Yocto Project and OpenEmbedded Training

Beaglebone Black variant

Practical Labs

https://bootlin.com

May 06, 2024

About this document

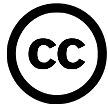
Updates to this document can be found on `https://bootlin.com/doc/training/yocto`.

This document was generated from LaTeX sources found on `https://github.com/bootlin/training-materials`.

More details about our training sessions can be found on `https://bootlin.com/training`.

Copying this document

© 2004-2024, Bootlin, `https://bootlin.com`.



This document is released under the terms of the Creative Commons CC BY-SA 3.0 license . This means that you are free to download, distribute and even modify it, under certain conditions.

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/yocto/yocto-labs.tar.xz
$ tar xvf yocto-labs.tar.xz
```

Lab data are now available in an `yocto-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

© 2004-2024 Bootlin, CC BY-SA license

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

 © 2004-2024 Bootlin, CC BY-SA license

Lab1: First Yocto Project build

Your first dive into Yocto Project and its build mechanism

During this lab, you will:

- Set up an OpenEmbedded environment
- Configure the project and choose a target
- Build your first Poky image

Setup

Before starting this lab, make sure your home directory is not encrypted using eCryptFS. OpenEmbedded cannot be used on top of an eCryptFS file system due to limitations in file name lengths.

Go to the `$HOME/yocto-labs/` directory.

Install the required packages:

```
sudo apt install bc build-essential chrpath cpio diffstat gawk git python3 texinfo wget lz4
```

Download Yocto

Download the `kirkstone` version of Poky:

```
git clone https://git.yoctoproject.org/git/poky
cd $HOME/yocto-labs/poky
git checkout -b kirkstone-4.0.5 kirkstone-4.0.5
```

Return to your project root directory (`cd $HOME/yocto-labs/`) and download the `meta-arm` and `meta-ti` layers:

```
cd $HOME/yocto-labs
git clone https://git.yoctoproject.org/git/meta-arm
cd meta-arm
git checkout -b yocto-4.0.1 yocto-4.0.1
```

```
cd $HOME/yocto-labs
git clone https://git.yoctoproject.org/git/meta-ti
cd meta-ti
git checkout -b kirkstone-labs 2a5a0339d5bd28d6f6aedaf02a6aaa9b73a248e4
git am $HOME/yocto-labs/bootlin-lab-data/\
    0001-Simplify-linux-ti-staging-recipe-for-the-Bootlin-lab.patch
```

Set up the build environment

Check you're using Bash. This is the default shell when using Ubuntu.

Export all needed variables and set up the build directory:

```
cd $HOME/yocto-labs
source poky/oe-init-build-env
```

You must specify which machine is your target. By default it is qemu. We need to build an image for a beaglebone. Update the MACHINE configuration variable accordingly. Be careful, beaglebone is different from the beagleboard machine!

Also, if you need to save disk space on your computer you can add INHERIT += "rm_work" in the previous configuration file. This will remove the package work directory once a package is built.

Don't forget to make the configuration aware of the ARM and TI layers. Edit the layer configuration file ($BUILDDIR/conf/bblayers.conf) and append the full path to the meta-arm-toolchain, meta-arm, meta-ti-bsp directories to the BBLAYERS variable.

Build your first image

Now that you're ready to start the compilation, simply run:

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

Once the build finished, you will find the output images under $BUILDDIR/tmp/deploy/images/beaglebone.

Set up the SD card

In this first lab we will use an SD card to store the bootloader, kernel and root filesystem files. The SD card image has been generated and is named core-image-minimal-beaglebone.wic.xz.

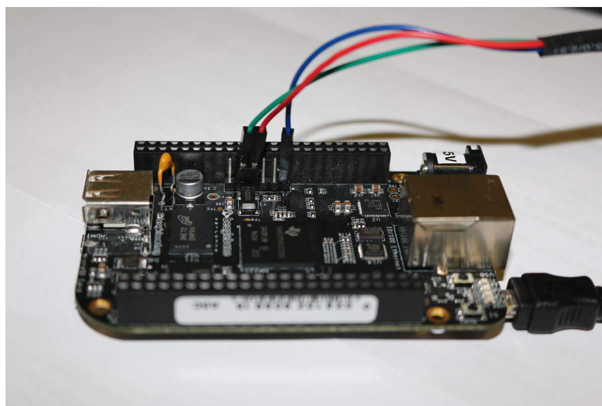
Now uncompress and flash the image with the following command:

```
xz -dc $BUILDDIR/tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.wic.xz | sudo dd \
    of=/dev/sdX conv=fdatasync bs=4M
```

Setting up serial communication with the board

The Beaglebone serial connector is exported on the 6 pins close to one of the 48 pins headers. Using your special USB to Serial adapter provided by your instructor, connect the ground wire (blue) to the pin closest to the power supply connector (let's call it pin 1), and the TX (red) and RX (green) wires to the pins 4 (board RX) and 5 (board TX)[2].

You always should make sure that you connect the TX pin of the cable to the RX pin of the board, and vice-versa, whatever the board and cables that you use.



Once the USB to Serial connector is plugged in, a new serial port should appear: /dev/ttyUSB0. You can also see this device appear by looking at the output of dmesg.

To communicate with the board through the serial port, install a serial communication program, such as picocom:

[2] See https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-Serial-Cable-F/ for details about the USB to Serial adapter that we are using.

 © 2004-2024 Bootlin, CC BY-SA license

```
sudo apt install picocom
```

If you run `ls -l /dev/ttyUSB0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. Therefore, you need to add your user to the `dialout` group:

```
sudo adduser $USER dialout
```

Important: for the group change to be effective, in Ubuntu 18.04, you have to *completely reboot* the system [3]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Now, you can run `picocom -b 115200 /dev/ttyUSB0`, to start serial communication on `/dev/ttyUSB0`, with a baudrate of `115200`. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

Boot

Insert the SD card in the dedicated slot on the BeagleBone Black. Press the S2 push button (located just above the previous slot), plug in the USB cable and release the push button. You should see boot messages on the console.

Wait until the login prompt, then enter `root` as user. Congratulations! The board has booted and you now have a shell.

[3] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.

© 2004-2024 Bootlin, CC BY-SA license

Lab2: Advanced Yocto configuration

Configure the build, customize the output images and use NFS

During this lab, you will:

- Customize the package selection
- Configure the build system
- Use the rootfs over NFS

Set up the Ethernet communication and NFS on the board

Later on, we will mount our root filesystem through the network using NFS. We will use Ethernet over USB device and therefore will only need the USB device cable that is already used to power up the board.

First we need to set the kernel boot arguments U-Boot will pass to the Linux kernel at boot time. For that, mount the `bootfs` partition of the SD card on your PC and edit the extlinux configuration file: `extlinux/extlinux.conf`.

Change the `APPEND` line to be (in just 1 line):

```
APPEND root=/dev/nfs rw nfsroot=192.168.0.1:/nfs,nfsvers=3,tcp ip=192.168.0.100:::::usb0
       g_ether.dev_addr=f8:dc:7a:00:00:02 g_ether.host_addr=f8:dc:7a:00:00:01
       rootwait rw console=${console},${baudrate}
```

Set up the Ethernet communication on the workstation

To configure your network interface on the workstation side, we need to know the name of the network interface connected to your board. You won't be able to see the network interface corresponding to the Ethernet over USB device connection yet, because it's only active when the board turns it on, from U-Boot or from Linux. When this happens, the network interface name will be `enx<macaddr>`. Given the value we gave to `g_ether.host_addr`, it will therefore be `enxf8dc7a000001`.

Then, instead of configuring the host IP address from Network Manager's graphical interface, let's do it through its command line interface, which is so much easier to use:

```
nmcli con add type ethernet ifname enxf8dc7a000001 ip4 192.168.0.1/24
```

Set up the NFS server on the workstation

First install the NFS server on the training computer and create the root NFS directory:

```
sudo apt install nfs-kernel-server
sudo mkdir -m 777 /nfs
```

Then make sure this directory is used and exported by the NFS server by adding the below line to the `/etc/exports` file:

```
/nfs *(rw,sync,no_root_squash,subtree_check)
```

Finally, make the NFS server use the new configuration:

```
sudo exportfs -r
```

 © 2004-2024 Bootlin, CC BY-SA license

Add a package to the rootfs image

You can add packages to be built by editing the local configuration file `$BUILDDIR/conf/local.conf`. The `IMAGE_INSTALL` variable controls the packages included into the output image.

To illustrate this, add the Dropbear SSH server to the list of enabled packages.

Tip: do not overwrite the default enabled package list, but append the Dropbear package instead.

Boot with the updated rootfs

First we need to put the rootfs under the NFS root directory so that it is accessible by NFS clients. Simply uncompress the archived output image in the previously created `/nfs` directory:

```
sudo tar xpf $BUILDDIR/tmp/deploy/images/beaglebone/\
  core-image-minimal-beaglebone.tar.xz -C /nfs
```

Then boot the board.

The Dropbear SSH server was enabled a few steps before, and should now be running as a service on the BeagleBone Black. You can test it by accessing the board through SSH:

```
ssh root@192.168.0.100
```

You should see the BeagleBone Black command line!

Choose a package variant

Dependencies of a given package are explicitly defined in its recipe. Some packages may need a specific library or piece of software but others only depend on a functionality. As an example, the kernel dependency is described by `virtual/kernel`.

To see which kernel is used, dry-run BitBake:

```
bitbake -vn virtual/kernel
```

In our case, we can see the `linux-ti-staging` provides the `virtual/kernel` functionality:

```
NOTE: selecting linux-ti-staging to satisfy virtual/kernel due to PREFERRED_PROVIDERS
```

We can force Yocto to select another `kernel` by explicitly defining which one to use in our local configuration. Try switching from `linux-ti-staging` to `linux-dummy` only using the local configuration.

Then check the previous step worked by dry-running again BitBake.

```
bitbake -vn virtual/kernel
```

Tip: you may need to define the more specific information here to be sure it is the one used. The `MACHINE` variable can help here.

As this was only to show how to select a preferred provider for a given package, you can now use `linux-ti-staging` again.

BitBake tips

BitBake is a powerful tool which can be used to execute specific commands. Here is a list of some useful ones, used with the `virtual/kernel` package.

- The Yocto recipes are divided into numerous tasks, you can print them by using: `bitbake -c listtasks virtual/kernel`.
- BitBake allows to call a specific task only (and its dependencies) with: `bitbake -c <task> virtual/kernel`. (`<task>` can be `menuconfig` here).
- You can force to rebuild a package by calling: `bitbake -f virtual/kernel`

- `world` is a special keyword for all packages. `bitbake --runall=fetch world` will download all packages sources (and their dependencies).
- You can get a list of locally available packages and their current version with:
  `bitbake -s`
- You can also find detailed information on available packages, their current version, dependencies or the contact information of the maintainer by visiting:
  http://recipes.yoctoproject.org/

For detailed information, please run `bitbake -h`

Going further

If you have some time left, let's improve our setup to use TFTP, in order to avoid having to reflash the SD card for every test. What you need to do is:

1. Install a TFTP server (package `tftpd-hpa`) on your system.
2. Copy the Linux kernel image and Device Tree to the TFTP server home directory (specified in `/etc/default/tftpd-hpa`) so that they are made available by the TFTP server.
3. Change the U-Boot `bootcmd` to load the kernel image and the Device Tree over TFTP.

See the training materials of our *Embedded Linux system development* course for details!

 © 2004-2024 Bootlin, CC BY-SA license

Lab3: Add a custom application

Add a new recipe to support a required custom application

During this lab, you will:

- Write a recipe for a custom application
- Integrate this application into the build

This is the first step of adding an application to Yocto. The remaining part is covered in the next lab, "Create a Yocto layer".

Setup and organization

In this lab we will add a recipe handling the `nInvaders` application found at https://ninvaders.sourceforge.net/. Before starting the recipe itself, find the `recipes-extended` directory originating from OpenEmbedded-Core and add a subdirectory for your application.

First hands on nInvaders

The nInvaders application is a terminal based game following the space invaders family. In order to deal with the text based user interface, nInvaders uses the ncurses library.

First try to find the project homepage, download the sources and have a first look: license, Makefile, requirements…

Write a minimal recipe

Create a file that respects the Yocto nomenclature: `${PN}_${PV}.bb`

Specify the source URL of the latest nInvaders archive and give a try at building your recipe:

```
bitbake ninvaders
```

Archive checksum and license

BitBake will refuse to go any further if it cannot validate the downloaded bundle using a checksum. You'll also need to provide some information about the license of the package.

Testing and troubleshooting

Try to make the recipe on your own. Also eliminate the warnings related to your recipe: some configuration variables are not mandatory but it is a very good practice to define them all.

If you hang on a problem, check the following points:

- The checksum and the URI are valid
- The dependencies are explicitly defined
- The internal state has changed, clean the working directory:
  `bitbake -c cleanall ninvaders`

One of the build failures you will face will generate many messages such as `multiple definition of `skill_level'; aliens.o:(.bss+0x674): first defined here`.

The `multiple definition` issue is due to the code base of *nInvaders* being quite old, and having multiple compilation units redefine the same symbols. While this was accepted by older *gcc* versions, since *gcc 10* this

© 2004-2024 Bootlin, CC BY-SA license

is no longer accepted by default.

While we could fix the *nInvaders* code base, we will take a different route: ask *gcc* to behave as it did before *gcc 10* and accept such redefinitions. This can be done by passing the `-fcommon` *gcc* flag.

To achieve this, make sure to add `-fcommon` to the `CFLAGS` variable.

Tip: BitBake has command line flags to increase its verbosity and activate debug outputs. Also, remember that you need to cross-compile nInvaders for ARM! Maybe, you will have to configure your recipe to resolve some mistakes done in the application's Makefile (which is often the case). A bitbake variable permits to add some Makefile's options, you should look for it.

Update the rootfs and test

Now that you've compiled the `nInvaders` application, generate a new rootfs image with `bitbake core-image-minimal`. Then update the NFS root directory. You can confirm the `nInvaders` program is present by running:

```
find /nfs -iname ninvaders
```

Access the board command line through SSH. You should be able to launch the `nInvaders` program. Now, it's time to play!

Inspect the build

The `nInvaders` application was unpacked and compiled in the recipe's work directory. Can you spot `nInvaders`' directory in the build work directory?

Once you found it, look around. You should at least spot some directories:

- The sources. Remember the `S` variable?
- `temp`. There are two kinds of files in there. Can you tell what are their purposes?
- Try to see if the licences of `nInvaders` were extracted.

 © 2004-2024 Bootlin, CC BY-SA license

Lab4: Create a Yocto layer

Add a custom layer to the Yocto project for your project needs

During this lab, you will:

- Create a new Yocto layer
- Interface this custom layer to the existing Yocto project
- Use applications from custom layers

This lab extends the previous one, in order to fully understand how to interface a custom project to the basic Yocto project.

Tools

You can access the configuration and state of layers with the `bitbake-layers` command. This command can also be used to retrieve useful information about available recipes. Try the following commands:

```
bitbake-layers show-layers
bitbake-layers show-recipes 'linux-*'
bitbake-layers show-overlayed
bitbake-layers create-layer
```

Create a new layer

With the above commands, create a new Yocto layer named `meta-bootlinlabs` with a priority of `7`. Before doing that, return to your project root directory, where by convention all layers are stored (`cd $HOME/yocto-labs/`).

Before using the new layer, we need to configure its generated configuration files. You can start with the `README` file which is not used in the build process but contains information related to layer maintenance. You can then check, and adapt if needed, the global layer configuration file located in the `conf` directory of your custom layer.

Integrate a layer to the build

To be fair, we already used and integrated a layer in our build configuration during the first lab, with `meta-ti`. This layer was responsible for `BeagleBone Black` support in Yocto. We have to do the same for our `meta-bootlinlabs` now.

There is a file which contains all the paths of the layers we use. Try to find it without looking back to the first lab. Then add the full path to our newly created layer to the list of layers.

Validate the integration of the `meta-bootlinlabs` layer with:

```
bitbake-layers show-layers
```

and make sure you don't have any warning from bitbake.

Add a recipe to the layer

In the previous lab we introduced a recipe for the nInvaders game. We included it to the existing `meta` layer. While this approach give a working result, the Yocto logic is not respected. You should instead always **use a custom layer** to add recipes or to customize the existing ones. To illustrate this we will move our previously created nInvaders recipe into the `meta-bootlinlabs` layer.

You can check the nInvaders recipe is part of the `meta` layer first:

```
bitbake-layers show-recipes ninvaders
```

Then move the nInvaders recipe to the `meta-bootlinlabs` layer. You can check that the nInvaders recipe is now part of the layer with the `bitbake-layers` command.

 © 2004-2024 Bootlin, CC BY-SA license

Lab5: Extend a recipe

Add your features to an existing recipe

During this lab, you will:

- Apply patches to an existing recipe
- Use a custom configuration file for an existing recipe
- Extend a recipe to fit your needs

Create a basic appended recipe

To avoid rewriting recipes when a modification is needed on an already existing one, BitBake allows to extend recipes and to overwrite, append or prepend configuration variables values through the so-called BitBake append files.

We will first create a basic BitBake append file, without any change made to the original recipe, to see how it is integrated into the build. We will then extend some configuration variables of the original recipe.

We here aim to extend the `linux-ti-staging` kernel recipe.

Try to create an appended recipe using the guidelines given in the slides.

You can see available `bbappend` files and the recipe they apply to by using the `bitbake-layers` tool (again!):

```
bitbake-layers show-appends
```

If the BitBake append file you just created is recognized by your Yocto environment, you should see:

```
linux-ti-staging_5.10.bb:
  $HOME/yocto-labs/meta-bootlinlabs/recipes-kernel/linux/linux-ti-staging_5.10.bbappend
```

Add patches to apply in the recipe

We want our extended `linux-ti-staging` kernel to support the Nunchuk as a joystick input. We can add this by applying patches during the `do_patch` task. The needed patches are provided with this lab. You can find them under `~/yocto-labs/bootlin-lab-data/nunchuk/linux`. For more details about how to write the driver handling the Nunchuk, have a look at our embedded Linux kernel and driver development training course at https://bootlin.com/training/kernel/.

Applying a patch is a common task in the daily Yocto process. Many recipes, appended or not, apply a specific patch on top of a mainline project. It's why patches do not have to be explicitly applied, if the recipe inherits from the patch class (directly or not), but only have to be present in the source files list.

Try adding the patches included in this lab to your BitBake append file. Do not forget to also add the `defconfig` file provided alongside the patches. This file contains the kernel configuration. It is handled automatically in the `linux-ti-staging` original recipe.

You can now rebuild the kernel to take the new patches into account:

```
bitbake virtual/kernel
```

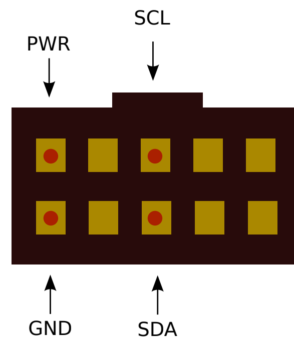
Connect the Nunchuk

Take the Nunchuk device provided by your instructor.

We will connect it to the second I2C port of the CPU (`i2c1`), with pins available on the `P9` connector.

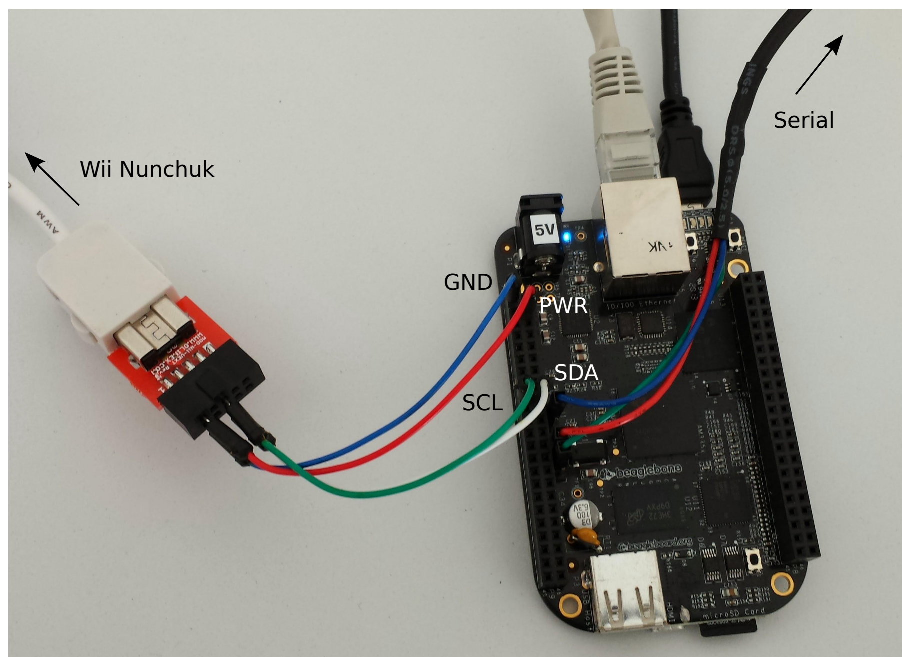
© 2004-2024 Bootlin, CC BY-SA license

Identify the 4 pins of the Nunchuk connector:



Connect the Nunchuk pins:

- The GND pin to P9 pins 1 or 2 (GND)
- The PWR pin to P9 pins 3 or 4 (DC_3.3V)
- The SCL pin to P9 pin 17 (I2C1_SCL)
- The SDA pin to P9 pin 18 (I2C1_SDA)



Test the Nunchuk

Copy the newly generated kernel and device tree images into the first SD card partition (or to the TFTP server home directory in case you enabled TFTP boot). Then boot the board and wait until you have access to the `busybox` command line.

You can then make sure that the Nunchuk is recognized and is working by checking the presence of the `js0` device file:

```
ls /dev/input/js0
```

Now display the raw events generated by the Nunchuk:

 © 2004-2024 Bootlin, CC BY-SA license

```
cat /dev/input/js0
```

You should see random characters appearing while playing with the Nunchuk. Be aware that the driver we integrated also handles accelerometer events. Therefore, moving the device will produce many events!

Patch nInvaders

The nInvaders game uses keyboard events for its controls. We first need to apply a patch introducing joystick support. The patch is located at `~/yocto-labs/bootlin-lab-data/nunchuk/ninvaders/`.

Add the patch to the nInvaders `SRC_URI`.

Then build a full `core-image-minimal` and update the NFS root directory.

Play nInvaders!

After booting the board you should be able to play nInvaders with the keyboard…and the Nunchuk! The `C` button is used to confirm and to fire, and `Z` to pause the game.

Access the board command line through SSH, and launch the game:

```
$ ninvaders
```

Lab6: Create a custom machine configuration

Let Poky know about your hardware!

During this lab, you will:

- Create a custom machine configuration
- Understand how the target architecture is dynamically chosen

Create a custom machine

The machine file configures various hardware related settings. That's what we did in lab1, when we chose the `beaglebone` one. While it is not necessary to make our custom machine image here, we'll create a new one to demonstrate the process.

Add a new `bootlinlabs` machine to the previously created layer, which will make the BeagleBone properly boot.

This machine describes a board using the `cortexa8thf-neon` tune and is a part of the `ti33x` SoC family. Add the following lines to your machine configuration file:

```
require conf/machine/include/ti-soc.inc
SOC_FAMILY:append = ":ti33x"

DEFAULTTUNE = "armv7athf-neon"
require conf/machine/include/arm/armv7a/tune-cortexa8.inc
```

Populate the machine configuration

This `bootlinlabs` machine needs:

- To select `linux-ti-staging` as the preferred provider for the kernel.
- To build `am335x-boneblack.dtb` and the `am335x-boneblack-wireless.dtb` device trees.
- To select `u-boot-ti-staging` as the preferred provider for the bootloader.
- To use `arm` as the U-Boot architecture.
- To use `am335x_evm_config` as the U-Boot configuration target.
- To use `0x80008000` as the U-Boot entry point and load address.
- To use a `zImage` kernel image type.
- To configure one serial console to `115200;ttyS0`
- To support some features:
  - `apm`
  - `usbgadget`
  - `usbhost`
  - `vfat`

 © 2004-2024 Bootlin, CC BY-SA license

- `ext2`
- `alsa`

Build an image with the new machine

You can now update the `MACHINE` variable value in the local configuration and start a fresh build.

Check generated files are here and correct

Once the generated images supporting the new `bootlinlabs` machine are generated, you can check all the needed images were generated correctly.

Have a look in the output directory, in `$BUILDDIR/tmp/deploy/images/bootlinlabs/`.

Is there anything missing?

Update the rootfs

You can now update your root filesystem, to use the newly generated image supporting our `bootlinlabs` machine!

Going further

We chose a quite generic tune (`armv7athf-neon`). It's the same one as meta-ti's definition for the Beaglebone machine. You can see what Bitbake did in `$BUILDDIR/tmp/work`.

Now, we can change the tune to `cortexa8thf-neon`. Rebuild the image, and look at `$BUILDDIR/tmp/work`. What happened?

Lab7: Create a custom image

The highest level of customization in Poky

During this lab, you will:

- Write a full customized image recipe
- Choose the exact packages you want on your board

Add a basic image recipe

A build is mainly defined by two files: the machine configuration and the image recipe. The image recipe is the top level file for the generated rootfs and the packages it includes. Our aim in this lab is to define a custom image from scratch to allow a precise selection of packages on the target. To show how to deal with real world configuration and how the Yocto Project can be used in the industry we will, in addition to the production image recipe you will use in the final product, create a development one including debug tools and show how to link the two of them to avoid configuration duplication.

First add a custom image recipe in the `meta-bootlinlabs` layer. We will name it `bootlinlabs-image-minimal`. You can find information on how to create a custom image on the dedicated Yocto Project development manual at https://docs.yoctoproject.org/dev-manual/index.html. There are different ways to customize an image, we here want to create a full recipe, using a custom `.bb` file.

Do not forget to inherit from the `core-image` class.

Select the images capabilities and packages

You can control the packages built and included into the final image with the `IMAGE_INSTALL` configuration variable. It is a list of packages to be built. You can also use package groups to include a bunch of programs, generally enabling a functionality, such as `packagegroup-core-boot` which adds the minimal set of packages required to boot an image (i.e. a shell or a kernel).

You can find the package groups under the `packagegroups` directories. To have a list of the available ones:

```
find -name packagegroups
```

Open some of them to read their description and have an idea about the capabilities they provide. Then update the installed packages of the image recipe and don't forget to add the nInvaders one!

Add a custom package group

We just saw it is possible to use package groups to organize and select the packages instead of having a big blob of configuration in the image recipe itself. We will here create a custom package for game related recipes.

With the above documentation, create a `packagegroup-bootlinlabs-games` group which inherits from the `packagegroup` class. Add the nInvaders program into its runtime dependencies.

Now update the image recipe to include the package group instead of the nInvaders program directly.

Differentiate the production recipe from the debug one

You can enable the debugging capabilities of your image just by changing the BitBake target when building the whole system. We want here to have a common base for both the production and the debug images, but also take into account the possible differences. In our example only the built package list will change.

 © 2004-2024 Bootlin, CC BY-SA license

Create a debug version of the previous image recipe, and name it `bootlinlabs-image-minimal-dbg`. Try to avoid duplicating code! Then add the `dbg-pkgs` to the image features list. It is also recommended to update the recipe's description, and to add extra debugging tools.

Build the new debug image with BitBake and check the previously included packages are present in the newly generated rootfs.

Lab8: Develop your application in the Poky SDK

Generate and use the Poky SDK

During this lab, you will:

- Build the Poky SDK
- Install the SDK
- Compile an application for your machine in the SDK environment

Build the SDK

Two SDKs are available, one only embedding a toolchain and the other one allowing for application development. We will use the latter one here.

Build an SDK for the `bootlinlabs-image-minimal` image, with the `populate_sdk` task.

Once the SDK is generated, a script will be available in `tmp/deploy/sdk`.

Install the SDK

Open a new terminal to be sure that no extra environment variable is set. We mean to show you how the SDK sets up a fully working environment.

Install the SDK in `$HOME/yocto-labs/sdk` by executing the script generated at the previous step.

```
$BUILDDIR/tmp/deploy/sdk/poky-glibc-x86_64-bootlinlabs-image-minimal-cortexa8hf-neon-toolchain-2.5.sh
```

Set up the environment

Go into the directory where you installed the SDK (`$HOME/yocto-labs/sdk`). Source the environment script:

```
source environment-setup-cortexa8hf-vfp-neon-poky-linux-gnueabi
```

Have a look at the exported environment variables:

```
env
```

Compile an application in the SDK

Download the essential `Ctris` sources at https://download.mobatek.net/sources/ctris-0.42-1-src.tar.bz2

Extract the source in the SDK:

```
tar xf ctris-0.42-1-src.tar.bz2
tar xf ctris-0.42.tar.bz2
cd ctris-0.42
```

Then modify the Makefile, to make sure that the environment variables exported by the SDK script are not overridden.

Try to compile the application. Just like nInvaders, ctris is also an old program and won't build with a recent toolchain. You will face these errors:

 © 2004-2024 Bootlin, CC BY-SA license

- The ctris makefile uses the native compiler, not the cross compiler provided by the SDK; while you could fix it using `make -e` as done for nInvaders, try fixing it by editing the `Makefile` this time; hint: you don't need to write any code, just to delete two lines
- Building with a recent GCC will give the following error, not reported by older versions:

  ```
  error: format not a string literal and no format arguments [-Werror=format-security]
  ```

  Fix this by adding `-Wno-error=format-security` to `CFLAGS`
- As for nInvaders, you will see the `multiple definition of...` error; add the `-fcommon` flag to `CFLAGS` also for ctris

You can check the application was successfully compiled for the right target by using the `file` command. The `ctris` binary should be an ELF 32-bit LSB executable compiled for ARM.

Finally, you can copy the binary to the board, by using the `scp` command. Then run it and play a bit to ensure it is working fine!

© 2004-2024 Bootlin, CC BY-SA license

Lab9: Using devtool

Automate recipe development and debugging using devtool

During this lab, you will:

- Use devtool to generate a new recipe more quickly
- Modify a recipe to add a new patch using devtool
- Upgrade a recipe to a newer version using devtool

Generate a new recipe

The `devtool` executable is already available in your shell after sourcing the `oe-init-build-env` script. Take a look at the sub-commands it offers:

```
devtool --help
```

We now want to add a new recipe for the "GNU Hello" program (https://www.gnu.org/software/hello/). We can do so using the `add` subcommand.

However we want to use version 2.10 instead of the latest mainline version, so we can use the `--version` option:

```
devtool add --version 2.10 https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

You can observe that `devtool` calls `bitbake` multiple times. In the output messages, some lines are particularly interesting:

```
INFO: Creating workspace layer in .../build/workspace
...
INFO: Using default source tree path .../build/workspace/sources/hello
...
INFO: Recipe .../build/workspace/recipes/hello/hello_2.10.bb has been automatically created; \
    further editing may be required to make it fully functional
```

The first `INFO` line means devtool has created the workspace in the `workspace` directory inside the `build` directory. Take a moment to inspect how the workspace looks like, but remember to never modify it manually: the workspace is the internal state of devtool, so it may stop working if it is modified externally. The workspace now looks like this:

```
$ tree workspace/ | head -n20
workspace/
|-- README
|-- appends
| `-- hello_2.10.bbappend
|-- conf
| `-- layer.conf
|-- recipes
| `-- hello
| `-- hello_2.10.bb
`-- sources
```

 © 2004-2024 Bootlin, CC BY-SA license

```
    `-- hello
        ...
        |-- GNUmakefile
        |-- INSTALL
        |-- Makefile.am
        |-- Makefile.in
$
```

As you can see the workspace is a layer, having a `conf/layer.conf` file. It also has a directory for recipes which already holds a recipe for GNU Hello 2.10. The `sources` directory contains the source code of the hello recipe, that devtool uses internally to manage patches.

You can see that the workspace layer has been enabled by checking your `conf/bblayers.conf` or by running `bitbake-layers show-layers`.

It's time to try building the GNU Hello program via devtool:

```
devtool build hello
```

In the output messages, look for:

```
NOTE: hello: compiling from external source tree .../workspace/sources/hello
```

This means that the recipes managed by devtool do not download the source code in the usual way, but rather they use the local copy of the sources that has been previously populated is in `workspace/sources/<RECIPENAME>`.

You can have a look at the output of the build, which is in the usual location inside the workdir: `tmp/work/<ARCHITECTURE>/hello/2.10-r0/image/`.

Using `devtool deploy-target` is a handy way to try the newly built code on the target:

```
devtool deploy-target hello root@192.168.0.100
```

This will send to the device all the files in the `image` subdirectory of the recipe work directory, keeping the directory layout and file permissions. You can now test the program on the target:

```
$ ssh root@192.168.0.100
root@bootlinlabs:~# hello
Hello, world!
root@bootlinlabs:~#
```

Before moving the recipe to the meta-bootlinlabs layer, have a look at the recipe code as it has been generated by devtool:

```
devtool edit-recipe hello
```

This command will open the `hello_2.10.bb` file of the workspace in an editor. You can see that the recipe has various comments added by devtool: you should review them, fix or adapt whatever is needed, and save the recipe.

First of all, check the exact license by reading the first few lines of `workspace/sources/hello/src/hello.c` and you will discover it is a "GPL 3.0 or later". The license guessed by devtool is `GPL-3.0-only`, thus replace it by `GPL-3.0-or-later`.

Devtool has already computed the hashes for you, but there are several `SRC_URI` hashed, thus feel free to remove all of them except `SRC_URI[sha256sum]`.

You can also simplify the `SRC_URI` line using `GNU_MIRROR`, getting:

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

Note the `inherit` line: from the content of the source code files of the GNU Hello program, devtool already guessed that it is configured using the Autotools and it is using GNU Gettext. This saved us a lot of time! There is a comment above the `inherit` line: do not remove it for the moment.

Finally there is a line setting `EXTRA_OECONF` to an empty string. This line is useless unless you know you need to set some configuration flags, thus you can remove it together with the comment line.

The resulting recipe (`workspace/recipes/hello/hello_2.10.bb`) should now look like:

```
LICENSE = "GPL-3.0-or-later"
LIC_FILES_CHKSUM = "file://COPYING;md5=d32239bcb673463ab874e80d47fae504"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
SRC_URI[sha256sum] = "31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f516b"

# NOTE: if this software is not capable of being built in a separate build directory
# from the source, you should replace autotools with autotools-brokensep in the
# inherit line
inherit gettext autotools
```

You can double check that your recipe still works as expected using `devtool build` and `devtool deploy-target`. When you are done you can remove the files from the target:

```
devtool undeploy-target hello root@192.168.0.100
```

You can now stop having the hello recipe handled by devtool and move it to the meta-bootlinlabs layer:

```
devtool finish -f hello ../meta-bootlinlabs/
```

Now the recipe is in `../meta-bootlinlabs/recipes-hello/hello/hello_2.10.bb`. You can read the .bb file and verify the content has not changed. Move the recipe to a more reasonable directory name:

```
mv ../meta-bootlinlabs/recipes-hello ../meta-bootlinlabs/recipes-utils
```

Now check the content of the workspace: the hello recipe is not there anynmore. However the source code of the GNU Hello program as still in the `workspace/sources/hello` directory. Devtool does not delete it, in case you have done any valuable work in it that you still haven't saved to a patch. As it is not your case, just delete it:

```
rm -fr workspace/sources/hello/
```

Now double check that the recipe is still building correctly in the meta-bootlinlabs layer. There's no reason it should fail, right?

```
bitbake hello
```

Oops, it failed! To have a hint about the reason, have a look at the comment we didn't remove from the recipe:

```
# NOTE: if this software is not capable of being built in a separate build directory
# from the source, you should replace autotools with autotools-brokensep in the
# inherit line
```

This points exactly to the problem with GNU Hello 2.10: it fails building out-of-tree, i.e. with a build directory different from the source directory, as is done by default when using `autotools.bbclass`. Just fix the recipe as the comment suggests by changing the `inherit` line. You can then remove the comment as well. Your work dir is now polluted so you need to clean it before running a new build:

 © 2004-2024 Bootlin, CC BY-SA license

```
bitbake -c clean hello
bitbake hello
```

That's all: you now have a very concise (and working!) hello recipe in the meta-bootlinlabs layer!

Modify a recipe

Now use devtool to modify the hello recipe adding a patch. This can be very useful if you have to fix a bug in a third-party program and there is no patch around to fix it yet.

Use `devtool modify` to put an existing recipe under the control of devtool:

```
devtool modify hello
```

Now you have two hello recipes: one in the meta-bootlinlabs layers and one in the workspace layer. To ensure about which will be used by bitbake, you can inspect the layer priorities.

Now enter the source directory. You can notice devtool created a git repository into it:

```
cd workspace/sources/hello/
git log
```

The generated git repository contains only one commit which contains the pristine sources as extracted from the downloaded tarball. Now open the `src/hello.c` with an editor, go around line 60 and edit the "Hello, world" string to print whatever you prefer. Save and exit the editor. Check your modification using `git diff -- src/hello.c` and test it as done earlier:

```
devtool build hello
devtool deploy-target hello root@192.168.0.100
```

Edit again the source code if needed. When you are satisfied with your changes, just commit them using git:

```
git add src/hello.c
git commit -m 'Change the greeting message'
```

Check your changes in the git repository, then exit the workspace:

```
git log
cd $BUILDDIR
```

You are now ready to update the original recipe to take into account your changes:

```
devtool update-recipe hello
```

Looking at the `recipes-utils/hello` directory in the meta-bootlinlabs layer you can notice that a new patch has been created and added to `SRC_URI`. The patch applies the same change that you have just committed. With devtool you don't need to handle the patch syntax, but rather you can use git as you are probably already used to.

Now remove the hello recipe from under the control of devtool, then cleanup as done earlier:

```
devtool reset hello
rm -fr workspace/sources/hello/
```

Upgrade a recipe to the latest mainline version

devtool can be used to simplify the upgrade of a recipe to a newer mainline version.

First, it can detect which is the latest version available on the original site. This is based on heuristics that work for projects that store their source tarballs in a standard way, which most do. To check for the latest

version, use:

```
devtool latest-version hello
```

At the time of this writing, the output shows that 2.12.1 is the latest version. You can modify the recipe to use the newest version, including the computation of the new hashes and renaming the .bb file, with a single command:

```
devtool upgrade hello
```

Note the INFO line about license changes that you have to verify. Double check in the sources that the license is still “GPL 3.0 or later”, then open the recipe as it is currently in the devtool workspace:

```
devtool edit-recipe hello
```

The recipe file contains a diff between the old and the new version of the recipe. If you can see that the license changes are not relevant, as is the case for the upgrade from 2.10 to 2.12.1, then you can simply delete the comment, save and exit.

Now check that everything works as done earlier, then apply your changes to the layer.

```
devtool finish hello ../meta-bootlinlabs/
rm -fr workspace/sources/hello/
```

Now the meta-bootlinlabs layer contains the latest version of GNU Hello!

 © 2004-2024 Bootlin, CC BY-SA license