Boot time optimization Training

Lab Book

free electrons
http://free-electrons.com

June 2, 2017

About this document

Updates to this document can be found on `http://free-electrons.com/doc/training/{}/`.

This document was generated from LaTeX sources found on `http://git.free-electrons.com/training-materials`.

More details about our training sessions can be found on `http://free-electrons.com/training`.

Copying this document

Preparing hardware

Preparing a PC and a USB drive before the workshop.

Make sure the prerequisites are met before the workshop. Make sure you do not follow the below instructions on the morning the workshop starts, as these preparation steps can take up to several hours, depending on the speed of your Internet connection.

Here is the required hardware:

- Atmel SAMA5D3x evaluation kit
- 5V power adaptor for the kit
- At least one USB-A (male) to micro-USB (male) cable. Two such cables if possible.
- Recent PC with at least 2 GB of RAM, a decently fast processor and 20 GB of free contiguous space for installing Linux.
- 1 USB flash drive with at least 4 GB of free space

Installing Linux on the PC

Do this before joining the workshop. This can be either done by participants themselves on their own PCs, or on PCs supplied by the entity which organizes the event.

For the practical labs in this workshop, we need you to install **Ubuntu Desktop 12.04**. We support both 32 and 64 bit variants, though 64 bit will probably give you the best performance if you have a recent CPU.

Get the Desktop edition of Ubuntu 12.04 at `http://www.ubuntu.com/download/desktop`, and follow the instructions. Note that Xubuntu and Kubuntu variants are fine too.

The way your disk is partition will also matter in the practical labs. What matters is that `/opt` is in a partition with at least 15 GB of disk space. It's still because of the pre-compiled build environment which must have a specific path to be useful.

The easiest way to achieve this is to have only one big root (`/`) partition, which would then contain `/opt`. If you have enough space, you can of course have a dedicated partition for `/opt`.

Important notes

- We do not support Linux installations in a virtual machine. VMs will suffer from long compile times and will complicate access to the real hardware.
- Using Ubuntu in live mode (from with installing it) will not work either. Memory may get scarse, compile time will be much longer because of the slow disk, and exporting the root filesystem through NFS won't work.

At the end of installation, we recommend to install the latest updates:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Preparing a USB disk for the workshop

The USB disk will be needed by workshop participants:

- To access files which could take a long time to download in the workshop venue (if it has a slow connection to the Internet or if there are too many people downloading the same files at once).
- To get the pre-compiled build environment, saving at least one hour of compile time. This file is several hundreds of MB big, and would also take a long time to download.

Now take a USB disk with at least 4 GB of free space, and connect it to your Ubuntu PC. You should see a window showing its contents.

We are going to reformat this disk, so that every participant accesses his disk through the same name and path.

The first thing you need to do is find which device file is associated to this newly inserted disk. Type the `dmesg` command and look at the last lines:

```
...
[ 6103.687852]  sdb: sdb1
...
```

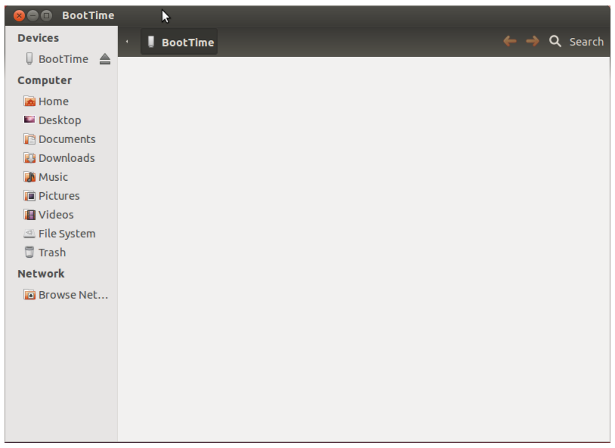
Let's assume that this disk associated to `/dev/sdb`:

- `/dev/sdb` represents the whole disk
- `/dev/sdb1` represents only the first partition on this disk. There could be more partitions.

Now, let's format this first partition for the `FAT32` filesystem (make sure you replace `sdb1` by the right device name, otherwise you may destroy data on other disks!):

```
sudo umount /dev/sdb1
sudo mkfs.vfat -F 32 -n BootTime /dev/sdb1
```

Now, remove your USB drive and insert it again. A file explorer window should appear:

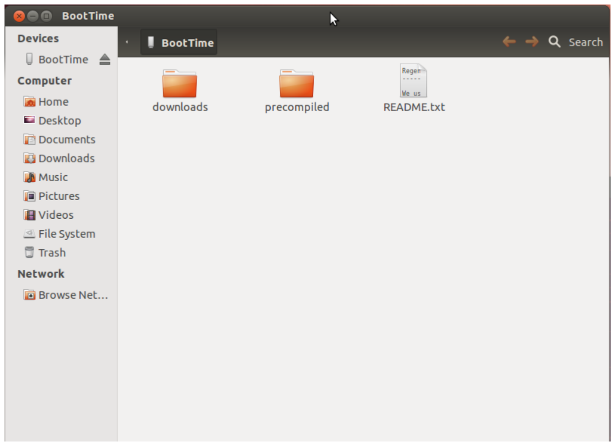


Now let's fill this disk with data needed during the workshop:

```
cd /media/$USER/BootTime
wget -r -np -nH --cut-dirs=2 --reject "index.html*" http://free-electrons.com/labs/boottime/
sudo umount /media/$USER/BootTime
```

Depending on your Internet connection speed, downloading the files could take up to several hours.

At the end, your USB drive contents should look like this:



Installing the lab archive

Please follow the below instructions carefully. If you don't use the same directories, your pre-compiled build environment won't work and you will waste a lot of time compiling software.

Type the below commands:

```
cd /opt
sudo chown -R $USER.$USER .
wget http://free-electrons.com/doc/training/boot-time/boot-time-labs.tar.xz
sudo tar xf boot-time-labs.tar.xz
sudo chown -R $USER.$USER .
```

Installing software packages needed during the workshop

This step is recommended for all workshops. It will pre-install all the software packages needed during the workshop.

To keep our instructions as close as real life as possible, our labs ask to install software packages exactly when they are needed. However, if all the participants to a workshop download the same packages at the same time, downloads are very likely to be slow.

This step can be skipped if you are just a few people doing the labs from home or from an office with fast Internet access.

Here is the command to pre-install the software packages:

```
/opt/boot-time-labs/setup/install-packages
```

Ready for the workshop

You are now ready to do the workshop!

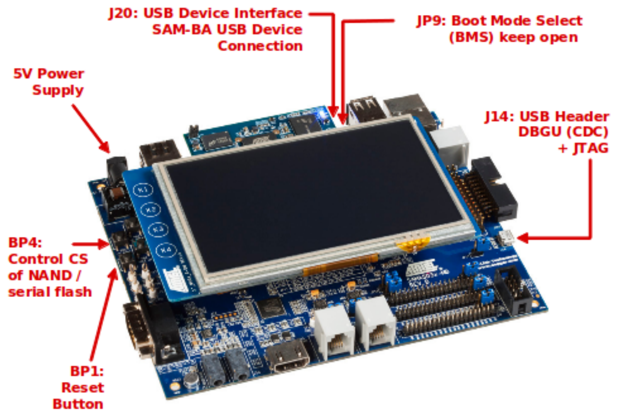
Getting started

Learn how to interact with the board and how to reflash it

Board access through the serial line

The first thing you need when working with a board is to access one of the serial ports of the processor.

On the Atmel board, the processor's DBGU port can be accessed through the J14 connector.


J20: USB Device Interface SAM-BA USB Device Connection
JP9: Boot Mode Select (BMS) keep open
5V Power Supply
J14: USB Header DBGU (CDC) + JTAG
BP4: Control CS of NAND / serial flash
BP1: Reset Button

SAMA5D3x EK board connectors
(Source: http://www.at91.com/linux4sam/bin/view/Linux4SAM/GettingStarted_a5d3x)

Power on your board and take a USB to micro USB cable provided by your instructor. Connect the micro USB end to J14, and the other end to your PC.

You should now have a device file named /dev/ttyACM0:

```
$ ls -la /dev/ttyACM0
crw-rw---- 1 root dialout 166, 0 Jul  9 16:48 /dev/ttyACM0
```

You will use this device file to access the serial connection to the board. As you can see, Ubuntu only allows the root user and dialout group to access this file. A solution is to add your user to this dialout group:

```
sudo adduser $USER dialout
```

You need to log out and in again for the group change to be effective.

To communicate with the board through the serial port, install a serial communication program, such as picocom[1]:

```
sudo apt-get install picocom
```

[1] picocom is one of the simplest utilities to access a serial console. It works in a simple terminal. minicom has more features, but is also more complex to configure.

Now, run the below command, to start serial communication on `/dev/ttyACM0`, with a baudrate of 115200:

```
$ picocom -b 115200 /dev/ttyACM0
picocom v1.4

port is        : /dev/ttyACM0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
escape is      : C-a
noinit is      : no
noreset is     : no
nolock is      : no
send_cmd is    : ascii_xfr -s -v -l10
receive_cmd is : rz -vv

Terminal ready
```

If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

Flash the board

First, create the working directory for this lab:

```
mkdir /opt/boot-time-labs/flashing
cd /opt/boot-time-labs/flashing
```

It is now time to flash the board with the filesystem used in this workshop. To do so, we will use a tool named SAM-BA, provided by ATMEL. You can download SAM-BA at `http://atmel.com/Images/sam-ba_2.12.zip`, or take it from the USB flash drive as follows:

```
cp /media/$USER/BootTime/downloads/sam-ba_2.12.zip .
unzip sam-ba_2.12.zip
```

Now add the SAM-BA directory to your `PATH` environment variable by adding the below line at the end of your `~/.bashrc` file:

```
export PATH=$PATH:/opt/boot-time-labs/flashing/sam-ba_cdc_cdc_linux/
```

Now, source your `.bashrc` to load the new definition in your current terminal:

```
source ~/.bashrc
```

We will use the Buildroot demo from Linux4SAM[2].

Let's take the copy of the demo from the USB flash drive and extract it:

```
cp /media/$USER/BootTime/downloads/sama5d3xek-demo.tar.xz .
tar xf sama5d3xek-demo.tar.xz
```

SAM-BA requires an extra Ubuntu package:

```
sudo apt-get install libxss1
```

[2]The original demo was found on `ftp://at91.com/pub/demo/linux4sam_4.0/linux4sam-buildroot-sama5d3xek_linux4sam_4.0.zip`, but it may have changed since the time we prepared these instructions. Using the original copy is a way to make sure that the instructions are not broken by Atmel updates.

If you have a 64 bit system (if the `arch` command returns `x86_64`, SAM-BA also needs the 32-bit libraries:

```
sudo apt-get install ia32-libs
```

Make sure that `JP9` is open to enable booting from the on-chip boot ROM.

To be able to flash the board, you have to stop the CPU in RomBOOT mode. While in this mode, you will be able to use SAM-BA to communicate directly with the CPU. To enter the RomBOOT mode, you have to press and hold `PB4` while resetting the board using `PB1`. `RomBOOT` should appear on your serial console and the board will stop there.

If provided by the instructor, take a second USB to micro USB cable. If you only have one such cable, exit `picocom` and disconnect the one you were using for the serial console, for the time of the flashing operation.

Connect your cable to `J20`, the SAM-BA interface. A device named `/dev/ttyACM1` (or `/dev/ttyACM0` if you disconnected the serial line) should then appear on your system:

```
$ ls -la /dev/ttyACM1
crw-rw---- 1 root dialout 166, 1 Jul  9 16:48 /dev/ttyACM1
```

You are now ready to flash the board using SAM-BA.

```
cd sama5d3xek-demo/
sam-ba /dev/ttyACM1 AT91SAMa5d3x-EK sama5d3xek_demo_linux_nandflash.tcl
```

If everything went smoothly, you should see the `DONE` keyword after a few minutes in your terminal:

```
...
-I- Complete 99%
-I-  Writing: 0x20000 bytes at 0x2420000 (buffer addr : 0x2001052C)
-I-  0x20000 bytes written by applet
-I- === DONE. ===
```

If you are facing trouble or just need more details about how to use and flash the Atmel board, or if you wish to run the same steps from Windows, you should read the `http://www.at91.com/linux4sam/bin/view/Linux4SAM/GettingStarted_a5d3x` page.

Now, remove the USB cable used for SAM-BA. If you have only one cable, connect it again to the serial line and start `picocom` again. Then, reset your board with `PB1`.

In the serial console, you will see your device boot. The first boot will be slow, because of the time needed to generate an SSH key pair. After a few minutes at most, the board will start showing a video on its LCD.

Measuring boot time

Measuring the various components of boot time

Our first goal is to measure boot time in a coarse way. In a second step, we will measure the time spent in the various components of boot time.

Setting your goals

As explained in the lectures, the first thing to do is to choose what do measure. In the demo running on this particular board, a natural choice is to measure the time taken to start the demo video.

Ideally, we would measure the time elapsed since power-on or since a reset event. However, we have no oscilloscope here, and therefore we can only rely on messages sent on the serial port.

- The earliest message we can have on the serial port is the `RomBOOT` one. We will take its time of arrival as the origin of time.
- We will implement a way to send a message to the serial line when the video starts playing.

Measuring overall boot time

We will use `grabserial` to measure the time spent while booting. It prints the timing information of each message received on the serial line[3], and display the time elapsed since a given event (like receiving a special string), used as the origin of time.

You will find `grabserial` on `http://elinux.org/Grabserial`. You can also get it from your USB flash drive:

Download it and put it in your `PATH`:

```
cd
mkdir bin && cd bin
cp /media/$USER/BootTime/downloads/grabserial .
chmod a+x grabserial
```

The command we will be using is:

```
~/bin/grabserial -d /dev/ttyACM0 -m RomBOOT -t -e 30
```

Here are details about this command:

- `-m "RomBOOT"` specifies the string marking the origin of time.
- `-t` makes `grabserial` show the time when **the first character of each line is received**.
- `-e 30` tells `grabserial` to stop measuring after 30 seconds.

[3]Beware that grabserial displays the arrival time of the beginning of a line on a serial port. If a message was sent without a trailing line feed, you could get the impression that the following characters have been received much earlier. This is particularly true when only a whitespace was first sent to the serial line.

First, you will have to exit `picocom` using `C-a C-x` to run `grabserial`.

Start `grabserial` and reset the board using `PB1`. You will see that it takes around 13.5 seconds to reach the Buildroot prompt and show the demo video:

```
$ ~/bin/grabserial -d /dev/ttyACM0 -m RomBOOT  -t -e 30
[0.000002 0.000002] RomBOOT
[0.054440 0.054440]
[0.054594 0.000154]
[0.054719 0.000125] AT91Bootstrap 3.5.2 (Wed Jan 30 18:42:28 CET 2013)
[0.059009 0.004290]
[0.059185 0.000176] 1-Wire: Loading 1-Wire information ...
[0.062645 0.003460] 1-Wire: ROM Searching ... Done, 0x3 1-Wire chips
found
[0.110066 0.047421]
...
[12.158893 0.064820] Starting portmap: done
[12.284482 0.125589] Initializing random number generator... done.
[12.393179 0.108697] ALSA: Restoring mixer settings...
[12.447447 0.054268] Starting network...
[13.077587 0.630140] Starting sshd: OK
[13.467746 0.390159]
[13.470225 0.002479] Welcome to Buildroot
[13.472141 0.001916] buildroot login:
$
```

Note that there's no information on exactly when the first user space code gets executed. We will fix that in the *Add instrumentation* section.

You can make a visual check that the `buildroot login:` message is issued at about the same time as the video starts playing. Of course, the accuracy is about 1 or 2 seconds, and it's time to measure the video starting time in a more accurate way.

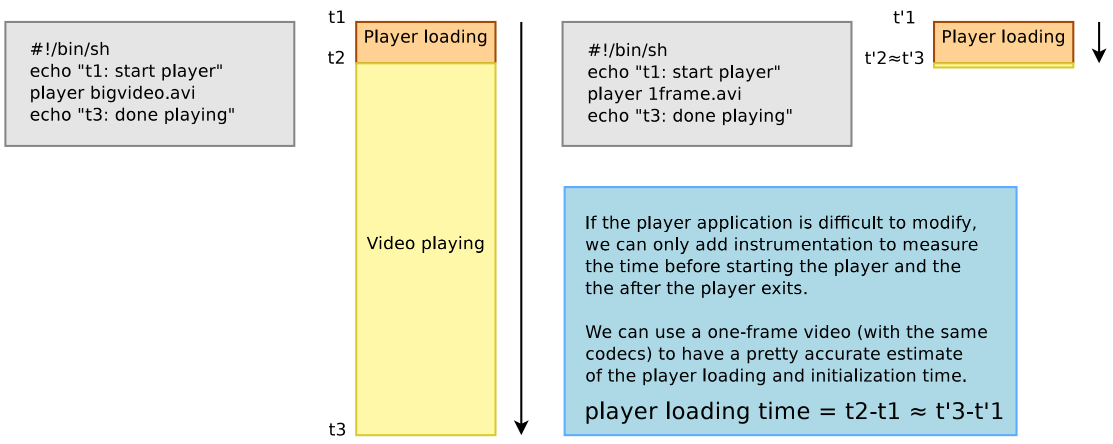
Trick to estimate the video start time

For this kind of need, finding out when a device starts functioning, the ideal solution would be to monitor the LCD screen pins with an oscilloscope, and based on the recorded activity, find out when the video actually starts playing.

When only software tracing is available, a standard solution is to add instrumentation to the application (here the video player) to write a message to the serial line.

In our case, however, our video player is based on GStreamer, a very complex piece of software. We can write a message on the serial line before starting the player application, but the time to load this application and actually start playing the video is far from being negligible. This time is one of the components of boot time that we want to measure.

A trick here is to replace the video by the shortest video possible (with the same codecs), containing only one frame. Assuming that the time to play the video itself is negligible, we can have a pretty accurate measurement of the time to load the video player and start playing the video.


#!/bin/sh
echo "t1: start player"
player bigvideo.avi
echo "t3: done playing"
t1
t2
t3
Player loading
Video playing
#!/bin/sh
echo "t1: start player"
player 1frame.avi
echo "t3: done playing"
t'1
t'2≈t'3
Player loading
If the player application is difficult to modify, we can only add instrumentation to measure the time before starting the player and the the after the player exits.
We can use a one-frame video (with the same codecs) to have a pretty accurate estimate of the player loading and initialization time.
player loading time = t2-t1 ≈ t'3-t'1

Reconstitute the build environment for the root filesystem

In every boot time reduction project, you have to rebuild the root filesystem, either by using the original build system, or by compiling the software stack with a new build system which could make it easier to optimize and simplify the root filesystem.

Another option would have been to make direct changes to the root filesystem. However, what you do is difficult to reproduce, and would make the system more difficult to maintain. In a real production project, it's much better to make changes to the configuration of the build system used to generate the root filesystem.

This way, your changes to reduce boot time are always applied, whatever the other changes you make on the system.

This is also true when you need to add instrumentation. If you do this by hand, you will lose all your instrumentation the next time you have to update or optimize a component.

Install development software

To update and recompile the root filesystem and components like the Linux kernel, you will need to install several software development packages:

```
sudo apt-get install git libncurses5-dev u-boot-tools g++ bison
sudo apt-get install flex gettext texinfo libqt4-dev lzop
```

Also install the `meld` package. It is a very convenient utility to compare different versions of a file, which is often useful in an optimization project.

```
sudo apt-get install meld
```

Now, to be able to update the root filesystem, you need to reconstitute the Buildroot environment that was used to generate it.

Do it by following the instructions available on `http://free-electrons.com/labs/boottime/README.txt`.

Study how user space boots

It's important to understand how the various user space programs are started in the system. They all originate from the `init` program, which is the first and only program executed by the

Linux kernel[4], when it's done booting the device.

The programs that are executed by the `init` program can found by reading the `/etc/inittab` file, which is a configuration file for `init`.

Start `picocom` if needed and study this file. You will see that a `/etc/init.d/rcS` script is also executed, and executes more scripts at his turn. Take some time to read these scripts, until you understand how the video player is started.

Add instrumentation

Make sure you are in the `/opt/boot-time-labs/buildroot/buildroot-at91` directory.

Copy the `../data/1frame.avi` file[5] to `board/atmel/sama5d3ek_demo/`, where the original video is stored.

Now, modify the `board/atmel/sama5d3ek_demo/post-build.sh` file[6]:

- To copy the `1frame.avi` video to `/root` on the target file system. Do this by looking for the instruction that copies the original video, and by using a similar command.
- To modify the contents of the `/root/go.sh` script which starts the video player:
  - To add an `echo -e "\nStarting video player"` command [7] right before invoking `gst-launch`, the video player that the demo uses.
  - To play the `1frame.avi` video instead of the original one.
  - To add an `echo -e "\nDone playing video"` right after playing the video.

The next thing we need is to know when the `init` program starts to execute.

Let's have a look at home this program is built. Run the below command to check the configuration of `BusyBox`, which implements `init`:

```
make busybox-menuconfig
```

Go to the `Init utilities` menu, and disable `Be _extra_ quiet on boot`. Exit and save your new configuration. This way, we will have a message on the serial line when `init` starts to execute.

It's now time to rebuild your root filesystem:

```
make
```

The above command should only take a few seconds to run.

Make sure that the `output/target/root/go.sh` file contains the right modifications and that there are no syntax errors. Update the `board/atmel/sama5d3ek_demo/post-build.sh` file until you get this file right.

If you can't manage to get it right, have a look at the `../solutions/post-build1.sh` file.

[4]There is one exception: the kernel can also run helpers to handle hotplug events.

[5]Here's the command that was used to generate a video containing only the first frame of the original video: `avconv -i MPEG2_480_272.avi -vcodec copy -acodec copy -frames 1 1frame.avi`

[6]This script is called after compiling the root filesystem components, and is used to add custom scripts and files to the system.

[7]We are using a newline (`\n`) at the beginning of the string because `grabserial` only pays attention to the arrival time of the first character in each line. Without this, we would get the wrong time if the current line already had some characters sent earlier. If we didn't use `echo -e` (e: suppport for escape characters), `\n` wouldn't have been considered as a newline.

Re-flash the root filesystem

We will use SAM-BA again to update the root filesystem.

Let's copy our new root filesystem to it to our SAM-BA directory:

```
cp output/images/rootfs.ubi \
/opt/boot-time-labs/flashing/sama5d3xek-demo/
```

Put your board in RomBOOT mode (follow the procedure in the previous chapter), and assuming that the SAM-BA device is /dev/ttyACM1, reflash your board, using the same instructions as before:

```
cd /opt/boot-time-labs/flashing/sama5d3xek-demo/
sam-ba /dev/ttyACM1 AT91SAMa5d3x-EK sama5d3xek_demo_linux_nandflash.tcl
```

Press the reset button and make sure that your board boots as expected. You can also check that the new instrumentation messages are there. Don't pay attention to timing this time, as SSH keys had to be generated from scratch again.

To stop the video player from running in a quick and endless way, you can run the below command in the serial console:

```
killall go.sh
```

We will remove the endless loop a little bit later.

Measure boot time components

You can now exit picocom and run grabserial again.

We can now time all the various steps of system booting in a pretty accurate way. Do it by filling writing down the time stamps in the below table.

Then, in each row, compute the elapsed time, which is the difference between the time stamp for the next step and the time stamp for the current step. For this purpose, you can use the bc -l command line calculator.

Boot phase	Start string	Time stamp	Elapsed
RomBOOT	`RomBOOT`		
Bootstrap	`AT91Bootstrap 3.5.2`		
Bootloader	`U-Boot 2012.10`		
Kernel	`Booting Linux on physical CPU 0`		
Init scripts	`init started: BusyBox v1.21.0`		
Critical application	`Starting video player`		
Critical application ready	`Done playing video`		
Total			

In the next labs, we will work on reducing boot time for each of the boot phases.

Init script optimizations

Analysing 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.

Use bootchartd on the board

Add `bootchartd` to your `BusyBox` configuration[8].

```
cd /opt/boot-time-labs/buildroot/buildroot-at91/
make busybox-menuconfig
make
```

The above command should only take a few seconds to run.

Re-flash the root filesystem

Reflash your root filesystem in the same way you did it in the previous labs. Reboot it as usual to let it generate its SSH keys.

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.

To go to the `u-boot` prompt, go to the `picocom` window showing the boards's serial line, reset your board with `PB1`, and press a key before the timer expires.

First, have a look at the current value of the `bootargs` environment variable:

```
U-Boot> printenv bootargs
bootargs=console=ttyS0,115200
mtdparts=atmel_nand:8M(bootstrap/uboot/kernel)ro,-(rootfs) rw
rootfstype=ubifs ubi.mtd=1 root=ubi0:rootfs
```

Now add the `init` kernel parameter as follows:

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

This will make the system boot and the resulting bootlog will be located in `/var/log/bootlog.tgz`. Copy that file on your host. You can do this by inserting the USB disk provided by your instructor into one of the USB hosts ports of the board:

[8] Hint: to find where `bootchartd` support can be added in the `BusyBox` configuration interface, press the `/` key (search command). This will tell you in which submenu the corresponding option can be found.

```
# mount /dev/sda1 /mnt
# cp /var/log/bootlog.tgz /mnt
# umount /mnt
```

Analyse bootchart data on your workstation

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

```
sudo apt-get install ant openjdk-6-jdk
```

Now, get the `bootchart` source code[9] [10] , compile it and use `bootchart` to generate the boot chart:

```
cd /opt/boot-time-labs/buildroot/
cp /media/$USER/BootTime/downloads/bootchart-0.9.tar.bz2 .
tar xf bootchart-0.9.tar.bz2
cd bootchart-0.9
ant
java -jar bootchart.jar /media/$USER/BootTime/bootlog.tgz
```

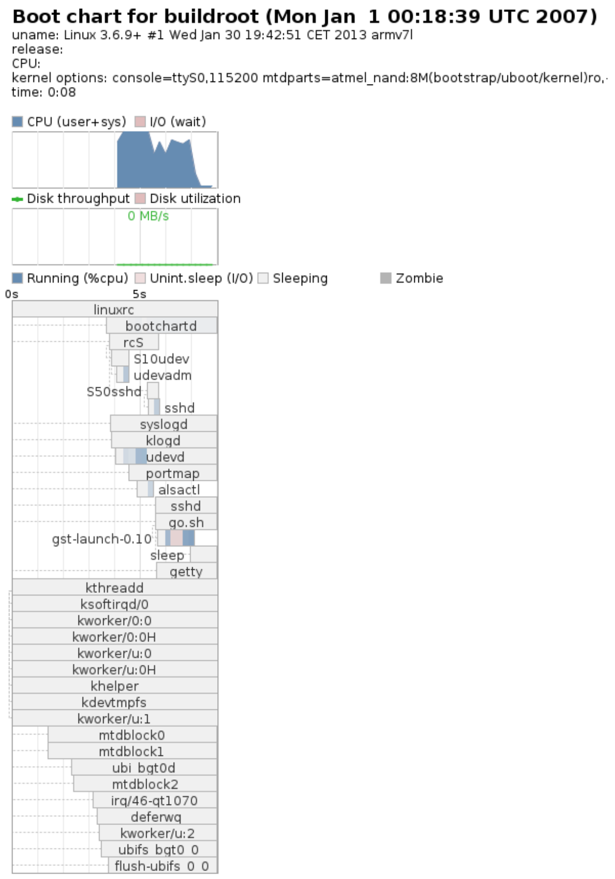
This produces the `bootlog.png` image which you can visualize to study and optimize your startup sequence:

```
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 (Ubuntu, Xubuntu, Kubuntu...), it will run the particular image viewer available in that particular flavor.

[9] The source code was originally found on `http://prdownloads.sourceforge.net/bootchart/bootchart-0.9.tar.bz2`.

[10] 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.


Boot chart for buildroot (Mon Jan 1 00:18:39 UTC 2007)
uname: Linux 3.6.9+ #1 Wed Jan 30 19:42:51 CET 2013 armv7l
release:
CPU:
kernel options: console=ttyS0,115200 mtdparts=atmel_nand:8M(bootstrap/uboot/kernel)ro,·
time: 0:08

Remove unnecessary functionality

The above graph shows that there are `udev` processes running during the startup process, taking a significant amount of CPU time (the graph shows blue rectanges when the processes actually uses the CPU).

`udev` is pretty likely to be unnecessary in this system. Managing device files (even for hotplugged devices) is perfectly taken care of by the kernel's `devtmpfs` filesystem. `udev` might be needed to run specific programs to react to hotplugging events, but it has no visible usefulness in the basic demo usage scenario.

So let's remove `udev`! To do so, disable it in the Buildroot configuration:

```
cd /opt/boot-time-labs/buildroot/buildroot-at91/
make menuconfig
```

In `System configuration` and then in `/dev management`, choose `Dynamic using devtmpfs only`.

The next step would be to re-run Buildroot. However, we're hitting one of the current limitations of Buildroot. To keep its design simple and efficient, Buildroot currently doesn't support removing software even if it has been removed from the configuration. The clean way *would* be to run `make clean` and then run `make` again.

This is usually fine, as Buildroot is pretty fast. In our case however, we made it Buildroot build its own toolchain, and doing this is the most time consuming part. We would lose a lot of time.

So, let's remove `udev` manually instead. Again, this is a quick and dirty trick to save time in a time limited workshop. Don't do this for a production project! It may have unexpected side effects.

```
find output/target/ -name "*udev*"
rm -r output/target/etc/udev/
rm output/target/etc/init.d/S10udev
rm output/target/usr/lib/libudev.so
rm output/target/usr/bin/udevadm
rm -r output/target/lib/udev/
rm output/target/lib/libudev.so.0*
```

Let's also take this opportunity to disable `bootchartd` compilation. Do this through `make busybox-menuconfig`, and remove the `output/target/sbin/bootchartd` file manually.

Rebuild your root filesystem (run `make`) and reflash the device once again. You don't have to worry about removing `bootchartd` from the kernel command line as the flashing process got you back to the initial kernel command line.

After the first reboot (as always for SSH key generation), reboot your board through `grabserial` and measure the time stamp for the `Done playing video` message.

Postpone services

If we get back to the bootchart that we generated, we can see that there are several other services which might be needed for the device to have all its features, but which should be run after the demo video, instead of before it. One example is the SSH server.

Let's customize the `/etc/init.d/rcS` file which starts such services, in order to start the video player first, and the other services later.

First, let's make a copy of what Buildroot generated:

```
cp output/target/etc/init.d/rcS board/atmel/sama5d3ek_demo/
```

Then, add the below lines after the first line in this copy:

```
/root/go.sh &
sleep 5
```

The first added line corresponds to the contents of `/etc/init.d/S99demo`. We remove this executable, because we will continue to use the `S??*` wildcard to start the other services.

According to the measures we have already made, the delay of 5 seconds should be more than enough to let the player play the 1 frame video. With this pessimistic delay, we make sure that the CPU will only be dedicated to the video player for 3 seconds.

Now, let's copy this replacement `/etc/init.d/rcS` file by modifying `board/atmel/sama5d3ek_demo/post-build.sh` once again:

- By removing the code that creates `/etc/init.d/S99demo`
- By coping our new `rcS` file to the target filesystem.

The last thing to do before running `make` is to remove the `output/target/etc/init.d/S99demo` file manually. Remember that Buildroot doesn't clean the target directory!

Now run `make` and make sure that `output/target/etc/init.d/rcS` has the new right contents.

Reflash your system, and reboot it through `grabserial` to measure the new total boot time[11].

Write down your results in the table at the end of this chapter.

While the total boot time is now smaller, you can still notice that the time to run the player is slightly longer. It's not surprising. You can guess that the video player is sharing some libraries (mainly C library components) with the executables for the other services. Since the services are no longer started before itself, it must be the first one to load such libraries, which explains the longer time to start.

Optimize necessary functionality

It's now time to optimize the `go.sh` script that runs the video, by making this script as simple as possible.

Simplify shell programming

First, let's make a simple, one-time experiment that we won't keep. Directly on the board, modify `/root/go.sh`, replacing

```
while `ls /dev/fb0 > /dev/null 2>&1`
```

by

```
while [ -e /dev/fb0 ]
```

This is a much simpler test, with no output to manage. It is even very likely that the test command is a shell built-in, meaning that there is no sub-process to spawn (very costly).

Run `halt` and then reboot the board with `grabserial`, and measure how much time you saved by simplifying this test. In our own tests, we saved 34 ms! This is not negligible at all when similar cases accumulate and you are aiming at booting in just a few seconds.

The lesson to learn is that the shell scripts should be programmed with very good care, to avoid complex constructs causing multiple children processes to be run while fewer would be sufficient.

Radical simplification of the launcher script

Now, let's make `/root/go.sh` as simple as possible:

- By removing the soc subtype check. This check was very complex, and completely unnecessary when you have the video capable SoC.
- By running the video player just once, instead of through an endless `while` loop.

Simplify the contents of this file by removing lines in `board/atmel/sama5d3ek_demo/post-build.sh`.

As usual, update your root filesystem, reflash it, measure the new results and store them in the table below.

[11] Note that we are no longer bothered by SSH key generation, as it starts after our video is played. Therefore, one reboot after reflashing is now sufficient.

Going further

If you are ahead of the others in the workshop, there are other thing you could try to do to start the video player earlier: You could

- Take the commands that are run before `/etc/init.d/rcS` in `/etc/inittab`, and move them after starting the video in `/etc/init.d/rcS`.
- In `/etc/init.d/rcS`, source `/root/go.sh` instead of executing it. This way, this shell script will be interpreted by the current shell, instead of having to spawn a child process (another shell, this is expensive).

Last but not least, there's another thing we can simplify. We could recompile `BusyBox` with only the capabilities that are needed in this demo. It would make the `init` program and the other executables faster to load (being smaller) and run.

However, we prefer to make such a simplification at the end, because it is convenient to interact with a full-featured set of UNIX commands when you experiment with your system.

Results

Fill the below table with the results from your experiments:

Technique	Total time stamp	Difference with previous experiment
Original time		N/A
Remove `udev`		
Postpone services		
Optimize launcher		
Other trick		
Other trick		
Total gain		

Application optimization

Optimize the startup time of your applications

Measuring

The general rule stays the same. You have to measure the time taken to execute the various pieces of code in your application.

Here, except for possible compiler optimization, modifying the application is outside the scope of this workshop, because it requires a good knowledge about the application itself.

However, we are going to use a few techniques which should help you to improve your own application when you are back to real life.

Compiling utilities

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`.

Note that with this version of Buildroot and Linux 3.6.9, we didn't manage to compile the `perf` utility. We will try again when a newer stable kernel is available for this board.

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 it 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 /root/go.sh
```

Also have `strace` generate a summary:

```
strace -c -f -o strace-summary.log /root/go.sh
```

Take some time to read `strace.log`[12] , and look for the time when the program opens the video file.

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

Removing unnecessary functionality

Based on what you learned by tracing and profiling your application, you could recompile it to remove support for the features that you know are not used in your system. This should speed up its execution, at least slightly.

In our case, we could recompile GStreamer with just the features and plugins required to play the exact video format and codec that we have.

Postponing, reordering

In our particular case, modifying `gst-launch` would be very difficult. It could make sense with your own application though, for which the code is familiar to you.

Optimizing necessary functionality

In this particular case, `gst-launch` is far from being the most efficient way of opening the video. It is really meant to help creating a multimedia pipeline. Once it is well defined, the `GStreamer` developers recommend to directly program with the `GStreamer` API[13].

[12] 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 `http://free-electrons.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.

[13] See the details on `http://docs.gstreamer.com/display/GstSDK/Basic+tutorial+10%3A+GStreamer+tools`

Kernel optimizations

Measure kernel boot components and optimize the kernel boot time

Measuring

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

The first thing we need to do is to recompile the kernel to make sure it has the right options.

So, let's get the kernel sources for our system. A source archive is available on your USB disk. We will need them to recompile our kernel in a few minutes anyway.

Plug-in your USB disk and type the below commands:

```
cd /opt/boot-time-labs/kernel
cp /media/$USER/BootTime/downloads/linux-3.6.9-at91.tar.xz .
tar xf linux-3.6.9-at91.tar.xz
source env.sh
cd linux-3.6.9-at91
make sama5d3_defconfig
make xconfig
```

Here are the contents of the `env.sh` file:

```
export ARCH=arm
export CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabi-
export PATH=$PATH:/opt/boot-time-labs/buildroot/buildroot-at91/output/host/usr/bin/
```

Now configure the kernel with the below settings:

- `CONFIG_LOG_BUF=16`. That's the size of the kernel ring buffer. If we keep the default size (14), the earliest kernel messages are overwritten by the latest ones.
- Make sure that `CONFIG_PRINTK_TIME` is enabled (it should already be)

Save your configuration. We are ready to compile our kernel:

```
make -j 8 uImage
```

Keep a backup copy of your original kernel image, and copy the new one[14]:

```
cd ../../flashing/sama5d3xek-demo
cp uImage uImage.orig
cp ../../kernel/linux-3.6.9-at91/arch/arm/boot/uImage .
```

Reflash your system and make it boots in the same way as before.

Now, let's enable `initcall_debug`. Reboot your board and press a key to stop the U-boot countdown.

[14]So far, we are lucky to have a kernel with most drivers compiled inside the kernel. If we had important drivers compiled as modules, we would need to compile the kernel with Buildroot, to put the updated modules in the root filesystem. We are also lucky not to have to update the Device Tree Binary, as we are using the same kernel sources as the demo.

In the U-boot command line, add settings to the kernel command line[15] and boot your system:

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

Once booted, you can store the debug information in a file:

```
dmesg > initcall_debug.log
```

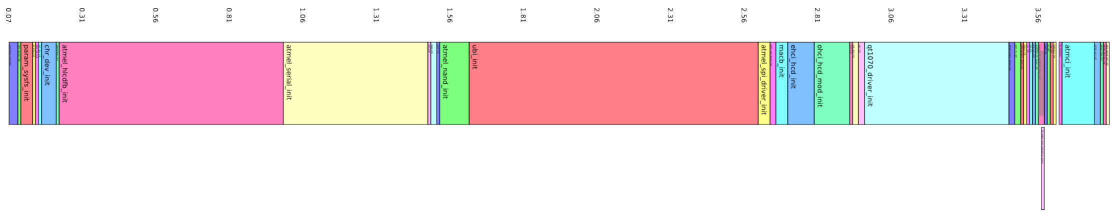
Copy this file to the USB disk.

Let's use the kernel script to generate a nice boot graph from this debug information:

```
cd /opt/boot-time-labs/kernel
cp /media/$USER/BootTime/initcall_debug.log .
perl linux-3.6.9-at91/scripts/bootgraph.pl initcall_debug.log > boot.svg
```

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

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



Now review the longuest 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[16], and what each driver corresponds to.

Then, you can look the source code and 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).

Before going on, reboot your board through `grabserial` to measure the total boot time. The kernel rebuild could have modified it a little bit (in case the Atmel people preparing the demo didn't use exactly the same toolchain or kernel options as we do). Write down your result at the end of this chapter.

Reordering and postponing functionality

The boot graph revealed drivers which could be compiled as modules and loaded later. Let's compile such drivers that you found as modules and install the corresponding modules in the root filesystem.

Because of this requirement, we now need to compile the kernel with Buildroot.

First let's prepare a kernel configuration in which the selected drivers are now compiled as modules:

```
cd /opt/boot-time-labs/kernel/linux-3.6.9-at91/
make xconfig
```

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

[16] 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: http://elixir.free-electrons.com

For each driver, you have to look for the parameter which enables it. If the parameter name is not trivial to find in the kernel configuration interface, already knowing the source file implementing the function in the boot chart, you can look at the `Makefile` file in the same directory, and find which parameter is used to compile the source file in a conditional way.

Once you are done converting static drivers into modules, copy your configuration file:

```
cp .config ../config-3.6.9-at91-with-modules
```

Now go back to the Buildroot directory:

```
cd ../../buildroot/buildroot-at91
make menuconfig
```

Go to the `Kernel` menu, and enable `Linux Kernel`. Now fill the kernel related settings:

- `Kernel version`: `Custom tarball`
- `URL of custom kernel tarball`: `http://free-electrons.com/labs/boottime/downloads/linux-3.6.9-at91.tar.xz`
- `Kernel configuration`: `Using a custom config file`
- `Configuration file path`: `/opt/boot-time-labs/kernel/config-3.6.9-at91-with-modules`
- Enable `Device tree support`
- `Device tree source`: `Use a device tree present in the kernel`
- `Device Tree Source file names`: `sama5d34ek`

To save download time, put the Linux sources in the Buildroot download directory. Then run Buildroot:

```
cp ../../kernel/linux-3.6.9-at91.tar.xz dl/
```

Check that your modules have been added to the root filesystem:

```
find output/target -name *.ko
```

The last thing to do is to load the necessary modules in a manual way[17] from the `/etc/init.d/rcS` file, before starting the services (SSH will need the network driver to be loaded, for example).

Modify the `board/atmel/sama5d3ek_demo/rcS` file to run the `modprobe` command for each of the modules before starting the services (example: `modprobe macb`).

Rerun Buildroot. Before reflashing your device, this time do not only copy `output/images/rootfs.ubi` but also `output/images/uImage`! Reflash your device, measure the new boot time and write it down at the end of this chapter.

Removing unnecessary functionality

The boot graph that we generated doesn't show any obvious kernel driver that would consume a significant amount of time and could be taken away because it is completely useless.

Of course, there will be kernel features that we will be able to remove, in order to reduce the kernel size and make the kernel faster to load in the bootloader. However, this shouldn't have much impact on the kernel's execution time.

[17] Oops, we don't have `udev` any more, and it would have done it for us. However, manually loading modules is no big deal, and a simpler solution for hotplug events can still be put in place anyway.

There's one thing we can remove though, and didn't appear on the boot graph: we can disable console output. Writing messages on the serial line can be very slow, especially as the serial line has a slow bandwidth.

You can do this by adding the `quiet` parameter to the kernel command line. Since we reflash the device frequently, let's store the new setting in the flashing script.

Look for the `bootargs` setting in the `sama5d3x_demo_linux_nandflash.tcl` file and add the `quiet` parameter.

Reflash your device, measure boot time, and write in down in the summary table.

There is another thing that is unnecessary too: the calibration of the delay loop, as explained in the lectures. Read the `lpj` value from a previous boot log, and pass this value on the kernel command line.

Measure the new boot time and write your result in the summary table.

Optimizing necessary functionality

The boot graph revealed the existence of drivers with initcalls taking a long time to execute. It would be worth spending time analysing their code, looking for opportunities to reduce the initialization time taken by these drivers.

However, such investigation work could take days, unless you find obvious issues (such as big delay loops).

Results

Fill the below table with the results from your experiments:

Technique	Total time stamp	Difference with previous experiment
Original time		N/A
Postpone functionality (drivers compiled as modules)		
Remove unnecessary functionality (`quiet` option)		
Remove unnecessary functionality (skip delay loop calibration)		
Total gain		

Bootloader optimizations

Reduce boot time by using and optimizing Barebox

Switching to Barebox

We could have applied the same methodology as in the previous labs, and started to reduce boot time by measuring all the U-boot steps, by making it simpler and by tuning its options.

However, there's another bootloader available, called Barebox. It supports an increasing number of hardware platforms, in particular the Atmel based ones. One of its strengths is that it can copy the kernel image with the CPU RAM caches enabled, allowing for faster copy to RAM and execution from it.

Let's download the sources for Barebox:[18]

```
cd /opt/boot-time-labs/barebox
cp /opt/BootTime/downloads/barebox-2013.08.tar.xz .
tar xf barebox-2013.08.tar.xz
```

We also apply some patches providing configuration and environments files that reduce the Barebox features to a minimum:

```
cd barebox-2013.08
cat ../patches/*.patch | patch -p1
```

We can now configure and compile Barebox:

```
source ../../kernel/env.sh
make sama5d3xek_fast_defconfig
make
```

You can flash the generated image `arch/arm/pbl/zbarebox.bin` to replace `u-boot`. This time you will have to use the `zImage` instead of the `uImage` for the kernel. Edit the `sama5d3x_demo_linux_nandflash.tcl` script to do that.

Optimizing the kernel

Try to remove as much as possible from the kernel configuration to be faster.

Measure your new boot time.

[18] Once again, the bootloader sources can also be found in the USB disk provided by the instructor. See also http://free-electrons.com/labs/boottime/README.txt for details about where we got these sources from.

Appendix - Results

Results obtained in the various steps

Initial boot time components

Here are the results we got after making the initial measurements:

Boot phase	Start string	Time stamp	Elapsed
RomBOOT	`RomBOOT`	0.000000	55 ms
Bootstrap	`AT91Bootstrap 3.5.2`	0.054626	978 ms
Bootloader	`U-Boot 2012.10`	1.032380	7,720 ms
Kernel	`Booting Linux on physical CPU 0`	8.752495	2,795 ms
Init scripts	`init started: BusyBox v1.21.0`	11.547943	1,890 ms
Critical application	`Starting video player`	13.438071	1,244 ms
Critical application ready	`Done playing video`	14.682485	
Total			14,682 ms

The original boot log is available in `/opt/boot-time-labs/measuring/boot-instrumented1.log`.

Init script optimizations

Results

Here are the results we obtained from our own experiments:

Technique	Total time stamp	Difference with previous experiment
Original time	14.682485	N/A
Remove `udev`	13.667242	-1,015 ms
Postpone services	13.003734	-666 ms
Optimize launcher	12.920196	-84 ms
Other trick		
Other trick		
Total gain		1,765 ms

The original boot logs are available in `/opt/boot-time-labs/init-scripts/`.

Kernel optimizations

Measuring

Here are the longest initcalls that we discovered on the boot graph:

- `atmel_hlcdfb_init` (about 800 ms!): obviously the framebuffer driver for the Atmel's LCD controller. A quick look at the source code `drivers/video/atmel_hlcdfb.c` doesn't reveal any obvious delay loop and anything pathological. There is no module parameter either which could be used to shorten probe time.

- `atmel_serial_init` (about 500 ms!): obviously Atmel's serial driver. No obvious issue found in the code. Another driver that needs optimizing!
- `atmel_nand_init` (about 100 ms). Nothing suspicious found. May be scanning for bad blocks though (not immediately found in the code).
- `ubi_init` (about 1000 ms!). That's the longuest one. This corresponds to the time running the `ubi_attach` functionality. A *UBI fastmap* feature was added in Linux 3.7 to address this issue, but our kernel version doesn't have it yet. A solution would be to switch to a newer stable kernel when available. At least, you can't remove this one as long as the rootfilesystem (in a UBI partition) depends on it.
- `macb_init` (Ethernet driver), `ehci_hcd_init` and `ohci_hcd_mod_init` (USB host controller drivers): the corresponding devices are not used in the boot sequence. We can build their drivers as modules and load them later.
- `qt1070_driver_init` (about 500 ms)! Reading the code reveals that this is a driver for the touch keys on the left hand of the LCD (`K1` to `K2` keys). Another driver which would be worth optimizing, but which could be built as a module.
- `atmci_init` (about 100 ms). That's the Atmel Multimedia Card (MMC) interface. It can also be build as a module.

Postponing, compiling some drivers as modules

Thanks to the boot graph highlighting them, here are the kernel configuration settings that we modified, to compile some drivers as modules: `CONFIG_MACB, CONFIG_USB_OHCI_HCD, CONFIG_USB_EHCI_HCD, CONFIG_KEYBOARD_QT1070`.

Results

Here are the results we obtained from our own experiments:

Technique	Total time stamp	Difference with previous experiment
Original time	13.252827	N/A
Postpone functionality (drivers compiled as modules)	12.302508	-950 ms
Remove unnecessary functionality (`quiet` option)	11.166088	-1.136 ms
Remove unnecessary functionality (skip delay loop calibration)	Not tested	
Total gain		-2,087 ms