Yocto Project and OpenEmbedded Training Labs — BeaglePlay variant

Root Commit

January 27, 2026

Contents

1 Licensing information **6**
- 1.1 License . . . 6
- 1.2 Sources . . . 6

2 Preparing Your Environment **7**
- 2.1 Goals . . . 7
- 2.2 PC performance check . . . 7
- 2.3 GNU/Linux distribution check . . . 7
- 2.4 Matrix channel check . . . 7
- 2.5 Download and extract lab archive . . . 7
- 2.6 Download precompiled binary artifacts . . . 8
- 2.7 Next Steps . . . 8

3 Building a Standard Image for BeaglePlay **9**
- 3.1 Goals . . . 9
- 3.2 Install Development Packages . . . 9
  - 3.2.1 Ubuntu and Debian . . . 9
  - 3.2.2 Other supported distros . . . 9
- 3.3 Improve Build Performance . . . 9
- 3.4 Setting Up Build Environment . . . 10
  - 3.4.1 Cloning Source Repositories . . . 10
  - 3.4.2 Build and Environment Setup . . . 10
  - 3.4.3 Add Layers . . . 10
  - 3.4.4 Set `MACHINE` . . . 11
  - 3.4.5 Building the Image . . . 11

4 First Yocto Boot on BeaglePlay **12**
- 4.1 Goals . . . 12
- 4.2 Board preparation . . . 12
  - 4.2.1 Connect the serial line . . . 12
  - 4.2.2 Connect the serial line . . . 12
  - 4.2.3 Access the serial line . . . 13
  - 4.2.4 Flash the micro SD card . . . 13
- 4.3 Boot the board . . . 14

5 Working with Variables 15
5.1 Goals . . . 15
5.2 Quick preparation . . . 15
5.3 Challenges . . . 15
5.3.1 Standard assignment . . . 15
5.3.2 Overriding a default value . . . 15
5.3.3 Editing variable values . . . 15
5.3.4 Additional operators . . . 16
5.4 Cleaning up . . . 16

6 BeaglePlay Networking 17
6.1 Goals . . . 17
6.2 Introduction . . . 17
6.3 Connection through a local network . . . 17
6.4 Direct connection to your PC . . . 18
6.4.1 Networking setup on the PC side . . . 18
6.4.2 Networking setup on the board side . . . 19

7 Adding Packages to Image 21
7.1 Goals . . . 21
7.2 Introduction . . . 21
7.3 SSH access to the board . . . 21
7.4 Adding the `evtest` program . . . 21
7.5 Build an Image for an Emulated ARM64 Machine . . . 23
7.6 Goals . . . 23
7.7 Setup and build . . . 23
7.8 Testing the image . . . 23

8 Create your first recipe 25
8.1 Goals . . . 25
8.2 Create a new layer . . . 25
8.3 Create a recipe for GNU Hello . . . 25
8.3.1 Creating the recipe . . . 25
8.3.2 Building the recipe . . . 26
8.3.3 Testing the recipe . . . 26
8.3.4 Finalizing the recipe . . . 27

9 Create a recipe for the MyMan ASCII game 28
9.1 Goals . . . 28
9.2 Introduction . . . 28
9.3 Draft recipe from devtool . . . 28
9.4 Fix licensing information . . . 28
9.5 First build and tests on the target . . . 29

9.6 Update the image 29
9.7 Improve the recipe 29
9.8 Closing work 30

10 Tweaking the Device Tree **31**
10.1 Goals 31
10.2 Board setup 31
10.2.1 Through a JST-SH cable 31
10.2.2 Through regular breadboard wires 31
10.3 Test declared hardware 32
10.4 Finding the Bootloader Recipe 32
10.5 Create recipe `bbappend` 32
10.5.1 Empty bbappend 32
10.5.2 Add a u-boot patch 32
10.5.3 Update and test image 33

11 Switching to the Mainline Linux Kernel **34**
11.1 Goals 34
11.2 Create a new `meta-mainline` layer 34
11.3 Create a new kernel recipe 34
11.4 Change the kernel recipe 35
11.5 Create a kernel configuration 35
11.5.1 Add the kernel configuration to your recipe 36
11.6 Test 36

12 Modify MyMan to Support Joystick Control **37**
12.1 Goals 37
12.2 Modify the MyMan recipe 37
12.3 Compiling the program 37
12.4 Test your program and have fun 38
12.5 Finalizing the recipe 38
12.6 Update image 38
12.7 Checking your patch 38
12.8 Looking back 39

13 Smarter Kernel Recipe **40**
13.1 Goal 40
13.2 Support two machines 40
13.2.1 Modify the `genericarm64` setup 40
13.2.2 Enable the `linux-mainline` recipe 40
13.2.3 Create a new Linux kernel configuration for QEMU on genericarm64 40
13.2.4 Add the configuration to the `linux-mainline` recipe 40
13.2.5 Update the image 41

13.3 Support Linux 6.13.11 too . . . 41
13.3.1 New recipe . . . 41
13.3.2 Select a particular version . . . 41
13.3.3 Make configurations version specific too . . . 41
13.3.4 Configuration for BeaglePlay . . . 42
13.3.5 Configuration for genericarm64 . . . 42
13.4 Factorize code between the supported versions . . . 42

14 Create a New Machine 44
14.1 Goals . . . 44
14.2 New machine configuration . . . 44
14.3 Moving settings from `conf/local.conf` . . . 44
14.4 Update overrides . . . 45
14.5 Add dependency to `meta-ti-bsp` layer . . . 45

15 Create a New Image 46
15.1 Goals . . . 46
15.2 Create a new image recipe . . . 46
15.3 Move settings from `conf/local.conf` . . . 46
15.4 Make the root partition bigger . . . 46

16 Create Your Own Distro 47
16.1 Goals . . . 47
16.2 Create a new distro . . . 47
16.3 Changing the init manager . . . 47
16.4 Setting custom welcome text . . . 48
16.5 Remove unnecessary layers . . . 48

17 Setting up a binary distribution 49
17.1 Goals . . . 49
17.2 Build a new package . . . 49
17.3 Change the package manager . . . 49
17.4 Enable package management . . . 49
17.5 Remove a package . . . 49
17.6 Start a package feed server . . . 50
17.7 Network configuration tweaks . . . 50
17.8 Configure your image to use this package feed . . . 50

18 Managing Vulnerabilities 52
18.1 Goals . . . 52
18.2 Enabling vulnerability checks . . . 52
18.3 Ignore irrelevant reports . . . 52
18.4 Use the latest stable kernel . . . 52

18.5 Mark a vulnerability as irrelevant . . . 52

19 Using Kas **53**

19.1 Goals . . . 53

19.2 Setup . . . 53

19.3 Create missing Git repositories . . . 53

19.4 Reconstitute our project image . . . 53

19.5 Test your image . . . 54

20 Final Challenge: Media Player **55**

20.1 Goals . . . 55

20.2 New Project . . . 55

20.2.1 Freeing Disk Space . . . 55

20.2.2 New Workspace . . . 55

20.2.3 Image to Generate . . . 55

20.2.4 Specific Guidelines . . . 56

20.3 Image Execution . . . 56

20.3.1 Install QEMU emulator for ARM64 . . . 56

20.3.2 Run Final Command . . . 56

20.3.3 Tips . . . 56

20.3.4 Completion codes . . . 57

1 Licensing information

Copyright Root Commit, 2024 – 2025

1.1 License

This document is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International* (CC BY-SA 4.0) license. You are free to share and adapt the material for any purpose, even commercially, provided that appropriate credit is given, a link to the license is provided, and any derivative works are distributed under the same license. For more information, see https://creativecommons.org/licenses/by-sa/4.0/.

1.2 Sources

LaTeX sources for this document are available at https://gitlab.com/rootcommit/training-materials.

2 Preparing Your Environment

2.1 Goals

Prepare your computer for the practical labs that follow

2.2 PC performance check

To avoid trouble and wasting a lot of time later make sure that your PC has:

- At least 4 physical CPU cores
- At least 16 GB of RAM
- At least 200 GB of free disk space
- High-speed connection to the Internet

Also double check that you have all the required hardware for the labs: https://rootcommit.com/training/yocto-hardware/. Note that in an in-person setting, the electronic board and its accessories are provided by the instructor, and not all at once.

2.3 GNU/Linux distribution check

Make sure that you're using a distribution that's supported by the Yocto Project: https://docs.yoctoproject.org/ref-manual/system-requirements.html#supported-linux-distributions

This is a step we can run while the system image is still being built (probably).

2.4 Matrix channel check

At this point, you should also be connected to the session's Matrix channel, if you are attending an online course. You should have received instructions when your participation to our course was confirmed, but your instructor can still help you complete this step.

2.5 Download and extract lab archive

You need a directory structure for the labs, which contains a few ready-made files for your work:

```
cd $HOME
wget https://rootcommit.com/data/sessions/2026/yocto-public-jan/yocto-labs.tar.xz
tar xf yocto-labs.tar.xz
```

2.6 Download precompiled binary artifacts

To save time in the next lab, as we have a big file to download, also retrieve the `https://rootcommit.com/data/sessions/sstate-data.tar` archive.

Your instructor will give your the credentials in the session's Matrix channel.

If you are reading this on your own (outside of a Root Commit training session), you can safely skip this step. Your first build will just take more time, that's it.

You are now ready to run the next labs.

2.7 Next Steps

- Get the Yocto reference distribution (Poky) and additional layers
- Build your first image

3 Building a Standard Image for BeaglePlay

3.1 Goals

This lab introduces the Yocto Project by guiding you through the steps to generate a Linux image for the BeaglePlay board using the Poky reference distribution.

3.2 Install Development Packages

3.2.1 Ubuntu and Debian

Install the necessary development packages by running:

```
sudo apt install build-essential chrpath cpio debianutils diffstat \
file gawk gcc git iputils-ping libacl1 liblz4-tool locales python3 \
python3-git python3-jinja2 python3-pexpect python3-pip \
python3-subunit socat texinfo unzip wget xz-utils zstd
```

On Ubuntu 24.04 specifically, you will have enable the use of unprivileged user namespaces that BitBake needs, by creating a new `/etc/sysctl.d/60-apparmor-namespace.conf` file with the following contents:

```
kernel.apparmor_restrict_unprivileged_userns=0
```

Then reboot.

3.2.2 Other supported distros

See https://docs.yoctoproject.org/ref-manual/system-requirements.html#required-packages-for-the-build-host for package requirements for all distributions.

3.3 Improve Build Performance

To accelerate the building of binaries, let's try to use prebuilt artifacts from the Yocto Project. Currently, this represents about 6.5 GB of data.

You should have already retrieved a `sstate-data.tar` archive during the previous lab. Let's assume it was stored in your `Downloads` directory:

```
cd $HOME
tar xf ~/Downloads/sstate-data.tar
```

You also need to add this to your `$HOME/.bashrc` file:

```
export BB_ENV_PASSTHROUGH_ADDITIONS="DL_DIR SSTATE_DIR"
export DL_DIR="${HOME}/data/bitbake.downloads"
export SSTATE_DIR="${HOME}/data/bitbake.sstate"
```

To apply these settings, reload this file:

```
source ~/.bashrc
```

3.4 Setting Up Build Environment

3.4.1 Cloning Source Repositories

The first thing is to clone the sources of the *Poky* reference distribution:

```
mkdir $HOME/yocto-labs/
cd $HOME/yocto-labs/
git clone -b scarthgap git://git.yoctoproject.org/poky.git
```

To avoid regressions during the course, let's check out a commit tested by Root Commit:

```
cd poky
git checkout 72983ac391008ebceb45edc7a8f0f6d5f4fe715c
cd ..
```

Of course, if you are running this on your own and wish to test the latest updates, feel free to stay on the tip of the branches!

You also need to clone extra layers that we need to support the BeaglePlay board:

```
git clone -b scarthgap https://git.yoctoproject.org/git/meta-arm
cd meta-arm
git checkout a81c19915b5b9e71ed394032e9a50fd06919e1cd
cd ..

git clone -b scarthgap https://git.yoctoproject.org/git/meta-ti
cd meta-ti
git checkout da28ae30cabd39ebce054ea2e230548d9b9c3322
cd ..
```

3.4.2 Build and Environment Setup

Let's source the environment setup script which will create the `build` directory and go into it:

```
cd poky
source oe-init-build-env
```

3.4.3 Add Layers

We now need to add the layers we need to the newly created `conf/bblayers.conf` file. This could be done by editing this file, but that's even easier to do with the below commands:

```
bitbake-layers add-layer ../../meta-arm/meta-arm-toolchain
bitbake-layers add-layer ../../meta-arm/meta-arm
bitbake-layers add-layer ../../meta-ti/meta-ti-bsp
```

3.4.4 Set MACHINE

The last thing to do is to define the name of the machine we want to use. Add the below line at the end of the `conf/local.conf` file:

```
MACHINE = "beagleplay-ti"
```

3.4.5 Building the Image

Start the build process by running:

```
bitbake core-image-minimal
```

The build process may take several minutes or up to several hours depending on the performance of your PC, and on whether you are using the prebuilt artifacts or not. And if you didn't set `MACHINE` as instructed, you could also spend hours building an image for the wrong machine!

4 First Yocto Boot on BeaglePlay

4.1 Goals

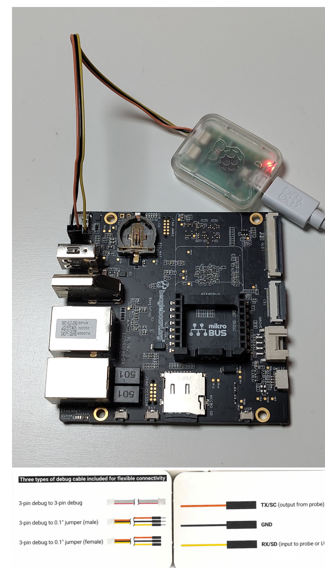
Learn how to prepare the board, connect to a serial line, flash the image generated by Yocto and boot the board.

4.2 Board preparation

This is a step we can run while the system image is still being built (probably).

4.2.1 Connect the serial line

The first thing to do is to connect the board UART port (between the USB-C and USB-A) port. This way, we will have access to the first messages from the bootloader and from the kernel, and will be able to interact with the system even before we set up networking.



You can either use a regular USB-serial dongle, or use the Raspberry Pi Debug Probe if you have one. In any case, you should:

- Connect the `GND` pin of the cable to the `GND` pin of the board.
- Connect the `TX` pin of the cable to the `RX` pin of the board (UART pins are crossed!).
- Connect the `RX` pin of the cable to the `TX` pin of the board.

Later, when we boot the board, if you don't get anything on the line, it's probably because you swapped the signal pins.

4.2.2 Connect the serial line

Now connect your USB-serial cable to your PC, and run the `sudo dmesg` command to check that a new serial device file has appeared:

For the Raspberry Pi Debug Probe, you should have a `/dev/ttyACMx` device file:

```
[311822.061148] cdc_acm 5-1.4.3:1.1: ttyACM0: USB ACM device
```

And for a more traditional USB-serial dongle, you're likely to get a `/dev/ttyUSBx` device file:

```
[311990.458934] usb 5-1.4.4: pl2303 converter now attached to ttyUSB0
```

Now, you want to have permission to access this file as a regular user. It's nice not to need `sudo` to access serial lines. So, check the UNIX group for this file:

```
ls -la /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 Mar 14 06:43 /dev/ttyUSB0
```

Here on Ubuntu, we need to add our user to the `dialout` group:

```
sudo adduser $USER dialout
```

At least in Ubuntu 24.04, you have to reboot for the change to be effective. In theory, logging out and logging back in should be sufficient, so that could work with other distributions.

4.2.3 Access the serial line

You need a terminal emulator program to use the serial line. One of the easiest ones is `tio`:

```
sudo apt install tio
```

Then, just pass `tio` the name of your serial device file:

```
tio /dev/ttyUSB0
```

Hit `[Ctrl][t] + [q]` whenever you want to exit `tio`.

4.2.4 Flash the micro SD card

At this stage, you need BitBake to be done generating the image for your board. Otherwise, just wait a little more and let your instructor know.

First, insert your micro SD card in a card reader:

- If that's an external card reader, the corresponding device file should be `/dev/sda`, `/dev/sdb`...
- If that's an internal card reader, the device file should be `/dev/mmcblk0`.

⚠ **If you have an old PC with a rotating disk, be careful to never flash with the device file corresponding to your hard disk, which is likely to be `/dev/sda`. You'll brick your distribution otherwise.**

Check by yourself by running `sudo dmesg`:

```
[868458.948951] mmc0: new UHS-I speed SDR104 SDHC card at address 0007
[868459.043023] mmcblk0: mmc0:0007 SD16G 14.4 GiB
[868459.044230]  mmcblk0: p1 p2
```

So, let's assume the micro SD card is represented by `/dev/mmcblk0`.

Before flashing, you also need the `bmaptool` utility:

```
sudo apt install bmap-tools
```

Another thing to do is unmount any existing partitions on the SD card:

```
sudo umount /dev/mmcblk0p?
```

or, assuming your SD card is represented by `/dev/sdb`:

```
sudo umount /dev/sdb?
```

This will unmount (safely remove) any existing partitions: `/dev/sdb1`, `/dev/sdb2`, etc, if they were mounted automatically by your GNU/Linux distribution.

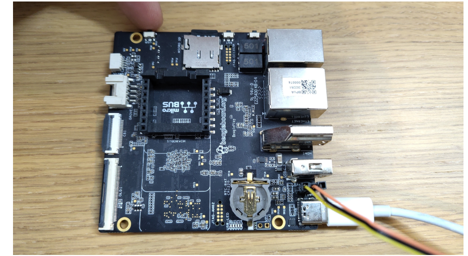
Now, you are ready to flash your image on the micro SD card:

```
sudo bmaptool copy \
     deploy-ti/images/beagleplay-ti/core-image-minimal-beagleplay-ti.rootfs.wic.xz \
     /dev/mmcblk0
```

i This path for the generated image is a bit unusual, most probably because the TI layer is generating two images. The normal path for images is
tmp/deploy/<machine>/<image>-<machine>.rootfs.wic[.bz2|.gz|.xz].

4.3 Boot the board

Insert the flash SD card into the slot on the board.



To boot the board and make it boot off the SD card (instead of the on-board eMMC storage), you have to press and hold the USR key, plug in a USB-C cable for power, and then release the USR key.

You should then see the board boot in the serial console. Log in with the root user and an empty password:

```
...
Poky (Yocto Project Reference Distro) 5.0.15 beagleplay-
ti /dev/ttyS2

beagleplay-ti login: root

WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

root@beagleplay-ti:~#
```

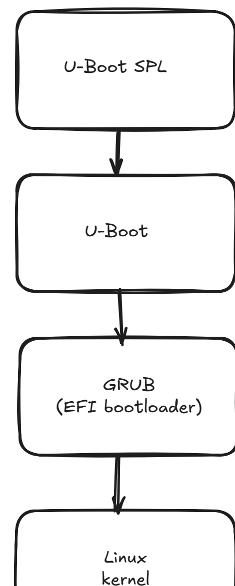
At any time during these labs, to reboot the board while still booting off the SD card, you'll have to:

1. Press and hold the USR button
2. Press and release the RST button
3. Release the USR button

i You can also reboot via the reboot command. In this case, you don't need to use the USR button, and your board will continue to boot off the SD card.

So, reboot the board, and pause the boot process in U-Boot by pressing the space key (then type boot to continue), you can see that booting is done through 3 stages:

Boot flow of Poky's standard image for BeaglePlay (arm64)



This is an unusually complicated boot flow, as U-Boot could directly boot the Linux kernel. However, using a EFI bootloader (as on x86 platforms), makes the image (including the Linux kernel and device tree images) bootloader independent. It could boot in the same way on any EFI compatible bootloader (here GRUB).

5 Working with Variables

5.1 Goals

Quick practice with basic variable syntax and operators, through small challenges to overcome.

5.2 Quick preparation

Copy the `vars-before.conf` and `vars-after.conf` files from `$HOME/yocto-labs/conf/` to the `conf/` directory in the build directory, next to `local.conf`.

At the beginning of `local.conf`, add this line:

```
include vars-before.conf
```

And at the end, add this line:

```
include vars-after.conf
```

During this whole lab, you are **not allowed** to modify these files. You are only allowed to modify `local.conf`.

5.3 Challenges

5.3.1 Standard assignment

Check the value of the `DRINK` variable.

Now, by editing `local.conf` (before the `vars-after.conf` include, of course), make this variable get the `coquito` value.

5.3.2 Overriding a default value

Another easy one: check the value of `SOFTDRINK` and set it to `cherry mint` instead.

Now, check the value of `INGREDIENTS`, and without modifying it, replace `mango` by `coconut` in the string value.

5.3.3 Editing variable values

Add `queso fundido` at the beginning of the value of `LUNCH`

Now, without changing the prior code, add a dot at the end of the `LUNCH` value.

5.3.4 Additional operators

Check the value of the `COCKTAIL` variable. Find a way to remove its second ingredient 😜.

Check the value of the `IMAGE_INSTALL` variable. Look for a way to add `retirement-package` to its value.

At least, this last part illustrates the order of operations.

5.4 Cleaning up

Time to clean up, to avoid interfering with the next labs:

```
$ rm conf/vars*.conf
```

Also remove your changes to `conf/local.conf`, without forgetting the `include` statements at the beginning and at the end.

6 BeaglePlay Networking

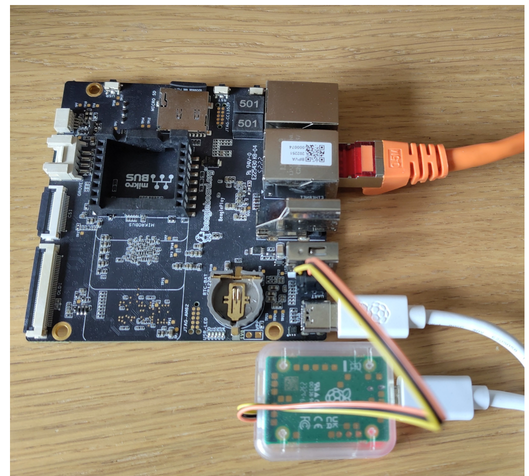
6.1 Goals

- Set up networking for the BeaglePlay board.
- Make your first changes to the generated image

6.2 Introduction

In this series of labs, we are going to consider two ways of setting up networking on the BeaglePlay board and connecting your board to your PC:

- Network connection through a local network. This assumes you have a networking switch or hub to connect the board to, or you have a spare Ethernet plug at your desk. This may not be possible in a classroom environment.
- Direct network connection to your PC. Just connect your board to your PC using a networking cable, possibly through a USB-network adapter connected to your PC. This is pretty easy too in the end, though more explanations are necessary.



In any case, connect a networking cable to your board, using the standard RJ45 connector in the middle [1]

6.3 Connection through a local network

If this option is not available to you, skip to the next section.

Connect the board to your networking equipment or plug. Boot the board and access the command line prompt as you did previously.

You can run the `ip a` command to check that your board now has an IP address for its standard Ethernet interface:

```
root@beagleplay-ti:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

[1] The smaller connector next to the standard one, is also a networking connector, but for Single Pair Ethernet. BeaglePlay was one of the first boards to feature this technology.

```
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether 34:08:e1:84:e9:f4 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.69/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
```

In the above example, you can see that the `eth0` interface has the `192.168.1.69` IP address.

It could happen that the board fails to get an IP address through DHCP at the first attempt. If this happen, you can try to run `udhcpc` (DHCP client) in the root shell.

If this works well enough for you, and you always get the same IP address, you may skip the next section if you are in a hurry. However, the next section should be instructive. You may at least read its instructions.

6.4 Direct connection to your PC

This solution should always be available if you have a spare Ethernet port on your PC (typically if it is connected to the Internet via WLAN), or if you have a USB to Ethernet adapter (part of the recommended accessories for this course).

Since we're going to connect the board directly to the PC, there will be no other networking equipment, no DHCP server assigning dynamic IP addresses, so we will need to use arbitrary static IP addresses:

The IP address and subnet that you choose must not collide with the address range of your local network. Otherwise, you may end up with two different machines having the same IP address, which causes very hard to debug networking issues. Sometimes packets go through, sometimes they don't... and for a good reason!

6.4.1 Networking setup on the PC side

The first thing is to find the name the corresponding networking device. Type the `ip a` command to list all networking devices on your system:

- `enp` devices correspond to internal network cards (`p` like PCI)
- `enx<macaddr>` correspond to USB networking dongles
- `wl` devices correspond to Wireless LAN adapters

Now that your network interface is identified, assign a static IPv4 address to it. Use the below command, changing the IP address and network interface name:

```
nmcli con add type ethernet ifname enx6c1ff71acf05 ip4 172.24.0.1/24
```

6.4.2 Networking setup on the board side

Here, we want to set a static IP address for the board too. The easiest way is to pass an `ip=` setting to the kernel command line.

Rather than manually changing the kernel command line on the SD card, let's have Yocto do this for us, and our changes won't be lost in the new images we will generate in the future.

Let's check the value of this command line first. To do this, on the Linux shell on the board, run this command:

```
# cat /proc/cmdline
BOOT_IMAGE=/Image root=PARTUUID=076c4a2a-02 rootwait rootfstype=ext4
```

So, let's look for this value in the sources. This took a bit of digging, but eventually, looking for `rootfstype=ext4` in the `meta-ti` layer did the trick:

```
$ cd meta-ti
$ git grep "rootfstype=ext4"
meta-ti-bsp/wic/sdimage-2part-efi.wks.in:bootloader --timeout=3
--append="rootfstype=ext4 ${TI_WKS_BOOTLOADER_APPEND}"
```

So, the kernel command line is set in the part that specifies the image contents. We are lucky because the `meta-ti` maintainers added a variable to add extra parameters.

So, all you have to do is add the below line to `conf/local.conf`:

```
TI_WKS_BOOTLOADER_APPEND = "ip=172.24.0.2::172.24.0.1:255.255.255.0::eth0:off"
```

This also declares `172.24.0.1` as the gateway to access the outside network.

Regenerate the image and reflash it, boot the board, and check that the kernel has the expected command line. From the last kernel boot up messages (look for `IP-Config: Complete:`), you can also see that the correct IP address is set.

However, on the Linux command line, double check the IP address of the board:

```
# ip a
```

You should see that it is not set any more. What happens is that the `/etc/network/interfaces` file specifies that `eth0` should be initialized by DHCP. This is what smashes the static IP address we set previously.

Fortunately, Yocto offers multiple ways to modify generated images. With our current level of knowledge, the easiest way is probably to get rid of the `/etc/network/interfaces` and the mechanism that relies on it.

We have a very handy command that allows to find which package a file was installed from:

```
$ oe-pkgdata-util find-path /etc/network/interfaces
init-ifupdown: /etc/network/interfaces
```

Therefore, all we have to do is remove the `init-ifupdown` package from the final image, by adding a `PACKAGE_EXCLUDE` setting to `conf/local.conf`: [2]

```
PACKAGE_EXCLUDE = "init-ifupdown"
```

Update the image, and you should be able to ping the board from the host machine:

[2] If you are already a little more familiar with Yocto, you may wonder why we're not just removing the package from `IMAGE_INSTALL`. To answer your question, just check the value of this variable for `core-image-minimal`, and you will see that its value is defined not by a list of variables, but by a **package group**. Package groups cannot be edited as variables can, so that's why a feature such as `PACKAGE_EXCLUDE` exists.

```
$ ping 172.24.0.2
```

7 Adding Packages to Image

7.1 Goals

Learn how to find and add packages to the initial image.

7.2 Introduction

The main goals of this lab, in addition to learning by practicing, are to:

- Now that networking is in place, access the board through SSH. For this, we will need to install an SSH server.
- Install the `evtest` program, which we will need in the next labs.

7.3 SSH access to the board

Following what you learned in the lectures, install look for the recipe implementing an SSH server.

Once you found the right recipe, check all the packages that are generated by the recipe. In this case, the package we want to add bears the same name as the recipe. That's very often the case!

Now, after updating the image and rebooting the board, you should be able to access it through SSH, and not only through the serial console:

```
ssh root@172.24.0.2
```

We will still have to regenerate the image in the next steps and labs. This will cause the private key of the SSH server to be regenerated and cause this type of error:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

As explained in the SSH client output, you will have to reset your copy of the server's public key:

```
ssh-keygen -f $HOME/.ssh/known_hosts -R 172.24.0.2
```

7.4 Adding the `evtest` program

In a later lab, we are going to need the `evtest` program. This programs allows to get information about any input device and test it too.

In our case, we will test a Linux driver for the Adafruit mini I2C gamepad the course requires.

You have been given all the information needed to do this. Don't hesitate to share your questions with your instructor if needed, though.

Once you updated the image and rebooted the board, please run the `evtest` command:

```
# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      gpio-keys
Select the device event number [0-0]:
```

As you can see, there is only one input device on the system. Choose `0` and then you can test the buttons on your system.

Actually, as the `evtest` output shows, only one button is supported:

```
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "gpio-keys"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 256 (BTN_0)
```

Press and release the `USR` button several time to see its state change in the `evtest` output.

There's another button that ”works”, but in a very different way 😉.

7.5 Build an Image for an Emulated ARM64 Machine

7.6 Goals

In this lab, we are going to setup and create a Yocto image build from another target machine. This will be useful later when we want to implement a kernel recipe supporting multiple machines at once.

This will be for a virtual machine emulated by QEMU, so you won't need to procure a second board for this part.

As the build will take a little time to complete, it's good to start it now.

It's also a good opportunity to repeat what we did at the begining.

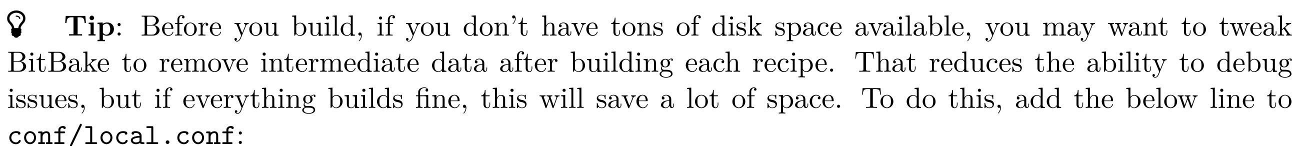
7.7 Setup and build

Create a new directory in your main lab directory:

```
$ cd $HOME/yocto-labs
$ mkdir genericarm64
$ cd genericarm64
```

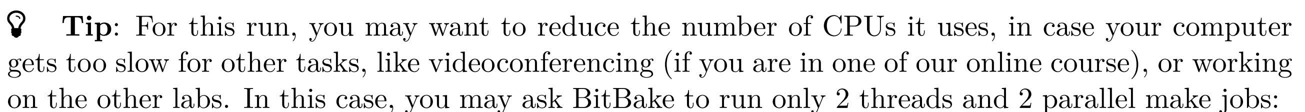
In this new directory:

- Clone the `styhead` branch of Poky
- Source the environment setup script
- Define `MACHINE` as `"genericarm64"`
- Add the `u-boot` package to your images. Otherwise, a generic arm64 filesystem will be built, but without anything for a specific machine. With the U-Boot bootloader, we will be able to boot our image on a virtual arm64 machine emulated by QEMU.



Tip: Before you build, if you don't have tons of disk space available, you may want to tweak BitBake to remove intermediate data after building each recipe. That reduces the ability to debug issues, but if everything builds fine, this will save a lot of space. To do this, add the below line to `conf/local.conf`:

```
INHERIT += "rm_work"
```



Tip: For this run, you may want to reduce the number of CPUs it uses, in case your computer gets too slow for other tasks, like videoconferencing (if you are in one of our online course), or working on the other labs. In this case, you may ask BitBake to run only 2 threads and 2 parallel make jobs:

```
BB_NUMBER_THREADS = "2"
PARALLEL_MAKE = "-j 2"
```

Build the `core-image-minimal` image.

This is likely to take a pretty long time, as we're using a different machine, a different version of Poky, and we don't have ready made binary artifacts either.

Anyway, this part is meant to run in the background while you are busy with other tasks.

7.8 Testing the image

When the image is done building, let's boot it on QEMU. Poky ships with a convenient `runqemu` script that starts the QEMU emulator for the right target architecture and on the image that you built:

```
$ runqemu nographic slirp
```

Some details:

- `nographic` starts QEMU directly in your terminal. That's convenient when you just need to access a text console.
- `slirp` selects networking in user-mode. You won't need root privileges to bring up the network in the virtual machine, and therefore, you won't be prompted for your password when you run this command.
- To exit QEMU, type `[Ctrl][a]` followed by `[x]`.

8 Create your first recipe

8.1 Goals

We are going to create our first recipe. For this purpose, to follow a good old tradition, we are going to use GNU Hello from https://www.gnu.org/software/hello/.

8.2 Create a new layer

In Yocto, every recipe needs to be in a layer.

As it's against the Yocto Law to modify a layer maintained by somebody else, let's create our own.

Within your Yocto build environment:

```
$ cd $HOME/yocto-labs
$ bitbake-layers create-layer meta-homebrew
```

Have a look at the initial contents of this layer, in particular `conf/layer.conf` and also at the sample recipe in `recipes-example/example/example_0.1.bb`.

Then, add this layer to `conf/bblayers.conf` in your build environment. Don't forget the command that does this for you 😉.

8.3 Create a recipe for GNU Hello

8.3.1 Creating the recipe

Though we showed you what the sources of a recipe look like in the lectures, it's unusual to create such sources from scratch.

Yocto ships with a very useful tool to create draft recipes from sources: `devtool`. We will elaborate more about it in the next lectures, but we can already get introduced to it through a few examples.

Of course, we will finalize all our recipes by editing the recipe files.

So, let's go ahead and create this draft recipe for the latest version of GNU Hello, as found on the project's downloads page.

```
$ cd $BUILDDIR
$ devtool add https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz
```

As explained by the `devtool` output, the recipe file should be created in `workspace/recipes/hello/hello_2.12.1.bb`.

Have a look at `workspace/conf/layer.conf`. You can see that `workspace/` is a temporary layer created by `devtool`. Even `devtool` respects the Yocto Law, by storing its recipes in a layer.

Open the workspace/recipes/hello/hello_2.12.1.bb file. Very simple, isn't it? Where's the magic?

What keeps this recipe simple is this line:

```
inherit gettext autotools
```

The recipe inherits two **classes** that define the standard tasks that this recipe should have:

- autotools: the sources are packaged by the GNU Autotools. This is a set of tools to check for requirement, pass configuration options (configure script), generate a Makefile to build and install the software component from sources.
- gettext: the sources are using GNU gettext for translating interface strings. Because of this, the recipe will generate multiple packages corresponding to each supported language.

We will cover classes in further details in due time.

Now, what needs fixing in the Hello recipe is the license. devtool didn´t manage to figure it out, so let's do it by looking at the sources.

First, find the location of the sources:

```
$ bitbake-getvar -r hello S
```

Then, open the COPYING file in the sources to get the license. By the way devtool had a look at the right file but couldn't infer the license from it. That's very easy though.

So, fix the license in the recipe. When you are done, remove the unnecessary comments left by devtool.

8.3.2 Building the recipe

Testing is easy. You can ask BitBake to just build this recipe, and without having to update an image:

```
$ bitbake hello
```

You could also have run:

```
$ devtool build hello
```

8.3.3 Testing the recipe

Here, we are going to use an extremely useful features: devtool deploy-target. It allows to test a newly built recipe on the target without having to rebuild the image, drink some coffee, update the image, drink more coffee, and ultimately reboot the board, over and over again, so that at the end of the day, you are in caffeine overdose — forced to do more Yocto work because you can't sleep at all.

This mechanism works over SSH (you now understand why we set it up in the previous lab), so we have to create a new "host" definition for SSH.

If necessary, create the $HOME/.ssh/config file and add the below code block to it:

```
Host beagleplay
                User root
                Hostname 172.24.0.2
                Port 22
                StrictHostKeyChecking no
                UserKnownHostsFile /dev/null
```

Thanks to these settings, SSH has everything it needs to know to connect to your board. You can now connect with just:

```
$ ssh beagleplay
```

Now, you can deploy the recipe's build output (the output of the `do_install` task) to the board:

```
$ devtool deploy-target hello beagleplay
```

You should now be able to run the `hello` command one your board.

8.3.4 Finalizing the recipe

Now that the recipe seems to work fine, `devtool` can help us one more time by deploying the recipe to a permanent layer:

```
$ devtool finish -f hello ../../meta-homebrew
```

You can run this command to see where the new recipe was stored:

```
$ tree ../../meta-homebrew
```

You can also check that `hello` is no longer in `workspace/recipes/`.

9 Create a recipe for the MyMan ASCII game

9.1 Goals

We are going to create a more complex recipe, for the MyMan game, which is ASCII game unlicensed clone of the old Pac-Man game.

9.2 Introduction

The original sources of the game are available on GitHub: `https://github.com/kragen/myman`.

Unfortunately, these sources haven't been modified since 2009, and while they still compile on a modern GNU/Linux distribution, they turned out to be very complicated to cross-compile with BitBake. In addition, the `configure` script and `Makefile` were created by hand instead of being generated by the GNU Autotools.

Fortunately, an autotooled clone has been created by Michael Opdenacker, and is available on GitHub too: `https://github.com/michaelopdenacker/myman/`.

Thanks to the `autotools` class [1], BitBake will be able to build this game in a simple way.

9.3 Draft recipe from devtool

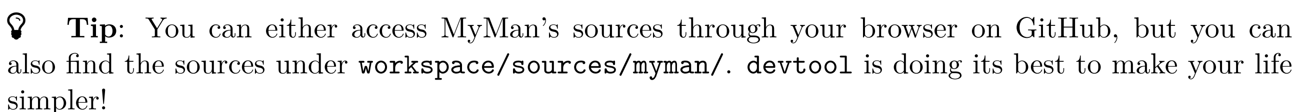
Using a command similar to the one used in the previous lab, use `devtool` to create a draft recipe from the source repository.

Even though we're not giving `devtool` a source archive, you will see that the tool will work equally well with a source repository!

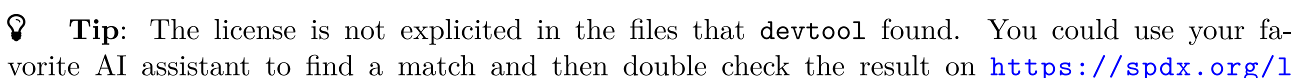
Looked at the generated recipes sources, and try to understand what was figured out by `devtool`.

9.4 Fix licensing information

Read the guidelines written by `devtool` in the draft recipe code, open the license files in the sources, and look for the actual license category this software is released with.



Tip: You can either access MyMan's sources through your browser on GitHub, but you can also find the sources under `workspace/sources/myman/`. `devtool` is doing its best to make your life simpler!



Tip: The license is not explicited in the files that `devtool` found. You could use your favorite AI assistant to find a match and then double check the result on `https://spdx.org/l`

[1] You can look for the class file in Poky, as a quick exercise

icenses/. Here, another way to confirm this is to check this documentation file in the sources: `website/htdocs/myman.xml`.

Once you have the right SPDX identifier, fix the `LICENSE` setting in the new recipe, and remove the no longer necessary comments in the recipe file.

9.5 First build and tests on the target

Build the new recipe with `devtool` and then deploy it on the target as we did in the previous lab.

If you try to run the `myman` binary, you should get:

```
myman: error while loading shared libraries: libncurses.so.5: cannot open shared
object file: No such file or directory
```

That shouldn't be a surprise, as `devtool` doesn't take care of copying the dependencies (unlike what happens when you generate an image).

We could regenerate the image with the `ncurses` packages installed, but we would have to deploy `myman` to the image again, which isn't very convenient at a stage we are just experimented with our new recipe.

Fortunately, we can work around this issue by deploying `ncurses` packages on the board too.

First, let's try to build this dependency with `devtool`:

```
$ devtool build ncurses
```

This should complain that only recipes under `workspace/` can be built. Not a problem for us, let's add the `ncurses` recipe to our workspace, as if we were going to modify it:

```
$ devtool modify ncurses
```

Now we can build and deploy `ncurses`

```
$ devtool build ncurses
$ devtool deploy-target ncurses beagleplay
```

Now, you can SSH to your board and run `myman` on it. We're suggesting to use SSH instead of the serial console, if you want a proper terminal and play with color.

9.6 Update the image

Now, let's add `myman` to our image and regenerate it, to make sure everything is all set in the recipe.

Update the SD card and test `myman` on the new image.

9.7 Improve the recipe

Look at all the files in the `myman` package:

```
$ rpm -qlp tmp/deploy/rpm/aarch64/myman-0.1+git-r0.aarch64.rpm
```

You will find lots of files under `/usr/share/` corresponding to data files for the various levels, sprites, etc.

As these files are the same on all architectures, it would be nice to store them in an architecture independent package: `myman-data`.

So, let's see what got added to the main package:

```
$ bitbake-getvar -r myman FILES:myman
```

Looking at the output, you can see that `/usr/share/myman` was included, and looking at the pre-expansion values, it must come from `${datadir}/${BPN}`.

So, using the `flac` recipe or any other one as an example, modify the recipe to create an additional `myman-data` package containing such data.

Also make sure it gets installed too when the main `myman` package is installed in an image. What's the cleanest way to achieve this? Don't hesitate to ask your instructor if you have a doubt.

When you're done, update and test the image again.

9.8 Closing work

Now that everything works, it's time to store the completed recipe in our `meta-homebrew` layer, under a `recipes-games` subdirectory.

So, first create this subdirectory and then use the same `devtool` command as in the previous lab to store the `myman` recipe in there and remove it from the `devtool` workspace.

10 Tweaking the Device Tree

10.1 Goals

In this lab, we are going to connect the Adafruit Mini I2C ”Seesaw” Gamepad device that this course requires, and for Linux to be able to use this device, it will have to be added to the Device Tree, which is the description of the hardware.

This will also be an opportunity to experiment with modifying recipes from third party layers.

10.2 Board setup

Let’s connect the Adafruit Mini I2C Gamepad to the BeaglePlay board.

10.2.1 Through a JST-SH cable

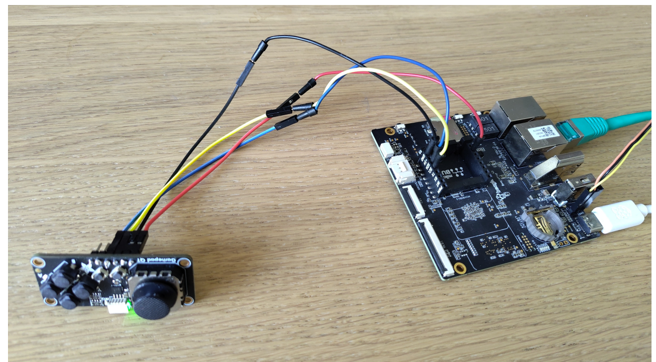
This is the default option, and also the safest one, as you can’t make mistakes connecting the cables. The cables won’t disconnect either.

So, just connect the gamepad to the board with the JST-SH female 4-pin cable.

The green LED of the cable should turn on.

10.2.2 Through regular breadboard wires

If you don’t have the JST-SH cable, another possibility is to connect the gamepad device to the I2C pins on the Mikrobus connector.



To do so, you will have to solder male or female headers to the gamepad PCB. Then, using suitable breadboard wire cables, connect:

- Gamepad `VIN` to Mikrobus `+3.3V`
- Gamepad `GND` to Mikrobus `GND`
- Gamepad `SCL` to Mikrobus `SCL`
- Gamepad `SCA` to Mikrobus `SCA`

Make sure the green LED of the cable is on when the board is on.

10.3 Test declared hardware

To compare with want we want to achieve with lab, boot the board, and `cd` into the `/sys/firmware/devicetree/` directory.

Then check that no `gamepad` device is found yet:

```
find . -name gamepad
```

10.4 Finding the Bootloader Recipe

Even though the board device tree sources originate from the kernel, in our particular case, the device tree for our board actually comes from the U-Boot bootloader sources, which originate from the kernel source tree as far as the device tree source files (`.dts`) are concerned.

Therefore, we need to tweak the BeaglePlay device tree that the U-Boot recipe used. We are going to use the `bbappend` mechanism to do that.

So, first, we have to find the recipe that builds U-Boot.

The first way is to run the `bitbake -s` command, that shows all the recipes which can be built under the current configuration:

```
bitbake -s | grep "u-boot"
```

In the output of this command, you will see that `u-boot-ti-staging 2025.01+git-r0` is what we are looking for.

Another good way of doing this search is to use the `bitbake-layers show-recipes` command. Not only does it show the eligible recipes, but also the other ones that are skipped under the current configuration:

```
bitbake-layers show-recipes "u-boot*"
```

Here again, you will find several u-boot recipes, but only `u-boot-ti-staging` isn't skipped.

So, that's the recipe that we will modify.

i A difference with `bitbake -s` is that all available recipe versions are available, and not just the one that will be built.

10.5 Create recipe `bbappend`

10.5.1 Empty bbappend

In your `homebrew` layer, create a `bbappend` recipe for the current version of `u-boot-ti-staging`. Add the code to add specific files to this `bbappend`.

Now check that your `bbappend` indeed applies to the `linux-ti-staging` recipes:

```
$ bitbake-layers show-appends u-boot-ti-staging
```

10.5.2 Add a u-boot patch

We now need to modify the description of the board, the Device Tree in the U-Boot sources, to add the I2C device that we have just connected.

To create such a patch, you also need to have access to U-Boot sources, and preferably in its git repository. We will do this kind of operation with `devtool` in a later lab, but for the moment, we will

take a ready-made patch from `/yocto-labs/patches/`.

Two cases:

- If your gamepad is connected through the JST-SH connector, use the `0001-k3-am625-beagleplay.dts-connect-Adafruit-mini-I2C-ga.patch` file
- If your gamepad is connected through regular wires to the Mikrobus connector, use `0001-k3-am625-beagleplay.dts-connect-gamepad-on-Mikrobus-.patch`

Copy this patch file to your recipe, and add it to `SRC_URI`.

Now run:

- A command to check the new value of `SRC_URI`
- A command to build the U-Boot recipe, to make sure the recipe and its inputs are fine.

10.5.3 Update and test image

Update the image and reflash the SD card with it.

Look for the `gamepad` file again in `/sys/firmware/devicetree`, to check that the Device Tree changes have indeed been applied.

The next step is to configure the kernel to include a driver for our gamepad, but this will happen in the next lab.

11 Switching to the Mainline Linux Kernel

11.1 Goals

The end goal of this lab is to configure the Linux kernel so that it includes a driver for our gamepad device.

We could do it with the `linux-ti-staging` kernel recipe, but the developers of this recipe have made it so smart and so customized that it's hard to find how to apply the standard methods for configuring the Linux kernel. Of course, in a course that should be as hardware independent as possible, we should use the standard methods as much as possible.

Therefore, we will create a very simple `linux-mainline` recipe that compiles a mainline Linux kernel, using the standard `kernel.bbclass` class.

You will see it's very easy to customize!

11.2 Create a new `meta-mainline` layer

Here we are going to create a new layer, because the one we've already created (`meta-homebrew`) is quite specific to our project, as it has `bbappend`s that only apply to recipes on top of `meta-ti`. Therefore, `meta-homebrew` won't be usable on projects with other machines.

So, create a new `meta-mainline` layer and add it to our current project.

You can run `bitbake-layers show-layers` to double check that this was done successfully.

11.3 Create a new kernel recipe

Start by creating a `recipes-kernel` subdirectory in your `meta-mainline` layer (just as in the `meta-ti-bsp` layer).

Then create a `linux-mainline` recipe directory containing a `linux-mainline_6.14.2.bb` recipe file with the below contents:

```
DESCRIPTION = "Mainline Linux kernel"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://COPYING;md5=6bc538ed5bd9a7fc9398086aedcd7e46"

inherit kernel

SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.14.y;protocol=https"

SRCREV = "9bc5c94e278f780af15b3f6e13ae08310aeae880"
```

```
S = "${WORKDIR}/git"
```

A few notes:

- Sources are fetched from the `linux-stable` git tree, which is where the stable release sources are published.
- Since we're fetching from a git repository, we have to specify a specific commit ID to fetch, through the `SRCREV` variable. You cannot rely on a specific tag, as tags can be modified in git, and therefore are not as unforgeable as a SHA hash.

  To get a commit ID from a given release tag, you can clone the upstream repository and query the commit ID as follows:

  ```
  $ git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
  $ cd linux
  $ git show v6.14.2 | grep commit
  ```

- For sources extracted by the `git` fetcher, you have to set the `S` variable to `"${WORKDIR}/git"`.

11.4 Change the kernel recipe

Using the virtual provider mechanism, modify the new recipe and your `conf/local.conf` file to make BitBake use this new recipe instead of `linux-ti-staging`.

To check that you achieved your goal, use the `bitbake-layers show-recipes` command.

11.5 Create a kernel configuration

Wow, those who have already configured and compiled a kernel manually may find this part difficult. To produce the kernel configuration file, you need to have the same sources as what Yocto is going to build (not difficult for a mainline Linux kernel), and more importantly, the same toolchain, as some kernel configuration settings depend on the capabilities of the compiler.

Fortunately, the Yocto Project is here to help. There is a special `menuconfig` task that you can run, to configure the Linux kernel, U-Boot, Barebox and BusyBox. This task is implemented by the `cml1` class, which is inherited by the recipes to build the above projects, in our case through the `kernel.bbclass` class.

First, let's double check that such a task is available:

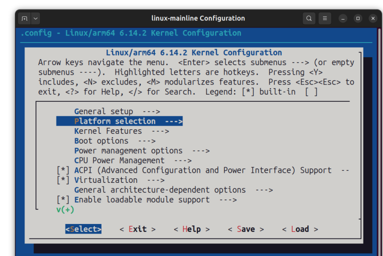
```
$ bitbake -c listtasks linux-mainline
```

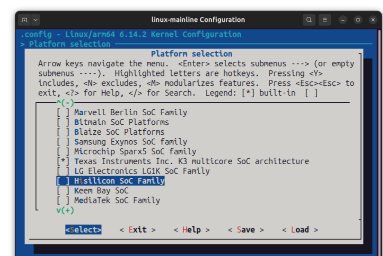
Indeed, you should find a `menuconfig` task in the output. So, let's run it:

```
bitbake -c menuconfig linux-mainline
```

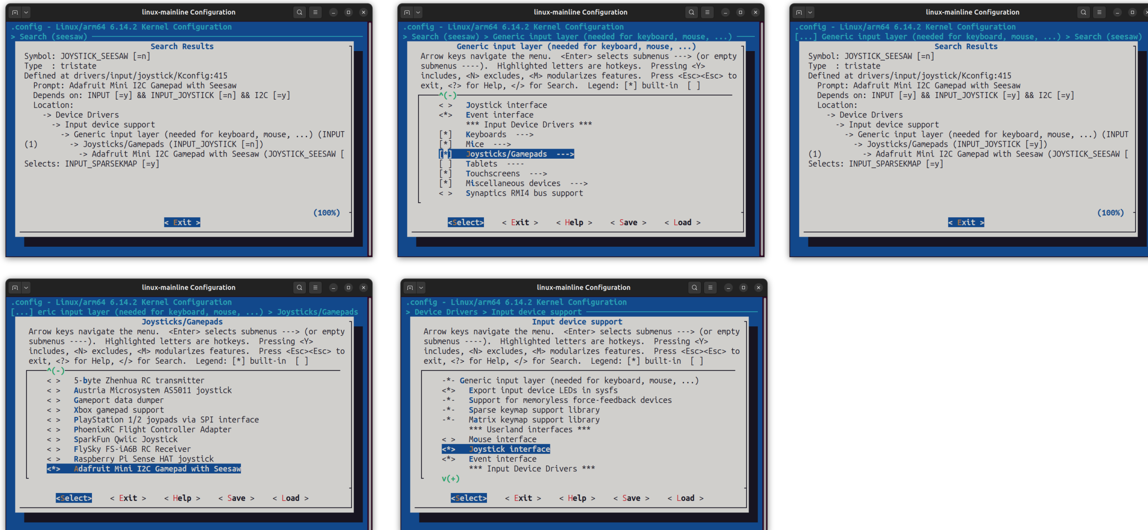
In the new terminal running the `menuconfig` interface, the first thing we want to do is remove support for other SoCs, to make the kernel lighter, and therefore faster to compile and boot.

So, using the `/` key, look for `SEESAW` and configure the driver to be built as a kernel built-in. In the search results, press the number corresponding to the relevant match, in our case `1`, to jump to where you can set a value to the parameter. You'll have to enable joystick support first, and then to add the Seesaw gamepad driver.





Please also add support for the Joystick interface (CONFIG_INPUT_JOYDEV.



When you are done, save and exit `menuconfig`.

Note that the task will show you where the `.config` file was saved.

We are not completely there yet, as the Linux kernel recipe will want to have a `defconfig` file. That's like a `.config` file, but without the parameters which have a value that is the default one.

Fortunately, we can also easily generate such a file without having to do this by hand in the Linux kernel sources (this would have been the `make savedefconfig` command), as there is also a `savedefconfig` task:

```
bitbake -c savedefconfig linux-mainline
```

Here again, you can see how the `defconfig` file was created.

11.5.1 Add the kernel configuration to your recipe

Copy the new `defconfig` file to a subfolder associated to your recipe.

Modify your recipe too to add the `defconfig` file to SRC_URI. When such a file is provided, it's automatically used by the `kernel` class as kernel configuration.

Now, verify that this worked, by checking the variable value for your recipe:

```
bitbake-getvar -r linux-mainline SRC_URI
```

Also check that the `linux-mainline` recipe build fine.

11.6 Test

Update your image and boot the board again.

Use the `uname -r` command to check the kernel version.

Run the `evtest` command. You should now see an `Adafruit Seesaw Gamepad` option. Once selected, you can press the keys of the gamepad and move the joystick, and you should see the corresponding input events in the `evtest` output.

Our gamepad is ready to be used!

12 Modify MyMan to Support Joystick Control

12.1 Goals

We are going to learn how to `devtool` can be used to develop applications.

Our practical goal is to use our Gamepad as an input device to play the MyMan game instead of using the keyboard arrow keys.

As we will modify the MyMan sources in several steps, it will be a little like developing an application and compiling it using BitBake, directly editing the sources but without having to make commits every time you want to test changes, which you would have to do if you used in a formal recipe.

12.2 Modify the MyMan recipe

Run the `devtool` command to get a copy of the MyMan recipe into the `devtool` workspace.

Then, go into the source directory for MyMan. Using the `git status` command, you should see that you code is in a git tree, in a `devtool` branch.

It's now time to modify the code to support driving the game through our joystick.

To do so, open the `src/myman.c` file and add a `getjs()` function, using the code available in `src/getjs.c` in your `$HOME/yocto-data` directory.

Then, in the same file, add a call to this function near the beginning of the `gameinput()` function, right after the call to `my_getch()`:

```
if (k == ERR) k = getjs();
```

This reads data from the joystick when no character was read from the terminal.

You may wonder why we are not giving you a patch. It's because we want you to see how `devtool` can help you test experimental code as you are developing it. Then, when your code works, `devtool` will create the final patch for you when you run `devtool finish`.

12.3 Compiling the program

Compile the program by building the recipe. Though that's not exactly as fast as directly invoking a compiler, this is orders of magnitude faster than having to commit your changes or creating a release archive so that BitBake can compile your code through a regular recipe.

Looking at the errors that you get, it turns out that you just need to add `"linux/joystick.h"` and `<fcntl.h>` to the source includes in the same file.

Once this is done, your program should compile well.

12.4 Test your program and have fun

Deploy your program on the board, and start it.

After pressing a few keys on your keyboard to start the game, you should now be able to steer your character using only the joystick pad.

Well done! We had to go through a number of interesting challenges to get there. Now, if you survive the hungry ghosts, you should be able to create and modify a recipe for many types of applications.

12.5 Finalizing the recipe

Now that the application works as expected, it's time to share the code with others.

You won't be able to run `devtool finish` right away.

You first need to commit your changes in the temporary source repository in `devtool`'s workspace:

```
git commit -s src/myman.c
```

Add a relevant title line and then a quick description of the change in the following lines.

i : Git will ask you to set your name and e-mail address if you haven't done that yet. That's needed to sign your commits, corresponding to the `-s` option. All patches in Yocto and OpenEmbedded have to be signed to trace their origin. This means that your information is part of the commit log. In this case, just follow the instructions. In our lab, it's fine to give a dummy name and/or dummy address. Those won't be shared unless your share your patch with other people.

Now, you can run the `devtool finish` command to update the recipe in the `meta-homebrew` layer.

Look at the updated contents of the layer. You should see a new patch in the MyMan recipe and added to `SRC_URI`.

12.6 Update image

So far, the new version of the program has just been deployed to the board in a temporary way. Now that the recipe has been updated, you can update the image and reflash it.

12.7 Checking your patch

For the case you have a real patch to share with the OpenEmbedded and Yocto community, OpenEmbedded comes with a handy `patchtest` tool to check the compliance of your patch against various rules.

First, install its prerequisites by following the guidelines given by the Yocto Contributor Guide (https://docs.yoctoproject.org/contributor-guide/submit-changes.html#validating-patches-with-patchtest).

Then, you should be able to run `patchtest`:

```
patchtest --patch ~/yocto-labs/meta-homebrew/recipes-games/myman/myman/0001-<...>.patch
```

Do fix issues, you will probably have to run `devtool modify` again and modify your commit (`git commit --amend` command), unless you can directly edit the patch file which will be indeed much simpler to do if it's possible.

12.8 Looking back

Remember the Device Tree patch for the `u-boot-ti-staging` recipe? It was precisely created in the same way by using `devtool modify`, making the change, committing it in Git and running `devtool finish`.

13 Smarter Kernel Recipe

13.1 Goal

Our goal is to make the mainline kernel recipe more generic so that it can support multiple machines at the same time, as well as multiple kernel versions.

In particular, this will be a great opportunity to take advantage of the overrides capability of BitBake.

13.2 Support two machines

We want to allow our `linux-mainline` recipe to support the `genericarm64` machine too, as we ran it through QEMU.

So, go back to the BitBake environment you used to build the `genericarm64`.

13.2.1 Modify the `genericarm64` setup

First, to reuse your `meta-mainline` layer, use a command to add it to your list of layers, or add it manually by editing `conf/bblayers.conf`.

To do so, you'll have to modify the `layer.conf` file to declare your layer as compatible with the Poky release we're using, currently `styhead` in this environment.

13.2.2 Enable the `linux-mainline` recipe

Find out what's the enabled recipe for building the Linux kernel.

Now make `linux-mainline` replace it, as you did in an earlier lab for the project on BeaglePlay.

13.2.3 Create a new Linux kernel configuration for QEMU on genericarm64

Launch the kernel configuration interface, and, in it:

- In `Platform selection`, remove all supported SoCs
  (QEMU runs a "virt" machine, which doesn't rely on any real SoC)
- Look for `DRM`, and disable it.
  This removes a lot of unnecessary graphics and GPU code.

After exiting `menuconfig`, run the task that creates a `defconfig` file.

13.2.4 Add the configuration to the `linux-mainline` recipe

It's now time to store the `defconfig` file in the recipe, but we don't want to lose the `defconfig` file for BeaglePlay either.

Reorganize the directory structure of the recipe so that it stores each `defconfig` file for each machine. Have a look at the slides again if necessary.

13.2.5 Update the image

Update `core-image-minimal` and test that it boots fine. Check the version of the Linux kernel inside:

```
# uname -r
```

13.3 Support Linux 6.13.11 too

Get back to the working directory for BeaglePlay.

Our goal now is to make the recipe more generic so that it supports both 6.13.11 and 6.14.2 versions.

13.3.1 New recipe

To begin with, copy the `linux-mainline_6.14.2.bb` file to `linux-mainline_6.13.11.bb`. Update `SRC_URI` and modify `SRCREV` to `"f2333ac1f914b6e22e04354f5cab8ad4631b0202"`. That's the commit id for release 6.13.11.

To check that your new recipe version is properly registered, you can run:

```
$ bitbake-layers show-recipes linux-mainline
```

13.3.2 Select a particular version

An issue with BitBake is that it will automatically pick up the most recent version of the recipe. To bypass this, you will have to use the `PREFERRED_VERSION` setting. So, add the below line to `conf/local.conf`:

```
PREFERRED_VERSION_linux-mainline = "6.13.11"
```

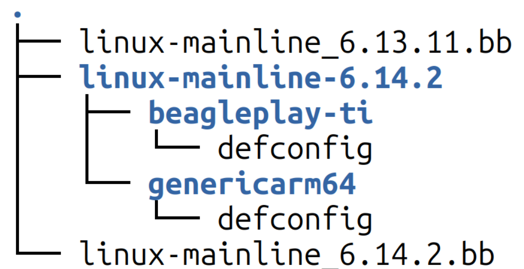
ℹ The `%` character allows to match any update for the 6.13 recipe that we could make in the future.

At this stage, we could build our new recipe version, but there's an issue: the `defconfig` files are for the 6.14.2 version, not for 6.13.11. The parameters between the two versions are slightly different.

13.3.3 Make configurations version specific too

Do you remember that the file overrides are also looked for in version specific directories? It's time to use this feature, so, in a separate terminal, go to the `linux-mainline` recipe directory and run:

```
mv files linux-mainline-6.14.2
```



⚠ **Caution: unlike in recipe names, the separator character in ${BP} is -, not _. This is a mistake your instructor makes frequently!**

This way, our `defconfig` files are now version and machine specific.

Now, try to build the recipe. Unfortunately, that won't work because BitBake will no longer be able to find the `defconfig` file. That's because it's added unconditionally, so each supported version should have one.

Let's fix that by making the addition of `defconfig` conditional to each machine providing one. So, as we have `defconfig` files only for Linux 6.14.2 so far:

- In `linux-mainline_6.14.2.bb`, remove the `defconfig` line from `SRC_URI` and add it back as follows:

  ```
  SRC_URI:append:beagleplay-ti = " file://defconfig"
  SRC_URI:append:genericarm64 = " file://defconfig"
  ```

- In `linux-mainline_6.13.11.bb`, remove the `defconfig` line for now, as we don't have such `defconfig` files yet.

You can no check that at least the `configure` task of the recipe executes well:

```
bitbake -c configure linux-mainline
```

13.3.4 Configuration for BeaglePlay

Using the same techniques as before, you can now prepare a Linux 6.13.11 configuration for the BeaglePlay:

- `Platform selection`: Keep only TI K3 SoC support
- Enable `CONFIG_JOYSTICK_SEESAW` and `CONFIG_INPUT_JOYDEV` as built-ins

Generate a new `defconfig` file from this configuration, and modify the recipe to store this new file.

Tip: Check the beginning of the `defconfig` to double check that it was produced for the expected Linux kernel version.

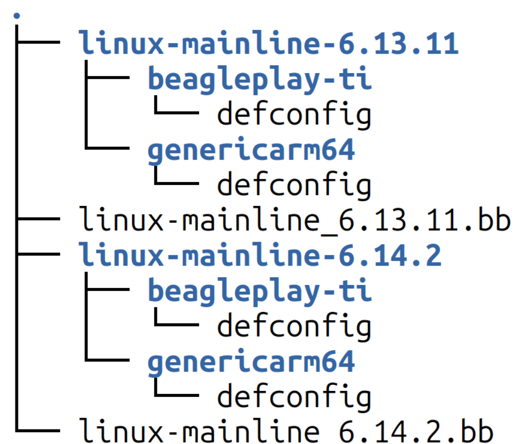
Rebuild the image and check:

- Your kernel version is indeed 6.13.11.
- You can still play MyMan with the joystick

13.3.5 Configuration for genericarm64

Do the same, create and add a configuration for the `genericarm64` machine run through QEMU. Use the same guidelines as for 6.14.2 for choosing the configuration settings.

Update the image and test that the Linux kernel version is the expected one.



13.4 Factorize code between the supported versions

As the Linux 6.13.x and 6.14.x recipes have a lot in common, it's time to remove the code duplication between them.

So, create a `linux-mainline.inc` with all the shared code. A trick, in this file, is to calculate a `KBRANCH` variable value from the value of `${PV}`. For example, for `6.14.2`, it would be equivalent to setting:

```
KBRANCH = "6.14"
```

Here's the tricky code, using a Python expression to isolation the branch version:

```
KBRANCH = "${@'.'.join(d.getVar('PV').split('.')[0:2])}"
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
${KBRANCH}.y;protocol=https"
```

Now, modify the individual `.bb` files to just include `linux-mainline.inc`, set `SRCREV` (which is unique), and keep the `defconfig` lines.

Though the `defconfig` files have the same names for all versions, it's handy not to mention them in the include file. This way, for a new version, you may just provide `defconfig` files for machines which already have one, and not for the machines that do not have one yet. In this case, the recipes don't fail to find a `defconfig` file which isn't available yet, and this allows you to prepare one by running the `menuconfig` task.

Test once again that your recipe and image build.

14 Create a New Machine

14.1 Goals

The primary goal of this lab is to create a new machine definition for our project, that will replace the custom settings that we set in `conf/local.conf`. This way, a user will just need to set the right `MACHINE` to inherit all the correct settings.

14.2 New machine configuration

We are going to create a new `beagleplay-gaming` machine.

Create the file in your `homebrew` layer, and include the definition of the `beagleplay-ti` machine. This way, the new machine inherits all the features of the existing one.

Then, in `conf/local.conf`, set `MACHINE` to the new name and try to build your usual image.

Unfortunately, everything's not ready yet, and you should get this error message:

```
MACHINE=beagleplay-gaming-k3r5 is invalid. Please set a valid MACHINE in your local.conf,
environment or other configuration file.
```

That's because of the **multiconfig** nature of the BeaglePlay build: two images are being generated, one of the regular ARM cores running Linux, and one for the R5 core in the TI AM625 SoC.

The addition of an extra machine is actually done in `meta-ti-bsp/conf/multiconfig/k3r5.conf`:

```
MACHINE:append = "-k3r5"
```

Check the other settings in this file, they are interesting too.

So, it turns out you also have to define a `beagleplay-gaming-k3r5` machine from the `beagleplay-k3r5` one.

Check that your image builds fine now.

14.3 Moving settings from `conf/local.conf`

To simplify your `conf/local.conf`, and allow other project to use your machine easily, move the below settings to your machine definition:

- `PREFERRED_PROVIDER` settings
- `PREFERRED_VERSION` settings
- Bootloader settings: `TI_WKS_BOOTLOADER_APPEND`

Of course, don't move the image related settings. They don't belong in a machine definition file.

Generate your image again to make sure everything's fine.

Also reflash your micro-SD card with your new image and make sure it's updated correctly. The shell prompt should now be:

```
root@beagleplay-gaming:~#
```

Also make sure you can still play MyMan with the joystick. This may not be the case if you overlooked something 😉.

14.4 Update overrides

Since we modified the machine name, modify the machine-based overrides in your recipes. Typically, you should add `beagleplay-gaming` when `beagleplay-ti` was used, so that the overrides continue to work.

14.5 Add dependency to `meta-ti-bsp` layer

Now that we're including machine definitions from `meta-ti-bsp`, we need to declare the dependency on this layer. This dependency actually started to exist when we added the `bbappend` recipe for `u-boot-ti-staging`.

Look for where to declare this dependency and add it there.

Run BitBake to generate the image once again. This way, if there is a syntax issue or any other misconfiguration, you will be notified.

15 Create a New Image

15.1 Goals

Our goal is to create a new image recipe so that we can eliminate the custom imagine settings in `conf/local.conf`. We also want to add more free space to our root partition.

15.2 Create a new image recipe

Create a new `core-image-gaming` image recipe in your `meta-homebrew` layer, based on the `core-image-minimal` one.

Tip: Look at the implementation of `core-image-minimal-dev`

Of course, make sure that your new image builds as expected.

From this example and the previous lab, we now know that you know to include files from other layers!

15.3 Move settings from `conf/local.conf`

Move all the package related settings from `conf/local.conf` to your new image recipe.

Rebuild and reflash your image, and make sure you can still ssh to your board and play MyMan with the joystick.

Congratulations, `conf/local.conf` just contains the `MACHINE` setting. You could reuse your image with other machines, typically to play the same game on multiple machines.

15.4 Make the root partition bigger

Before doing this, check the amount of free space in your root partition:

```
# df -h
```

`-h` in `df` (display filesystems) means human readable output, in KB, MB, GB instead of in blocks.

Now, modify your image recipe to add 256 MB of extra space.

Regenerate and reflash your image, and see the added space on your root partition.

16 Create Your Own Distro

16.1 Goals

In this lab, we are going to experiment with the creation of own distro. This will allow us to try different init manager, and configure our own login welcome message.

16.2 Create a new distro

In your `meta-homebrew` layer, create a new `asciigaming` distro, based on the `conf/distro/defaultsetup.conf` configuration. This way, we don't depend on the default Poky settings any more, and have something closer to a production distro.

At this stage, just set the `DISTRO` and `DISTRO_NAME` variables.

Update `conf/local.conf` and generate your image again.

You should see `WARNING: Duplicate inclusion for .../poky/meta/conf/distro/defaultsetup.conf` warning messages. This means you can remove the explicit inclusion of `conf/distro/defaultsetup.conf` which is already included in some way.

💡 **Tip**: You may also need to remove the `ti-deploy` directory because of conflicts in generated files.

Update your image, flash it and boot it.

In the serial console, you should see a description of the distribution:

```
Ascii Gaming nodistro.0 beagleplay-gaming /dev/ttyS2
```

Obviously, something is missing here. Look at all the variables to see where this `nodistro.0` string comes from, and correct this in `asciigaming.conf`.

Then, generate the image again, and check that the description looks better now.

16.3 Changing the init manager

Check the value of the `INIT_MANAGER` variable. It should be set to `"none"`.

On the target, nothing seems to have changed, though. You can check what the actual `init` program is:

```
# ls -la /sbin/init
lrwxrwxrwx    1 root     root            19 Mar  9 12:34 /sbin/init -> /sbin/init.sysvinit
```

Now, modify your distribution to use the `mdev-busybox` setting instead. This way, we can test the simple BusyBox init solution.

Boot your board and see the small difference.

Also try to SSH to the board, and it should... fail. It turns out that this init manager is not well tested. If you run the `ps` command, you will see that there is no SSH process.

You can start the SSH server manually and find the issue by yourself:

```
# /etc/init.d/ssh start
```

Before this issue is fixed (it's added to Root Commit's TODO list), let's try the systemd init manager instead.

Update your distribution, regenerate and reflash the image and see the difference! You can see many new services being started. This adds to the boot time of the platform[1] so that , but this opens new possibilities too in terms of making your system more secure.

ℹ There should be one error in systemd bootup messages: `[FAILED] Failed to start Generate network units from Kernel command line.` That's because we set an IP address on the kernel command line. You can safely ignore this issue.

You can also check the output of `bmaptool`, to compare the new size of the system with what we had before. Boot time is not the only time to pay...

Of course, you can make sure that SSH works again.

16.4 Setting custom welcome text

Remember the message warning that Poky is only meant to be used for testing purposes? We could also create our own to greet our users, give them directions and possibly a website to get more information from.

For your information, in most UNIX like systems, this message is stored in the /code/etc/motd.

So, run commands to find which package provides this file and then which recipe produces this package.

Then, provide your own text by adding a `bbappend` recipe to your layer. If you are in a hurry, you can also check how the Poky distro did that.

To create your text, here is a suggestion:

```
$ sudo apt install cowsay
$ cowsay -s "\
Welcome to ASCII Gaming - \
Zero polygons. Infinite possibilities." > motd
```

Whatever you choose, your ASCII gaming distribution deserves some ASCII art!

Regenerate your image and make sure everything works as expected.

16.5 Remove unnecessary layers

As your distribution doesn't depend on Poky anymore, remove the following layers from your configuration, to make sure they are no longer needed:

- `meta-poky`
- `meta-yocto-bsp`

Then, build your image again. This should work without any issue.

[1] For the moment, systemd is noticeably slower that SysV Init, because of all the new services it starts. However, it can be configured in such a way that the most critical services are started first. Systemd can even make your system boot faster because of its ability to start multiple services in parallel, unlike the other solutions that Yocto supports.

17 Setting up a binary distribution

17.1 Goals

The objective of this lab is to practice with binary distributions, by creating a package feed, and installing and removing packages directly on the live system.

17.2 Build a new package

Please copy the `vitetris` recipe from the `recipes` subdirectory in your lab archive to the `recipe-games` subdirectory in your `homebrew` layer.

i One more recipe that was created with `devtool`!

Then, build this recipe but without adding it to your image.

17.3 Change the package manager

First, modify your `asciigaming` distribution to use Debian packages.

Then, regenerate your `core-image-gaming` image. You should see many `do_package_write_deb` tasks being run, and also see a new `deb` subdirectory under `tmp-glibc/deploy/`.

Flash and boot your new image. Try to run the `dpkg -l` command on the target to list packages.

Why doesn't it work?

17.4 Enable package management

We need to add package management to our image to get access to the `dpkg` and `apt` tools for managing the packages on the target.

As the setting is part of `IMAGE_FEATURES`, add this feature to your image settings.

Regenerate your image and boot it again.

Now you should be able to list the packages on your target!

17.5 Remove a package

As an exercise, remove the `evtest` package directly on the target:

```
# apt purge evtest
```

At this stage, however, you can't add new packages yet. We need to set up a package feed and configure the image to use it.

17.6 Start a package feed server

In a new terminal window, go to $HOME/yocto-labs/poky/build/tmp-glibc/deploy/deb/, and start an HTTP server:

```
$ python3 -m http.server
```

Keep this server running at least until the end of the lab.

You also need need to generate an index for this feed, don't forget!

```
$ bitbake package-index
```

17.7 Network configuration tweaks

For the next stages, we will need the board to be able to connect to the Internet to get a correct date. Otherwise, the package lists on the feed will be considered as being in the future and won't be usable.

Fortunately, we have systemd that takes care of running an NTP client to synchronize the clock. It just needs to have access to the Internet.

Check the current date on your board. It should be out of date 😉:

```
# date
```

There are two things you need to do to make your PC act as an Internet gateway for your board:

- Enable IP forwarding:

  ```
  $ sudo sysctl -w net.ipv4.ip_forward=1
  ```

- Enable Network Address Translation (NAT):

  ```
  $ sudo iptables -t nat -A POSTROUTING -o <outif> -j MASQUERADE
  ```

  Here <outif> is a network interface connecting your PC to the Internet, as shown in the ip a command output.

There are ways to make these changes permanent. Check with your favorite search engine!

Check the date again. It should be correct now.

17.8 Configure your image to use this package feed

All you need to do is configure the package feeds in your image, but adding the below line to your conf/local.conf file:

```
PACKAGE_FEED_URIS = "http://172.24.0.1:8000"
```

The IP address is really specific to our setup, so it's right to set it in conf/local.conf.

Regenerate and boot your updated image.

You should now be able to fetch the catalog of the package feed:

```
# apt update
```

And install our new package:

```
# apt install vitetris
```

You can now play this new game:

```
# tetris
```

18 Managing Vulnerabilities

18.1 Goals

We are going to enable vulnerability checks and try to remove the number of positives.

18.2 Enabling vulnerability checks

Modify your `conf/local.conf` file to enable such checks, and build your `core-image-gaming` image as useful. The first run will take a bit of time to retrieve a local copy of the vulnerability database.

Count how many unpatched vulnerabilities you got.

18.3 Ignore irrelevant reports

Modify your distro configuration to ignore irrelevant reports.

Run the checks again. How many unpatched vulnerabilities are left?

18.4 Use the latest stable kernel

In particular, how many unpatched vulnerability were related to the Linux kernel? You can find such information in the check log file for this recipe.

Let's try to reduce this number!

Switch back to the recipe for the Linux `6.14` kernel. How many unpatched vulnerabilities are left?

Now, create a new recipe for the **latest** version of this kernel:

- You can identify this latest version on https://kernel.org/
- You can quickly get the commit ID for this version by clicking on the "changelog" link for that release.

Check the number of unpatched vulnerabilities again. Disappointing, isn't it? It turns out there are lots of vulnerabilities that are either unpatched or for which the vulnerability database is out of date, failing to mention that newer versions are no longer impacted.

18.5 Mark a vulnerability as irrelevant

Take the oldest CVE reported for the Linux kernel, have a look at its description and modify the recipe to ignore it.

Check that the number of unpatched vulnerabilities is reduced by one.

19 Using Kas

19.1 Goals

In this lab, our goal is to use Kas to generate our `core-image-gaming` image for the BeaglePlay. This should make it easier to share our project with other people.

19.2 Setup

Create a `$HOME/yocto-labs/kas` directory.

In this directory, create a `beagleplay-gaming.yml` file, taking https://github.com/mendersoftware/meta-mender-community/blob/scarthgap/kas/beagleplay-ti.yml as an example.

Of course, remove the Mender related parts that you don't need. Also remove the includes, as we will put everything in the toplevel file for better clarity.

19.3 Create missing Git repositories

The standard way of operation of Kas is to work with source repositories. Therefore, we need to turn our `meta-homebrew` and `meta-mainline` layers into Git repositories.

For example, here's how you would do this with `meta-homebrew`:

```
$ cd $HOME/yocto-labs/meta-homebrew
$ git init
$ git add .
$ git commit -as -m "Initial commit"
```

19.4 Reconstitute our project image

Using lecture contents and the Kas files in https://github.com/mendersoftware/meta-mender-community/kas as examples, fill the `beagleplay-gaming.yml` file with:

- The repositories and layers you had in `conf/bblayers.conf`.
  - To find the URL of each repository, you can check the `.git/config` file.
  - For each repository, to make your build update-proof, you should check the current Git commit id by running:

    ```
    git show | grep commit
    ```
- Your `MACHINE` setting
- Your `DISTRO` setting

- The name of the image you were building with BitBake setting
- Any options left in `conf/local.conf`

When you are done, run `kas` from inside the `kas` directory. Everything should be generated in a new `build` subdirectory.

⚠ **Make sure you don't run `kas build` inside the `poky` directory. Otherwise, `kas` would generate its output in the same output directory as used previously. This could cause trouble.**

19.5 Test your image

You can now flash the newly generated image and test it. Everything should be the same.

Tip: If there is a mismatch, look twice at `conf/local.conf`, you may have forgotten some settings...

20 Final Challenge: Media Player

20.1 Goals

Welcome to the last challenge in this course. The goals of this lab is to let you prove your new experience with OpenEmbedded and Yocto by resolving all the issues building a Media Player.

In this lab even more than in the others, you won't be given much guidance as for what to do to overcome the issues. If no explanations are given, it means either that what you have to do was already done in previous labs, or that explanations have anyway been given in the lectures.

You should also trust BitBake to give you helpful error messages. Don't hesitate to ask your instructor for tips when you are stuck, remembering that looking for solutions by yourself is the best way to build your experience.

Once you manage to build a fully functional system, you will be given a command to run. If this command executes correctly, it will give you a code to submit to your instructor, to receive a well-deserved course completion certificate.

20.2 New Project

20.2.1 Freeing Disk Space

This lab will be running an entirely new project, creating an image for QEMU on ARM64. So, check the remaining space on your storage and remove temporary data if necessary.

Don't hesitate to ask your instructor for advice.

20.2.2 New Workspace

Create the `mediaplayer` directory under `$HOME/yocto-labs` directory.

In this new directory, clone the Styhead version of Poky.

Source your environment setup script and add the layer in `$HOME/yocto-labs/layers/meta-broken/` to your layers.

That's where you'll encounter your first error. This new layer is seriously broken! However, here, you have the right to fix its contents as much as needed, as if an incompetent colleague of yours or ChatGPT had tried to create it, and you have been assigned to clean up all the mess.

20.2.3 Image to Generate

Once you have addressed this first issue, your goal is to generate a `core-image-mediaplayer` image for the `qemuarm64` machine.

From now on, you are mostly on your own to investigate and solve the issues your are facing. You are free to modify the layer and your configuration files. Note that removing recipes from the image is one

way to get the image to compile, but it won't get you anywhere as you will need the corresponding packages at the end.

The next subsection will still give you a few specific guidelines.

20.2.4 Specific Guidelines

You will encounter trouble cross-compiling the `sl` binary. From the error message you get, using the `file` command, you will be able to see that the `sl` binary was generated for your host.

That's where you will need to fix the recipe:

- To force `make` to use environment variables instead of the ones hard-coded for native compiling in `Makefile`.
- To add a source patch that modifies `Makefile` adds `$(LDFLAGS)` to the compiler command line.

Don't be impressed. You already modified recipes and created patches before.

20.3 Image Execution

20.3.1 Install QEMU emulator for ARM64

The image you managed to build is meant to be executed through QEMU for ARM64. However, though you will be able to run the `runqemu` command, the QEMU version built by Yocto currently lacks sound support. Until this issue is fixed, you will have to rely on QEMU as provided by your own GNU/Linux distribution.

On Ubuntu or Debian, you will have to run:

```
$ sudo apt install qemu-system-arm
```

Then, to know how with which parameters you should invoke QEMU, you need to run `runqemu` first. We will add parameters which are supposed to add sound support:

```
runqemu slirp qemuparams="-audiodev alsa,id=snd0 -device virtio-sound-pci,audiodev=snd0"
```

Then, from the `runqemu - INFO` lines, replace the path to the Yocto built `qemu-system-aarch64` by `/usr/bin/qemu-system-aarch64`. That's what you should run to test your image.

💡 **Tip**: Also replace `rootfs-<date>.ext4` by `rootfs.ext4` in the QEMU command line. This way, the command will still work without change after generating new versions of the image, to fix the final issues.

20.3.2 Run Final Command

In the emulated machine, it's now time to run the final command which will play free videos (2 x 3 minutes approximately), but before this will check that your image has all the required components:

```
# play-videos
```

You will have to make a few further corrections to get all remaining issues sorted out.

20.3.3 Tips

To add sound support to your kernel, you will have to find and fix a bug in the kernel recipe code.

20.3.4 Completion codes

When the `play-videos` script starts playing the two videos, everything should be all right. Just enjoy both videos until the end to get your completion codes. [1]

Then, share your completion codes with your instructor who will validate them, and will prepare a completion certificate for you, as a proof of the level of experience that you reached on OpenEmbedded and Yocto.

Thank you for this very nice achievement: having completed all these labs!

[1] If you wish to watch these videos again in higher resolution, go to https://studio.blender.org/films/.