**5.11. Configuring the Kernel**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#configuring-the-kernel)

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

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

For more information on kernel configuration, see the "[Changing the Configuration](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#changing-the-configuration)" section in the Yocto Project Linux Kernel Development Manual.

**5.11.1. Using  menuconfig**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#using-menuconfig)

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

To use the menuconfig tool in the Yocto Project development environment, you must launch it using BitBake. Thus, the environment must be set up using the [oe-init-build-env](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#structure-core-script) or [oe-init-build-env-memres](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#structure-memres-core-script) script found in the [Build Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-directory). You must also be sure of the state of your build in the [Source Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#source-directory). The following commands run menuconfig assuming the Source Directory's top-level folder is ~/poky:

$ cd poky

$ source oe-init-build-env

$ bitbake linux-yocto -c kernel\_configme -f

$ bitbake linux-yocto -c menuconfig

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

Consider an example that configures the linux-yocto-3.14 kernel. The OpenEmbedded build system recognizes this kernel aslinux-yocto. Thus, the following commands from the shell in which you previously sourced the environment initialization script cleans the shared state cache and the [WORKDIR](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-WORKDIR) directory and then runs menuconfig:

$ bitbake linux-yocto -c menuconfig

Once menuconfig launches, use the interface to navigate through the selections to find the configuration settings in which you are interested. For example, consider the CONFIG\_SMP configuration setting. You can find it at Processor Type and Features under the configuration selection Symmetric Multi-processing Support. After highlighting the selection, use the arrow keys to select or deselect the setting. When you are finished with all your selections, exit out and save them.

Saving the selections updates the .config configuration file. This is the file that the OpenEmbedded build system uses to configure the kernel during the build. You can find and examine this file in the Build Directory in tmp/work/. The actual .config is located in the area where the specific kernel is built. For example, if you were building a Linux Yocto kernel based on the Linux 3.14 kernel and you were building a QEMU image targeted for x86 architecture, the .config file would be located here:

poky/build/tmp/work/qemux86-poky-linux/linux-yocto-3.14.11+git1+84f...

...656ed30-r1/linux-qemux86-standard-build

**Note**

The previous example directory is artificially split and many of the characters in the actual filename are omitted in order to make it more readable. Also, depending on the kernel you are using, the exact pathname for linux-yocto-3.14... might differ.

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

# CONFIG\_SMP is not set

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

**Note**

Be sure to make a copy of the .config and don't just rename it. The build system needs an existing .configfrom which to work.

**5.11.2. Creating a  defconfig File**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#creating-a-defconfig-file)

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

**Note**

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

To create a defconfig, start with a complete, working Linux kernel .config file. Copy that file to the appropriate ${[PN](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-PN)} directory in your layer's recipes-kernel/linux directory, and rename the copied file to "defconfig". Then, add the following lines to the linux-yocto .bbappend file in your layer:

FILESEXTRAPATHS\_prepend := "${THISDIR}/${PN}:"

SRC\_URI += "file://defconfig"

The [SRC\_URI](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-SRC_URI) tells the build system how to search for the file, while the [FILESEXTRAPATHS](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-FILESEXTRAPATHS) extends the [FILESPATH](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-FILESPATH) variable (search directories) to include the ${PN} directory you created to hold the configuration changes.

**Note**

The build system applies the configurations from the defconfig file before applying any subsequent configuration fragments. The final kernel configuration is a combination of the configurations in the defconfig file and any configuration fragments you provide. You need to realize that if you have any configuration fragments, the build system applies these on top of and after applying the existing defconfig file configurations.

For more information on configuring the kernel, see the "[Changing the Configuration](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#changing-the-configuration)" and "[Generating Configuration Files](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#generating-configuration-files)" sections, both in the Yocto Project Linux Kernel Development Manual.

**5.11.3. Creating Configuration Fragments**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#creating-config-fragments)

Configuration fragments are simply kernel options that appear in a file placed where the OpenEmbedded build system can find and apply them. Syntactically, the configuration statement is identical to what would appear in the .config file, which is in the [Build Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-directory):

tmp/work/*arch*-poky-linux/linux-yocto-*release\_specific\_string*/linux-*arch*-*build\_type*

It is simple to create a configuration fragment. For example, issuing the following from the shell creates a configuration fragment file named my\_smp.cfg that enables multi-processor support within the kernel:

$ echo "CONFIG\_SMP=y" >> my\_smp.cfg

**Note**

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

Where do you put your configuration fragment files? You can place these files in the same area pointed to by [SRC\_URI](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-SRC_URI). The OpenEmbedded build system picks up the configuration and adds it to the kernel's configuration. For example, suppose you had a set of configuration options in a file called myconfig.cfg. If you put that file inside a directory named linux-yocto that resides in the same directory as the kernel's append file and then add a SRC\_URI statement such as the following to the kernel's append file, those configuration options will be picked up and applied when the kernel is built.

SRC\_URI += "file://myconfig.cfg"

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

SRC\_URI += "file://myconfig.cfg \

file://eth.cfg \

file://gfx.cfg"

**5.11.4. Fine-Tuning the Kernel Configuration File**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#fine-tuning-the-kernel-configuration-file)

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

As part of the kernel build process, the do\_kernel\_configcheck task runs. This task validates the kernel configuration by checking the final .config file against the input files. During the check, the task produces warning messages for the following issues:

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

**Note**

The do\_kernel\_configcheck task can also optionally report if an option is overridden during processing.

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

To streamline the configuration, do the following:

1. Start with a full configuration that you know works - it builds and boots successfully. This configuration file will be your baseline.
2. Separately run the do\_kernel\_configme and do\_kernel\_configcheck tasks.
3. Take the resulting list of files from the do\_kernel\_configcheck task warnings and do the following:
   * Drop values that are redefined in the fragment but do not change the final .config file.
   * Analyze and potentially drop values from the .config file that override required configurations.
   * Analyze and potentially remove non-board specific options.
   * Remove repeated and invalid options.
4. After you have worked through the output of the kernel configuration audit, you can re-run the do\_kernel\_configme anddo\_kernel\_configcheck tasks to see the results of your changes. If you have more issues, you can deal with them as described in the previous step.

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

**5.11.5. Determining Hardware and Non-Hardware Features for the Kernel Configuration Audit Phase**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#determining-hardware-and-non-hardware-features-for-the-kernel-configuration-audit-phase)

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

In order to not overwhelm the user with configuration warnings, by default the system only reports on missing "hardware" options because a missing hardware option could mean a boot failure or that important hardware is not available.

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

kernel-cache/features/drm-psb/hardware.cfg

kernel-cache/features/kgdb/hardware.cfg

kernel-cache/ktypes/base/hardware.cfg

kernel-cache/bsp/mti-malta32/hardware.cfg

kernel-cache/bsp/fsl-mpc8315e-rdb/hardware.cfg

kernel-cache/bsp/qemu-ppc32/hardware.cfg

kernel-cache/bsp/qemuarma9/hardware.cfg

kernel-cache/bsp/mti-malta64/hardware.cfg

kernel-cache/bsp/arm-versatile-926ejs/hardware.cfg

kernel-cache/bsp/common-pc/hardware.cfg

kernel-cache/bsp/common-pc-64/hardware.cfg

kernel-cache/features/rfkill/non-hardware.cfg

kernel-cache/ktypes/base/non-hardware.cfg

kernel-cache/features/aufs/non-hardware.kcf

kernel-cache/features/ocf/non-hardware.kcf

kernel-cache/ktypes/base/non-hardware.kcf

kernel-cache/ktypes/base/hardware.kcf

kernel-cache/bsp/qemu-ppc32/hardware.kcf

The following list provides explanations for the various files:

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

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

CONFIG\_SERIAL\_8250

CONFIG\_SERIAL\_8250\_CONSOLE

CONFIG\_SERIAL\_8250\_NR\_UARTS

CONFIG\_SERIAL\_8250\_PCI

CONFIG\_SERIAL\_CORE

CONFIG\_SERIAL\_CORE\_CONSOLE

CONFIG\_VGA\_ARB

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

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

**5.12. Patching the Kernel**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#patching-the-kernel)

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

**Note**

You can use the yocto-kernel script found in the [Source Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#source-directory) under scripts to manage kernel patches and configuration. See the "[Managing kernel Patches and Config Items with yocto-kernel](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#managing-kernel-patches-and-config-items-with-yocto-kernel)" section in the Yocto Project Board Support Packages (BSP) Developer's Guide for more information.

This example creates a simple patch by adding some QEMU emulator console output at boot time through printk statements in the kernel's calibrate.c source code file. Applying the patch and booting the modified image causes the added messages to appear on the emulator's console.

The example assumes a clean build exists for the qemux86 machine in a [Source Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#source-directory) named poky. Furthermore, the [Build Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-directory) is build and is located in poky and the kernel is based on the Linux 3.4 kernel.

Also, for more information on patching the kernel, see the "[Applying Patches](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#applying-patches)" section in the Yocto Project Linux Kernel Development Manual.

**5.12.1. Create a Layer for your Changes**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#create-a-layer-for-your-changes)

The first step is to create a layer so you can isolate your changes. Rather than use the yocto-layer script to create the layer, this example steps through the process by hand. If you want information on the script that creates a general layer, see the "[Creating a General Layer Using the yocto-layer Script](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#creating-a-general-layer-using-the-yocto-layer-script)" section.

These two commands create a directory you can use for your layer:

$ cd ~/poky

$ mkdir meta-mylayer

Creating a directory that follows the Yocto Project layer naming conventions sets up the layer for your changes. The layer is where you place your configuration files, append files, and patch files. To learn more about creating a layer and filling it with the files you need, see the "[Understanding and Creating Layers](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#understanding-and-creating-layers)" section.

**5.12.2. Finding the Kernel Source Code**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#finding-the-kernel-source-code)

Each time you build a kernel image, the kernel source code is fetched and unpacked into the following directory:

${S}/linux

See the "[Finding Temporary Source Code](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#finding-the-temporary-source-code)" section and the [S](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-S) variable for more information about where source is kept during a build.

For this example, we are going to patch the init/calibrate.c file by adding some simple console printk statements that we can see when we boot the image using QEMU.

**5.12.3. Creating the Patch**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#creating-the-patch)

Two methods exist by which you can create the patch: [devtool](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#using-devtool-in-your-workflow) and [Quilt](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#using-a-quilt-workflow). For kernel patches, the Git workflow is more appropriate. This section assumes the Git workflow and shows the steps specific to this example.

1. ***Change the working directory***: Change to where the kernel source code is before making your edits to the calibrate.cfile:
2. $ cd ~/poky/build/tmp/work/qemux86-poky-linux/linux-yocto-${PV}-${PR}/linux

Because you are working in an established Git repository, you must be in this directory in order to commit your changes and create the patch file.

**Note**

The [PV](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-PV) and [PR](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-PR) variables represent the version and revision for the linux-yocto recipe. The PV variable includes the Git meta and machine hashes, which make the directory name longer than you might expect.

1. ***Edit the source file***: Edit the init/calibrate.c file to have the following changes:
2. void calibrate\_delay(void)
3. {
4. unsigned long lpj;
5. static bool printed;
6. int this\_cpu = smp\_processor\_id();
7. printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
8. printk("\* \*\n");
9. printk("\* HELLO YOCTO KERNEL \*\n");
10. printk("\* \*\n");
11. printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
12. if (per\_cpu(cpu\_loops\_per\_jiffy, this\_cpu)) {
13. .
14. .
15. .

1. ***Stage and commit your changes***: These Git commands display the modified file, stage it, and then commit the file:
2. $ git status
3. $ git add init/calibrate.c
4. $ git commit -m "calibrate: Add printk example"

1. ***Generate the patch file***: This Git command creates the a patch file named 0001-calibrate-Add-printk-example.patch in the current directory.
2. $ git format-patch -1

**5.12.4. Set Up Your Layer for the Build**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#set-up-your-layer-for-the-build)

These steps get your layer set up for the build:

1. ***Create additional structure***: Create the additional layer structure:
2. $ cd ~/poky/meta-mylayer
3. $ mkdir conf
4. $ mkdir recipes-kernel
5. $ mkdir recipes-kernel/linux
6. $ mkdir recipes-kernel/linux/linux-yocto

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

1. ***Create the layer configuration file***: Move to the meta-mylayer/conf directory and create the layer.conf file as follows:
2. # We have a conf and classes directory, add to BBPATH
3. BBPATH .= ":${LAYERDIR}"
4. # We have recipes-\* directories, add to BBFILES
5. BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb \
6. ${LAYERDIR}/recipes-\*/\*/\*.bbappend"
7. BBFILE\_COLLECTIONS += "mylayer"
8. BBFILE\_PATTERN\_mylayer = "^${LAYERDIR}/"
9. BBFILE\_PRIORITY\_mylayer = "5"

Notice mylayer as part of the last three statements.

1. ***Create the kernel recipe append file***: Move to the meta-mylayer/recipes-kernel/linux directory and create the linux-yocto\_3.4.bbappend file as follows:
2. FILESEXTRAPATHS\_prepend := "${THISDIR}/${PN}:"
3. SRC\_URI += "file://0001-calibrate-Add-printk-example.patch"

The [FILESEXTRAPATHS](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-FILESEXTRAPATHS) and [SRC\_URI](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-SRC_URI) statements enable the OpenEmbedded build system to find the patch file. For more information on using append files, see the "[Using .bbappend Files in Your Layer](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#using-bbappend-files)" section.

1. ***Put the patch file in your layer***: Move the 0001-calibrate-Add-printk-example.patch file to the meta-mylayer/recipes-kernel/linux/linux-yocto directory.

**5.12.5. Set Up for the Build**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#set-up-for-the-build)

Do the following to make sure the build parameters are set up for the example. Once you set up these build parameters, they do not have to change unless you change the target architecture of the machine you are building:

* ***Build for the correct target architecture:*** Your selected [MACHINE](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-MACHINE) definition within the local.conf file in the [Build Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-directory)specifies the target architecture used when building the Linux kernel. By default, MACHINE is set to qemux86, which specifies a 32-bit Intel® Architecture target machine suitable for the QEMU emulator.
* ***Identify your meta-mylayer layer:*** The [BBLAYERS](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#var-BBLAYERS) variable in the bblayers.conf file found in the poky/build/conf directory needs to have the path to your local meta-mylayer layer. By default, the BBLAYERS variable contains paths to meta, meta-poky, and meta-yocto-bsp in the poky Git repository. Add the path to your meta-mylayer location:
* BBLAYERS ?= " \
* $HOME/poky/meta \
* $HOME/poky/meta-poky \
* $HOME/poky/meta-yocto-bsp \
* $HOME/poky/meta-mylayer \
* "

**5.12.6. Build the Modified QEMU Kernel Image**[**¶**](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-the-modified-qemu-kernel-image)

The following steps build your modified kernel image:

1. ***Be sure your build environment is initialized***: Your environment should be set up since you previously sourced the [oe-init-build-env](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#structure-core-script) script. If it is not, source the script again from poky.
2. $ cd ~/poky
3. $ source oe-init-build-env

1. ***Clean up***: Be sure to clean the shared state out by using BitBake to run from within the Build Directory the [do\_cleansstate](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#ref-tasks-cleansstate)task as follows:
2. $ bitbake -c cleansstate linux-yocto

**Note**

Never remove any files by hand from the tmp/deploy directory inside the [Build Directory](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#build-directory). Always use the various BitBake clean tasks to clear out previous build artifacts. For information on the clean tasks, see the "[do\_clean](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#ref-tasks-clean)", "[do\_cleanall](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#ref-tasks-cleanall)", and "[do\_cleansstate](http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html#ref-tasks-cleansstate)" sections all in the Yocto Project Reference Manual.

1. ***Build the image***: Next, build the kernel image using this command:

$ bitbake -k linux-yocto