

Buildroot system development

BeagleBone Black variant

Practical Labs

  
<https://bootlin.com>

June 15, 2026

## About this document

Updates to this document can be found on <https://bootlin.com/training/buildroot>.

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

© 2004-2026, Bootlin, <https://bootlin.com>.



This document is released under the terms of the [Creative Commons CC BY-SA 3.0 license](https://creativecommons.org/licenses/by-sa/3.0/). 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/buildroot/buildroot-bbb-labs.tar.xz
$ tar xvf buildroot-bbb-labs.tar.xz
```

Lab data are now available in an `buildroot-bbb-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<sup>1</sup>, *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.

---

<sup>1</sup>This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- 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/`

# Basic Buildroot usage

*Objectives:*

- *Get Buildroot*
- *Configure a minimal system with Buildroot for the BeagleBone Black*
- *Do the build*
- *Prepare the BeagleBone Black for usage*
- *Flash and test the generated system*

## Setup

Go to the `$HOME/buildroot-bbb-labs/` directory.

As specified in the Buildroot manual<sup>2</sup>, Buildroot requires a few packages to be installed on your machine. Let's install them using Ubuntu's package manager:

```
sudo apt install sed make binutils gcc g++ bash patch \  
gzip bzip2 perl tar cpio python3 unzip rsync wget libncurses-dev
```

## Download Buildroot

Since we're going to do Buildroot development, let's clone the Buildroot source code from its Git repository:

```
git clone https://gitlab.com/buildroot.org/buildroot.git
```

Go into the newly created `buildroot` directory.

We're going to start a branch from the `2025.02.6` Buildroot release, with which this training has been tested.

```
git checkout -b bootlin 2025.02.6
```

## Configuring Buildroot

If you look under `configs/`, you will see that there is a file named `beaglebone_defconfig`, which is a ready-to-use Buildroot configuration file to build a system for the BeagleBone Black Wireless platform. However, since we want to learn about Buildroot, we'll start our own configuration from scratch!

Start the Buildroot configuration utility:

```
make menuconfig
```

Of course, you're free to try out the other configuration utilities `nconfig`, `xconfig` or `gconfig`.

Now, let's do the configuration:

- Target Options menu
  - It is quite well known that the BeagleBone Black Wireless is an ARM based platform, so select ARM (little endian) as the target architecture.

---

<sup>2</sup><https://buildroot.org/downloads/manual/manual.html#requirement-mandatory>

- According to the BeagleBone Black Wireless website at <https://beagleboard.org/BLACK>, it uses a Texas Instruments AM335x, which is based on the ARM Cortex-A8 core. So select `cortex-A8` as the `Target Architecture Variant`.
- On ARM two *Application Binary Interfaces* are available: EABI and EABIhf. Unless you have backward compatibility concerns with pre-built binaries, EABIhf is more efficient, so make this choice as the `Target ABI` (which should already be the default anyway).
- The other parameters can be left to their default value: ELF is the only available `Target Binary Format`, VFPv3-D16 is a sane default for the *Floating Point Unit*, and using the ARM instruction set is also a good default (we could use the Thumb-2 instruction set for slightly more compact code).
- We don't have anything special to change in the `Build options` menu, but take nonetheless this opportunity to visit this menu, and look at the available options. Each option has a help text that tells you more about the option.
- `Toolchain` menu
  - By default, Buildroot builds its own toolchain. This takes quite a bit of time, and for ARMv7 platforms, there is a pre-built toolchain provided by ARM. We'll use it through the *external toolchain* mechanism of Buildroot. Select `External toolchain` as the `Toolchain type`. Do not hesitate however to look at the available options when you select `Buildroot toolchain` as the `Toolchain type`.
  - Select `Bootlin toolchains` as the `Toolchain`. It will automatically select the `armv7-eabihf glibc bleeding-edge 2024.05-1` variant, which is fine for our needs. Buildroot can either use pre-defined toolchains such as the ones provided by ARM or Bootlin, or custom toolchains (either downloaded from a given location, or pre-installed on your machine).
- `System configuration` menu
  - For our basic system, we don't need a lot of custom *system configuration* for the moment. So take some time to look at the available options, and put some custom values for the `System hostname`, `System banner` and `Root password`.
- `Kernel` menu
  - We obviously need a Linux kernel to run on our platform, so enable the `Linux kernel` option.
  - By default, the most recent Linux kernel version available at the time of the Buildroot release is used. In our case, we want to use a specific version, to make sure our build is reproducible. So select `Custom version` as the `Kernel version`, and enter `6.12.47` in the `Kernel version` text field that appears.
  - Now, we need to define which kernel configuration to use. We'll start by using a default configuration provided within the kernel sources themselves, called a *defconfig*. To identify which *defconfig* to use, you can look in the kernel sources directly, at <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/arm/configs/?id=v6.12>. In practice, for this platform, it is not trivial to find which one to use: the AM335x processor is supported in the Linux kernel as part of the support for many other Texas Instruments processors: OMAP2, OMAP3, OMAP4, etc. So the appropriate *defconfig* is named `omap2plus_defconfig`. You can open up this file in the Linux kernel Git repository viewer, and see it contains the line `CONFIG_SOC_AM33XX=y`, which is a good indication that it has the support for the processor used in the BeagleBone Black. Now that we have identified the *defconfig* name, enter `omap2plus` in the `Defconfig name` option.
  - The `Kernel binary format` is the next option. Since we are going to use a recent U-Boot bootloader, we'll keep the default of the `zImage` format.
  - On ARM, all modern platforms now use the *Device Tree* to describe the hardware. The BeagleBone Black Wireless is in this situation, so you'll have to enable the `Build a Device Tree Blob` option. At <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/>

[arm/boot/dts/?id=v6.12](#), you can see the list of all Device Tree files available in the 6.12 Linux kernel (note: the Device Tree files for boards use the `.dts` extension). The one for the BeagleBone Black Wireless is `am335x-boneblack-wireless.dts` located in `ti/omap/`. Even if talking about Device Tree is beyond the scope of this training, feel free to have a look at this file to see what it contains. Back in Buildroot, enable `Build a Device Tree Blob (DTB)` and type `ti/omap/am335x-boneblack-wireless` as the `In-tree Device Tree Source` file names.

- The kernel configuration for this platform requires having OpenSSL available on the host machine. To avoid depending on the OpenSSL development files installed by your host machine Linux distribution, Buildroot can build its own version: just enable the `Needs host OpenSSL` option.
- **Target packages** menu. This is probably the most important menu, as this is the one where you can select amongst the 3000+ available Buildroot packages which ones should be built and installed in your system. For our basic system, enabling `BusyBox` is sufficient and is already enabled by default, but feel free to explore the available packages. We'll have the opportunity to enable some more packages in the next labs.
- **Filesystem images** menu. For now, keep only the `tar the root filesystem` option enabled. We'll take care separately of flashing the root filesystem on the SD card.
- **Bootloaders** menu.
  - We'll use the most popular ARM bootloader, *U-Boot*, so enable it in the configuration.
  - Select `Kconfig` as the `Build system`. U-Boot is transitioning from a situation where all the hardware platforms were described in C header files to a system where U-Boot re-uses the Linux kernel configuration logic. Since we are going to use a recent enough U-Boot version, we are going to use the latter, called *Kconfig*.
  - Use the custom version of U-Boot `2024.04`.
  - Look at <https://gitlab.denx.de/u-boot/u-boot/-/tree/v2024.04/configs> to identify the available U-Boot configurations. For many AM335x platforms, U-Boot has a single configuration called `am335x_evm_defconfig`, which can then be given the exact hardware platform to support using a Device Tree. So we need to use `am335x_evm` as `Board defconfig` and `DEVICE_TREE=am335x-boneblack-wireless` as `Custom make options`
  - U-Boot on AM335x is split in two parts: the first stage bootloader called `MLO` and the second stage bootloader called `u-boot.img`. So, select `u-boot.img` as the `U-Boot binary format`, enable `Install U-Boot SPL binary image` and use `MLO` as the `U-Boot SPL binary image name`.

You're now done with the configuration!

## Building

You could simply run `make`, but since we would like to keep a log of the build, we'll redirect both the standard and error outputs to a file, as well as the terminal by using the `tee` command:

```
make 2>&1 | tee build.log
```

While the build is on-going, please go through the following sections to prepare what will be needed to test the build results.

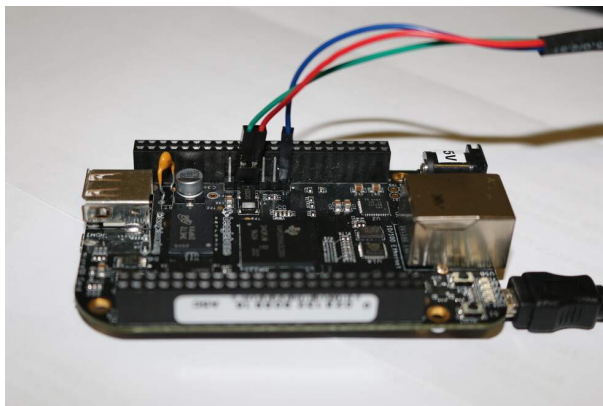
## Prepare the BeagleBone Black Wireless

The BeagleBone Black is powered via the USB-A to mini-USB cable, connected to the mini-USB connector labeled `P4` on the back of the board.

The Beaglebone serial connector is exported on the 6 male 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)<sup>3</sup>.

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`:

```
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<sup>4</sup>. 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.

## Prepare the SD card

Our SD card needs to be split in two partitions:

- A first partition for the bootloader. It needs to comply with the requirements of the AM335x SoC so that it can find the bootloader in this partition. It should be a FAT32 partition. We will store the bootloader (MLO and `u-boot.img`), the kernel image (`zImage`) and the Device Tree (`am335x-boneblack.dtb`).
- A second partition for the root filesystem. It can use whichever filesystem type you want, but for our system, we'll use `ext4`.

First, let's identify under what name your SD card is identified in your system: look at the output of `cat /proc/partitions` and find your SD card. In general, if you use the internal SD card reader of a laptop, it will be `mmcblk0`, while if you use an external USB SD card reader, it will be `sdX` (i.e. `sdb`, `sdc`, etc.). **Be careful:** `/dev/sda` is generally the hard drive of your machine!

<sup>3</sup>See <https://www.olimex.com/Products/USB-Modules/Interfaces/USB-SERIAL-F> for details about the USB to Serial adapter that we are using.

<sup>4</sup>As explained on <https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/>.

If your SD card is `/dev/mmcblk0`, then the partitions inside the SD card are named `/dev/mmcblk0p1`, `/dev/mmcblk0p2`, etc.

To format your SD card, do the following steps:

1. Unmount all partitions of your SD card (they are generally automatically mounted by Ubuntu)
2. Erase the beginning of the SD card to ensure that the existing partitions are not going to be mistakenly detected:  
`sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=16.`
3. Create the two partitions.
  - Start the `cdisk` tool for that:  
`sudo cfdisk /dev/mmcblk0`
  - Choose the `dos` partition table type
  - Create a first small partition (128 MB), primary, with type `e` (*W95 FAT16*) and mark it bootable
  - Create a second partition, also primary, with the rest of the available space, with type `83` (*Linux*).
  - Exit `cdisk`
4. Format the first partition as a *FAT32* filesystem:  
`sudo mkfs.vfat -a -F 32 -n boot /dev/mmcblk0p1.`
5. Format the second partition as an *ext4* filesystem:  
`sudo mkfs.ext4 -L rootfs -E nodiscard /dev/mmcblk0p2.`
  - `-L` assigns a volume name to the partition
  - `-E nodiscard` disables bad block discarding. While this should be a useful option for cards with bad blocks, skipping this step saves long minutes in SD cards.

Remove the SD card and insert it again, the two partitions should be mounted automatically, in `/media/$USER/boot` and `/media/$USER/rootfs`.

Now everything should be ready. Hopefully by that time the Buildroot build should have completed. If not, wait a little bit more.

## Flash the system

Once Buildroot has finished building the system, it's time to put it on the SD card:

- Copy the `MLO`, `u-boot.img`, `zImage` and `am335x-boneblack-wireless.dtb` files from `output/images/` to the boot partition of the SD card.
- Extract the `rootfs.tar` file to the `rootfs` partition of the SD card, using:  
`sudo tar -C /media/$USER/rootfs/ -xf output/images/rootfs.tar .`
- Create a file named `extlinux/extlinux.conf` in the boot partition. This file should contain the following lines:

```
label buildroot
  kernel /zImage
  devicetree /am335x-boneblack-wireless.dtb
  append console=ttyO0,115200 root=/dev/mmcblk0p2 rootwait
```

These lines teach the U-Boot bootloader how to load the Linux kernel image and the Device Tree, before booting the kernel. It uses a standard U-Boot mechanism called *distro boot command*, see <https://source.denx.de/u-boot/u-boot/-/raw/master/doc/README.distro> for more details.

Cleanly unmount the two SD card partitions, and eject the SD card.

## Boot the system

Insert the SD card in the BeagleBone Black. Push the S2 button (located near the USB host connector) and plug the USB power cable while holding S2. Pushing S2 forces the BeagleBone Black to boot from the SD card instead of from the internal eMMC.

You should see your system booting. Make sure that the U-Boot SPL and U-Boot version and build dates match with the current date. Do the same check for the Linux kernel.

Login as `root` on the BeagleBone Black, and explore the system. Run `ps` to see which processes are running, and look at what Buildroot has generated in `/bin`, `/lib`, `/usr` and `/etc`.

Note: if your system doesn't boot as expected, make sure to reset the U-Boot environment by running the following U-Boot commands:

```
env default -f -a
saveenv
```

and reset. This is needed because the U-Boot loaded from the SD card still loads the U-Boot environment from the eMMC. Ask your instructor for additional clarifications if needed.

## Explore the build log

Back to your build machine, since we redirected the build output to a file called `build.log`, we can now have a look at it to see what happened. Since the Buildroot build is quite verbose, Buildroot prints before each important step a message prefixed by the `>>>` sign. So to get an overall idea of what the build did, you can run:

```
grep ">>>" build.log
```

You see the different packages between downloaded, extracted, patched, configured, built and installed.

Feel free to explore the `output/` directory as well.

# Root filesystem construction

*Objectives:*

- *Explore the build output*
- *Customize the root filesystem using a rootfs overlay*
- *Customize the Linux kernel configuration*
- *Use a post-build script*
- *Customize the kernel with patches and*
- *Add more packages*
- *Use defconfig files and out of tree build*

## Explore the build output

Now that we have discussed during the lectures the organization of the Buildroot *output* tree, take some time to look inside *output/* for the different build artefacts. And especially:

- Identify where the cross-compiler has been installed.
- Identify where the source code for the different components has been extracted, and which packages have been built.
- Identify where the target root filesystem has been created, and read the `THIS_IS_NOT_YOUR_ROOT_FILESYSTEM` file.
- See where the `staging` symbolic link is pointing to.

## Use a *rootfs overlay* to setup the network

The BeagleBone Black Wireless does not have any Ethernet interface, so we will use Ethernet over USB to provide network connectivity between our embedded system and the development PC. To achieve this we will need to:

1. Add an init script to setup network over USB
2. Add a configuration file that configures the network interface with the appropriate IP address

## Init script for USB network setup

There are different mechanisms to configure *USB gadget* with Linux: we will use the *gadget configs* interface, which allows from user-space to create USB devices providing an arbitrary set of functionalities<sup>5</sup>.

Since the setup of such a *USB gadget* is not trivial, we provide a ready-to-use shell script that we will add to the *init scripts* of the Buildroot system. The script is called `S30usb gadget` and is available from this lab data directory at `$HOME/buildroot-bbb-labs/buildroot-rootfs/`.

---

<sup>5</sup>See [https://elinux.org/images/e/ef/USB\\_Gadget\\_Configs\\_API\\_0.pdf](https://elinux.org/images/e/ef/USB_Gadget_Configs_API_0.pdf) for more details

We could copy this script directly to our SD card, but this would mean that the next time we reflash the SD card with the root filesystem produced by Buildroot, we would lose those changes.

In order to automate the addition of this script to the root filesystem as part of the Buildroot build, we will use the **rootfs overlay** mechanism. Since this *overlay* is specific to our project, we will create a custom directory for our project within the Buildroot sources: `board/bootlin/beagleboneblack/`.

Within this directory, create a `rootfs-overlay` directory, and in `menuconfig`, specify `board/bootlin/beagleboneblack/rootfs-overlay` as the *rootfs overlay* (option `BR2_ROOTFS_OVERLAY`).

Copy the `S30usb gadget` script to your overlay so that it is located in `board/bootlin/beagleboneblack/rootfs-overlay/etc/init.d/S30usb gadget`. At boot time, the default init system used by Buildroot will execute all scripts named `SXX*` in `/etc/init.d`.

## IP address configuration

By default, Buildroot uses the `ifup` program from BusyBox, which reads the `/etc/network/interfaces` file to configure network interfaces. So, in `board/bootlin/beagleboneblack/rootfs-overlay`, create a file named `etc/network/interfaces` with the following contents:

```
auto lo
iface lo inet loopback

auto usb0
iface usb0 inet static
    address 192.168.42.2
    netmask 255.255.255.0
```

Then, rebuild your system by running `make`. Here as well, we don't need to do a full rebuild, since the *rootfs overlays* are applied at the end of each build. You can check in `output/target/etc/init.d/` and `output/target/etc/network/` if both the init script and network configuration files were properly copied.

Reflash the root filesystem on the SD card, and boot your BeagleBone Black. It should now have an IP address configured for `usb0` by default.

## Configure the network on your host

In the next sections of this lab, we will want to interact with the BeagleBone Black over the network, through USB. So in this section, we'll configure your host machine to assign an appropriate IP address for the USB network interface.

On Ubuntu, the network interfaces corresponding to Ethernet-over-USB connections are named `enx<macaddr>`. The host MAC address is hardcoded in the `S30usb gadget` script to `f8:dc:7a:00:00:01`, so the interface will be named `enxf8dc7a000001`.

To configure an IP address for this interface on your host machine, we'll use NetworkManager and its command line interface:

```
nmcli con add type ethernet ifname enxf8dc7a000001 ip4 192.168.42.1/24
```

*Note: using `ip` in the command line is not recommended, because Network Manager will unconfigure and reconfigure the network interface each time the board is rebooted.*

Once this is done, make sure you can communicate with your target using `ping`.

## Add *dropbear* as an SSH server

As a first additional package to add to our system, let's add the *dropbear* SSH client/server. The server will be running on the BeagleBone Black, which will allow us to connect over the network to the BeagleBone Black.

Run `make menuconfig`, and enable the `dropbear` package. You can use the search capability of `menuconfig` by typing `/`, enter `DROPBEAR`. It will give you a list of results, and each result is associated with a number between parenthesis, like (1). Then simply press 1, and `menuconfig` will jump to the right option.

After leaving `menuconfig`, restart the build by running `make`.

In this case, we do not need to do a full rebuild, because a simple `make` will notice that the `dropbear` package has not been built, and will therefore trigger the build process.

Re-extract the root filesystem tarball in the `rootfs` partition of the SD card. Don't forget to replace the entire root filesystem:

```
rm -rf /media/$USER/rootfs/*
sudo tar -C /media/$USER/rootfs/ -xf output/images/rootfs.tar
```

Now, boot the new system on the BeagleBone Black. You should see a message:

```
Starting dropbear sshd: OK
```

Now, from your PC, if you try to SSH to the board by doing:

```
ssh root@192.168.42.2
```

## Use a post-build script

Write a shell script that creates a file named `/etc/build-id` in the root filesystem, containing the Git commit id of the Buildroot sources, as well as the current date. Since this script will be executed as a post-build script, remember that the first argument passed to the script is `$(TARGET_DIR)`.

Register this script as a post-build script in your Buildroot configuration, run a build, and verify that `/etc/build-id` is created as expected.

## Patch the Linux kernel

Now, we would like to connect an additional peripheral to our system: the *Wii Nunchuk*. Using this custom peripheral requires adding a new driver to the Linux kernel, making changes to the Device Tree describing the hardware, and changing the kernel configuration. This is the purpose of this section.

We will first create a new directory to store our kernel patches. It will sit next to our *rootfs overlay* in our project-specific directory:

```
mkdir board/bootlin/beagleboneblack/patches/linux/
```

Copy in this directory the two patches that we provided with the data of this lab, in `$HOME/buildroot-bbb-labs/buildroot-rootfs/linux/`:

```
cp $HOME/buildroot-bbb-labs/buildroot-rootfs/linux/*.patch \
  board/bootlin/beagleboneblack/patches/linux/
```

The first patch adds the driver, the second patch adjusts the Device Tree. Feel free to look at them. If you're interested, you can look at our training course *Embedded Linux kernel driver development*, which precisely covers the development of this driver.

Now, we need to tell Buildroot to apply these patches before building the kernel. To do so, run `menuconfig`, go to the *Build options* menu, and adjust the `Global patch directories` option to `board/bootlin/beagleboneblack/patches/`.

Let's now clean up completely the `linux` package so that its sources will be re-extracted and our patches applied the next time we do a build:

```
make linux-dirclean
```

If you check in `output/build/`, the `linux-<version>` directory will have disappeared.

Now, we need to adjust our kernel configuration to enable the *Wii Nunchuk* driver. To start the Linux kernel configuration tool, run:

```
make linux-menuconfig
```

This will:

- Extract the Linux kernel sources
- Apply our two patches
- Load the defined kernel configuration, from `omap2plus_defconfig`
- Start the kernel menuconfig tool

Once in the kernel menuconfig, enable the option `CONFIG_JOYSTICK_WIICHUCK`, and make sure it is enabled statically. Also, make sure the `CONFIG_INPUT_EVDEV` option is enabled statically (by default it is enabled as a module). Once those options are set, leave the kernel menuconfig.

Your kernel configuration has now been customized, but those changes are only saved in `output/build/linux-<version>/config`, which will be deleted at the next `make clean`. So we need to save such changes persistently. To do so:

1. Run Buildroot menuconfig
2. In the Kernel menu, instead of Using a defconfig, chose Using a custom config file. This will allow us to use our own custom kernel configuration file, instead of a pre-defined *defconfig* that comes with the kernel sources.
3. In the Configuration file path, enter `board/bootlin/beagleboneblack/linux.config`.
4. Exit menuconfig
5. Run `make linux-update-defconfig`. This will generate the configuration file in `board/bootlin/beagleboneblack/linux.config`. It will be a *minimal* configuration file (i.e. a *defconfig*). In this file, verify that the option `CONFIG_JOYSTICK_WIICHUCK` is properly set to `y`.

You can now restart the build of the kernel:

```
make
```

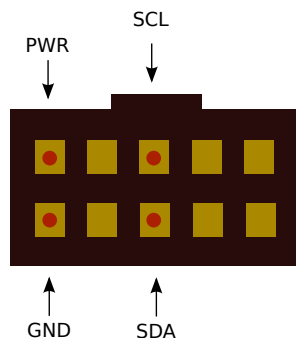
It should hopefully end successfully, and if you look closely at the build log, you should see the file `wiichuck.c` being compiled.

## Connect the Wii 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.

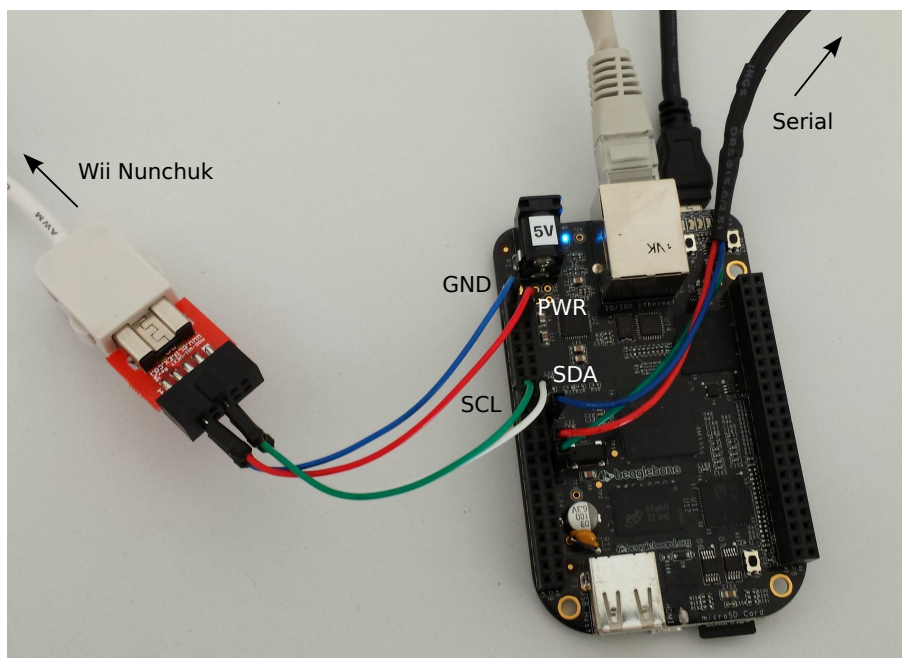
Identify the 4 pins of the nunchuk connector:



Nunchuk i2c pinout  
(UEXT connector from Olimex, front view)

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 CLK pin to P9 pin 17 (I2C1\_SCL)
- The DATA pin to P9 pin 18 (I2C1\_SDA)



## Test the *nunchuk*

Reflash your system, both the *Device Tree*, Linux kernel image and root filesystem, and boot it.

In the kernel boot log, you should see a message like:

```
input: Wiichuck expansion connector as /devices/platform/ocp/4802a000.i2c/i2c-1/1-0052/input/input0
```

You can also explore *sysfs*, and see that your Nunchuk device is handled by the system:

```
cat /sys/bus/i2c/devices/1-0052/name
```

Now, to get the raw events coming from the Nunchuk, you can do:

```
cat /dev/input/event0
```

or, if you prefer to see hexadecimal values instead of raw binary:

```
cat /dev/input/event0 | hexdump -C
```

You should see events when moving the Nunchuk (it has an accelerometer), when moving the joystick and pushing the buttons.

## Add and use *evtest*

Since the raw events from the Nunchuk are not very convenient to read, let's install an application that will decode the raw input events and display them in a more human readable format: *evtest*.

Enable this package in Buildroot, restart the build, reflash the root filesystem and reboot the system. Now you can use *evtest*:

```
evtest /dev/input/event0
```

## Generate a *defconfig*

Now that our system is already in a good shape, let's make sure its configuration is properly saved and cannot be lost. Go in *menuconfig*, and in the Build options menu. There is an option called *Location to save buildroot config* which indicates where Buildroot will save the *defconfig* file generated by *make savedefconfig*. Adjust this value to `$(TOPDIR)/configs/bootlin_defconfig`.

Then, exit *menuconfig*, and run:

```
make savedefconfig
```

Read the file `configs/bootlin_defconfig` generated in the Buildroot sources. You will see the values for all the options for which we selected a value different from the default. So it's a very good summary of what our system is.

Identify the options related to the following aspects of the system:

- The architecture specification
- The toolchain definition
- The system configuration
- The Linux kernel related configuration
- The selection of packages
- The U-Boot related configuration

## Testing a full rebuild

To make sure that we are able to rebuild our system completely, we'll start a build from scratch. And to learn something new, we'll use *out of tree* build.

To do so, create a build directory anywhere you want, and move inside this directory:

```
mkdir ~/bootlin/buildroot-build/  
cd ~/bootlin/buildroot-build/
```

Now, we will load the `bootlin_defconfig`:

```
make -C ~/bootlin/buildroot/ O=$(pwd) bootlin_defconfig
```

Let's explain a little bit what happens here. By using `-C ~/bootlin/buildroot/`, we in fact tell `make` that the `Makefile` to analyze is not in the current directory, but in the directory passed as the `-C` argument. By passing `O=`, we tell Buildroot where all the output should go: by default it goes in `output/` inside the Buildroot sources, but here we override that with the current directory (`$(pwd)`).

This command will have two main effects:

1. It will load the `bootlin_defconfig` as the current configuration. After running the command, read the file named `.config`. It's much longer than the `defconfig`, because it contains the values for all options.
2. It will create a minimal `Makefile` in this output directory, which will allow us to avoid doing the `make -C ... O=...` dance each time.

Now that this is done, start the build. You can again save the build log:

```
make 2>&1 | tee build.log
```

# New packages in Buildroot

*Objectives:*

- *Create a new package for nInvaders*
- *Understand how to add dependencies*
- *Add patches to nInvaders for Nunchuk support*

## Preparation

After doing a Google search, find the *nInvaders* website and download its source code. Analyze its build system, and conclude which Buildroot package infrastructure is the most appropriate to create a package for *nInvaders*.

## Minimal package

Create a directory for the package in the Buildroot sources, `package/ninvaders`. Create a `Config.in` file with one option to enable this package, and a minimal `ninvaders.mk` file that specifies what is needed just to *download* the package.

For reference, the download URL of the *nInvaders* tarball is <https://sourceforge.net/projects/ninvaders/files/ninvaders/0.1.1/>.

Note: to achieve this, only two variables need to be defined in `.mk` file, plus the call to the appropriate package infrastructure macro.

Now, go to `menuconfig`, enable *nInvaders*, and run `make`. You should see the *nInvaders* tarball being downloaded and extracted. Look in `output/build/` to see if it was properly extracted as expected.

## Make it build!

As you have seen in the previous steps, *nInvaders* uses a simple `Makefile` for its build process. So you'll have to define the `build_commands` variable to trigger the build of *nInvaders*. To do this, you will have to use four variables provided by Buildroot:

- `TARGET_MAKE_ENV`, which should be passed in the environment when calling `make`.
- `MAKE`, which contains the proper name of the `make` tool with potentially some additional parameters to parallelize the build.
- `TARGET_CONFIGURE_OPTS`, which contains the definition of many variables often used by `Makefiles`: `CC`, `CFLAGS`, `LDFLAGS`, etc.
- `@D`, which contains the path to the directory where the *nInvaders* source code was extracted.

When doing Buildroot packages, it is often a good idea to look at how other packages are doing things. Look for example at the `jhead` package, which is going to be fairly similar to our `ninvaders` package.

Once you have written the *nInvaders* build step, it's time to test it. However, if you just run `make` to start the Buildroot build, the `ninvaders` package will not be rebuilt, because it has already been built.

So, let's force Buildroot to rebuild the package by removing its source directory completely:

```
make ninvaders-dirclean
```

And then starting the build:

```
make
```

This time, you should see the `ninvaders 0.1.1 Building` step actually doing something, but quickly failing with a message saying that the `ncurses.h` file could not be found.

Move on to the next section to see how to solve this problem!

## Handling dependencies

The `ncurses.h` header file is missing, because `nInvaders` depends on the `ncurses` library for drawing its interface on a text-based terminal. So we need to add `ncurses` in the dependencies of `nInvaders`. To do this, you need to do two things:

- Express the dependency in the package `Config.in` file. Use a `select` statement to make sure the `ncurses` package option is automatically selected when `ninvaders` is enabled. Check that the `ncurses` package does not have itself some dependencies that need to be propagated up to the `ninvaders` package.
- Express the dependency in the package `.mk` file.

Restart again the build of the package by using `make ninvaders-dirclean all` (which is the same as doing `make ninvaders-dirclean` followed by `make`).

Now the package build fails at link time with messages such as `multiple definition of 'skill_level'; aliens.o:(.bss+0x674): first defined here`.

## Customizing CFLAGS

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 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 that `CFLAGS` is set to `$(TARGET_CFLAGS) -fcommon` in `NINVADERS_BUILD_CMDS`.

Restart the build with `make ninvaders-dirclean all`.

Now the package should build properly! If you look in `output/build/ninvaders-0.1.1/`, you should see a `nInvaders` binary file. Run the file program with `nInvaders` as argument to verify that it is indeed built for ARM.

However, while `nInvaders` has been successfully compiled, it is not installed in our target root filesystem!

## Installing and testing the program

If you study the `nInvaders Makefile`, you can see that there is no provision for installing the program: there is no `install:` rule.

So, in `ninvaders.mk`, you will have to create the *target installation commands*, and simply manually install the `nInvaders` binary. Use the `$(INSTALL)` variable for that. Again, take example on the `jhead` package to know how to achieve that.

Rebuild once again the `ninvaders` package. This time, you should see the `nInvaders` binary in `output/target/usr/bin/`!

Reflash your root filesystem on the SD card and reboot the system. `nInvaders` will not work very well over the serial port, so log to your system through `ssh`, and play `nInvaders` with the keyboard!

Note: if you get the error `Error opening terminal: xterm-256color`. when running `nInvaders`, issue first the command `export TERM=xterm`.

## Support the Nunchuk

Playing with the keyboard is nice, but playing with our Nunchuk would be even nicer! We have written a patch for `nInvaders` that makes this possible.

This patch is available in the lab data directory, under the name `0001-joystick-support.patch`. Copy this patch to the right location so that it gets applied after `nInvaders` is extracted by Buildroot, and before it is built. Rebuild once again the `ninvaders` package. Verify that the patch gets applied at the `ninvaders 0.1.1 Patching step`.



However, this patch relies on the Linux kernel *joystick interface*, that we need to enable. Go to the Linux kernel configuration using `make linux-menuconfig`, and enable `CONFIG_INPUT_JOYDEV`. Exit, and make sure to save your kernel configuration safely using `make linux-update-defconfig`. Restart the overall build by running `make`.

Then reflash your kernel image and root filesystem on the SD card, reboot, and start `nInvaders` in a SSH session. You should now be able to control it using the Nunchuk joystick, and fire with the C button.

## Adding a hash file

To finalize the package, add the missing *hash file*, so that people building this package can be sure they are building the same source code. To know the hash, SourceForge provides this information: go to the `nInvaders` download page, and next to the file name, there is a small information icon that will provide the MD5 and SHA1 hashes. Add both hashes to the hash file.

Home / [ninvaders](#) / 0.1.1 📶

Name ↕	Modified ↕	Size ↕	Downloads / Week ↕
↑ <a href="#">Parent folder</a>			
<a href="#">ninvaders-0.1.1.tar.gz</a>	2003-05-08	31.3 kB	39  
<b>Totals: 1 Item</b>		<b>31.3 kB</b>	<b>39</b>

Click here to see the hashes

Once the *hash file* is added, rebuild the package completely by doing `make ninvaders-dirclean all`.

Look at the build output, and before the `ninvaders 0.1.1 Extracting step`, you should see a message like this:

```
ninvaders-0.1.1.tar.gz: OK (sha1: ....)
ninvaders-0.1.1.tar.gz: OK (md5: ....)
```

## Testing package removal

Now, to experiment with Buildroot, do the following test: disable the `ninvaders` package in `menuconfig` and restart the build doing `make`. Once the build is done (which should be very quick), looked in `output/target/`. Is `nInvaders` still installed? If so, why?

## Sanity checking your package

If you want to verify if your package matches the coding style rules of Buildroot, you can run:

```
make check-package
```

While a successful result doesn't mean your package is perfect, it at least verifies a number of basic requirements.

# Advanced packaging

*Objectives:*

- *Package an application with a mandatory dependency and an optional dependency*
- *Package a library, hosted on GitHub*
- *Use hooks to tweak packages*
- *Add a patch to a package*

## Start packaging application bar

For the purpose of this training, we have created a completely stupid and useless application called `bar`. Its home page is <https://bootlin.com/~thomas/bar/>, from where you can download an archive of the application's source code.

Create an initial package for `bar` in `package/bar`, with the necessary code in `package/bar/bar.mk` and `package/bar/Config.in`. Don't forget `package/bar/bar.hash`. At this point, your `bar.mk` should only define the `<pkg>_VERSION`, `<pkg>_SOURCE` and `<pkg>_SITE` variables, and a call to a package infrastructure.

Enable the `bar` package in your Buildroot configuration, and start the build. It should download `bar`, extract it, and start the configure script. And then it should fail with an error related to `libfoo`. And indeed, as the `README` file available in `bar`'s source code says, it has a mandatory dependency on `libfoo`. So let's move on to the next section, and we'll start packaging `libfoo`.

## Packaging libfoo: initial packaging

According to `bar`'s `README` file, `libfoo` is only available on *GitHub* at <https://github.com/tpetazzoni/libfoo>.

Create an initial package for `libfoo` in `package/libfoo`, with the relevant minimal variables to get `libfoo` downloaded properly. Since it's hosted on *GitHub*, remember to use the `github make` function provided by Buildroot to define `<pkg>_SITE`. To learn more about this function, `grep` for it in the Buildroot tree, or read the Buildroot reference manual.

Also, notice that there is a version tagged `v0.1` in the *GitHub* repository, you should probably use it.

Enable the `libfoo` package and start the build. You should get an error due to the configure script being missing. What can you do about it? Hint: there is one Buildroot variable for *autotools* packages to solve this problem.

`libfoo` should now build fine. Look in `output/target/usr/lib`, the dynamic version of the library should be installed. However, if you look in `output/staging/`, you will see no sign of `libfoo`, neither the library in `output/staging/usr/lib` or the header file in `output/staging/usr/include`. This is an issue because the compiler will only look in `output/staging` for libraries and headers, so we must change our package so that it also installs to the *staging directory*. Adjust your `libfoo.mk` file to achieve this, restart the build of `libfoo`, and make sure that you see `foo.h` in `output/staging/usr/include` and `libfoo.*` in `output/staging/usr/lib`.

Now everything looks good, but there are some more improvements we can do.

## Improvements to libfoo packaging

If you look in `output/target/usr/bin`, you can see a program called `libfoo-example1`. This is just an example program for `libfoo`, it is typically not very useful in a real target system. So we would like this example program to not be installed. To achieve this, add a *post-install target hook* that removes `libfoo-example1`. Rebuild the `libfoo` package and verify that `libfoo-example1` has been properly removed.

Now, if you go in `output/build/libfoo-v0.1`, and run `./configure --help` to see the available options, you should see an option named `--enable-debug-output`, which enables a debugging feature of `libfoo`. Add a sub-option in `package/libfoo/Config.in` to enable the debugging feature, and the corresponding code in `libfoo.mk` to pass `--enable-debug-output` or `--disable-debug-output` when appropriate.

Enable this new option in `menuconfig`, and restart the build of the package. Verify in the build output that `--enable-debug-output` was properly passed as argument to the `configure` script.

Now, the packaging of `libfoo` seems to be alright, so let's get back to our `bar` application.

## Finalize the packaging of bar

So, `bar` was failing to configure because `libfoo` was missing. Now that `libfoo` is available, modify `bar` to add `libfoo` as a dependency. Remember that this needs to be done in two places: `Config.in` file and `bar.mk` file.

Restart the build, and it should succeed! Now you can run the `bar` application on your target, and discover how absolutely useless it is, except for allowing you to learn about Buildroot packaging!

## bar packaging: *libconfig* dependency

But there's some more things we can do to improve `bar`'s packaging. If you go to `output/build/bar-1.0` and run `./configure --help`, you will see that it supports a `--with-libconfig` option. And indeed, `bar`'s README file also mentions `libconfig` as an optional dependency.

So, change `bar.mk` to add *libconfig* as an optional dependency. No need to add a new `Config.in` option for that: just make sure that when *libconfig* is enabled in the Buildroot configuration, `--with-libconfig` is passed to `bar`'s `configure` script, and that *libconfig* is built before `bar`. Also, pass `--without-libconfig` when *libconfig* is not enabled.

Enable `libconfig` in your Buildroot configuration, and restart the build of `bar`. What happens?

It fails to build with messages like `error: unknown type name 'config_t'`. Seems like the author of `bar` messed up and forgot to include the appropriate header file. Let's try to fix this: go to `bar`'s source code in `output/build/bar-1.0` and edit `src/main.c`. Right after the `#if defined(USE_LIBCONFIG)`, add a `#include <libconfig.h>`. Save, and restart the build of `bar`. Now it builds fine!

However, try to rebuild `bar` from scratch by doing `make bar-dirclean all`. The build problem happens again. This is because doing a change directly in `output/build/` might be good for doing a quick test, but not for a permanent solution: everything in `output/` is deleted when doing a `make clean`. So instead of manually changing the package source code, we need to generate a proper patch for it.

There are multiple ways to create patches, but we'll simply use Git to do so. As the `bar` project home page indicates, a Git repository is available on GitHub at <https://github.com/tpetazzoni/bar>.

Start by cloning the Git repository:

```
git clone https://github.com/tpetazzoni/bar.git
```

Once the cloning is done, go inside the `bar` directory, and create a new branch named `buildroot`, which starts the `v1.0` tag (which matches the `bar-1.0.tar.xz` tarball we're using):

```
git branch buildroot v1.0
```

Move to this newly created branch<sup>6</sup>:

```
git checkout buildroot
```

Do the `#include <libconfig.h>` change to `src/main.c`, and commit the result:

```
git commit -a -m "Fix missing <libconfig.h> include"
```

Generate the patch for the last commit (i.e. the one you just created):

```
git format-patch HEAD^
```

and copy the generated `0001-*.patch` file to `package/bar/` in the Buildroot sources.

Now, restart the build with `make bar-dirclean all`, it should build fully successfully!

You can even check that `bar` is linked against `libconfig.so` by doing:

```
./output/host/usr/bin/arm-linux-readelf -d output/target/usr/bin/bar
```

On the target, test `bar`. Then, create a file called `bar.cfg` in the current directory, with the following contents:

```
verbose = "yes"
```

And run `bar` again, and see what difference it makes.

Congratulations, you've finished packaging the most useless application in the world!

## Preparing for the next lab

In preparation for the next lab, we need to do a clean full rebuild, so simply issue:

```
make clean all 2>&1 | tee build.log
```

---

<sup>6</sup>Yes, we can use `git checkout -b` to create the branch and move to it in one command

# Advanced aspects

*Objectives:*

- *Use build time, dependency and filesystem size graphing capabilities*
- *Use licensing report generation, and add licensing information to your own packages*
- *Use BR2\_EXTERNAL*

## Build time graphing

When your embedded Linux system grows, its build time will also grow, so it is often interesting to understand where the build time is spent.

Since we just did a fresh clean rebuild at the end of the previous lab, we can analyze the build time. The raw data has been generated by Buildroot in `output/build/build-time.log`, which contains for each step of each package the start time and end time (in seconds since Epoch).

Now, let's get a better visualization of this raw data:

```
make graph-build
```

Note: you may need to install `python-matplotlib` and `graphviz` on your machine.

The graphs are generated in `output/graphs`:

- `build.hist-build.pdf`, build time of each package, by build order
- `build.hist-duration.pdf`, build time of each package, by build duration
- `build.hist-name.pdf`, build time of each package, by package name
- `build.pie-packages.pdf`, build time of each package, in proportion of the total build time
- `build.pie-steps.pdf`, build time of each step

Explore those graphs, see which packages and steps are taking the biggest amount of time.

Note that when you don't do a clean rebuild, the `build-time.log` file gets appended and appended with all the successful builds, making the resulting graphs unexploitable. So remember to always do a clean full rebuild before looking at the build time graphs.

## Dependency graphing

Another useful tool to analyze the build is graphing dependencies between packages. The dependency graph is generated for your current configuration: depending on the Buildroot configuration, a given package may have different dependencies.

To generate the full dependency graph, do:

```
make graph-depend
```

The graph is also generated in `output/graphs`, under the name `graph-depends.pdf`. On the graph, identify the `bar` and `ninvaders` packages you have created, and look at their dependencies to see if they match your expectations.

Now, let's draw a graph for a much bigger system. To do this, create a completely separate Buildroot output directory:

```
mkdir $HOME/buildroot-bbb-labs/buildroot-output-test-graph/  
cd $HOME/buildroot-bbb-labs/buildroot-output-test-graph/
```

We're going to create a Buildroot configuration, so create a file named `.config` and put the following contents:

```
BR2_TOOLCHAIN_BUILDROOT_GLIBC=y  
BR2_TOOLCHAIN_BUILDROOT_CXX=y  
BR2_PACKAGE_MESA3D=y  
BR2_PACKAGE_MESA3D_GALLIUM_DRIVER_SWRAST=y  
BR2_PACKAGE_MESA3D_OPENGL_EGL=y  
BR2_PACKAGE_MESA3D_OPENGL_ES=y  
BR2_PACKAGE_XORG7=y  
BR2_PACKAGE_XSERVER_XORG_SERVER=y  
BR2_PACKAGE_LIBGTK3=y  
BR2_PACKAGE_WEBKITGTK=y
```

It represents a configuration that builds an internal toolchain, with a X.org graphic server, the Mesa3D OpenGL implementation, the Gtk3 library, and the Webkit Web rendering engine. We're not going to build this configuration, as it would take quite a bit of time, but we will generate the dependency graph for it.

First, let's run `make menuconfig` to expand this minimal configuration into a full configuration:

```
make -C $HOME/buildroot-bbb-labs/buildroot/ O=$(pwd) menuconfig
```

Feel free to explore the configuration at this stage. Now, let's generate the dependency graph:

```
make graph-depends
```

Look at `graphs/graph-depends.pdf` and how complex it is. Now, let's look at the dependencies of one specific package, let's say `libgtk3`:

```
make libgtk3-graph-depends
```

Now, open the graph generated at `graphs/libgtk3-graph-depends.pdf`. As you can see, it is a lot more readable.

Such dependencies graphs are very useful to understand why a package is being built, and help identifying what you could do to reduce the number of packages that are part of the build.

## Filesystem size graphing

Run `make graph-size` and watch the PDF generated at `output/graphs/graph-size.pdf`. You can also look at the CSV files generated in `output/graphs/`.

## Licensing report

Go back to our original build directory, in `$HOME/buildroot-bbb-labs/buildroot/`.

As explained during the lectures, Buildroot has a built-in mechanism to generate a licensing report, describing all the components part of the generated embedded Linux system, and their corresponding licenses.

Let's generate this report for our system:

```
make legal-info
```

In the output, you can see some interesting messages:

```
WARNING: bar: cannot save license (BAR_LICENSE_FILES not defined)
WARNING: libfoo: cannot save license (LIBFOO_LICENSE_FILES not defined)
WARNING: ninvaders: cannot save license (NINVADERS_LICENSE_FILES not defined)
```

So, now update your `ninvaders`, `libfoo` and `bar` packages to include license information. Run again `make legal-info`.

Now, explore `output/legal-info`, look at the `.csv` files, the `.txt` files, and the various directories. Buildroot has gathered for you most of what is needed to help with licensing compliance.

## Use BR2\_EXTERNAL

We should have used `BR2_EXTERNAL` since the beginning of the training, but we were busy learning about so many other things! So it's finally time to use `BR2_EXTERNAL`.

The whole point of `BR2_EXTERNAL` is to allow storing your project-specific packages, configuration files, root filesystem overlay or patches outside of the Buildroot tree itself. It makes it easier to separate the open-source packages from the proprietary ones, and it makes updating Buildroot itself a lot simpler.

So, as recommended in the slides, the goal now is to use `BR2_EXTERNAL` to move away from the main Buildroot tree the following elements:

- The `bar` and `libfoo` packages. We will keep the `ninvaders` package in the Buildroot tree, since it's a publicly available open-source package, so it should be submitted to the official Buildroot rather than kept in a `BR2_EXTERNAL` tree.
- The Linux kernel patch and Linux kernel configuration file.
- The *rootfs overlay*
- The *post-build script*
- The *defconfig*

Your `BR2_EXTERNAL` tree should look like this:

```
+-- board/
| +-- bootlin/
|   +-- beagleboneblack/
|     +-- linux.config
|     +-- post-build.sh
|     +-- patches/
|       +-- linux/
|         +-- 0001-Add-nunchuk-driver.patch
|         +-- 0002-Add-i2c1-and-nunchuk-nodes-in-dts.patch
|       +-- rootfs-overlay/
|       +-- etc
|         +-- network
|         +-- interfaces
|       +-- init.d
|       +-- S30usb gadget
+-- package/
| +-- bar
|   +-- 0001-Fix-missing-libconfig.h-include.patch
|   +-- bar.mk
|   +-- Config.in
| +-- libfoo
```

```
|      +-- libfoo.mk
|      +-- Config.in
+-- configs
|  +-- bootlin_defconfig
+-- Config.in
+-- external.desc
+-- external.mk
```

Now, do a full rebuild using your `BR2_EXTERNAL` tree, and check that your system builds and runs fine!

## Going further

If you have some time left, let's improve our setup to use *genimage*. This way, we will be able to generate a complete SD card image, which we can flash on a SD card, without having to manually create partitions. Follow those steps:

- Change the Buildroot configuration to generate an *ext4* filesystem image
- Take example on `board/stmicroelectronics/common/stm32mp157/genimage.cfg.template` to create your own `board/bootlin/stm32mp1/genimage.cfg`. Keep only the single Device Tree we need for our project.
- Adjust the Buildroot configuration to use the `support/scripts/genimage.sh` script as a *post-image* script, and pass `-c board/bootlin/stm32mp1/genimage.cfg` as *post-image* script arguments. Make sure to enable `BR2_PACKAGE_HOST_GENIMAGE`.

# Application development with Buildroot

*Objectives:*

- *Build and run your own application*
- *Remote debug your application*
- *Create a package for your application*

## Build and run your own application

Let's create your own little application that we will use for demonstration in this lab. Create a folder `$HOME/buildroot-bbb-labs/myapp`, and inside this folder a single C file called `myapp.c` with the following contents:

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

To build this application, we'll use the cross-compiler generated by Buildroot. To make this easy, let's add the Buildroot host directory into our `PATH`:

```
export PATH=$HOME/buildroot-bbb-labs/buildroot/output/host/bin:$PATH
```

Now you can build your application easily:

```
arm-linux-gcc -o myapp myapp.c
```

Copy the `myapp` binary to your target using `scp` (we use the legacy *SCP* protocol, as we haven't installed a *SFTP* server, hence the `-O` option):

```
scp -O myapp root@192.168.42.2:
```

And run the `myapp` application on your target.

Now, let's extend the application a little bit more to use a library, the `libconfig` library we've already used in a previous lab. Change the source code of the application to the one provided in this lab data directory, `myapp.c`.

If you try to build this application with just:

```
arm-linux-gcc -o myapp myapp.c
```

It fails to build because it does not link with `libconfig`. So you can manually do:

```
arm-linux-gcc -o myapp myapp.c -lconfig
```

Since `libconfig.so` is in `output/staging/usr/lib` and the compiler is configured to automatically look in `output/staging` as its `sysroot`, it works fine.

However, there's a better solution: using `pkg-config`. Buildroot has installed a special version of `pkg-config` in `output/host/bin`, which you can query for libraries available for the target. Run:

```
pkg-config --list-all
```

And check you have `libconfig` mentioned. You can query the compiler and linker flags for `libconfig`:

```
pkg-config --cflags --libs libconfig
```

And use that to build your application:

```
arm-linux-gcc -o myapp myapp.c $(pkg-config --cflags --libs libconfig)
```

In the case of `libconfig`, it doesn't simplify a lot because the compiler and linker flags are simple, but for some other libraries, they are more complicated.

Copy the new version of `myapp` to your target, and run it. Create a `myapp.cfg` config file, and run your application again.

## Remote debug your application

Our application is simple and works, but what if you need to debug it? So let's set up remote debugging.

The *ARM* toolchain is provided with a pre-compiled `gdbserver`, so we'll simply use it. Enable the option `BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY`, and then force the re-installation of the toolchain using:

```
make toolchain-external-bootlin-reinstall
```

Reflash your system, or alternatively, just copy `output/target/usr/bin/gdbserver` to the target `/usr/bin/` directory using `scp`.

To do some appropriate debugging, we need to have debugging symbols available. So we need to do two things:

1. Rebuild our application with the `-g` flag.
2. Rebuild the Buildroot system with debugging symbols, so that shared libraries have debugging symbols. However, since we don't want to rebuild the entire Buildroot system now, we'll use a trick and rebuild only the library we need to have the debugging symbols for: `libconfig`. To achieve this, first go to Buildroot `menuconfig`, and in `Build options`, enable `build packages with debugging symbols`. Then, do `make libconfig-dirclean all` to force the rebuild of just `libconfig`.

Now, on your target, start `gdbserver` in multi-process mode, listening on TCP port 2345:

```
gdbserver --multi localhost:2345
```

Back on the host, run the `cross-gdb` with the `myapp` application as argument:

```
arm-linux-gdb myapp
```

We need to tell `gdb` where the libraries can be found:

```
(gdb) set sysroot output/staging
```

And then connect to the target:

```
(gdb) target extended-remote 192.168.42.2:2345
```

Define which program we want to run on the target:

```
(gdb) set remote exec-file myapp
```

Let's put a breakpoint on the main function, and start the program:

```
(gdb) break main
(gdb) run
```

It stops on the first line of the main function, which is the call to `config_init`, implemented by the `libconfig` library. If you do the `gdb` instruction `step`, `gdb` will step into the function, so you can follow what happens. After having done `step` once, you can do `backtrace` to see that you are in the function `config_init` called by `main`:

```
(gdb) backtrace
#0 config_init (config=0xbefffc3c) at libconfig.c:725
#1 0x000106f0 in main () at myapp.c:11
```

Note that if you want `gdbserver` to stop on the target, you need to run the `gdb` command `monitor exit`.

## Create a package for your application

Building manually your own application is not desirable, we obviously want to create a Buildroot package for it. A useful mechanism to package your own applications is to use the `local site method`, which tells Buildroot that the source code of your application is available locally.

Create a new package called `myapp` in your `BR2_EXTERNAL` tree, and by using the `local site method`, make it use directly the `myapp` source code from `$HOME/buildroot-bbb-labs/myapp`. Remember that you can use `$(TOPDIR)` to reference the top-level directory of the Buildroot sources.

For now, directly call `gcc` in the build commands. Of course, if your application becomes more complicated, you should start using a proper build system (`Makefile`, `autotools`, `CMake`, etc.).

When the package builds, you should see as the first step being done that the `myapp` source code gets *rsynced* from `$(HOME)/bootlin/myapp`:

```
>>> myapp custom Syncing from source dir /home/thomas/bootlin/myapp
```

The build should now proceed to the end. Now, make a stupid but visible change to the source code in `myapp.c`.

Restart the build of `myapp` using `make myapp-rebuild`, you will see that Buildroot automatically *rsyncs* again the source code. Then `scp` the file `output/target/usr/bin/myapp` to `192.168.42.2:/usr/bin` and run `myapp` again on the target.

As you can see you can now develop your applications and libraries, using your normal version control system and relying on Buildroot to do all the configure, build and install steps for you.