4). Use git show command to see the commit log messages and the text differences:

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

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

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

14. Modifying Source Code

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

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

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

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

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

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

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

```
 ${WORKDIR}/linux-${MACHINE}-${KTYPE}-build
```

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

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

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

```
$ bitbake linux-yocto
```

15. Working With Your Own Sources

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

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

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

1. Copy the `linux-yocto-custom.bb` recipe to your layer and give it a meaningful name. The name should include the version of the Linux kernel you are using (e.g. `linux-yocto-myproject_3.5.bb`, where "3.5" is the base version of the Linux kernel with which you would be working).
2. In the same directory inside your layer, create a matching directory to store your patches and configuration files (e.g. `linux-yocto-myproject`).
3. Edit the following variables in your recipe as appropriate for your project:
    - `SRC_URI`: The `SRC_URI` should be a Git repository that uses one of the supported Git fetcher protocols (i.e. `file`, `git`, `http`, and so forth). The skeleton recipe provides an example `SRC_URI` as a syntax reference.
    - `LINUX_VERSION`: The Linux kernel version you are using (e.g. "3.4").
    - `LINUX_VERSION_EXTENSION`: The Linux kernel `CONFIG_LOCALVERSION` that is compiled into the resulting kernel and visible through the `uname` command.
    - `SRCREV`: The commit ID from which you want to build.
    - `PR`: Treat this variable the same as you would in any other recipe. Increment the variable to indicate to the OpenEmbedded build system that the recipe has changed.
4. `COMPATIBLE_MACHINE`: A list of the machines supported by your new recipe. This variable in the example recipe is set by default to a regular expression that matches only the empty string, "(^$)". This default setting triggers an explicit build failure. You must change it to match a list of the machines that your new recipe supports. For example, to support the `qemux86` and `qemux86-64` machines, use the following form:]

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

16. Incorporating Out-of-Tree Modules

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

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

To get started, copy this recipe to your layer and give it a meaningful name (e.g. `mymodule_1.0.bb`). In the same directory, create a directory named `files` where you can store any source files, patches, or other files necessary for

building the module that do not come with the sources. Finally, update the recipe as appropriate for the module.

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

```
obj-m := hello.o

SRC := $(shell pwd)

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

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

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

KERNEL_SRC: The location of the kernel sources. This variable is set to the value of the `STAGING_KERNEL_DIR` within the `module.bbclass` class.

To help maximize compatibility with out-of-tree drivers used to build modules, the OpenEmbedded build system also recognizes and uses the `KERNEL_PATH` variable, which is identical to the `KERNEL_SRC` variable. Both variables are common variables used by external Makefiles to point to the kernel source directory.

Note: KERNEL_PATH is the same as KERNEL_SRC variable.

STAGING_KERNEL_DIR: The directory with kernel headers that are required to build out-of-tree modules.

After you have prepared your recipe, you will likely want to include the module in your images. To do this, set one of them as appropriate in **your machine configuration file:**

1). MACHINE_ESSENTIAL_EXTRA_RDEPENDS:

A list of required machine-specific packages to install as part of the image being built. The build process depends on these packages being present. Furthermore,

because this is a "machine essential" variable, the list of packages are essential for the machine to boot. The impact of this variable affects images based on `packagegroup-core-boot`, including the `core-image-minimal` image.

This variable is similar to the MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS variable with the exception that the image being built has a build dependency on the variable's list of packages. In other words, the image will not build if a file in this list is not found.

As an example, suppose the machine for which you are building requires `example-init` to be run during boot to initialize the hardware. In this case, you would use the following in the machine's `.conf` configuration file:

```
MACHINE_ESSENTIAL_EXTRA_RDEPENDS += "example-init"
```

2). MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS

A list of recommended machine-specific packages to install as part of the image being built.

This variable is similar to the MACHINE_ESSENTIAL_EXTRA_RDEPENDS variable with the exception that the image being built does not have a build dependency on the variable's list of packages. In other words, the image will still build if a package in this list is not found. Typically, this variable is used to handle essential kernel modules, whose functionality may be selected to be built into the kernel rather than as a module, in which case a package will not be produced.

Consider an example where you have a custom kernel where a specific touchscreen driver is required for the machine to be usable. However, the driver can be built as a module or into the kernel depending on the kernel configuration. If the driver is built as a module, you want it to be installed. But, when the driver is built into the kernel, you still want the build to succeed. This variable sets up a "recommends" relationship so that in the latter case, the build will not fail due to the missing package. To accomplish this, assuming the

package for the module was called `kernel-module-ab123`, you would use the following in the machine's `.conf` configuration file:

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

3). MACHINE_EXTRA_RDEPENDS

A list of machine-specific packages to install as part of the image being built that are not essential for the machine to boot. However, the build process for more fully-featured images depends on the packages being present.

This variable affects all images based on `packagegroup-base`, which does not include the `core-image-minimal` or `core-image-full-cmdline` images.

The variable is similar to the `MACHINE_EXTRA_RRECOMMENDS` variable with the exception that the image being built has a build dependency on the variable's list of packages. In other words, the image will not build if a file in this list is not found.

An example is a machine that has WiFi capability but is not essential for the machine to boot the image. However, if you are building a more fully-featured image, you want to enable the WiFi. The package containing the firmware for the WiFi hardware is always expected to exist, so it is acceptable for the build process to depend upon finding the package. In this case, assuming the package for the firmware was called `wifidriver-firmware`, you would use the following in the `.conf` file for the machine:

```
MACHINE_EXTRA_RDEPENDS += "wifidriver-firmware"
```

4). MACHINE_EXTRA_RRECOMMENDS

A list of machine-specific packages to install as part of the image being built that are not essential for booting the machine. The image being built has no build dependency on this list of packages.

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

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

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

17. Changing the Configuration

If you have a final Linux kernel `.config` file you want to use, copy it to a directory named `files`, which must be in your layer's `recipes-kernel/linux` directory, and name the file "defconfig". Then, add the following lines to your linux-yocto `.bbappend` file in your layer:

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

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

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

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

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

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

18. Using an Iterative Development Process

1). "-dirty" String

If kernel images are being built with "-dirty" on the end of the version string, this simply means that modifications in the source directory have not been committed.

To force a pickup and commit of all such pending changes, enter the following:

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

2). Generating Configuration Files

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

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

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

The resulting `.config` file is located in `${WORKDIR}` under the `linux-${MACHINE}-${KTYPE}-build` directory. You can use the entire `.config` file as the `defconfig` file.

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

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

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

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

2). Run the `menuconfig` command:

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

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

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

The `diffconfig` command creates a file that is a list of Linux kernel `CONFIG_` assignments.

You can use kernel tools to produce warnings for when a requested configuration does not appear in the final `.config` file or when you override a policy configuration in a hardware configuration fragment.

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

    ...

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

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

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

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