Boot time optimization Training

Practical Labs

bootlin
https://bootlin.com

June 1, 2021

About this document

Updates to this document can be found on https://bootlin.com/doc/training/boot-time.

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

Goals

Implement a live camera system and optimize its boot time.

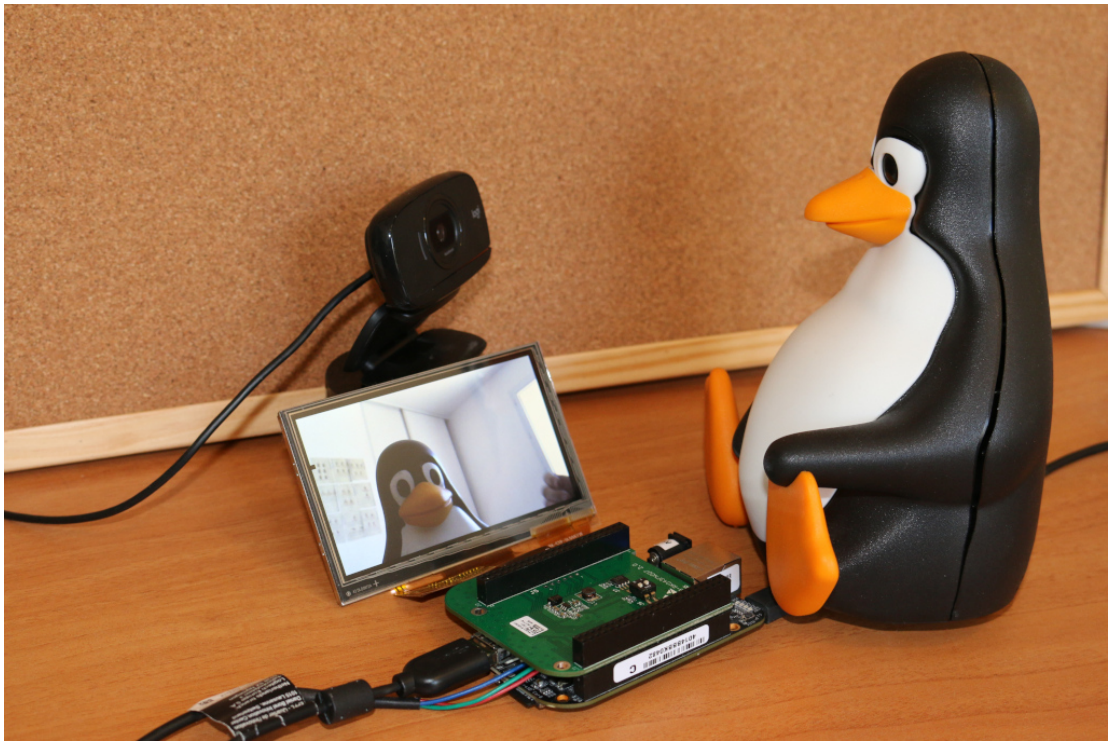
Here's a description of the system that we are going to build and optimize in terms of boot time:

Hardware:

- Main board: Beagle Bone Black (Regular or Wireless), with an ARM Cortex A8 SoC (AM335x from Texas Instruments).
- Extended by a 4 inch LCD cape
- Connected to a USB webcam

Software:

- Bootloader: U-Boot
- Operating system: Linux
- User space: ffmpeg video player
- Build system: Buildroot
- Functionality: as soon as the system has booted, display the video from the USB webcam.



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/boot-time/boot-time-labs.tar.xz
$ tar xvf boot-time-labs.tar.xz
```

Lab data are now available in an `boot-time-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.

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

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

the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

- 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: `$ chown -R myuser.myuser linux/`

Downloading bootloader, kernel and Buildroot source code

Save time and start fetching the source code that you will need during these labs

Installing git packages

We are going to access bootloader, kernel and Buildroot sources through their `git` repositories, which will allow us to track any changes that we make to the source code of these projects.

Depending on how fast your network connection is (and how many users share it), fetching such sources is likely to take a significant amount of time. That's why we're starting such downloads now.

So, let's install the below packages:

```
sudo apt install git gitk git-email
```

Git configuration

After installing `git` on a new machine, the first thing to do is to let `git` know about your name and e-mail address:

```
git config --global user.name 'My Name'
git config --global user.email me@mydomain.net
```

Such information will be stored in commits. It is important to configure it properly in case we need to generate and send patches.

Cloning the Buildroot source tree

Go to the `$HOME/boot-time-labs/rootfs` directory.

```
git clone git://git.buildroot.net/buildroot
```

or if you are behind a proxy blocking `git`, here's a slower alternative:

```
git clone https://git.buildroot.net/buildroot
```

If the connection to the Internet turns out to be not fast enough, your instructor can give you a USB flash drive with a `tar` archive of a recently cloned tree:

```
tar xf buildroot-git.tar.xz
cd buildroot
git checkout master
git pull
```

We will select a particular release tag later. Let's move on to the next source repository.

Cloning the U-Boot source tree

Create the `$HOME/boot-time-labs/bootloader` directory and go into it.

```
git clone git://git.denx.de/u-boot.git
```

or

```
git clone https://git.denx.de/u-boot.git
```

Similarly, your instructor can give you a pre-downloaded archive if needed.

Cloning the mainline Linux tree

Go to the `$HOME/boot-time-labs/kernel` directory.

This represents the biggest amount of sources to download, actually more than 1 GB of data! Again, you can use a pre-downloaded archive if that turns out to be to much for your actual connection.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

or

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Accessing stable Linux releases

Go to the `linux` source directory.

Having the Linux kernel development sources is great, but when you are creating products, you prefer to avoid working with a target that moves every day.

That's why we need to use the *stable* releases of the Linux kernel.

Fortunately, with `git`, you won't have to clone an entire source tree again. All you need to do is add a reference to a *remote* tree, and fetch only the commits which are specific to that remote tree.

```
git remote add stable git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
git fetch stable
```

As this still represents many git objects to download (about 300 MiB when 5.4 was the latest version), if you are using an already downloaded git tree, your instructor will probably have fetched the *stable* branch ahead of time for you too. You can check by running:

```
git branch -a
```

We will choose a particular stable version in the next labs.

Now, let's continue the lectures. This will leave time for the commands that you typed to complete their execution (if needed).

Board setup

Objective: setup communication with the board and configure the bootloader.

After this lab, you will be able to:

- Access the board through its serial line.
- Check the stock bootloader
- Attach the 4.3" LCD cape

Getting familiar with the board

Take some time to read about the board features and connectors:

- If you have the original BeagleBone Black:
  https://www.elinux.org/Beagleboard:BeagleBoneBlack
- If you have the newer BeagleBone Black Wireless:
  https://beagleboard.org/black-wireless in addition to the above URL.

Don't hesitate to share your questions with the instructor.

Download technical documentation

We are going to download documents which we will need during our practical labs.

The main document to download is the BeagleBone Black System Reference Manual found at https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true.

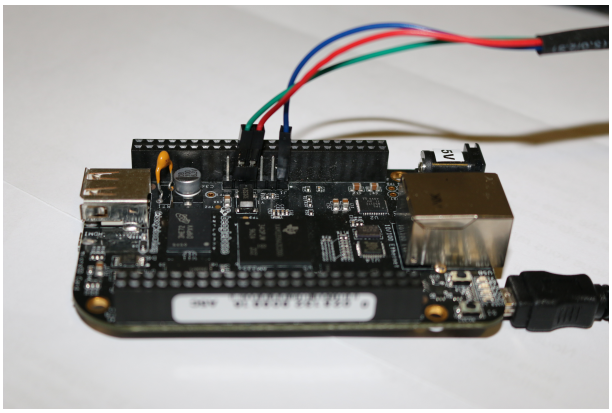
Even if you have the BeagleBoneBlack Wireless board, this is the ultimate reference about the board, in particular for the pinout and possible configurations of the P8 and P9 headers, and more generally for most devices which are the same in both boards. You don't have to start reading this document now but you will need it during the practical labs.

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.

[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.



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, you have to *completely log out* from your session and log in again (no need to reboot). 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.

It is now time to power up your board by plugging in the mini-USB (BeagleBone Black case) or micro-USB (BeagleBone Black Wireless case) cable supplied by your instructor (with your PC or a USB power supply at the other end of the cable).

See what messages you get on the serial line. You should see U-boot start on the serial line.

Bootloader interaction

Reset your board. Press the space bar in the `picocom` terminal to stop the U-boot countdown. You should then see the U-Boot prompt:

```
=>
```

This step was just to check that the serial line was connected properly. In a later lab, we will replace the existing bootloader by a version that we compiled ourselves.

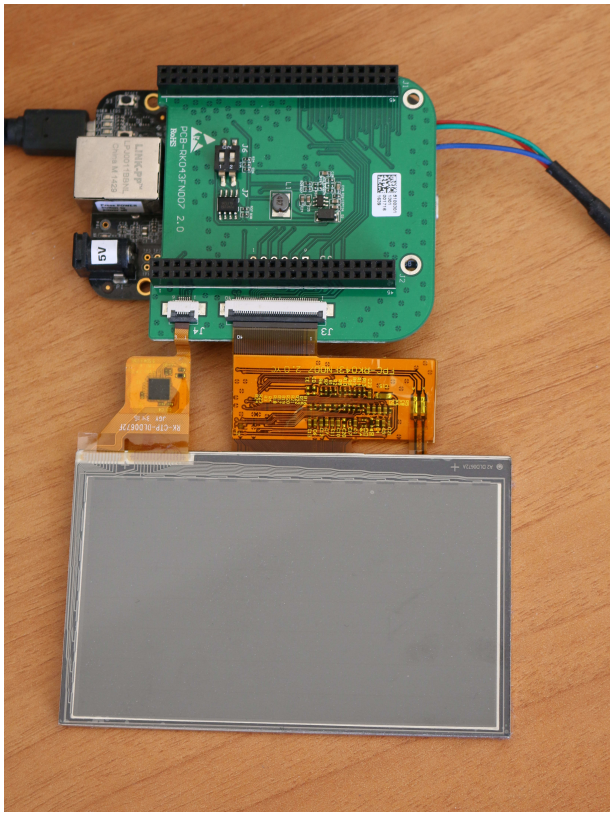
Attach the LCD cape

Switch off the board first, by pressing the POWER button until all the LEDs go off[3].

Now that we have successfully tested serial console, we are ready to attach the 4.3 LCD cape provided by your instructor.

Note that if you bought your own Beagle Bone Black board, you will have to **gently** bend the serial headers using pliers, otherwise the serial cable won't fit between the board and the cape.

Now, you can connect the LCD cape to your Bone Black board. There is just one way that fits, given the space taken by the RJ45 connector:



[3] That's strongly recommended by the board maker, to avoid hardware damage that can happen if the board is abruptly switched off.

Build the system

Objective: compile the root filesystem.

After this lab, you will have a ready to use root filesystem to boot your system with, including a video player application.

We haven't compiled the bootloader and kernel for our board yet, but since this part can take a long time (especially compiling the cross-compiling toolchain), let's start it now, while we are still fetching kernel sources or going through lectures.

Setup

As specified in the Buildroot manual[4], 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 python unzip rsync wget libncurses-dev
```

You will later also find that you also need extra packages:

```
sudo apt install bison flex
```

Choosing a Buildroot release

Buildroot is one of the best tools for building a custom root filesystem for a dedicated embedded system with a fixed set of features, typically like the one we're trying to build.

Go to the /code /boot-time-labs/rootfs/buildroot/ directory.

We will use the latest revision of the `2021.02` release, which is one of Buildroot's long term releases:

```
git tag | grep 2021.02
git checkout 2021.02
```

Configuring Buildroot

To minimize external dependencies and maximize flexibility, we will ask Buildroot to generate its own toolchain. This can be better than using external toolchains, as we have the ability to tweak toolchain settings in a fine way if needed.

Start the Buildroot configuration utility:

```
make menuconfig
```

- `Target Options` menu
  - `Target Architecture`: select `ARM (little endian)`
  - `Target Architecture Variant`: select `cortex-A8`

[4] https://buildroot.org/downloads/manual/manual.html#requirement-mandatory

  - 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 will later try the `Thumb-2` instruction set for slightly more compact code).

- `System configuration` menu
  - Unselect `remount root filesystem read-write during boot`. This way, we will keep the root filesystem in read-only mode. When we make tests and reboot the system multiple times, this avoids filesystem recovery with approximately takes 4 seconds and adds jitter to our measurements.

- `Toolchain` menu
  - `GCC compiler Version`: select `gcc 10.x`. This allows to have the latest version of the compiler (at the time of this writing), and the best available optimizations.
  - Keep all the other settings unmodified. We will get a toolchain with the *uClibc* library this way.

- `Target packages` menu
  - In `Audio and video applications`, select `ffmpeg` and inside the `ffmpeg` submenu, add the below options:
    * Select `Build libswscale`

- For the moment, all the remaining default settings are fine for us.

However, we need to do one thing to customize the root filesystem: we need to add a script that will automatically start the `ffmpeg` video player.

To do so, we will use Buildroot's *Root filesystem overlay* capability, which allows to drop ready-made files into the final root filesystem archive[5].

To begin with, let's start by creating a specific directory to store our Buildroot customizations for our project.

```
mkdir board/beaglecam
```

And in this directory, let's create a directory for root filesystem overlays:

```
mkdir board/beaglecam/rootfs-overlay
```

Now, let's copy a script that we're providing you to `etc/init.d`:

```
mkdir -p board/beaglecam/rootfs-overlay/etc/init.d/
cp ~/boot-time-labs/rootfs/data/S50playvideo board/beaglecam/rootfs-overlay/etc/init.d/
```

We can now run `make menuconfig` again and in `System configuration`, add `board/beaglecam/rootfs-overlay` to `Root filesystem overlay directories`.

[5] See https://buildroot.org/downloads/manual/manual.html#customize for details.

Running Buildroot

We are now ready to execute Buildroot:

```
make
```

Enjoy, and be patient, as building a cross-compiling toolchain takes time!

Build the bootloader

Objective: compile and install the bootloader.

After this lab, you will be able to compile U-Boot for your target platform and run it from a micro SD card provided by your instructor.

Setup

Go to the `~/boot-time-labs/bootloader/u-boot/` directory.

Let's use the 2021.04 version:

```
git checkout v2021.04
```

Compiling environment

As the previous Buildroot lab is probably not over yet, we will use the cross-compiling toolchain provided by Ubuntu:

```
sudo apt install gcc-arm-linux-gnueabihf
export CROSS_COMPILE=arm-linux-gnueabihf-
```

Configuring U-Boot

Let's use a ready-made U-Boot configuration for our hardware.

The `configs/` directory normally contains one or several configuration file(s) for each supported board. However, in our case, we are going to use a more generic configuration file that supports all Beaglebone Black variants and, according to the name, the AM335x EVM board too:

```
make am335x_evm_defconfig
```

Compiling U-Boot

Just run:

```
make
```

or, to compile faster:

```
make -j 8
```

This runs 8 compiler jobs in parallel (for example if you have 4 CPU cores on your workstation... using more jobs than cores guarantees that the CPUs and I/Os are always fully loaded, for optimum performance.

At the end, you have `MLO` and `u-boot.img` files that we will put on a micro SD card for booting.

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`), the Device Tree (`am335x-boneblack.dtb`) and a special U-Boot script for the boot.
- 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!**

If your SD card is `/dev/mmcblk0`, then the partitions inside the SD card are named `/dev/mmcblk0p1`, `/dev/mmcblk0p2`, etc. If your SD card is `/dev/sdc`, then the partitions inside are named `/dev/sdc1`, `/dev/sdc2`, 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`. Use `sdc` or `sdb` instead of `mmcblk0` if needed.
3. Create the two partitions.
   - Start the `cfdisk` tool for that:
     `sudo cfdisk /dev/mmcblk0`
   - Chose 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 `cfdisk`
4. Format the first partition as a *FAT32* filesystem:
   `sudo mkfs.vfat -F 32 -n boot /dev/mmcblk0p1`. Use `sdc1` or `sdb1` instead of `mmcblk0p1` if needed.
5. Format the second partition as an *ext4* filesystem:
   `sudo mkfs.ext4 -L rootfs -E nodiscard /dev/mmcblk0p2`. Use `sdc2` or `sdb2` instead of `mmcblk0p2` if needed.
   - `-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`.

Booting your new bootloader

On a board in a normal state, there should be a bootloader on the on-board MMC (eMMC) storage, and this will prevent you from using a bootloader on an external SD card (unless you hold the `USER` button while powering up your board, which is just suitable for exceptional needs).

Therefore, to override this behavior and use the external SD card, instead, let's wipe out the `MLO` file on the eMMC.

Power up or reset your board, and in the U-Boot prompt, run:

```
fatls mmc 1
```

You should see the `MLO` file in the list of files. Let's remove it by issuing the below command (erasing the first 1MiB on the eMMC):

```
mmc dev 1
mmc erase 0 100000
```

You can now see that `fatls mmc 1` no longer sees any file.

If your board doesn't boot at all, please fix it first using our instructions on https://raw.githubusercontent.com/bootlin/training-materials/master/lab-data/common/bootloader/beaglebone-black/README.txt.

Now, copy your newly compiled `MLO` and `u-boot.img` files to the SD card's `boot` partition, and after cleanly unmounting this partition, insert the SD card into the board and reset it.

You should now see your board booting with your own `MLO` and U-Boot binaries (the versions and compiled dates are shown in the console).

Build the kernel and boot the system

Objective: configure, compile and install the kernel, install the root file system and see the full system in action!

At the end of the lab, you'll have your system completely up and running.

Setup

Go to the `~/boot-time-labs/kernel/linux/` directory and install a package to we will need to compile the Linux kernel:

```
sudo apt install libssl-dev
```

First let's get the latest updates to the remote `stable` tree (that's needed if you started from a ready made archive of the Linux git repository):

```
git fetch stable
```

First, let's get the list of branches on our `stable` remote tree:

```
git branch -a
```

As we will do our labs with the Linux 5.11 stable branch, the remote branch we are interested in is `remotes/stable/linux-5.11.y`.

First, open the `Makefile` file just to check the Linux kernel version that you currently have.

Now, let's create a local branch starting from that remote branch:

```
git checkout -b 5.11-beaglecam stable/linux-5.11.y
```

This local branch will allow us to keep our modifications to the Linux kernel to support the 4.3" LCD cape that we're using.

Open `Makefile` again and make sure you now have a 5.11.<n> version.

Compiling environment

You need the same `PATH` and `CROSS_COMPILE` environment variables as when you compiled U-Boot, plus the `ARCH` one that corresponds to the target architecture.

```
export CROSS_COMPILE=arm-linux-gnueabihf-
export ARCH=arm
```

Adding support for the 4.3" LCD cape

To support using the 4.3" LCD cape, all we need to do is declare and configure the devices on this cape. This is typically done by customizing the board's *Device Tree* or by adding a *Device Tree Overlay*.

So, to avoid messing with the standard DTS for our board, let's use a such a customized device tree through a separate file:

```
cp ~/boot-time-labs/kernel/data/am335x-boneblack-lcd43.dts arch/arm/boot/dts/
```

You now have to modify `arch/arm/boot/dts/Makefile` so that the new DTS file gets compiled too.

There is another thing you need to do to make the Linux framebuffer work on the LCD display, avoiding conflicts with HDMI:

Open `arch/arm/boot/dts/am335x-boneblack-common.dtsi` and remove:

- The `port` subnode under the `lcdc` node.
- The `ports` subnode under the `tda19988` node.

Configuring the Linux kernel

First, lets pick the default kernel configuration for boards with a TI OMAP or AMxxxx SoC:

```
make help | grep omap
```

What we need is the configuration for OMAP2 and later SoCs:

```
make omap2plus_defconfig
```

Let's run `make menuconfig` or `make xconfig` and select the below options. Use the search capability of such configuration interfaces to find the corresponding parameters (remove `CONFIG_` when you search.

To enable support for the framebuffer and the PWM backlight:

- `CONFIG_PWM_TIEHRPWM=y`
- `CONFIG_FB_SIMPLE=y`
- `CONFIG_BACKLIGHT_PWM=y`
- `CONFIG_DRM=y`
- `CONFIG_DRM_TILCDC=y`
- `CONFIG_DRM_TI_TFP410=y`

For USB support:

- `CONFIG_USB=y`
- `CONFIG_USB_MUSB_HDRC=y`
- `CONFIG_USB_MUSB_DSPS=y`
- `CONFIG_MUSB_PIO_ONLY=y`
- `CONFIG_USB_GADGET=y`
  You may wonder why we need this while we're not using USB gadget at all. At least in the 5.1 kernel, this setting was required to get some of the USB host functionality we needed.
- `CONFIG_NOP_USB_XCEIV=y`
- `CONFIG_AM335X_PHY_USB=y`
- `CONFIG_USB_GPIO_VBUS=y`

- CONFIG_USB_GADGET_VBUS_DRAW=500
- CONFIG_USB_CONFIGFS_F_UVC=y

For the webcam

- CONFIG_MEDIA_SUPPORT=y
- CONFIG_MEDIA_USB_SUPPORT=y
- CONFIG_VIDEO_DEV=y
- CONFIG_USB_VIDEO_CLASS=y

For your convenience, of if you screw up your settings in a later lab, you can also use a reference configuration file found in `boot-time-labs/kernel/data`.

Compiling the kernel

To compile the device tree, just run:

```
make dtbs
```

To compile the kernel, just run:

```
make -j 8 zImage
```

Note that the default `make` target would have worked too, but with just `zImage`, we avoid compiling many modules that are configured in the default configuration. This saves quite a lot of time!

At the end, copy the kernel binary and DTB to the SD card's boot partition:

```
cp arch/arm/boot/zImage /media/$USER/boot/
cp arch/arm/boot/dts/am335x-boneblack-lcd43.dtb /media/$USER/boot/dtb
```

Installing the root filesystem

We are also ready to install the root filesystem. Still with the SD card connected to your workstation:

```
cd ~/boot-time-labs/rootfs/buildroot
rm -rf /media/$USER/rootfs/*
sudo tar -C /media/$USER/rootfs/ -xf output/images/rootfs.tar
sudo umount /media/$USER/rootfs
sudo umount /media/$USER/boot
```

Then insert the SD card in the board's slot.

Bootloader configuration

Back to the serial console for your board, let's define the default boot sequence, to load the kernel and DTB from the external SD card:

```
setenv bootcmd 'fatload mmc 0:1 81000000 zImage; fatload mmc 0:1 82000000 dtb; bootz 81000000 - 82000000'
```

The last thing to do is to define the kernel command line:

```
setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk0p2 rootwait ro
```

- `rootwait` waits for the root device to be ready before attempting to mount it. You may have a kernel panic otherwise.
- `ro` mounts the root filesystem in read-only mode. If this is possible, this is quite important to avoid random filesystem checks at boot time, depending on how the system was shut down, switched off or rebooted. Such filesystem checks can add a lot of jitter from one boot to another, making boot time measurements unpredicable and difficult to reproduce.

Last but not least, save your changes:

```
saveenv
```

Testing time!

First, connect the USB webcam provided by your instructor, and point it to an interesting direction ;)

Then, reset your board or power it on, and see it work as expected. If you don't get what you expected, check your serial console for errors, and if you're stuck, show your system to your instructor.

Measure boot time - Software solution

Objective: measure boot time with software only solutions

During this lab, we will use techniques to measure boot time using only software solutions.

Timing messages on the serial console

Let's use `grabserial` to time all the messages received in the serial console, from the first stage bootloader to launching the final application:

```
sudo apt install grabserial
```

We are now ready to time the messages in the serial console. First, exit from Picocom (`[Ctrl][a] [Ctrl][x]`). Then, power off your board, remove its USB power supply and run:

```
grabserial -d /dev/ttyUSB0 -t -e 30
```

- `-t` Displays the time when the first character of each line was received.
- `-e` Specifies the **e**nd time when `grabserial` will exit.

Now, insert the USB power cable and see the console messages with their timing information:

```
0.000000 0.000000]
[2.596791 2.596791] U-Boot SPL 2021.04 (Apr 05 2021 - 16:02:48 +0100)
[2.601680 0.004889] Trying to boot from MMC1
[3.013119 0.411439]
[3.013584 0.000465]
[3.013835 0.000251] U-Boot 2021.04 (Apr 05 2021 - 16:02:48 +0100)
...
```

As you can see, the time to start U-Boot SPL can be neglected. We can use the `U-Boot SPL` string as a reference point for timing. This way, we don't have to power off the board every time we wish to make a measurement. Resetting the board will be sufficient.

So, let's run `grabserial` again:

```
grabserial -d /dev/ttyUSB0 -m "U-Boot SPL" -t -e 30
```

Timing the execution of the application

Bootlin prepared a patch to `ffmpeg` to issue a message in its logs after decoding and displaying the first frame. Let's instruct Buildroot to apply it!

```
mkdir -p board/beaglecam/patches/ffmpeg/
cp ../data/0001-ffmpeg-log-notification-after-first-frame.patch \
   board/beaglecam/patches/ffmpeg/
```


Then, tell Buildroot to look for patches in this directory, by adding `board/beaglecam/patches` to the `BR2_GLOBAL_PATCH_DIR` configuration setting (through `make menuconfig` or by directly editing the `.config` file.

Then, rebuild `ffmpeg`:

```
make ffmpeg-dirclean
make
```

Note that `make ffmpeg-rebuild` wouldn't be sufficient to apply a newly added patch.

Let's add something else before updating the root filesystem image...

Timing the launching of the application

To measure the time the application takes to load and execute, it's also very useful to time the instant when the application is started.

So, let's also modify `board/beaglecam/rootfs-overlay/etc/init.d/S50playvideo` to add the below line before running `ffmpeg`:

```
echo "Starting ffmpeg"
```

Now we can update the root filesystem image:

```
make
```

After reflashing the SD card, reset the board and check that you are getting the new message. You can now measure the time between starting `ffmpeg` and finishing processing the first frame.

Thanks to the last message, we can now stop `grabserial` when it's received, replacing the `-e` argument:

```
grabserial -d /dev/ttyUSB0 -m "U-Boot SPL" -t -e 30 -q "First frame decoded"
```

We now have a complete measurement of the initial boot time of the system. We are going to write down the key figures, but it's always useful to keep the full log each of our experiments. This is very often useful to compare experiments, double check some measurements that are surprising and could have been copied in a wrong way, and investigate some differences between two different runs.

So, create the $HOME/boot-time-labs/logs directory and copy and paste the `grabserial` output to the `initial.log` file in this directory.

Initial measurements

Now, take your calculator and fill the below table with the results from your experiments:

Step	Duration	Description
U-Boot SPL		Between `U-Boot SPL 2021.04` and `U-Boot 2021.04`
U-Boot		Between `U-Boot 2021.04` and `Starting kernel`
Kernel		Between `Starting kernel` and `Run /sbin/init`
Init scripts		Between `Run /sbin/init` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total		

Measure boot time - Hardware solution

Objective: measure boot time with hardware

During this lab, we will use a hardware technique to measure boot time, from cold boot (or reset) to the instant when the first frame has been decoded.

Arduino setup

Take the Arduino Nano board provided by your instructor. Connect it in the middle of the breadboard provided too, so that you can connect wires to both sides of the Arduino.

Download the 1.8.13 version of the Arduino IDE from `https://www.arduino.cc/` (don't use the Arduino package in Ubuntu 20.04, as it has issues connecting to the serial port, even with root privileges, while the official version works without any problem). Extract the archive in `/usr/local/`.

Use the provided USB cable to connect the Arduino to your PC, and start the IDE:

```
/usr/local/arduino-1.8.13/arduino
```

Now, configure the IDE for your Arduino:

- In `Tools`, `Board`, select `Arduino Nano`
- In `Tools`, `Processor`, select `ATmega328p` (or `ATmega328p old bootloader` if you have a Nano clone)
- In `Tools`, `Port`, select `ttyUSB1` (or `ttyUSB0` if the serial line for your Bone Black board is no longer connected.

Now are now ready to use your Arduino board:

- Go to `Files`, `Examples`, `01. Basics` and select `Blink`. This program allows to blink the LED on the Nano.
- Press the `Upload` butter and you should see the *sketch* work (that's how the Arduino community call their programs).
- You can now unplug the Arduino and plug it back. The same program will be started automatically. Loading a program is just necessary once.

7-segment display setup

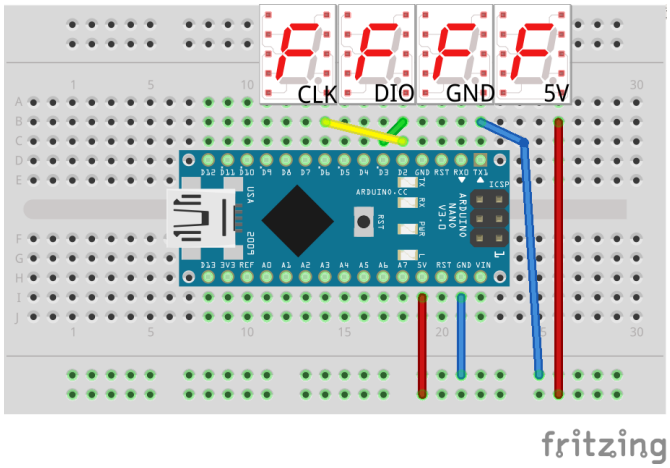
We are going to use a 7-segment display to display time elapsed since the last time the Beagle Bone Black board was last reset.

Take the TM1637 module provided by your instructor, and insert its pins into the breadboard at a convenient location.

Now, using breadboard wires, connect the `GND` pin of the Arduino to one of the blue rails of the breadboard, then to the `GND` pin of the 7-segment module. Please use blue or black wires!

Similarly, connect the `5V` pin of the Arduino to the red rail of the breadboard, then to the `5V` pin of the module. Using red or orange wires is recommended too.

Then, you can connect the Arduino `D2` pin to the `CLK` pin of the module, and the `D3` pin to the `DIO` pin of the module:



Now, let's test your wiring by loading the sketch in `~/boot-time-labs/arduino/test-7segment/`. Upload and try to run it.

Oops, a library is missing. You could have retrieved it through the IDE's library manager (`Tools`, `Manage Libraries`), but in this case, we absolutely need its latest version. So, go to https://github.com/Seeed-Studio/Grove_4Digital_Display, download a zip file and extract this archive into `~/Arduino/libraries/`. Rename `Grove_4Digital_Display-master` to `Grove_4Digital_Display` (removing the `-master` suffix added by GitHub, and you should be ready to go.

Configuring Bone Black pins as GPIOs

Our goal is to measure boot time with the Arduino system, in a way that is as accurate as hardware can be, and without having to rely on a slow device which is the serial console.

Finding a good start signal

A first possibility would be to watch the 3.3V VDD pins of the Bone Black board and start counting when they go high when the board is powered on. However, it would be cumbersome to have to power off the board each time we wish to make a measurement.

A second possibility is to watch the state of the `RESET` signal. When this pin goes from high to low, and back to high, it means that the board starts booting. That's a good time to start counting, and doing it after each reset is a convenient solution.

Look at the Bone Black System Reference and find which pin on the P8 or P9 headers is used to expose the `SYS_RESET` line.

Finding a free pin for the stop signal

Unfortunately, the schematics for the 4.3" LCD cape from Element 14 are currently unavailable. The community is looking for them. This would have been useful to find pins that are not used by this cape.

Fortunately, by reading the Board's Reference Manual about the `P8` and `P9` headers, you can guess what GPIOs could be available from the set of possible signals.

If you look for `Bootlin` in the Device Tree Source we provided, you can see in the pin definitions sections that we selected pin 13 from the P9 headers:

```
/* Bootlin boot time labs: use idle pin as custom GPIO on P9_13 */
AM33XX_PADCONF(AM335X_PIN_GPMC_WPN, PIN_OUTPUT, MUX_MODE7)
```

If you look at the `Expansion Header P9 Pinout` table in the Board's Reference Manual, you will see that `MODE7` allows to get GPIO bank 0 / number 31 on P9's pin 13.

Back to the pin for `SYS_RESET`, there is nothing to configure to get it. It's the only line connected to the pin.

Pull up or pull down?

We need to control the state of the pins we watch when they are not driven. That's particularly important for the `SYS_RESET` signal, as if it's left floating, it can reset the board or prevent it from booting at any time.

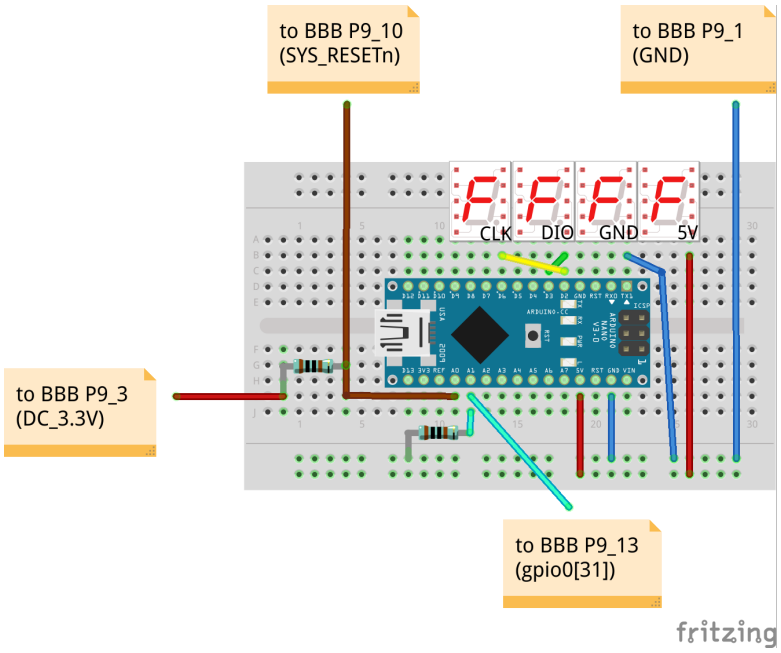
The ATmega328p CPU pins could be configured as pull-up (with internal pull-up resistors), but we are not allowed to have a 5V level connected to our board, as the Beagle Bone Black is **not 5V tolerant**, as explicited in its manual.

So, we will use a pull-up resistor to keep it at 3.3V when not driven, connected to a 3.3V voltage rail from the Beagle Bone Black.

For our custom GPIO pin, we will keep it low by default, using a pull-down resistor (the ATmega328p CPU pins don't have internal pull-down resistors), and drive it high, after displaying the first video frame.

Then which Arduino pins to connect to? As the Beagle Bone Black board has a 3.3V voltage level, it's best to use the Arduino's Analog pins to measure the voltage driven by the Bone Black with precision. We will measure a small integer value for 0V, and about 700 (out of a 1024 maximum) for 3.3V.

So, let's use Arduino pin `A0` for reset, and pin `A1` for the custom GPIO, and connect two 1 Kohm resistors provided by your instructor:



Last but not least, don't forget to connect the ground level on the Arduino board to the one on the Beagle Bone Black (pins `1` or `2` on either `P8` or `P9` connectors).

Making ffmpeg drive the custom GPIO

Once we know how to modify `ffmpeg` to write to its output after processing the first frame, it's easy to add code that configures our custom GPIO line and drives it.

We are going to use the GPIO Sysfs interface (documented on `admin-guide/gpio/sysfs`). This interface is now deprecated for several reasons, but it's convenient to use for simple needs.

The first thing to do is to find which number Linux will use to represent our custom GPIO. To do this, check that your kernel was compiled with `CONFIG_DEBUG_FS`. Then mount the DebugFS filesystem and access GPIO information as follows:

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/gpio
```

Here's what you should get:

```
...
 gpio-124 (ethernet]           )
 gpio-125 (ethernet]           )
 gpio-126 (P9_11 [uart4_rxd]   )
 gpio-127 (P9_13 [uart4_txd]   )
```

So, our custom GPIO in number `127`. Such descriptions actually come from the `arch/arm/boot/dts/am335x-boneblack.dts` file.

Here's example shell code to drive this GPIO to a high level, using the legacy interface:

```
cd /sys/class/gpio
echo 127 > export
```

A `gpio127` file appears. You can then configure the directory and write a value:


```
echo out > gpio127/direction
echo 1 > gpio127/value
```

If you have a multimeter, you could even check by yourself that this works.

Now, modify your Buildroot customizations so that you use the patch in `boot-time-labs/rootfs/data/0001-ffmpeg-first-frame-completion-log-and-gpio.patch` instead of the previously used one (see how this page does the same thing as above).

Rebuild and reflash your root filesystem.

Final version of the Arduino program

We should now be able to test our full system. Loading the sketch in `~/boot-time-labs/arduino/stopwatch/`. Upload and run it, and then press the `RESET` button on your board.

If things don't work as expected, you can also open the Arduino Serial Monitor (in the `Tools` menu), and read the pins values that displayed there by the program.

Once the system works, is it in line with the software measurements? Is it slightly more pessimistic as it's supposed to start counting time a few cycles earlier?

Toolchain optimization

Get the best cross-compiling toolchain for your application and system

The goal of this lab is to find the best toolchain for your application, in terms of performance and code size. Smaller code can be faster to load, and save time when using in an initramfs (when the whole filesystem is loaded at once in RAM).

In this lab, we will see how to test an alternative toolchain, measuring:

- Application execution time
- Application and total filesystem size

Measuring application execution time

At this stage, measuring the total system boot time is not accurate enough. We need to time the execution of the application more precisely.

Hence, we will call the `ffmpeg` player directly from the command line and temporarily apply a patch that will make `ffmpeg` exit after processing the first frame.

To achieve this:

- Temporarily replace the `0001-ffmpeg-log-notification-after-first-frame.patch` patch by `0001-ffmpeg-exit-after-first-frame.patch` found in `~/boot-time-labs/rootfs/data/`.
- Edit the `S50playvideo` script to comment out the line starting `ffmpeg` automatically. We don't want `ffmpeg` to run automatically at this stage, because otherwise when won't be able to time its first execution through the `time` command and then compare with the second time it is executed, to have an measurement of the `ffmpeg` loading time.

Now, rebuild the root filesystem:

```
make ffmpeg-dirclean
make
```

Let's measure the total root filesystem size and the size of the `ffmpeg` executable:

```
ls -l output/images/rootfs.tar
tar tvf output/images/rootfs.tar ./usr/bin/ffmpeg
```

Write these two numbers in the first row of the below table:

	Total rootfs size	`/usr/bin/ffmpeg` size
ARM toolchain		
Thumb2 toolchain		
Musl toolchain		

Reflash the root filesystem on the SD card and reboot your board. On the serial console, log in and run the video player through the `time` command (copying and pasting the command from these instructions or from the `/etc/init.d/S50playvideo` file:


```
time ffmpeg -f video4linux2 -video_size 544x288 -input_format mjpeg \
-i /dev/video0 -pix_fmt rgb565le -f fbdev /dev/fb0
```

Note that we removed `-t 10`. It's no longer needed to stop after 10 seconds as we stop after decoding the first frame.

Write down your first results in the below table (`total1`, `user1` and `sys1` columns):

	total1	user1	sys1	total2	user2	sys2	total3	user3	sys3
ARM toolchain									
Thumb2 toolchain									
Musl toolchain									

After the first run, the program and its shared libraries are now in the file cache. Lets run the command two more times and write down (in the `2` and `3` columns) how fast it can run in this ideal case.

Switching to a Thumb2 toolchain

Now, let's modify the Buildroot toolchain, so that it generates *Thumb2* code (instead of *ARM*) by default.

Before we switch, let's make a backup of the `buildroot` directory, in case results are disappointing and we wish to revert the changes without having to go through a full build again. In our particular case, we will start the next lab using this backup, waiting for the new build to be complete.

In case you didn't know, a time and disk space efficient way to do this is by using the `cp -al` command, which uses hard links instead of making new copies of each file:

```
cd ..
cp -al buildroot buildroot-arm
```

You can then check that the files correspond to the same inode:

```
ls -i buildroot/Makefile buildroot-arm/Makefile
```

Back to the `buildroot` directory, run `make menuconfig`, and in `Target options`, set `ARM instruction set` to `Thumb2`.

Save the changes and run the full toolchain and root filesystem build again:

```
make clean
make
```

This is probably going to run for at least 30 minutes. In the meantime, start working on the next lab.

When the build is over:

- Measure the new root filesystem archive and `ffmpeg` executable size, write it down in the table, and compute the difference percentages vs. the ARM code.
- Update the SD card with the new filesystem, run the same time measurements, and write down the results to compare them with the ARM ones. You can also add the percentage difference.

So, was it a good choice to switch to *Thumb2*? Where there any size and performance benefits?

Don't hesitate to show your results to your instructor.

Test the musl C library

The last thing to try in this lab is using a toolchain with the *Musl* C library, instead of *uClibc*, which is the C library that Buildroot uses by default.

Once again, keep a copy your current Buildroot directory:

```
cd ..
cp -al buildroot buildroot-thumb2
```

Back to the `buildroot` directory, run `make menuconfig`, and in `Toolchain`, set `C library` to to `musl`.

Save, run `make clean` and build the root filesystem once again.

Once again, write down the two sizes and measure `ffmpeg` execution time.

Now, what the best combination? *ARM* or *Thumb2*, *uClibc* or *Musl*?

If you have the same size and performance between *uClibc* or *Musl*, its better to choose the latter, as according to the slides, it will allow to generate smaller static executables (we will try that in later instructions). Another reason is that the *Musl* library has a more liberal license, making it easier to ship static executables.

Generate a Buildroot SDK to rebuild faster

Choose Buildroot configuration that worked best for you, renaming the directory to `buildroot` if that was not the last one you tried.

With Buildroot, it's frequent to need to run `make clean` and thus make a full rebuild, typically after configuration changes. As you've seen, such rebuilds are expensive with our Buildroot configuration that builds the toolchain too.

Now that we have finalized our toolchain, let's have Buildroot generate an SDK that we won't have to build from scratch every time we need a full rebuild. In the below instructions, you assume that you chose a *Musl* toolchain:

```
make sdk
cd ~/boot-time-labs/rootfs
tar xf buildroot/output/images/arm-buildroot-linux-musleabihf_sdk-buildroot.tar.gz
cd arm-buildroot-linux-musleabihf_sdk-buildroot
./relocate-sdk.sh
```

Let's then configure Buildroot to use this new toolchain:

```
cd ../buildroot/
make menuconfig
```

In the `Toolchain` menu:

- Set `Toolchain type` to `External toolchain`
- Set `Toolchain` to `Custom toolchain`
- Set `Toolchain origin` to `Pre-installed toolchain`
- Set `Toolchain path` to `/home/<user>/boot-time-labs/rootfs/arm-buildroot-linux-musleabihf_sdk-buildroot` (replace `<user>` by your actual user name)

- Set `External toolchain gcc version` to `9.x`
- Set `External toolchain kernel headers series` to `5.4.x or later`
- Set `External toolchain C library` to `musl (experimental)`

Now test that your settings are correct:

```
make clean
make
```

Application optimization

Optimize the size and startup time of your application

Measuring

We have already measured application startup time in the previous lab.

Remove unnecessary functionality

Compiling ffmpeg with a reduced configuration

In our system, we use a generic version of `ffmpeg` that was built with support for too many codecs and options that we actually do not need in our very special case.

So, let's try to find out what the minimum requirements for `ffmpeg` are.

A first thing to do is to look at the ffmpeg logs:

```
Input #0, video4linux2,v4l2, from '/dev/video0':
  Duration: N/A, start: 93.369296, bitrate: N/A
    Stream #0:0: Video: mjpeg (Baseline), yuvj422p(pc, bt470bg/unknown/unknown), 544x288, 30 fps, 30 tbr, 1000k tbn, 1000k tbc
Stream mapping:
  Stream #0:0 -> #0:0 (mjpeg (native) -> rawvideo (native))
Press [q] to stounable to decode APP fields: Invalid data found when processing input
[swscaler @ 0x80f50] deprecated pixel format used, make sure you did set range correctly
[swscaler @ 0x80f50] No accelerated colorspace conversion found from yuv422p to rgb565le.
Output #0, fbdev, to '/dev/fb0':
  Metadata:
    encoder         : Lavf58.29.100
    Stream #0:0: Video: rawvideo (RGB[16] / 0x10424752), rgb565le, 544x288, q=2-31, 75202 kb/s, 30 fps, 30 tbn, 30 tbc
    Metadata:
      encoder         : Lavc58.54.100 rawvideo
```

Here we see that `ffmpeg` is using:

- Input from a `video4linux` device, decoding an `mjpeg` stream.
- Encoding a `rawvideo` stream, written to an `fbdev` output device.
- A software scaler to resize the input video for our LCD screen

Let's check `ffmpeg`'s `configure` script, and see what its options are:

```
cd ~/boot-time-labs/rootfs/buildroot-arm/output/build/ffmpeg-4.2.4
./configure --help
```

We see that `configure` has precisely three interesting options: `--list-encoders`, `--list-decoders`, `--list-filters`, `--list-outdevs` and `--list-indevs`.

Run `configure` with each of those and recognize the features that we need to enable.

Following these findings, here's how we are going to modify Buildroot's configuration for `ffmpeg`.

This time, let's assume that the *Thumb2* build from the previous lab has completed. If that's the case, finish that lab (measuring and writing down size and performance), and come back here when you are done:


```
cd ~/boot-time-labs/rootfs/buildroot/
make menuconfig
```

In Buildroot's configuration interface:

- Set `Enabled encoders` to `rawvideo`
- Set `Enabled decoders` to `mjpeg`
- Empty the `Enabled muxers`, `Enabled demuxers`, `Enabled parsers`, `Enabled bitstreams` and `Enabled protocols` settings.
- Set `Enabled filters` to `scale`
- For `Enable output devices` and `Enable input devices`, individual device selection is not possible, so we will configure devices manually in the next field. So, empty such settings.
- Set `Additional parameters for ./configure` to `--enable-indev=v4l2 --enable-outdev=fbdev`

Now, let's get Buildroot to recompile `ffmpeg`, taking our new settings into account:

```
make ffmpeg-dirclean
make
```

You can now fill the below table, reusing data from the previous lab:

	Total rootfs size	`/usr/bin/ffmpeg` size
Initial configuration		
Reduced configuration		
Difference (percentage)		

Do you expect to see differences in execution time, with a reduced configuration? Run the measures with `time` again, and compare with what you got during the previous lab.

If the results surprise you, don't hesitate to show them to your instructor ask for her/his opinion.

Trying to remove further features

Looking at the `ffmpeg` log which displays enabled configuration settings, try to find further configuration switches which can be removed without breaking the player in our particular system.

Inspection of the whole root filesystem

Something that can help too is to inspect the whole root filesystem, looking for files that don't seem necessary.

The easiest way is to do this on the workstation:

```
sudo apt install tree
cd /media/$USER/rootfs
tree
```

The `tree` command really makes this task easier. For the moment, don't bother about Busybox and system files. They will be addressed later. Better focus on files and libraries related to `ffmpeg`.

Further analysis of the application

With a build system like Buildroot, it's easy to add performance analysis and debugging utilities.

Configure Buildroot to add `strace` to your root filesystem. You will find the corresponding configuration option in `Package selection for the target` and then in `Debugging, profiling and benchmark`.

Run Buildroot and reflash your device as usual.

Tracing and profiling with strace

With `strace`'s help, you can already have a pretty good understanding of how your application spends its time. You can see all the system calls that it makes and knowing the application, you can guess in which part of the code it is at a given time.

You can also spot unnecessary attempts to open files that do not exist, multiple accesses to the same file, or more generally things that the program was not supposed to do. All these correspond to opportunities to fix and optimize your application.

Once the board has booted, run `strace` on the video player application:

```
strace -tt -f -o strace.log ffmpeg -f video4linux2 -video_size 544x288 \
-input_format mjpeg -i /dev/video0 -pix_fmt rgb565le -f fbdev /dev/fb0
```

Also have `strace` generate a summary:

```
strace -c -f -o strace-summary.log ffmpeg ...
```

Take some time to read `strace.log`[6], and see everything that the program is doing. Don't hesitate to lookup the ioctl codes on the Internet to have an idea about what's going on between the player, the camera and the display.

Also have a look at `strace-summary.log`. You will find the number of errors trying to open files that do not exist, and where most time is spent, for example. You can also count the number of memory allocations (using the `mmap2` system call).

Optimizing necessary functionality

At this stage, there is nothing more we can really do to further optimize `ffmpeg`, unless we are ready to dig into the code and make changes.

However, if the player was your own application, I'm sure this would help to understand how it's actually behaving and how to improve it to make it even faster and smaller.

Putting things back together

Now that we have analyzed the execution of the video player, let's restore the normal configuration for the system:

- Remove support for `strace`

[6] At this stage, when you have to open files directly on the board, some familiarity with the basic commands of the `vi` editor becomes useful. See https://bootlin.com/doc/command_memento.pdf for a basic command summary. Otherwise, you can use the more rudimentary `more` command. You can also copy the files to your PC, using a USB drive, for example.

- Restore the `0001-ffmpeg-log-notification-after-first-frame.patch` patch, replacing the most recently applied patch.
- Restoring the automatic execution of `ffmpeg` in `/etc/init.d/S50playvideo`.

As explained in the Buildroot manual[7], you need to make a full rebuild after disabling packages (such as `strace` in our case). Otherwise, such packages will still be present in the filesystem image. Fortunately, full rebuilds are now fast with Buildroot when it's using a prebuilt toolchain:

```
make clean
make
```

Update your root filesystem and then reboot.

According to our tests, there should be an issue now: the video player is started even before the camera is ready, as you can see in the system console.

To address this issue for the time being, let's modify `/etc/init.d/S50playvideo` by adding a loop waiting for the `/dev/video0` device to appear. So, let's add the following lines to `board/beaglecam/rootfs-overlay/etc/init.d/S50playvideo`, before the call to `echo "Starting ffmpeg"`:

```
if ! [ -e /dev/video0 ]
then
   echo "Waiting for /dev/video0 to be ready..."
   while ! [ -e /dev/video0 ]
   do
       sleep 0.001
   done
fi
```

Notes:

- It seems you can not run an empty `while` loop with Busybox `sh`. That's why I had to put a real command (not a comment) inside the loop.
- Fortunately, the `sleep` command supports subsecond waiting. Did you know?
- When we optimize the kernel, we will try to address this camera readiness issue. If we can't fix it, at least we will display something on the screen to make the user wait.

Update and reboot your system through `grabserial`, and copy and paste your output to `boot-time-labs/logs/application.log`. Fill the below table with updated figures (we don't expect earlier parts of system bootup to be impacted... scripts may run faster after being recompiled in `Thumb2`):

Step	Duration	Description
Init scripts		Between `Run /sbin/init` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total boot time		

[7] https://buildroot.org/downloads/manual/manual.html#full-rebuild

Init script optimizations

Analyzing and optimizing init scripts

Measuring

Remember that the first step in optimization work is measuring elapsed time. We need to know which parts of the init scripts are the biggest time consumers.

Check and write down the initial size of the root filesystem archive.

Use bootchartd on the board

Add `bootchartd` support (`CONFIG_BOOTCHARTD`) to your `BusyBox` configuration:

```
cd ~/boot-time-labs/rootfs/buildroot
make busybox-menuconfig
```

In `Archival Utilities`, also enable `Make tar, rpm, modprobe etc understand .gz data` (`FEATURE_SEAMLESS_GZ`), which is needed by `bootchartd`.

After saving the configuration, the `make` command should only take a few seconds to run.

Re-flash the root filesystem

Update your root filesystem on the SD card.

The next thing to do is to use the `init` argument on the kernel command line (in `u-boot`, this is the `bootargs` environment variable) to boot using `bootchart` instead of using the `init` program provided by Busybox.

So, boot your board but stay in the U-Boot shell, by pressing the `Space` key before the timer expires.

Add `init=/sbin/bootchartd` to the bootargs variable:

```
U-Boot> setenv bootargs ${bootargs} init=/sbin/bootchartd
U-Boot> saveenv
```

This will make the system boot and the resulting bootlog will be located in `/var/log/bootlog.tgz`. As `/var/log` is actually stored in RAM (through the *tmpfs* filesystem, you will copy it to the root filesystem.

First, as the filesystem is read-only, remount it in read-write mode:

```
mount -o remount,rw /
```

Now, copy the file to the root filesystem storage and halt your board:

```
cp /var/log/bootlog.tgz /root/
halt
```


Remove the SD card, insert it in your PC, and copy that file on your host:

```
cd $HOME/boot-time-labs/rootfs
sudo cp /media/$USER/rootfs/root/bootlog.tgz .
sudo chown $USER.$USER bootlog.tgz
```

Analyze bootchart data on your workstation

To compile and use `bootchart` on your workstation, you first need to install a few Java packages:

```
sudo apt install ant openjdk-11-jdk
```

Note that `ant` is a Java based build tool like `make`.

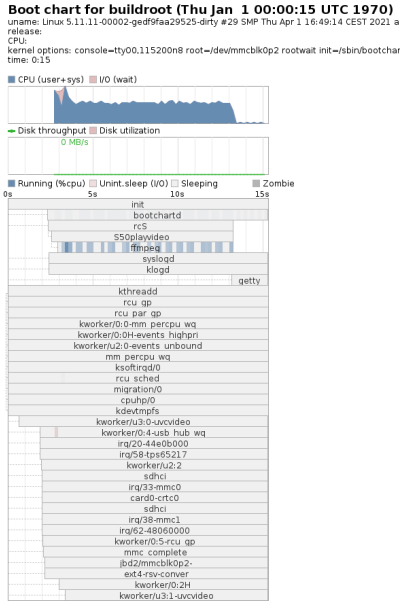
Now, get the `bootchart` source code for version 0.9 from https://bootlin.com/pub/source/bootchart-0.9.tar.bz2[8], compile it and use `bootchart` to generate the boot chart:

```
tar xf bootchart-0.9.tar.bz2
cd bootchart-0.9
ant
java -jar bootchart.jar ~/boot-time-labs/rootfs/bootlog.tgz
```

This produces the `bootlog.png` image which you can visualize to study and optimize your startup sequence:

```
xdg-open bootlog.png
```

`xdg-open` is a universal way of opening a file with a given MIME type with the associated application as registered in the system. According to the exact Ubuntu flavor that you are using, it will run the particular image viewer available in that particular flavor.



[8] Don't try to get the `bootchart` package supplied by Ubuntu instead. While it has similar functionality, it looks like a completely unrelated piece of software. To confirm this, it has no dependency whatsoever on Java packages.

Remove unnecessary functionality

Getting Buildroot to generate less files

The above graph shows several system processes running during the startup process, consuming CPU time, probably delaying the execution of our application (see the color bars showing when a task is using the CPU).

In the general case, there will be services that you want to keep. At least, you could change the order according to which services are started, by changing the alphanumeric order of startup files (that's reordering / postponing work).

Back to our case, we want to simplify our system as much as possible. In Buildroot's configuration interface, go to the `System Configuration` menu:

- Disable `Enable root login with password` and `Run a getty (login prompt) after boot`.
- Disable `Purge unwanted locales`

Don't forget to remove `bootchartd` support as well and `BR2_FEATURE_SEAMLESS_GZ` which we added earlier. Also disable `ifupdown scripts` in `Networking applications`. This allows to remove the `/etc/init.d/S40network` script.

Before we update the filesystem, let's make another experiment: boot your board, interrupt the video player, and unmount `/proc` and `/sys` manually. Then, run the video player command again, and you'll see that `ffmpeg` runs perfectly without these virtual filesystems mounted.

This means we could directly run the video player as the *init* process! To keep the possibility to interact with the board through a command line shell, we're going to run a shell after the video player.

So, let's remove `/etc/init.d/S50playvideo` from the root filesystem overlay and replace it by the `/playvideo` script provided in `~/boot-time-labs/rootfs/data/`.

Regenerate your root filesystem:

```
make clean
make
```

Update and reflash your SD card. Reboot the board, but before booting, stop in U-Boot to update the *init* program:

```
setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk0p2 rootwait rw init=/playvideo
saveenv
boot
```

Note that the `rw` setting is going to be important, as it will make Linux mount the root filesystem in read/write mode, which allows to record the access time of each file.

Detecting and eliminating unused files

If the video played ran fine as expected, you should now be in a shell in the serial console.

Type the `sync` command to flush the filesystem, remove the SD card and insert it on your PC again. Go to `/media/$USER/rootfs` and run the below command:

```
find . -atime -100 -type f
```


This lists the regular files that were not accessed during the boot sequence we've just executed:

```
./usr/lib/os-release
./usr/share/ffmpeg/libvpx-720p50_60.ffpreset
./usr/share/ffmpeg/libvpx-360p.ffpreset
./usr/share/ffmpeg/libvpx-720p.ffpreset
./usr/share/ffmpeg/ffprobe.xsd
./usr/share/ffmpeg/libvpx-1080p.ffpreset
./usr/share/ffmpeg/libvpx-1080p50_60.ffpreset
./usr/share/udhcpc/default.script
./bin/busybox
./lib/libatomic.so.1.2.0
./lib/libgcc_s.so.1
./etc/hostname
./etc/profile.d/umask.sh
./etc/init.d/S02klogd
./etc/init.d/S20urandom
./etc/init.d/rcS
./etc/init.d/rcK
./etc/init.d/S01syslogd
./etc/group
./etc/protocols
./etc/inittab
./etc/passwd
./etc/services
./etc/fstab
./etc/shadow
./etc/hosts
./etc/profile
./etc/issue
./etc/shells
```

How does it work?

`-atime -100` finds all the files which last access time is less than 100 minutes ago, actually when we extracted the archive. Why doesn't it find the files which were actually accessed during the boot sequence? That's because the board doesn't have a clock set and its date got back to January 1st, 1970. You can check the date of such a file:

```
stat etc/init.d/playvideo
  File: etc/init.d/playvideo
  Size: 300       Blocks: 8          IO Block: 4096   regular file
Device: b302h/45826d Inode: 376         Links: 1
Access: (0755/-rwxr-xr-x)  Uid: ( 1000/    mike)   Gid: ( 1000/    mike)
Access: 1970-01-01 01:00:02.250000000 +0100
Modify: 2019-05-21 22:29:21.348835786 +0200
Change: 2019-05-21 22:29:21.748834877 +0200
 Birth: -
```

A limitation is that it doesn't work with symbolic links and directories, so we don't know whether a symbolic link or directory was accessed or not. That's why we're keeping only regular files (`-type f`).

There's a discrepancy too with `/bin/busybox` which was for sure accessed, but through a symbolic link. We will have to remove it from the list.

If we had a board with a correct date, we would have still been able to use this technique, but this time only looking for files last accessed more than a few minutes ago (`-atime +5`).

We can now implement a Buildroot *Post build script* that will eliminate such files in the target directory. That's much easier than tweaking the recipes that generated these files, and this can be adapted to each case (each case is special, while recipes should be generic).

So, in the Buildroot directory, create a `board/beaglecam/post-fakeroot.sh` script that removing all the above files (except `/bin/busybox`):

```
#!/bin/sh
TARGET_DIR=$1
cd $TARGET_DIR
rm -rf \
./usr/lib/os-release \
./usr/share/ffmpeg/libvpx-720p50_60.ffpreset \
./usr/share/ffmpeg/libvpx-360p.ffpreset \
...
```

Don't forget to make this script executable!

The last thing to do is to configure `BR2_ROOTFS_POST_FAKEROOT_SCRIPT` to `board/beaglecam/post-fakeroot.sh`[9].

Rerun Buildroot and check that your root filesystem was simplified as expected:

```
make
```

See what's left in the final archive. Actually, we propose to be more aggressive and directly remove the entire `/etc` directory which we shouldn't need any more. Do the same for `/root`, `/tmp`, `/var`, `/media`, `/mnt`, `/opt`, `/run` and the `/lib32` link. We're keeping the `/proc` and `/sys` directories in case we need to mount the corresponding filesystems.

Why doing this so aggressively for files we won't access? On an *ext4* filesystem, files that are not accessed may not do any harm if they are not used, except perhaps marginally in terms of mounting time (if the filesystem is unnecessarily big). However, if we store the root filesystem in an *Initramfs* embedded in the kernel binary, every byte counts as it will make the kernel to load bigger.

Update your root filesystem, remove the `rw` kernel parameter from `bootargs` in U-Boot (better to keep the root filesystem mounted read-only as we don't cleanly shut down the system) and check that your system still boots fine.

Check and write down the new size of the root filesystem archive.

Reducing BusyBox to the minimum

While we're simplifying the root filesystem, it's time to reduce the configuration of Busybox, to only contain the features we need in our system.

Before we do this, check the size of the `busybox` executable in your root filesystem.

Buildroot helps us to configure BusyBox by providing a `make busybox-menuconfig` command, but it will be tedious to use because we will have to unselect countless options.

[9] We could have used Buildroot's post build scripts (`BR2_ROOTFS_POST_BUILD_SCRIPT`), but that would have been too early, as the `fakeroot` scripts make some customizations on files like `/etc/inittab`, which we want to remove.

Here's another way. Go to `output/build/busybox-1.29.3/` and run `make allnoconfig`, followed by `make menuconfig`. You'll see that most options are unselected!

Just select the below options, based on what we have in our `/playvideo` script:

- In `Settings`:
  - Enable `Support files > 2 GB`. Without this, BusyBox will fail to compile (at least with our toolchain)
- In `Shells`:
  - Enable `Use internal glob() implementation`, even if you don't select `ash`. Otherwise, compiling `hush` will fail.
  - Select the `hush` shell
  - Keep only `Support if/then/elif/else/fi` and `Support for, while and until loops`,
- In `Coreutils`:
  - Support for the `sleep` command, with support for fractional arguments.
  - Support for the `echo` command, without additional options.
  - Enable `test` and `test as [`
  - Disable `Extend test to 64 bit`

Now get back to the main Buildroot directory and copy this new configuration:

```
cp output/build/busybox-1.31.1/.config board/beaglecam/busybox.config
```

Then, run `make menuconfig` and set `BR2_PACKAGE_BUSYBOX_CONFIG` to this new file. In `System configuration`, also set `Init system` to `None`. Otherwise Buildroot will enable Busybox `init` into your configuration.

Run `make`, and update your SD card. Check the new size of `/bin/busybox`!

Also write down the new size of the root filesystem tar archive.

Switching to static executables

Since we now have only two executables (`busybox` and `ffmpeg`), let's explore the possibility to switch to static executables, hoping to reduce filesystem size by not having to copy the entire shared libraries.

In Buildroot's configuration interface, find and set `BR2_STATIC_LIBS=y`.

Run `make clean` and `make`.

Run `tar tvf output/images/rootfs.tar` to find out directories which are now empty and therefore can now be removed. Add such directories to your post-fakeroot script and regenerate the filesystem again. This should save a few extra bytes.

Once more, write down the new size of the root filesystem tar archive. You should observe substancial space reduction. Let's keep this option!

Testing

Boot your system again.

If everything works, it's time to boot the system again through `grabserial`, store the output to `logs/init-scripts.log` and update the below table:

Step	Duration	Description
Init scripts		Between `Run /playvideo as init process` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total boot time		

Filesystem optimizations

See what best filesystem options are in terms of boot time

During this lab, we will compare 3 ways of accessing the root filesystem

- Booting from an *ext4* filesystem
- Booting from a *SquashFS* filesystem
- Booting from an *initramfs*

Tests with ext4 and SquashFS

First, recompile your kernel with *SquashFS* support, and update it on the SD card.

Write the size of the `zImage` file in the first row of the below table:

Type	zImage size (bytes)
zImage without initramfs	
zImage with initramfs	

Go to `~/boot-time-labs/rootfs`. You should have a `rootfs` subdirectory where your root filesystem archive has been extracted.

Boot your system with `grabserial`, store its output in `logs/filesystem-ext4.log`, and start filling the table below:

Filesystem	Time for `Run /playvideo`	Time for `First frame decoded`	Time difference
ext4			
SquashFS			
Initramfs			

For *SquashFS*, you will have to create an image and copy it to the raw partition:

```
sudo apt install squashfs-tools
cd ~/boot-time-labs/rootfs
mksquashfs rootfs/ rootfs.sqfs -noappend
sudo dd if=rootfs.sqfs of=/dev/mmcblk0p2
```

Then remove the SD card and boot the board with it as usual, and record and store measurements.

Note that we also made tests with *ext2*, but they were very close to *ext4* ones, at least for very small filesystems like this one. So we decided to skip this filesystem here, to save time.

Testing further filesystems

If you have time, you could also test the *Btrfs*, *F2FS* and *EROFS* filesystems. That's very easy to do, as Buildroot can build images for such filesystems for you.

Initramfs tests

Booting from an *initramfs* is completely different. The strong advantage here is that the root filesystem will be extracted from an archive inside the kernel binary. So instead of several reads from the MMC, we will just have a single one (though bigger), in addition to the Device Tree binary. This can work well with small root filesystems as ours.

Booting the kernel should be faster too, as we won't need the MMC and filesystem drivers at all. So, let's configure the kernel accordingly.

To switch to an initramfs, there are a few things to do though:

- In an initramfs, you cannot have the kernel mount the *devtmpfs* filesystem mounted automatically on `/dev`. We'll mount it from our `playvideo` script. So:
  - Modify the Busybox configuration to add support for the `mount` command (found in `Linux System Utilities`), without any additional option.
  - Add the following line to the `/playvideo` file:
    `mount -t devtmpfs nodev /dev`
  - Run `make` to update your root filesystem.
  - Then extract it in your `rootfs` directory:

    ```
    cd ..
    mkdir rootfs
    cd rootfs
    tar xvf ../buildroot/output/images/rootfs.tar
    ```
- Go to the U-Boot command line on your board, and modify the kernel parameters:
  `setenv bootargs console=ttyO0,115200n8 rdinit=/playvideo`
  `root=` and `rootwait` are ignored when there is an Initramfs and `init` is replaced by `rdinit=`. Don't forget to run `saveenv`.

Go to `~/boot-time-labs/kernel/linux` and run the Linux configuration interface:

- In `General setup`
  - Fill `Initramfs source file(s)` with `../../rootfs/rootfs` (or the correct path to the directory containing your root filesystem)
  - Make sure you set `CONFIG_INITRAMFS_COMPRESSION_NONE=y` to avoid wasting space compressing the initramfs twice.
- Disable `Enable the block layer`
- Disable `MMC/SD/SDIO card support`
- We won't have to disable block filesystems as they are no longer compiled when block support is disabled.

Rebuild your kernel binary:

```
make -j8 zImage
```

Copy the new kernel image to your SD card and then boot the board. You will see that there is a new issue though: the messages from `echo` and `ffmpeg` don't go through. That's probably because `/dev/console` doesn't exit yet when the script is started.

So, let's fix it in our script. We are redirecting the important messages to `/dev/console`:


```
#!/bin/sh
mount -t devtmpfs nodev /dev
if ! [ -e /dev/video0 ]
then
   echo "Waiting for /dev/video0 to be ready..." > /dev/console
   while ! [ -e /dev/video0 ]
   do
       sleep 0.001
   done
fi
echo "Starting ffmpeg" > /dev/console
ffmpeg -t 10 -f video4linux2 -video_size 544x288 -input_format mjpeg -i /dev/video0 -pix_fmt rgb565le -f fbdev /dev/fb0 2> /dev/console
```

Now, you should be able to extract the measures and write them down in the table above. If your tests run the same way ours did, the initramfs approach should win by a few tens of milliseconds.

Also measure the size of your `zImage` file at write it in the table at the top of this chapter, to compare with your initial kernel.

Let's choose this solution with an initramfs. There are still many things we can accelerate during the execution of the bootloader and execution.

Kernel optimizations

Measure kernel boot components and optimize the kernel boot time

Measuring

We are going to use the kernel `initcall_debug` functionality.

Our default kernel already has the configuration settings that we need:

- `CONFIG_PRINTK_TIME=y`, to add a timestamp to each kernel message.
- `CONFIG_LOG_BUF_SHIFT=16`, to have a big enough kernel ring buffer.

That's not sufficient. We also need the output of the `dmesg` command.

We are going to make a few changes to the root filesystem. To save time later going back to the initial Buildroot configuration, make a copy of the `buildroot/` directory to `buildroot-dmesg/`:

```
rsync -aH buildroot/ buildroot-dmesg/
```

In this new directory, add support for `dmesg` command in BusyBox, and add the below line after the `ffmpeg` file in the `playvideo` scripts:

```
dmesg > /dev/console
```

Run Buildroot again, and update your `~/boot-time-labs/rootfs/rootfs` directory again. Compile your kernel again to to update the `zImage` with this root filesystem.

Now, let's enable `initcall_debug` in kernel parameters. Go to the U-Boot command line, and add the below settings to the kernel command line [10], and boot your system:

```
setenv bootargs ${bootargs} initcall_debug printk.time=1
boot
```

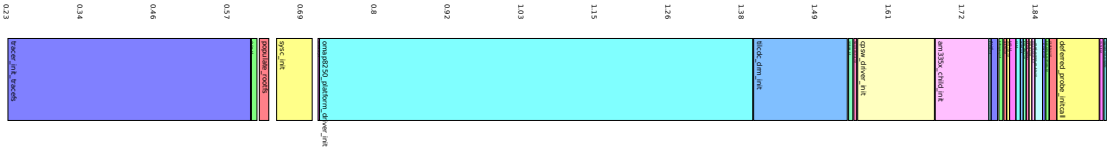
Boot the board with the new kernel image. If everything went well, you can now copy and paste the special `dmesg` output to a `~/boot-time-labs/kernel/initcall_debug.log` file on your workstation.

In `~/boot-time-labs/kernel` (at least where the kernel sources are), run the following command to generate a boot graph:

```
linux/scripts/bootgraph.pl initcall_debug.log > boot.svg
```

You can view the boot graph with the `inkscape` vector graphics editor:

```
sudo apt install inkscape
inkscape boot.svg
```



[10] Don't save these settings with `saveenv`. We will just need them once.

Now review the longest initcalls in detail. Each label is the name of a function in the kernel sources. Try to find out in which source file each function is defined[11], and what each driver corresponds to.

Then, you can look the source code and:

- See whether you need the corresponding driver or feature at all. If that's the case, just disable it.
- Otherwise, try look for obvious causes which would explain the very long execution time: delay loops (look for `delay`, parameters which can reduce probe time but are not used, etc).
- There could also be features than could be postponed. However, in our special case, we should only need to keep kernel features that we need to run our video player. However, in a real life system, the boot graph could indeed reveal drivers which could be compiled as modules and loaded later.

Recompile and reboot the kernel, updating the boot graph until there is nothing left that you can do.

When you are done exploiting data from the boot graphs, you can remove `dmesg` support from BusyBox and remove this command too from `playvideo`. Update your root filesystem and then kernel so that we get back to the original situation. We no longer need `initcall_debug`.

Removing unnecessary functionality

It's time to start simplifying the kernel by remove drivers and features that you won't need.

Do this **very progressively**. If you go too fast, you'll end up with a kernel that doesn't boot any more, but you won't be able to tell which parameter should have been kept.

Also, don't disable `CONFIG_PRINTK` too early as you would lose all the kernel messages in the console.

Also, for the moment, don't touch the options related to size and compression, including compiling the kernel with *Thumb2*, as the impact of each option could depend on the size of the kernel.

Make sure you go through all the possibilities covered in the slides, in particular to enable `CONFIG_EMBEDDED` to allow to unselect further features that should be present on a general purpose system[12].

At the end, you can disable `CONFIG_PRINTK`, and observe your total savings in terms of kernel size and boot time.

Last but not least, try to find other ways of reducing the kernel size. Go through the `.config` file and the kernel build log and look for ideas to further reduce size and boot time.

Optimizing required functionality

The time has come to make final optimizations on our kernel, mainly related to code size.

First, measure and write down your kernel size and the total boot time:

[11] You can do it with utilities such as `cscope`, which your instructor will be happy to demonstrate, or through our on-line service to explore the Linux kernel sources: `https://elixir.bootlin.com`

[12] Here we have a very specific system and we don't have to support programs that could be added in the future and could need more kernel features

Kernel type	Kernel size	Total boot time
ARM		
Thumb2		

Now, compile your kernel with `CONFIG_ARM_THUMB`. Before you do this, you could make a backup copy of your kernel source directory with `cp -al`, as a full rebuild of the kernel will be needed, and we may want to roll back later. Fortunately, thanks to our feature reduction work, the full rebuild should be faster than in the earlier labs.

Write down the kernel size and total boot time in the above table, and keep whatever option works best for you.

Then, continue by trying all the kernel compression schemes listed in the below table:

Compression type	Kernel size	Total boot time
Gzip		
LMZA		
XZ		
LZO		
LZ4		
None		

For the `None` row, there is no kernel configuration option, but all you have to do is take the `arch/arm/boot/Image` file, and make a `uImage` file out of it (as U-Boot's `bootz` command only works with `zImage` files):

```
sudo apt install u-boot-tools
mkimage -A arm -O linux -C none  -T kernel -a 80008000 -e 80008000 \
        -n 'Linux-5.11.11' -d arch/arm/boot/Image arch/arm/boot/uImage
```

Then, in U-Boot, you will have to boot it with `bootm` instead of `bootz`.

This option can make sense when the CPU is very slow and the storage is quite fast (like when you're booting Linux on a CPU emulated on an FPGA).

At the end, keep the option that gives you the best boot time, and update the below table:

Step	Duration	Description
U-Boot SPL		Between `U-Boot SPL 2021.04` and `U-Boot 2021.04`
U-Boot		Between `U-Boot 2021.04` and `Starting kernel`
Kernel + Init scripts		Between `Starting kernel` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total		

Note that we have merged the *Kernel* and *Init scripts* parts (the latter being very short anyway), because the kernel is now silent.

At the end of this lab, you can remove the `buildroot-dmesg` directory, which is no longer needed.

Bootloader optimizations

Reduce bootloader execution time

In this lab, we will run the final stage of boot time reduction:

- Improving the efficiency of the bootloader by optimizing its usage
- Recompiling the bootloader with the minimum set of options, and even completely skip the second stage of the bootloader.

Optimizing U-Boot usage

By following the indications given in the lectures, start by optimizing the way U-Boot is used.

At last, you can start by eliminating the infamous 2-second boot delay, something you've surely been longing to do.

Recompiling the bootloader

Importante note: for this lab, because of issues with Falcon Mode in U-Boot 2019.01 that we are currently investigating, you will have to switch back to the 2019.01 version of U-Boot, configured with its `am335x_boneblack_defconfig` file.

It's now time to eliminate useless features in U-Boot. Go to `~/boot-time-labs/bootloader/u-boot/` and run `make menuconfig` to unselect features that we don't need in our system.

For the moment, don't touch the `SPL / TPL` options, as we will try to use U-Boot's Falcon mode at the end.

In the same way you did when you reduced the kernel configuration, do the changes **progressively**, and even make backup copies of your intermediate configurations (`.config` file). You will be glad you did when you break U-Boot.

Once you have reached the minimum set of features, please measure boot time and fill the below table:

Step	Duration	Description
U-Boot SPL		Between `U-Boot SPL 2019.01` and `U-Boot 2019.01`
U-Boot		Between `U-Boot 2019.01` and `Starting kernel`
Kernel + Init scripts		Between `Starting kernel` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total		

Using faster storage

A last minute surprise: your instructor will give you new SD cards with faster read performance, at least as fast as the Beagle Bone Black seems to be able to go.

Why on earth didn't we use such SD cards right from the start of our labs?

It's because slower storage acts as a magnifying glass (or as a slow motion device) making it easier to observe elapsed time and the benefits of our optimizations. If the storage was lightning fast, it would be harder to appreciate speedups due to a small initramfs, for example.

So, edit the partition table of your new SD card, and create the first partition in the same way as when you prepared your original SD card. Then, copy the files over.

You can now go ahead and make tests again, and fill the table with your latest results:

Step	Duration	Description
U-Boot SPL		Between `U-Boot SPL 2019.01` and `U-Boot 2019.01`
U-Boot		Between `U-Boot 2019.01` and `Starting kernel`
Kernel + Init scripts		Between `Starting kernel` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total		

Using U-Boot's *Falcon* mode

It's now time to try U-Boot's capability to directly load the Linux kernel from its first stage (SPL), instead of loading U-Boot.

What follows is based on U-Boot's own documentation in its sources:

- `doc/README.falcon` (generic details)
- `board/ti/am335x/README` (specific details for boards with the am335x SoC)

The first thing to do is to generate a `uImage` file for the kernel binary. This image file contains information that U-Boot uses to know a few things about the kernel binary, most importantly the final load address, but also the type of file (binary, script, environment file), the target architecture and whether the binary is compressed or not.

This is called a *legacy image* for U-Boot. As you already know, U-Boot can now boot a `zImage` file, but according to the Falcon mode documentation, it does need a `uImage` file for SPL loading.

So, let's generate this file:

```
cd ~/boot-time-labs/kernel/linux/
make uImage LOADADDR=80008000
```

Copy this `uImage` file to your SD card boot partition.

To optimize the size of the U-Boot SPL, let's recompile it without the features we don't need. So, in U-Boot's `menuconfig` interface, go to the `SPL / TPL` menu and:

- Unselect `Support an environment`. Our own tests showed that this saves about 250 ms!
- Unselect `Suppport USB Gadget drivers`

Compile U-Boot again and copy the `u-boot.img` and `MLO` files to the boot partition too.

Now, let's run the final preparation step. We will set the `bootargs` environment variable, load the kernel and DTB, and use U-Boot's `spl export` command to prepare a ready to boot record with the DTB contents, the `bootargs`, the kernel loading addreses and other information that Linux would need to boot. Note that the U-Boot SPL will still load the `uImage` file from the FAT filesystem in the first partition of the SD card.

In the below command, you'll see that we can use U-Boot's ready made `loadaddr` and `fdtaddr` variables for addresses where to load the kernel and DTB. At least this works with U-Boot for our board.

```
load mmc 0:1 ${loadaddr} uImage
load mmc 0:1 ${fdtaddr} dtb
setenv bootargs console=ttyO0,115200n8 rdinit=/playvideo
spl export fdt ${loadaddr} - ${fdtaddr}
```

You can then see that `spl export` prepared everything to boot the Linux kernel with the provided DTB, but didn't do it. At the end, it tells you where the exported data were stored in RAM:

```
## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Linux-5.11.11-dirty
   Created:      2021-04-13   9:48:35 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2842016 Bytes = 2.7 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Kernel Image
   Loading Device Tree to 8ffec000, end 8ffff3ff ... OK
subcommand not supported
subcommand not supported
   Loading Device Tree to 8ffd5000, end 8ffeb3ff ... OK
Argument image is now in RAM: 0x8ffd5000
WARN: FDT size > CMD_SPL_WRITE_SIZE
```

The last thing to do is to store such information in an `args` file in the FAT partition on the MMC, using the starting RAM address provided above and its size (`0x8ffeb3ff - 0x8ffd5000`):

```
fatwrite mmc 0:1 0x8ffd5000 args 0x163ff
```

You're ready to go and reboot your board with the SD card inside. You should not longer see the U-Boot second stage being loaded, but just the SPL and the kernel.

If this doesn't work yet, please ask your instructor for advice and help.

When it works, update your table again:

Step	Duration	Description
U-Boot SPL		Between `U-Boot SPL 2019.01` and `Starting kernel`
Kernel + Init scripts		Between `Starting kernel` and `Starting ffmpeg`
Application		Between `Starting ffmpeg` and `First frame decoded`
Total		

Going further

There are several things we can do to try to further optimize things:

- As our storage is now faster, it can be interesting to explore the various kernel compression schemes again. The optimum solution may be a different one.

- Look for a solution to eliminate the delay detecting the USB webcam.
- If you don't manage to get rid of this delay, at least take advantage of this spare time to show signs of life on the screen, by implementing a splashscreen. You can even implement an animation. One thing you can do is use BusyBox's `fbsplash` tool, to first show an image on the framebuffer, and then even show a progress bar (knowing how much time you have to wait for the camera to be ready).

