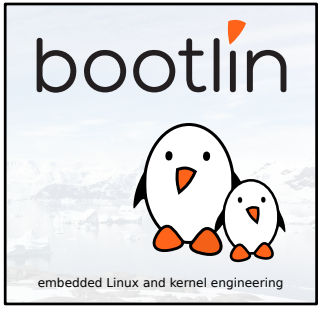
Embedded Linux boot time optimization training



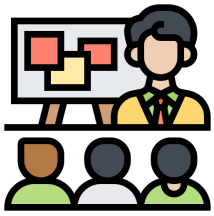
Latest update: February 24, 2023.

Document updates and training details:
https://bootlin.com/training/boot-time

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

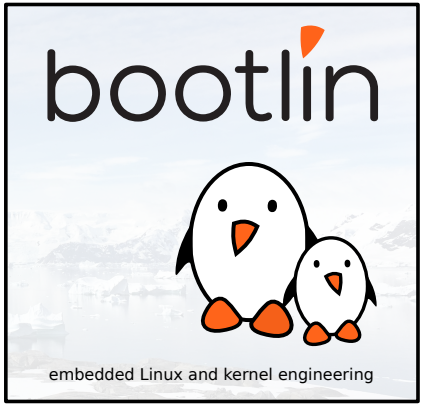
Embedded Linux boot time optimization training

- These slides are the training materials for Bootlin's *Embedded Linux boot time optimization* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/boot-time
- Contact: training@bootlin.com



Icon by Eucalyp, Flaticon

About Bootlin



Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

bootlin

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux boot time optimization On-site: 3 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 8000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 5000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- Twitter:
  https://twitter.com/bootlincom
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

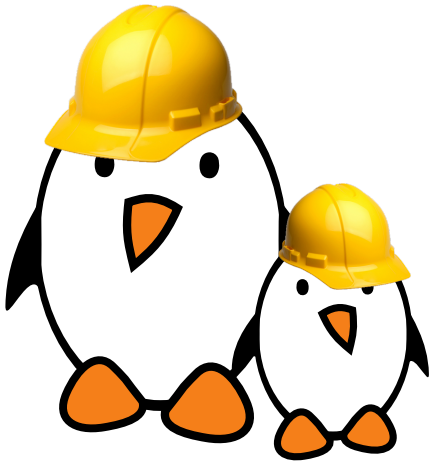
Icon by Freepik, Flaticon

Thanks

Special thanks to

- Zuehlke Engineering (Serbia)
  - For funding a major update to these materials and further development (2 days now)
- Microchip (formerly Atmel Corporation)
  - For funding the development of the first version of these materials (1 day course)

Practical lab - Lab setup and downloading sources



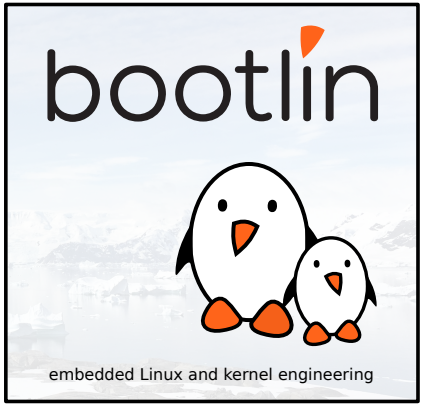
Prepare your lab environment

- Download and extract the lab archive

Start cloning source trees right away

- U-Boot
- Linux kernel
- Buildroot

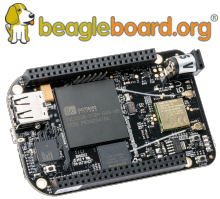
Generic course information



Supported hardware

BeagleBone Black or BeagleBone Black Wireless, from BeagleBoard.org

- Texas Instruments AM335x (ARM Cortex-A8 CPU)
- SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 4 GB of on-board eMMC storage
- USB host and USB device, microSD, micro HDMI
- WiFi and Bluetooth (wireless version), otherwise Ethernet
- 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards, called *capes*. See https://elinux.org/Beagleboard:BeagleBone_Capes.



The full system

- Beagle Bone Black board (of course). The Wireless variant should work fine too.
- Beagle Bone 4.3" LCD cape from Element 14
  https://www.element14.com/bbcape43
- Standard USB webcam (supported through the `uvcvideo` driver).

Shopping list: basic hardware for this course

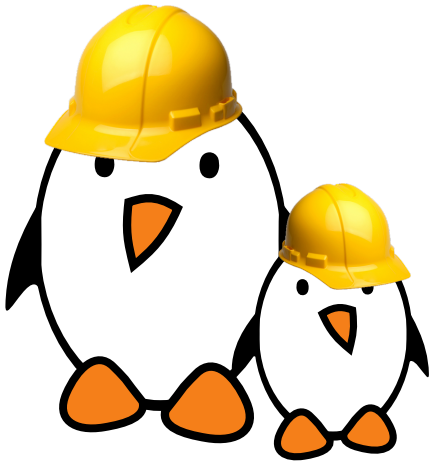
- BeagleBone Black or BeagleBone Black Wireless - Multiple distributors:
  See https://beagleboard.org/Products/.
- 5V power supply, at least 2A, for the BeagleBone Black, with a 5.5 mm barrel jack connector. Needed to drive the LCD cape!
  https://www.olimex.com/Products/Power/SY1005E/
- USB Serial Cable - 3.3 V - Female ends (for serial console):
  https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/
- Beagle Bone LCD4.3 cape from Element 14
  https://www.element14.com/bbcape43
- A standard micro SD card - 1 GB or more
- A faster micro SD card - 1 GB or more

Shopping list: optional hardware for this course

If you are interested in doing the optional hardware measurement lab

- A handful of breadboard wires (at least 10 to have sufficient different colors) [1]
- Arduino Nano board (or a clone), with its USB power cable
  https://store.arduino.cc/arduino-nano
- A breadboard large enough to plugin the Arduino Nano
  https://www.olimex.com/Products/Breadboarding/BREADBOARD-1/
- A four digit display based on the TM1637 chip
  https://www.seeedstudio.com/Grove-4-Digit-Display.html
- A 1,000 to 10,000 Ohm resistor for use as a pull-down resistor

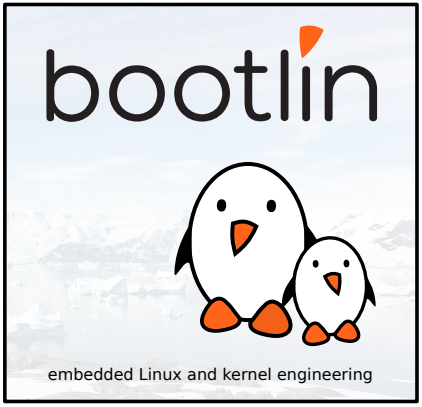
Practical lab - Board setup



Prepare your board

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

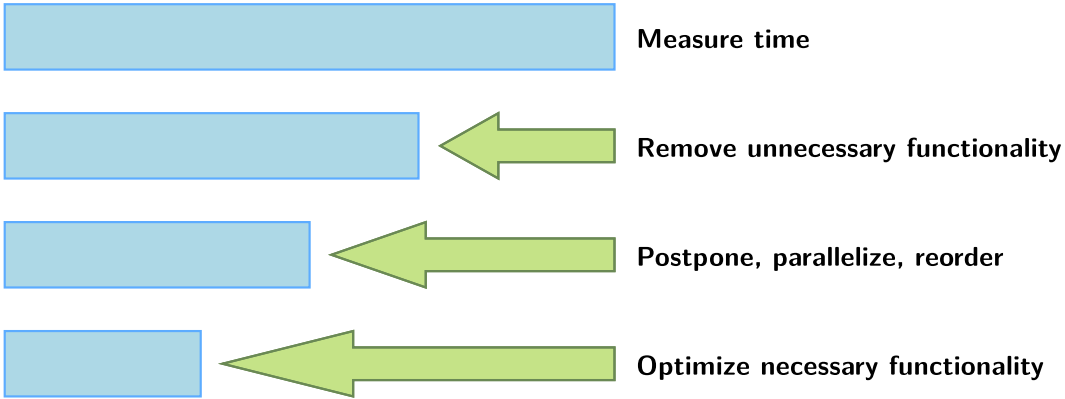
Principles



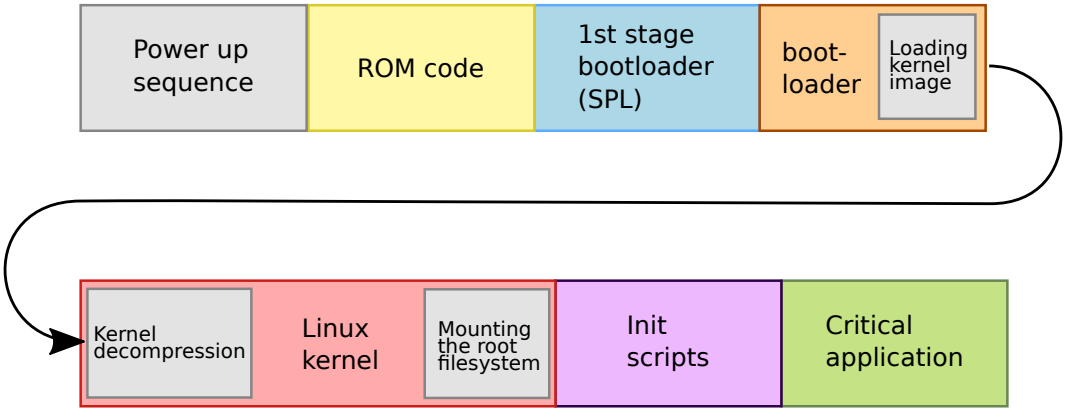
Set your goals

- Reducing boot time implies measuring boot time!
- You will have to choose reference events at which you start and stop counting time.
- What you choose will depend on the goal you want to achieve. Here are typical cases:
  - Showing a splash screen or an animation, playing a sound to indicate the board is booting
  - Starting a listening service to handle a particular message
  - Being fully functional as fast as possible

Boot time reduction methodology

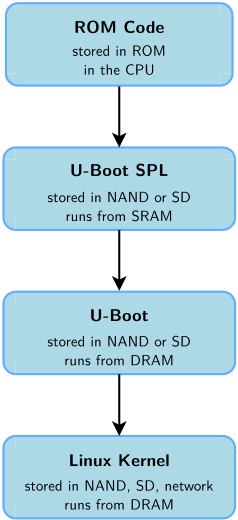


Boot time components



We are focusing on reducing *cold* boot time, from power on to the critical application.

Booting on ARM TI OMAP2+ / AM33xx



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM. Size limited to <128 KB (SRAM size on AM3358). No user interaction possible.
- **X-Loader** or **U-Boot SPL**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into DRAM and starts it. No user interaction possible. File called `MLO` (*Mmc LOader*).
- **U-Boot**: runs from DRAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to DRAM and starts it. Shell with commands provided. File called `u-boot.bin` or `u-boot.img`.
- **Linux Kernel**: runs from DRAM. Takes over the system completely (bootloaders no longer exists).

What to optimize first

Start by optimizing the **last steps** of the boot process!

- Don't start by optimizing things that will reduce your ability to make measurements and implement other optimizations.
- Start by optimizing your applications and startup scripts first.
- You can then simplify BusyBox, reducing the number of available commands.
- The next thing to do is simplify and optimize the kernel. This will make you lose debugging and development capabilities, but this is fine as user space has already been simplified.
- The last thing to do is implement bootloader optimizations, when kernel optimizations are over and when the kernel command line is frozen.

We will follow this order during the practical labs.

Worst things first and measurement methodology

Premature optimization is the root of all evil.
Donald Knuth

- Taking the time to measure time carefully is important.
  - Advice to make at least 3 measures for each configuration you want to measure.
  - Pay attention to variations between measures. Measures are only valuable when there is a low jitter between them.
  - Keep copies of all your logs. Always useful to double check or analyze measures which are inconsistent with the others.
- Find the worst consumers of time and address them first.
- You can waste a lot of time if you start optimizing minor spots first.

Build automation

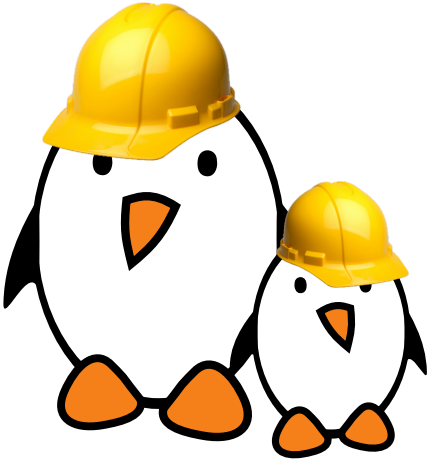
- Build automation is a very important part of a successful project.
- So, through the build system, you should automate any remaining manual step and all the new optimizations that you will implement to reduce boot time. Without such automation, you may forget some optimizations, or introduce new bugs when making further optimizations.
- Boot time optimization projects required countless rebuilds too, automating image generation will save a lot of time too.
- Kernel and bootloader compiling and optimizations can also be taken care of by the build system, though the need is less critical.

Generic ideas

Some ideas to keep in mind while trying to reduce the boot time:

- The fastest code is code that is not executed
- A big part of booting is actually loading code and data from the storage to RAM. Reading less means booting faster. I/O are expensive!
- The root filesystem may take longer to mount if it is bigger.
- So, even code that is not executed can make your boot time longer.
- Also, try to benchmark different types of storage. It has happened that booting from SD card was actually faster than booting from NAND.

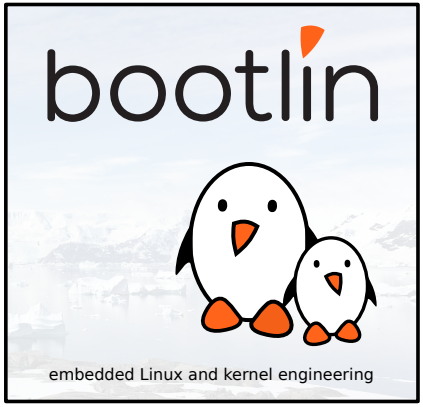
Practical lab - Build and boot the system



Compile your system components and get your system up and running

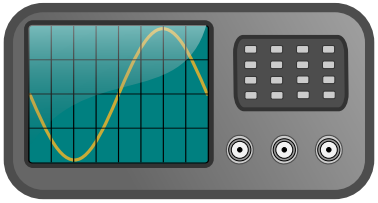
- Compile the root filesystem with Buildroot
- Compile, install and run the bootloader (U-Boot)
- Compile and install the Linux kernel
- Get the full system up and running

Measuring



Time measurement equipment: hardware

- The best equipment is an oscilloscope, if you can afford one
- Allows to time the "Power on" event (connected to a power rail), or any event (connected to a GPIO pin, for example), all this in a very accurate way.
- Easy to write to a GPIO at all the stages of system booting (we will explain how to do it)
- Some oscilloscopes are getting affordable. Example: Bitscope Pocket Analyzer (245 AUD, supported on Linux, https://www.bitscope.com/product/BS10/)



Measuring with hardware: using an Arduino

https://arduino.cc

- If you don't have an oscilloscope, an Arduino (or any general purpose MCU or MPU board) is a great solution too.
- The main strength of Arduino is its great ease of use and programming, plus all the hardware support libraries that are available.
- You can easily connect board pins to the Arduino analog pins, and watch their voltage.
- In our labs, we will use Arduino Nano boards for measuring the whole boot time.
- Arduino boards are Open Source Hardware. This project is definitely worth supporting!

Arduino Nano

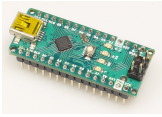
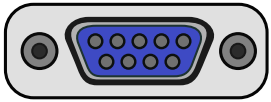


Image credits: https://frama.link/wdhQENrp (Wikimedia Commons)

Time measurement equipment: serial port

- Useful when you don't have monitoring hardware, or don't want to make take any risk connecting wires to the hardware.
- Usually relies on software which times messages received from the board's serial port (serial port absolutely required). Such software runs on a PC connected to the serial port.
- On the board, requires a real serial port (directly connected to the CPU), immediately usable from the earliest parts of the boot process. Attaching a USB-to-serial dongle to a USB **host** port on the device won't do: USB is available much later and messages go through more complex software stacks (loss of time accuracy).
- Limitation: won't be able to time the "Power on" event in an accurate way. But acceptable as you can assume that the time to run the ROM code is constant.

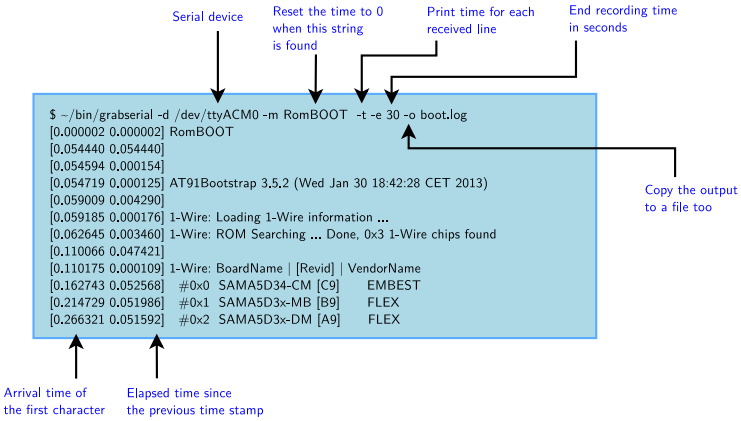


grabserial

https://elinux.org/Grabserial (by Tim Bird)

- A Python script to add timestamps to messages received on a serial console.
- Key advantage: starts counting very early (ROM code — if not silent, bootstrap and bootloader)
- Another advantage: no overhead on the target, because run on the host machine.
- Drawbacks: may not be precise enough. Can't measure power up time.
- Ubuntu package: `grabserial`
  Otherwise available on https://github.com/tbird20d/grabserial/

Using grabserial



Caution: `grabserial` shows the arrival time of the **first character** of a line. This doesn't mean that the entire line was received at that time.

grabserial tips

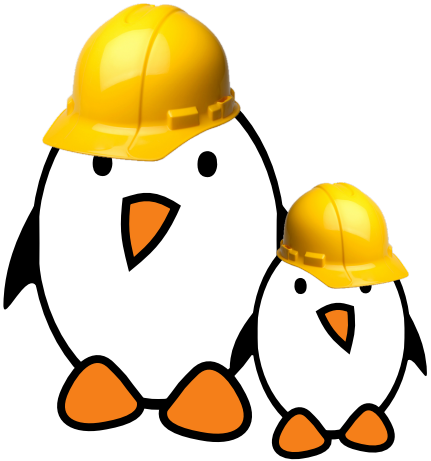
- You can interrupt `grabserial` manually (with `[Ctrl][c]`) when you have gone far enough.
- The `-m` (**m**atch start pattern) and `-q` (**q**uit pattern) options actually expect a regular expression. A normal string may not match in the middle of a line.
- Example: you may have to replace `-m "Starting kernel"` by `-m ".*Starting kernel.*"`.
- You can store a copy of the output to a file using the `-o` option. No need to copy / paste or redirect the output to keep it.

Dedicated measuring resources

Later, we will see specific resources for measuring time

- `time` for measuring application time
- `strace` for application tracing
- `bootchartd` for measuring and tracing the execution of system services.
- More specifically, `systemd-analyze` if your system is started with *Systemd*.
- `CONFIG_PRINTK_TIME` and `initcall_debug` for tracing and timestamping kernel code and functions.

Practical lab - Measuring time



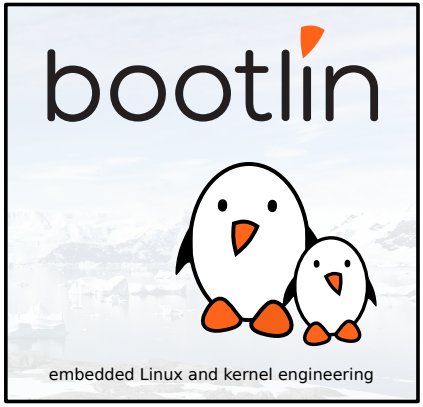
Measuring time with software

- Setting up `grabserial`
- Modify the video player to log a notification after the first frame is processed.
- Time the various components of boot time through messages written to the serial console.

Measuring time with hardware

- Setting up an Arduino system
- Timing reset on Beagle Bone Black
- Modifying the video player to toggle a GPIO after the first frame is processed.
- Display the live total time through a 7-segment display

Toolchain optimizations



Best toolchain for your project

Optimizing the cross-compiling toolchain is typically one of the first things to do:

- The benefits of a toolchain change will be more significant and easier to measure if other optimizations haven't been done yet.
- Here's what you can change in a toolchain, with a potential impact on boot time, performance and size:
  - Components: versions of *gcc* and *binutils*
    More recent versions can feature better optimization capabilities.
  - C library: *glibc*, *uClibc*, *musl*
    uClibc and *musl* libraries make a smaller root filesystem
  - Instruction set variant: *ARM* or *Thumb2* (on 32 bit only), *Hard Float* support or not. Can have an impact on code performance and code size.
    - *Thumb2*, available only on ARM 32, encodes the same instructions as *ARM* but in a more compact way, at least significantly reducing size.
    - *ARM EABIhf*, in addition to being more efficient, also reduces code size compared to *ARM EABI*, but only on binaries doing floating point computation. For example, `libavcodec` size is only reduced by 4K (-0.03%). That's negligible.

Choosing the C library

- The C library is hardcoded at toolchain creation time
- Available C libraries:
  - *glibc*: most standard and featureful
  - *uClibc*: smaller and configurable. Has been around for about 20 years.
  - *musl*: an alternative to *uClibc*, developed more recently but mature too.

glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- By default, quite big for small embedded systems. On armv7hf, version 2.31: `libc`: 1.5 MB, `libm`: 432 KB, source: https://toolchains.bootlin.com
- https://www.gnu.org/software/libc/

Image: https://bit.ly/2EzHl6m

uClibc-ng

- https://uclibc-ng.org/
- A continuation of the old uClibc project, license: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface.
  - Supports most embedded architectures, including MMU-less ones (ARM Cortex-M, Blackfin, etc.). The only library supporting ARM noMMU.
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Some features may be implemented later than on glibc (real-time, floating-point operations...)
  - Focus on size (RAM and storage) rather than performance
  - Size on armv7hf, version 1.0.34: `libc`: 712 KB, source: https://toolchains.bootlin.com
- Actively supported, but Yocto Project stopped supporting it.

musl C library

https://www.musl-libc.org/

- A lightweight, fast and simple library for embedded systems
- Created while uClibc's development was stalled
- In particular, great at making small static executables, which can run anywhere, even on a system built from another C library.
- More permissive license (MIT), making it easier to release static executables. We will talk about the requirements of the LGPL license (glibc, uClibc) later.
- Supported by build systems such as Buildroot and Yocto Project.
- Used by the Alpine Linux lightweight distribution (https://www.alpinelinux.org/)
- Size on armv7hf, version 1.2.0: `libc`: 748 KB, source: https://toolchains.bootlin.com

glibc vs uclibc-ng vs musl - small static executables

Let's compile and strip a `hello.c` program **statically** and compare the size

- With musl 1.2.0:
  9,084 bytes
- With uclibc-ng 1.0.34 :
  21,916 bytes.
- With glibc 2.31:
  431,140 bytes

Tests run with `gcc 10.0.2` toolchains for `armv7-eabihf`
(from https://toolchains.bootlin.com)

glibc vs uclibc vs musl - more realistic example

Let's compile and strip BusyBox 1.32.1 **statically**
(with the `defconfig` configuration) and compare the size

- With musl 1.2.0:
  1,176,744 bytes
- With uclibc-ng 1.0.34 :
  1,251,080 bytes.
- With glibc 2.31:
  1,852,912 bytes

Notes:

- Tests run with `gcc 10.0.2` toolchains for `armv7-eabihf`
- BusyBox is automatically compiled with `-Os` and stripped.
- Compiling with shared libraries will mostly eliminate size differences

Other smaller C libraries

- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- Choices:
  - Newlib, https://sourceware.org/newlib/, maintained by Red Hat, used mostly in Cygwin, in bare metal and in small POSIX RTOS.
  - Klibc, https://en.wikipedia.org/wiki/Klibc, from the kernel community, designed to implement small executables for use in an *initramfs* at boot time.

Advise for choosing the C library

- Advice to start developing and debugging your applications with *glibc*, which is the most standard solution, and is best supported by debugging tools (*ltrace* not supported by *musl* in Buildroot, for example).
- Then, when everything works, if you have size constraints, try to compile your app and then the entire filesystem with *uClibc* or *musl*.
- If you run into trouble, it could be because of missing features in the C library.
- In case you wish to make static executables, *musl* will be an easier choice in terms of licensing constraints. The binaries will be smaller too. Note that static executables built with a given C library can be used in a system with a different C library.

Time your commands using the time command

First run

```
> time ffmpeg ...
real 0m 2.06s
user 0m 0.17s
sys 0m 0.26s
```

← Total observed time

← Time in userspace (running the program and shared libs)

← Time in kernel space (accessing files, accessing device data)

Second run (program and libraries already in file cache)

```
> time ffmpeg...
real 0m 0.66s
user 0m 0.17s
sys 0m 0.25s
```

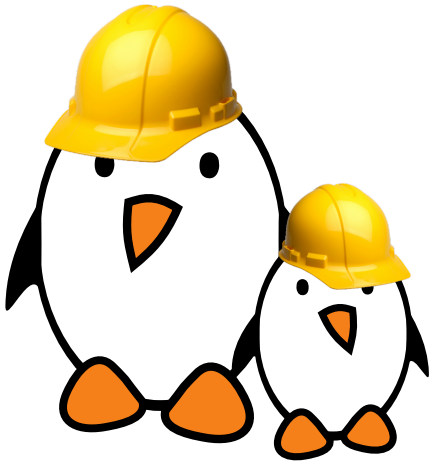
← Less waiting time!

real = user + sys + waiting time (at least on single core machines)

Your program cannot run faster than user + sys (unless you optimize the code)

This gives you the best time that can possibly be achieved (with the fastest storage).

Practical lab - Toolchain optimizations



- Measure filesystem and `ffmpeg` binary size. Time the execution of the application.
- Re-compile the root filesystem using a Thumb2 toolchain
- Re-compile the root filesystem with the *Musl* C library instead of *uClibc*
- Find the best toolchain in terms of size and execution time.
- Have Buildroot generate an external toolchain (*SDK*)

Lessons from labs: ARM vs Thumb2 (32 bit only)

- Testcase: root filesystem with `ffmpeg` and associated libraries (from our training labs), with uClibc
- Compiled with gcc 10.3, generating *ARM* code:
  Total filesystem size: 17.9 MB
  `ffmpeg` size: 239 KB
- Compiled with gcc 10.3, generating *Thumb2* code:
  Total filesystem size: 14.5 MB (-19 %)
  `ffmpeg` size: 191 KB (-20 %)
- Performance aspect: performance apparently slightly improved with Thumb2 (about 2 %, but there are slight variations in measured execution time, for one run to another).

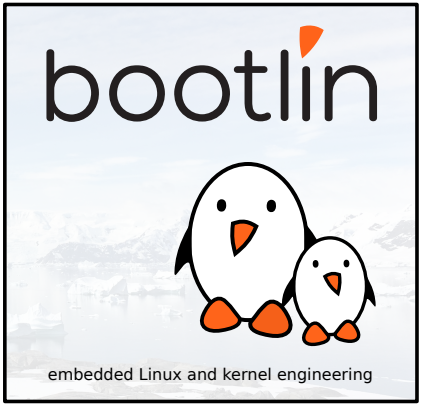
Lessons from labs: musl vs uClibc

Replacing *uClibc* by *musl* in our video player lab, keeping *Thumb2*. Here are data from an earlier run of our labs:

- Total system size with *uClibc*: 14.3 MB
- Total system size with *Musl*: 14.4 MB
- uClibc saves 80 KB (useful), but otherwise no other significant change in filesystem and code size. Not a surprise when the system is mostly filled with binaries relying on shared libraries.

Switching to *Musl* as it is supposed to allow for smaller static binaries, which will be useful in later labs.

Optimizing applications



Measuring: strace

- Allows to trace all the system calls made by an application and its children.
- Useful to:
  - Understand how time is spent in user space
  - For example, easy to find file open attempts (`open()`), file access (`read()`, `write()`), and memory allocations (`mmap2()`). Can be done without any access to source code!
  - Find the biggest time consumers (low hanging fruit)
  - Find unnecessary work done in applications and scripts. Example: opening the same file(s) multiple times, or trying to open files that do not exist.
- Limitation: you can't trace the `init` process!

System call tracer - https://strace.io

- Available on all GNU/Linux systems
  Can be built by your cross-compiling toolchain generator or by your build system.
- Allows to see what any of your processes is doing: accessing files, allocating memory... Often sufficient to find simple bugs.
- Usage:
  `strace <command>` (starting a new process)
  `strace -f <command>` (**f**ollow child processes too)
  `strace -p <pid>` (tracing an existing process)
  `strace -c <command>` (time statistics per system call)

See the strace manual for details.

Image credits: https://strace.io/

strace example output

```
> strace cat Makefile
[...]
fstat64(3, {st_mode=S_IFREG|0644, st_size=111585, ...}) = 0
mmap2(NULL, 111585, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f69000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320h\1\0004\0\0\0\344"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442180, ...}) = 0
mmap2(NULL, 1451632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e06000
mprotect(0xb7f62000, 4096, PROT_NONE) = 0
mmap2(0xb7f66000, 9840, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3) = 0
[...]
openat(AT_FDCWD, "Makefile", O_RDONLY)  = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=173, ...}, AT_EMPTY_PATH) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7290d28000
read(3, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 131072) = 173
write(1, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 173ifneq ($(KERNELRELEASE),)
```

Hint: follow the open file descriptors returned by `open()`. This tells you what files are handled by further system calls.

strace -c example output

```
> strace -c cheese
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 36.24    0.523807          19     27017           poll
 28.63    0.413833           5     75287       115 ioctl
 25.83    0.373267           6     63092     57321 recvmsg
  3.03    0.043807           8      5527           writev
  2.69    0.038865          10      3712           read
  2.14    0.030927           3     10807           getpid
  0.28    0.003977           1      3341        34 futex
  0.21    0.002991           3      1030       269 openat
  0.20    0.002889           2      1619       975 stat
  0.18    0.002534           4       568           mmap
  0.13    0.001851           5       356           mprotect
  0.10    0.001512           2       784           close
  0.08    0.001171           3       461       315 access
  0.07    0.001036           2       538           fstat
...
```

ltrace

A tool to trace **shared** library calls used by a program and all the signals it receives

- Very useful complement to `strace`, which shows only system calls.
- Of course, works even if you don't have the sources
- Allows to filter library calls with regular expressions, or just by a list of function names.
- With the `-S` option it shows system calls too!
- Also offers a summary with its `-c` option.
- Manual page: https://linux.die.net/man/1/ltrace
- Works better with *glibc*. `ltrace` used to be broken with *uClibc* (now fixed), and is not supported with *Musl* (Buildroot 2022.11 status).

See https://en.wikipedia.org/wiki/Ltrace for details

ltrace example output

```
# ltrace  ffmpeg -f video4linux2 -video_size 544x288 -input_format mjpeg -i /dev
/video0 -pix_fmt rgb565le -f fbdev /dev/fb0
__libc_start_main([ "ffmpeg", "-f", "video4linux2", "-video_size"... ] <unfinished ...>
setvbuf(0xb6a0ec80, nil, 2, 0)                    = 0
av_log_set_flags(1, 0, 1, 0)                      = 1
strchr("f", ':')                                  = nil
strlen("f")                                       = 1
strncmp("f", "L", 1)                              = 26
strncmp("f", "h", 1)                              = -2
strncmp("f", "?", 1)                              = 39
strncmp("f", "help", 1)                           = -2
strncmp("f", "-help", 1)                          = 57
strncmp("f", "version", 1)                        = -16
strncmp("f", "buildconf", 1)                      = 4
strncmp("f", "formats", 1)                        = 0
strlen("formats")                                 = 7
strncmp("f", "muxers", 1)                         = -7
strncmp("f", "demuxers", 1)                       = 2
strncmp("f", "devices", 1)                        = 2
strncmp("f", "codecs", 1)                         = 3
...
```

ltrace summary

Example summary at the end of the ltrace output (`-c` option)

```
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 52.64    5.958660     5958660         1 __libc_start_main
 20.64    2.336331     2336331         1 avformat_find_stream_info
 14.87    1.682895         421      3995 strncmp
  7.17    0.811210      811210         1 avformat_open_input
  0.75    0.085290         584       146 av_freep
  0.49    0.055150         434       127 strlen
  0.29    0.033008         660        50 av_log
  0.22    0.025090         464        54 strcmp
  0.20    0.022836       22836         1 avformat_close_input
  0.16    0.017788         635        28 av_dict_free
  0.15    0.016819         646        26 av_dict_get
  0.15    0.016753         440        38 strchr
  0.13    0.014536         581        25 memset
  0.09    0.009762        9762         1 avcodec_send_packet
...
------ ----------- ----------- --------- --------------------
100.00   11.318773                  4762 total
```

Valgrind

https://valgrind.org/

- *instrumentation framework for building dynamic analysis tools*
  - detect many memory management and threading bugs
  - profile programs
- Supported architectures: x86, x86-64, ARMv7, arm64, mips32, s390, ppc32 and ppc64
- Very popular tool especially for debugging memory issues
- Runs your program on a synthetic CPU → significant performance impact (100 x slower on SAMA5D3!), but very detailed instrumentation
- Runs on the target. Easy to build with Yocto Project or Buildroot.

Valgrind tools

- *Memcheck*: detects memory-management problems
- *Cachegrind*: cache profiler, detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code
- *Callgrind*: extension to Cachegrind, provides extra information about call graphs
- *Massif*: performs detailed heap profiling by taking regular snapshots of a program's heap
- *Helgrind*: thread debugger which finds data races in multithreaded programs. Looks for memory locations accessed by multiple threads without locking.
- More at https://valgrind.org/info/tools.html

Valgrind examples

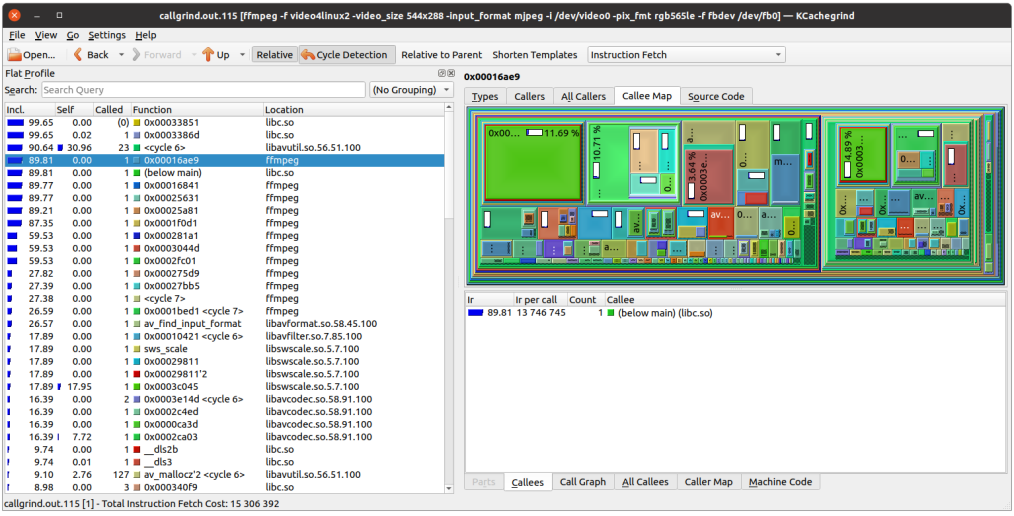
- *Memcheck*

```
$ valgrind --leak-check=yes <program>
  ==19182== Invalid write of size 4
  ==19182==    at 0x804838F: f (example.c:6)
  ==19182==    by 0x80483AB: main (example.c:11)
  ==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
  ==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
  ==19182==    by 0x8048385: f (example.c:5)
  ==19182==    by 0x80483AB: main (example.c:11)
```

- *Callgrind*

```
$ valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes <program>
$ ls callgrind.out.*
callgrind.out.1234
$ callgrind_annotate callgrind.out.1234
```

Kcachegrind - Visualizing Valgrind profiling data



https://github.com/KDE/kcachegrind

perf

- Uses hardware performance counters, much faster than Valgrind!
- Need a kernel with `CONFIG_PERF_EVENTS` and `CONFIG_HW_PERF_EVENTS`
- User space tool: `perf`. It is part of the kernel sources so it is always in sync with your kernel.
- Usage:

```
perf record /my/command
```

- Get the results with:

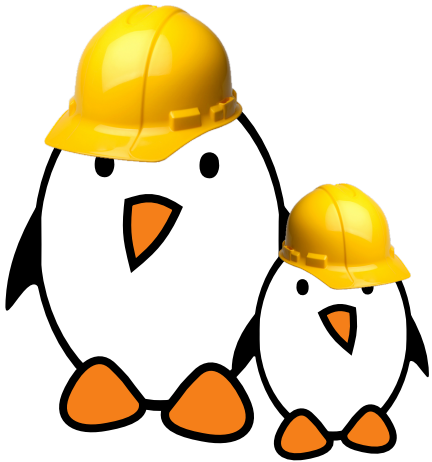
```
perf report
```

- Note: advice to run `perf` on a filesystem built with glibc. Didn't manage to compile `perf` on a Musl root filesystem (Buildroot 2021.02 status). Once again, glibc is recommended for debugging.

perf report output

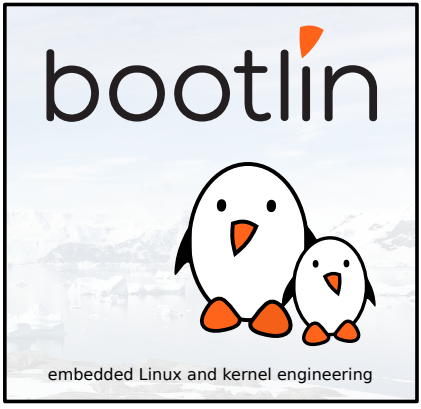
```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 5K of event 'cycles'
# Event count (approx.): 1392529663
#
# Overhead  Command  Shared Object             Symbol
# ........  .......  ........................  ..................................
#
    10.72%  ffmpeg   [kernel.kallsyms]         [k] video_get_user
    10.60%  ffmpeg   [kernel.kallsyms]         [k] vector_swi
     4.76%  ffmpeg   libc-2.31.so              [.] ioctl
     4.22%  ffmpeg   [kernel.kallsyms]         [k] __se_sys_ioctl
     3.81%  ffmpeg   [kernel.kallsyms]         [k] __video_do_ioctl
     3.42%  ffmpeg   libavformat.so.58.45.100  [.] avformat_find_stream_info
     2.83%  ffmpeg   [kernel.kallsyms]         [k] video_usercopy
     2.70%  ffmpeg   libc-2.31.so              [.] cfree
     2.58%  ffmpeg   [kernel.kallsyms]         [k] __fget_light
     2.53%  ffmpeg   libpthread-2.31.so        [.] __errno_location
     2.40%  ffmpeg   [kernel.kallsyms]         [k] arm_copy_from_user
     2.26%  ffmpeg   [kernel.kallsyms]         [k] memset
     2.09%  ffmpeg   [kernel.kallsyms]         [k] mutex_unlock
     2.06%  ffmpeg   [kernel.kallsyms]         [k] v4l2_ioctl
     2.05%  ffmpeg   libavcodec.so.58.91.100   [.] av_init_packet
     1.95%  ffmpeg   libc-2.31.so              [.] memset
...
```

Practical lab - Optimizing the application



- Compile the video player with just the features needed at run time.
- Trace and profile the video player with `strace`
- Observe size and time savings

Optimizing init scripts and system startup



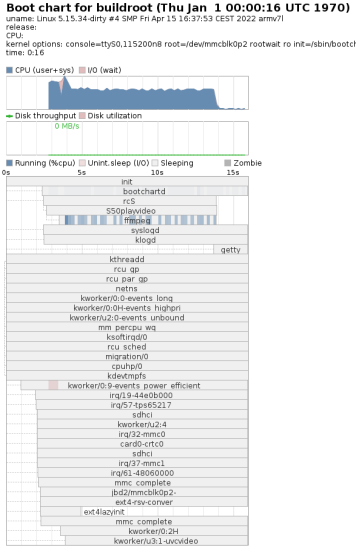
Methodology

There are multiple ways to reduce the time spent in init scripts before starting the application:

- Start the application as soon as possible after only the strictly necessary dependencies.
- Simplify shell scripts
- Even starting the application before `init`

Measuring - bootchart

- If you want to have a more detailed look at the userland boot sequence than with `grabserial`, you can use `bootchart`.
- http://www.bootchart.org



Measuring - bootchart

- You can use `bootchartd` from `busybox` (`CONFIG_BOOTCHARTD=y`)
- Boot your board passing `init=/sbin/bootchartd` on your kernel command line
- Copy the generated `/var/log/bootlog.tgz` file from your target to your host
- The last release of Bootchart is from 2007, and is now broken on `http://www.bootchart.org`. Download a copy from `https://bootlin.com/pub/source/bootchart-0.9.tar.bz2`
- Generate the timechart:

```
cd bootchart-<version>
java -jar bootchart.jar bootlog.tgz
```

- This produces a `bootlog.png` image

Measuring - systemd

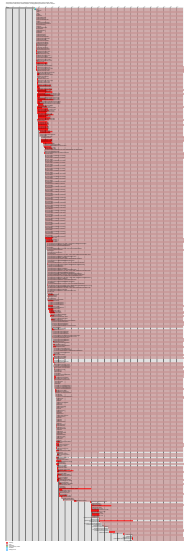
If you are using `systemd` as your `init` program, you can use `systemd-analyze`. See https://www.freedesktop.org/software/systemd/man/systemd-analyze.html.

```
$ systemd-analyze critical-chain
multi-user.target @47.820s
  pmie.service @35.968s +548ms
    pmcd.service @33.715s +2.247s
      network-online.target @33.712s
        systemd-networkd-wait-online.service @12.804s +20.905s
          systemd-networkd.service @11.109s +1.690s
            systemd-udevd.service @9.201s +1.904s
              systemd-tmpfiles-setup-dev.service @7.306s +1.776s
                kmod-static-nodes.service @6.976s +177ms
                  systemd-journald.socket
                    system.slice
                      -.slice
```

systemd-analyze plot

This command prints an SVG graphic detailing which system services have been started at what time, highlighting the time they spent on initialization.

```
$ systemd-analyze plot >bootup.svg
$ inkscape bootup.svg
```



Init optimizations

Goal to start your application as soon as possible after all the dependencies are started:

- Depends on your `init` program. Here we are assuming BusyBox `init` scripts.
- `init` scripts run in alphanumeric order and start with a letter (K for stop (**k**ill) and S for **s**tart).
- You want to use the lowest number you can for your application.
- You can even replace `init` with your application!
  However, that's easier to keep a standard `init`, which also acts as a universal parent to orphan processes (otherwise you get zombies), and also takes care of implementing system shutdown.

Optimizing init scripts

- Start all your services directly from a single startup script (e.g. `/etc/init.d/rcS`). This eliminates multiple calls to `/bin/sh`.
- An easier to maintain solution allowing to keep subscripts: `source` them (`.` command) if possible. This won't spawn new shell processes. Buildroot's `/etc/init.d/rcS` file already does this with `.sh` files.
- You could mount your filesystems directly in the C code of your application:

```c
#include <stdio.h>
#include <sys/mount.h>

int main (void)
{
        int ret;
        ret = mount("sysfs", "/tmp/test", "sysfs", 0, NULL);
        if(ret < 0)
                perror("Can't mount sysfs\n");
}
```

Reduce forking (1)

- `fork`/`exec` system calls are very expensive. Because of this, calls to executables from shells are slow.
- Try to use shell built-ins whenever possible. For example in BusyBox, you can use `echo`, `test`, `printf` and others as shell built-ins. At run time, use the `type` command to check whether a command is a built-in. Example: `type echo`.
- BusyBox also has a *exec prefer applets* setting (`CONFIG_FEATURE_PREFER_APPLETS`) trying to run the corresponding applet (instead of making an `exec` call), typically in shells or in commands such as `find -exec`.

Reduce forking (2)

Pipes and back-quotes are also implemented by `fork/exec`. You can reduce their usage in scripts. Example:

```
cat /proc/cpuinfo | grep model
```

Replace it with:

```
grep model /proc/cpuinfo
```

See https://elinux.org/Optimize_RC_Scripts

Reduce forking (3)

Replaced:

```
if [ $(expr match "$(cat /proc/cmdline)" '.* debug.*')\
       -ne 0 -o -f /root/debug ]; then
DEBUG=1
```

By a much cheaper command running only one process:

```
res=`grep " debug" /proc/cmdline`
if [ "$res" -o -f /root/debug ]; then
DEBUG=1
```

This only optimization allowed to save 87 ms on an ARM AT91SAM9263 system (200 MHz)!

Reduce size

- Strip your executables and libraries, removing ELF sections only needed for development and debugging. The `strip` command is provided by your cross-compiling toolchain. That's done by default in Buildroot.
- `superstrip`: https://muppetlabs.com/~breadbox/software/elfkickers.html. Goes beyond `strip` and can strip out a few more bits that are not used by Linux to start an executable. Buildroot stopped supporting it because it can break executables. Try it only if saving a few bytes is critical.

Quick splashscreen display (1)

Often the first sign of life that you are showing!

- A good solution seems to be BusyBox `fbsplash`:
  See `miscutils/fbsplash.c` in BusyBox sources.
- Alternative: `fbv`
  `http://s-tech.elsat.net.pl/fbv/`
- However, `fbv` is slow:
  878 ms on an Microchip AT91SAM9263 system!

Quick splashscreen display (2)

- To do it faster, you can dump the framebuffer contents:

```
fbv -d 1 /root/logo.bmp
cp /dev/fb0 /root/logo.fb
lzop -9 /root/logo.fb
```

- And then copy it back as early as possible in an initramfs:

```
lzopcat /root/logo.fb.lzo > /dev/fb0
```

Results on an Microchip AT91SAM9263 system:

	fbv	plain copy (dd)	lzopcat
Time	878 ms	54 ms	52.5 ms

https://bootlin.com/blog/super-fast-linux-splashscreen/

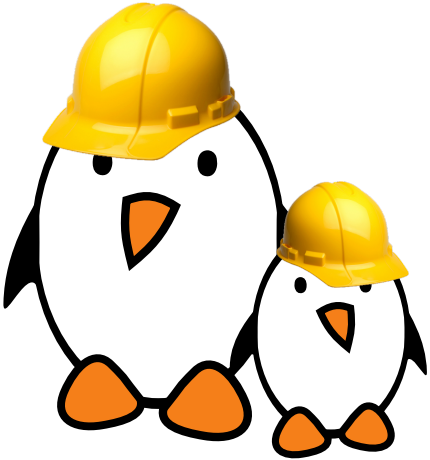
Note: *LZO* compression is the fastest in terms of decompression, and is supported by BusyBox.

Animated splashscreen

Still slow to read and write entire screens. Just draw useful pixels and even create an animation!

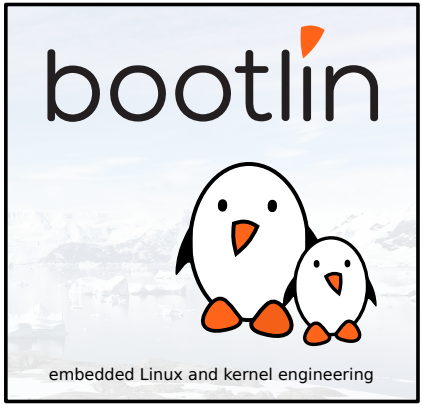
- Create a simple C program that just animates pixels and simple geometric shapes on the framebuffer!
- Example: https://bootlin.com/pub/code/fb/anim.c (Public Domain license).
  On a 400 MHz ARM9 system: starts drawing in 10 ms
  Size: 24 KB, compiled statically.

Practical lab - Reducing time in init-scripts



- Regenerate the root filesystem with Buildroot
- Use bootchartd to measure boot time

Filesystem optimizations



Filesystem impact on performance

Tuning the filesystem is usually one of the first things we work on in boot time projects.

- Different filesystems can have different initialization and mount times. In particular, the type of filesystem for the root filesystem directly impacts boot time.
- Different filesystems can exhibit different read, write and access time performance, according to the type of filesystem activity and to the type of files in the system.

Different filesystem for different storage types

- Block storage (including memory cards, eMMC)
  - ext2, ext4
  - xfs, jfs, reiserfs
  - btrfs
  - f2fs
  - SquashFS, EROFS
- Raw flash storage
  - JFFS2
  - YAFFS2
  - UBIFS
  - ubiblock + (SquashFS or EROFS)

See our embedded Linux training materials for full details:
https://bootlin.com/doc/training/embedded-linux/

Block filesystems

For block storage

- ext4: pretty good read and write performance.
- xfs, jfs, reiserfs: can be good in some read or write scenarii as well.
- btrfs, f2fs: can achieve best read and write performance, taking advantage of the characteristics of flash-based block devices.
- SquashFS: best mount time and read performance, for read-only partitions. Great for root filesystems which can be read-only.
- EROFS: new read-only file system for block storage. Worth testing too.

JFFS2

For raw flash storage

- Mount time depending on filesystem size: the kernel has to scan the whole storage at mount time, to read which block belongs to each file.
- Need to use the `CONFIG_JFFS2_SUMMARY` kernel option to store such information in flash. This dramatically reduces mount time.
- Benchmark on ARM:
  from 16 s to 0.8 s for a 128 MB partition.
- Rather poor read and write performance,
  compared to YAFFS2 and UBIFS.
- JFFS2 only makes sense on small storage space, where UBI would have too much overhead.

YAFFS2

For raw flash storage

- Good mount time
- Good read and write performance
- Drawbacks: no compression, not in the mainline Linux kernel

UBIFS

For raw flash storage, on top of the UBI layer

- Advantages:
  - Good read and write performance (similar to YAFFS2)
  - Other advantages: better for wear leveling (can operate on the whole UBI space, not only within a single partition).
- Drawbacks:
  - Not appropriate for small partitions (too much metadata overhead). Use JFFS2 or JAFFS2 instead.
  - Not so good mount time, because of the time needed to initialize UBI (*UBI Attach*: at boot time or running `ubi_attach` in user space).
  - Addressed by *UBI Fastmap*, introduced in Linux 3.7.
    See next slides.

How UBI Fastmap works

- *UBI Attach*: needs to read UBI metadata by scanning all erase blocks. Time proportional to the storage size.
- *UBI Fastmap* stores such information in a few flash blocks (typically at UBI detach time during system shutdown) and finds it there at boot time.
- This makes *UBI Attach* time constant.
- If *Fastmap* information is invalid (unclean system shutdown, for example), it falls back to scanning (slower, but correct, and *Fastmap* will work again during the next boot).
- Details: ELCE 2012 presentation from Thomas Gleixner: https://elinux.org/images/a/ab/UBI_Fastmap.pdf

Using UBI Fastmap

- Compile your kernel with `CONFIG_UBI_FASTMAP`
- Boot your system at least once with the `ubi.fm_autoconvert=1` kernel parameter.
- Reboot your system in a clean way
- You can now remove `ubi.fm_autoconvert=1`

UBI Fastmap benchmark

- Measured on the Microchip SAMA5D3 Xplained board (ARM), Linux 3.10
- UBI space: 216 MB
- Root filesystem: 80 MB used (Yocto)
- Average results:

	Attach time	Diff	Total time
Without *UBI Fastmap*	968 ms		
With *UBI Fastmap*	238 ms	-731 ms	-665 ms

- Expect to save more with bigger UBI spaces!

Note: total boot time reduction a bit lower probably because of other kernel threads executing during the attach process.

ubiblock + (SquashFS or EROFS)

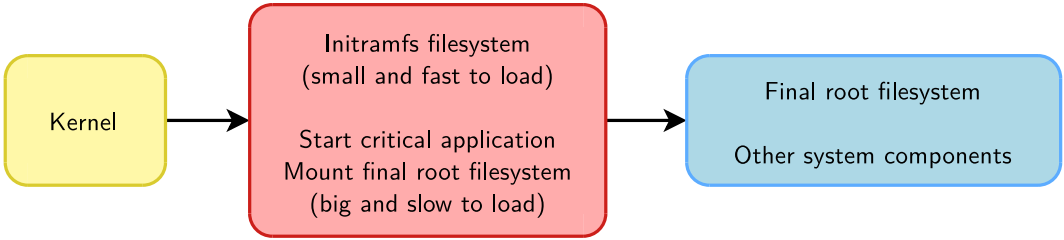
For raw flash storage

- *ubiblock*: read-only block device on top of UBI (`CONFIG_MTD_UBI_BLOCK`).
- Allows to put SquashFS or EROFS on a UBI volume.
- Expecting great boot time and read performance. Great for read-only root filesystems.
- Benchmarks not available yet.

Finding the best filesystem

- Raw flash storage: UBIFS with `CONFIG_UBI_FASTMAP` is probably the best solution.
- Block storage: SquashFS best solution for root filesystems which can be read-only. Btrfs and f2fs probably the best solutions for read/write filesystems.
- Fortunately, changing filesystem types is quite cheap, and completely transparent for applications. Just try several filesystem options, as see which one works best for you!
- Do not focus only on boot time.
  For systems in which read and write performance matters, we recommend to use separate root filesystem (for quick boot time) and data partitions (for good runtime performance).

An idea is to use a very small initramfs, just enough to start the critical application and then switch to the final root filesystem.



Root filesystem in memory: initramfs

It is also possible to boot the system with a filesystem in memory: *initramfs*

- Either from a compressed CPIO archive integrated into the kernel image
- Or from such an archive loaded by the bootloader into memory
- At boot time, this archive is extracted into the Linux file cache
- It is useful for two cases:
  - Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
  - As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.
- Details (in kernel documentation): `filesystems/ramfs-rootfs-initramfs`

External initramfs

- To create one, first create a compressed CPIO archive:

```
cd rootfs/
find . | cpio -H newc -o > ../initramfs.cpio
cd ..
gzip initramfs.cpio
```

- If you're using U-Boot, you'll need to include your archive in a U-Boot container:

```
mkimage -n 'Ramdisk Image' -A arm -O linux -T ramdisk -C gzip \
        -d initramfs.cpio.gz uInitramfs
```

- Then, in the bootloader, load the kernel binary, DTB and `uInitramfs` in RAM and boot the kernel as follows:

```
bootz kernel-addr initramfs-addr dtb-addr
```

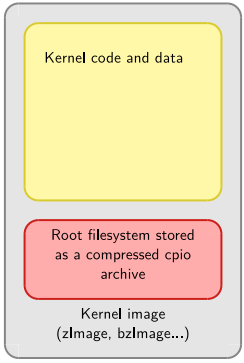
Built-in initramfs

To have the kernel Makefile include an initramfs archive in the kernel image: use the `CONFIG_INITRAMFS_SOURCE` option.

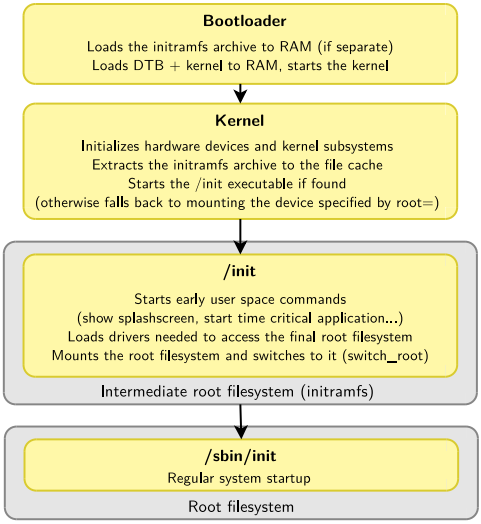
- It can be the path to a directory containing the root filesystem contents
- It can be the path to a ready made cpio archive
- It can be a text file describing the contents of the initramfs

See the kernel documentation for details: `driver-api/early-userspace/early_userspace_support`

WARNING: only binaries from GPLv2 compatible code are allowed to be included in the kernel binary using this technique. Otherwise, use an external initramfs.



Overall booting process with initramfs



Initramfs for boot time reduction

Create the smallest initramfs possible, just enough to start the critical application and then switch to the final root filesystem with `switch_root`:

- Use a light C library reduced to the minimum, *uClibc* or *Musl* if you are not yet using it for your root filesystem
- Reduce BusyBox to the strict minimum. You could even do without it and implement `/init` in C.
- Use statically linked applications (less CPU overhead, less libraries to load, smaller initramfs if no libraries at all), `BR2_STATIC_LIBS` in Buildroot.

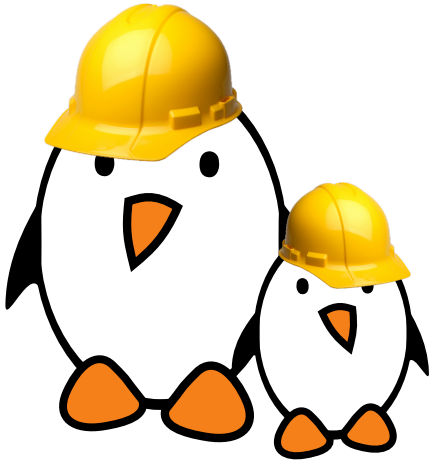
Statically linked executables: licensing constraints

- Statically linked executables are very useful to reduce size (especially in small initramfs), and require less work to start.
- However, the LGPL license in the uClibc and glibc libraries requires to leave the user the ability to relink the executable with a modified version of the library.
- Solution to keep static binaries:
  - Either provide the executable source code (even proprietary), allowing to recompile it with a modified version of the library. That's what you do when you ship a static BusyBox.
  - Or also provide a dynamically linked version of the executable (in a separate way), allowing to use another library version.
  - Easiest solution: build your static executables with the `musl` library (MIT license: no trouble)
- References:
  https://gnu.org/licenses/gpl-faq.html#LGPLStaticVsDynamic
  https://gnu.org/copyleft/lesser.html#section4

Do not compress your initramfs

- If you ship your initramfs inside a compressed kernel image, don't compress it (enable `CONFIG_INITRAMFS_COMPRESSION_NONE`).
- Otherwise, by default, your initramfs data will be compressed twice, and the kernel will be bigger and will take a little more time to load and uncompress.
- Example on Linux 5.1 with a 1.60 MB initramfs (tar archive size) on Beagle Bone Black: this allowed to reduce the kernel size from 4.94 MB to 4.74 MB (-200 KB) and save about 170 ms of boot time.

Practical lab - Filesystem optimizations



- Comparing the boot time performance of various filesystems
- Tests with initramfs booting too

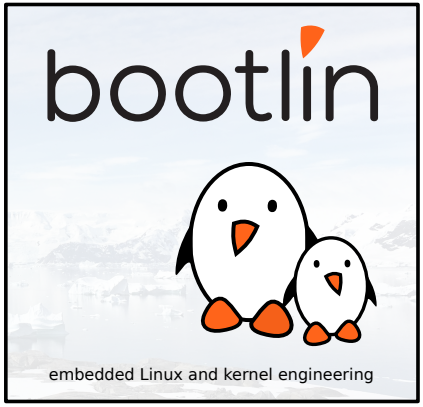
Filesystem optimizations - Results

Results on BeagleBone Black, Linux 5.11

FS image size	Buildroot image size	zImage size diff	Time to init	Total boot time	ffmpeg exec time
ext2 (rev1) only	62,914,560	+19,544	8.489s	9.704s	0.498s
ext4 only	62,914,560	+241,472	8.645s	9.862s	0.484s
btrfs	114,294,784	+546,376	11.789s	12.918s	0.487s
f2fs	104,857,600	+167,640	8.670s	9.803s	0.488s
squashfs with lzo	724,992	+19,016	8.500s	9.721s	0.436s
erofs	1,196,032	+28,072	8.510s	9.795s	0.491s
cramfs	737,280	*info lost*	8.499s	9.875s	0.656s
initramfs	N/A	+169,552	8.399s	9.660s	0.455s

Note: `zImage` kernel compressed with LZMA (best but slowest compressor), with a 1.162 MB filesystem (size of the tar archive generated by Buildroot).

Kernel optimizations



Measure - Kernel initialization functions

To find out which kernel initialization functions are the longest to execute, add `initcall_debug` to the kernel command line. Here's what you get on the kernel log:

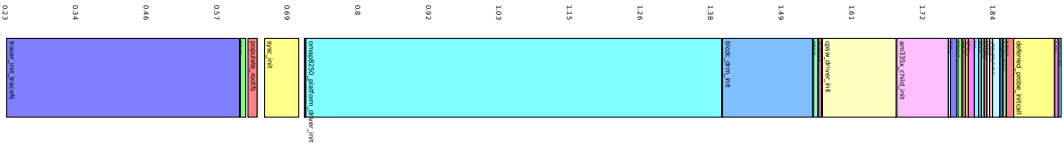
```
...
[    3.750000] calling  ov2640_i2c_driver_init+0x0/0x10 @ 1
[    3.760000] initcall ov2640_i2c_driver_init+0x0/0x10 returned 0 after 544 usecs
[    3.760000] calling  at91sam9x5_video_init+0x0/0x14 @ 1
[    3.760000] at91sam9x5-video f0030340.lcdheo1: video device registered @ 0xe0d3e340, irq = 24
[    3.770000] initcall at91sam9x5_video_init+0x0/0x14 returned 0 after 10388 usecs
[    3.770000] calling  gspca_init+0x0/0x18 @ 1
[    3.770000] gspca_main: v2.14.0 registered
[    3.770000] initcall gspca_init+0x0/0x18 returned 0 after 3966 usecs
...
```

You might need to increase the log buffer size with `CONFIG_LOG_BUF_SHIFT` in your kernel configuration. You will also need `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS`.

Kernel boot graph

With `initcall_debug`, you can generate a boot graph making it easy to see which kernel initialization functions take most time to execute.

- Copy and paste the output of the `dmesg` command to a file (let's call it `boot.log`)
- On your workstation, run the `scripts/bootgraph.pl` script in the kernel sources: `scripts/bootgraph.pl boot.log > boot.svg`
- You can now open the boot graph with a vector graphics editor such as `inkscape`:



Using the kernel boot graph (1)

Start working on the functions consuming most time first. For each function:

- Look for its definition in the kernel source code. You can use Elixir (for example https://elixir.bootlin.com).
- Be careful: some function names don't exist, the names correspond to *modulename*_`init`. Then, look for initialization code in the corresponding module.
- Remove unnecessary functionality:
  - Find which kernel configuration parameter compiles the code, by looking at the Makefile in the corresponding source directory.

Using the kernel boot graph (2)

- Postpone:
  - Find which module (if any) the function belongs to. Load this module later if possible.
- Optimize necessary functionality:
  - Look for parameters which could be used to reduce probe time, looking for the `module_param` macro.
  - Look for delay loops and calls to functions containing `delay` in their name, which could take more time than needed. You could reduce such delays, and see whether the code still works or not.

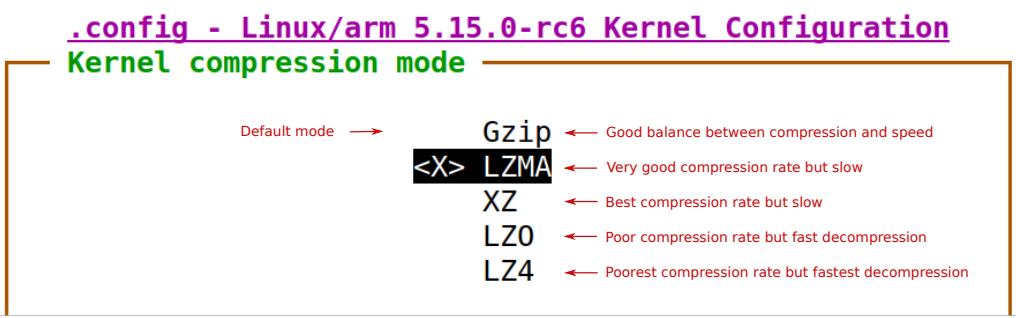
Reduce kernel size

First, we focus on reducing the size without removing features

- The main mechanism is to use kernel modules
- Compile everything that is not needed at boot time as a module
- Two benefits: the kernel will be smaller and load faster, and less initialization code will get executed
- Remove features that are not used by userland: `CONFIG_KALLSYMS`, `CONFIG_DEBUG_FS`, `CONFIG_BUG`
- Use features designed for embedded systems: `CONFIG_SLOB`, `CONFIG_EMBEDDED`

Kernel Compression

Depending on the balance between your storage reading speed and your CPU power to decompress the kernel, you will need to benchmark different compression algorithms. Also recommended to experiment with compression options at the end of the kernel optimization process, as the results may vary according to the kernel size.



Kernel compression options

Results on TI AM335x (ARM), 1 GHz, Linux 5.1

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.208 s	0.158 s	**0.154 s**	0.224 s	0.241 s
Time to userspace	1.451 s	2.167 s	1.999s	**1.416 s**	1.462 s

Gzip is close. It's time to try with faster storage (SanDisk Extreme Class A1)

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.150 s	0.114 s	**0.111 s**	0.161 s	0.173 s
Time to userspace	1.403 s	2.132 s	1.965 s	**1.363 s**	1.404 s

Lzo and Gzip seem the best solutions. Always benchmark as the results depend on storage and CPU performance.

Compressing the kernel with Zstandard

- Zstandard is a relatively recent compression scheme, implemented by Yann Collet.
- Unfortunately, not available on ARM yet.
  Only on x86, mips and s390 (Linux 5.15 status).
- Compressing better than gzip and decompressing as fast as LZO, it could be the best option.
- See https://en.wikipedia.org/wiki/Zstandard

```
config KERNEL_ZSTD
        bool "ZSTD"
        depends on HAVE_KERNEL_ZSTD
        help
          ZSTD is a compression algorithm targeting intermediate compression
          with fast decompression speed. It will compress better than GZIP and
          decompress around the same speed as LZO, but slower than LZ4. You
          will need at least 192 KB RAM or more for booting. The zstd command
          line tool is required for compression.
```

Booting an uncompressed kernel

- It is also possible to boot an uncompressed kernel: `arch/<arch>/boot/Image`
- This could be a worthy solution if you have a slow CPU and fast I/O, or if you're booting Linux in an emulated machine (hardware or software emulator).
- On U-Boot on ARM, you won't be able to boot with the `bootz` command. You will need to use `bootm` and a `uImage` file.

See https://bootlin.com/blog/uncompressed-linux-kernel-on-arm/

Optimize kernel for size (1)

- `CONFIG_CC_OPTIMIZE_FOR_SIZE`: possibility to compile the kernel with `gcc -Os` instead of `gcc -O2`.
- Such optimizations give priority to code size at the expense of code speed. `-Os` enables all `-O2` optimizations except those that often increase code size.
- Results: loading and decompressing the kernel is faster (smaller size), but then the kernel boots and runs slower.

Optimize kernel for size (2)

Results on BeagleBone Black, Linux 5.11, lzo compression

	O2	Os	Diff
Size	7372432	6594440	-10.5 %
Copy time	0.489 s	0.437s s	-52 ms
Decompression time	1.490 s	1.558 s	-68 ms
Time to userspace	1.303 s	1.462 s	+159 ms
Total boot time	5.739 s	5.796s	+57 ms

Deferring drivers and initcalls

- If you can't compile a feature as a module (e.g. networking or block subsystem), you can try to defer its execution.
- Your kernel will not shrink but some initializations will be postponed.
- Typically, you would modify `probe()` functions to return `-EPROBE_DEFER` until they are ready to be run.
- See https://lwn.net/Articles/485194/ for details about the infrastructure supporting this.

Turning off console output

- Console output is actually taking a lot of time (very slow device). Probably not needed in production. Disable it by passing the `quiet` argument on the kernel command line.
- You will still be able to use `dmesg` to get the kernel messages.
- Time between starting the kernel and starting the `init` program, on Microchip SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Without `quiet`	2.352 s	
With `quiet`	1.285 s	-1.067 s

- Less time will be saved on a reduced kernel, of course.
- Don't do it too early if you're using `grabserial`

Preset loops per jiffy

- At each boot, the Linux kernel calibrates a delay loop (for the `udelay()` function). This measures a number of loops per jiffy (*lpj*) value. You just need to measure this once! Find the `lpj` value in the kernel boot messages:

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

- Now, you can add `lpj=<value>` to the kernel command line:

```
Calibrating delay loop (skipped) preset value.. 996.14 BogoMIPS (lpj=4980736)
```

- Tests on BeagleBone Black (ARM), Linux 5.15: -85 ms

Multiprocessing support (CONFIG_SMP)

- SMP is quite slow to initialize
- It is usually enabled in default configurations, even if you have a single core CPU (default configurations should support multiple systems).
- So make sure you disable it if you only have one CPU core.
- Results on BeagleBone Black:
  Compressed kernel size: -188 KB

Kernel: last milliseconds (1)

To shave off the last milliseconds, you will probably want to remove unnecessary features:

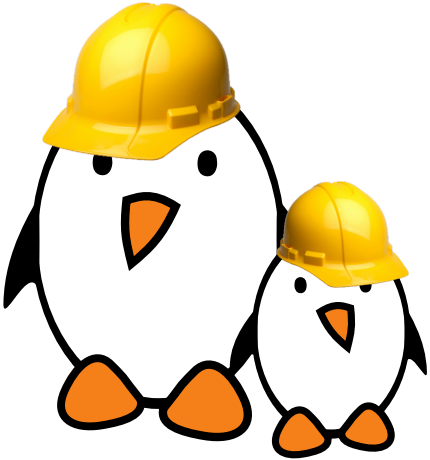
- `CONFIG_PRINTK=n` will have the same effect as the `quiet` command line argument but you won't have any access to kernel messages. You will have a significantly smaller kernel though.
- Compile your kernel in *Thumb2* mode (on ARM 32 bit): `CONFIG_THUMB2_KERNEL` (any ARM toolchain can do that).

Kernel last milliseconds (2)

More features you could remove:

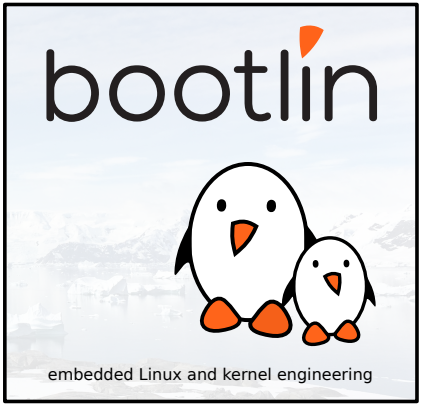
- Module loading/unloading
- Block layer
- Network stack
- USB stack
- Power management features
- `CONFIG_SYSFS_DEPRECATED`
- Input: keyboards / mice / touchscreens

Practical lab - Reduce kernel boot time



- Use `initcall_debug` to find the biggest time consumers
- Optimize existing functionality
- Remove unused features and drivers
- Select the best kernel compression method

Bootloader optimizations



Generic bootloader optimizations

Bootloader

- Remove unnecessary functionality.
  Usually, bootloaders include many features needed only for development. Compile your bootloader with fewer features.
- Optimize required functionality.
  Tune your bootloader for fastest performance.
  Skip the bootloader and load the kernel right away.

U-Boot - Remove unnecessary functionality

Recompile U-Boot to remove features not needed in production

- Disable as many features as possible through the `menuconfig` interface and through `include/configs/<soc>-<board>.h`
- Examples: MMC, USB, Ethernet, dhcp, ping, command line edition, command completion...
- A smaller and simpler U-Boot is faster to load and faster to initialize.

However, in this presentation, we will give the easiest optimizations in U-Boot, but won't be exhaustive, because the best way to save time is to skip U-Boot, using its *Falcon Mode* (covered in the next section).

U-Boot - Remove the boot delay

- Remove the boot delay:
  `setenv bootdelay 0`
- This usually saves several seconds!

U-Boot - Simplify scripts

Some boards have over-complicated scripts:

Running nested scripts

```
bootcmd=run bootf0
bootf0=run ${args0}; setenv bootargs ${bootargs} \
maximasp.kernel=maximasp_nand.0:kernel0; nboot 0x70007fc0 kernel0
```

Let's replace this by:

```
setenv bootargs 'mem=128M console=tty0 consoleblank=0
console=ttyS0,57600 \
mtdparts=maximasp_nand.0:2M(u-boot)ro,512k(env0)ro,512k(env1)ro,\
4M(kernel0),4M(kernel1),5M(kernel2),100M(root0),100M(root1),-(other)\
rw ubi.mtd=root0 root=ubi0:rootfs rootfstype=ubifs earlyprintk debug \
user_debug=28 maximasp.board=EEKv1.3.x \
maximasp.kernel=maximasp_nand.0:kernel0'
setenv bootcmd 'nboot 0x70007fc0 kernel0'
```

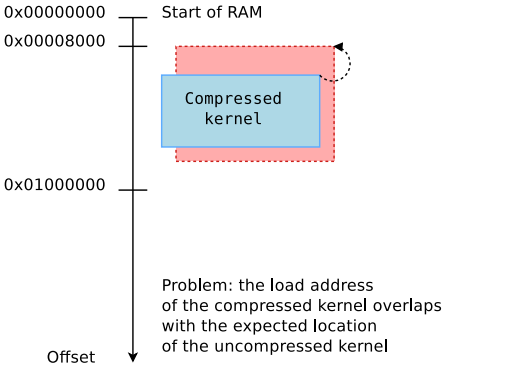
This saved 56 ms on this ARM9 system (400 MHz)!

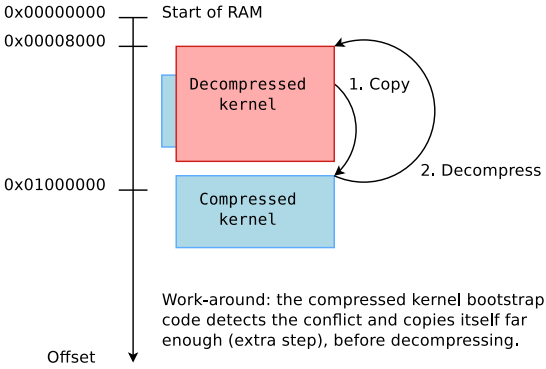
Bootloader: copy the exact kernel size

- When copying the kernel from **raw** flash or MMC to RAM, we still see many systems that copy too many bytes, not taking the exact kernel size into account.
- A solution is to store the exact size of the kernel in an environment variable, and use it a kernel loading time.
- Of course, that's not needed when the kernel is loaded from a filesystem, which knows how big the file is.

Bootloader: watch the compressed kernel load address

On ARM32, the uncompressed kernel is usually started at offset 0x8000 from the start of RAM. Load the compressed kernel at a far enough address!

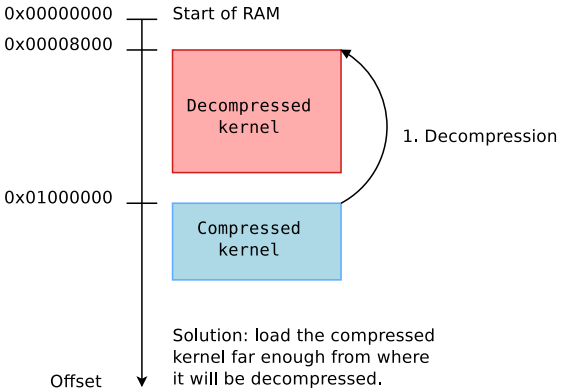




Source: https://people.kernel.org/linusw/how-the-arm32-linux-kernel-decompresses

Bootloader: load the compressed kernel far enough

On ARM32, a usual kernel load address is at offset 0x01000000 (16 MB)

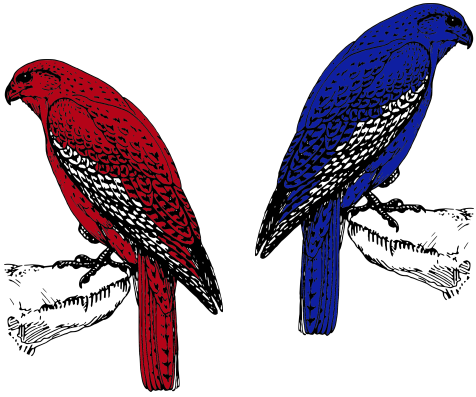


Tests on STM32MP157A (650 MHz): an overlap increases boot time by 107 ms.

U-Boot Falcon Mode

Goal: boot faster!

U-Boot Falcon Mode is about reducing the time spent in the bootloader.



Falcons are the fastest animals on Earth!

Image credits: https://openclipart.org/detail/287044/falcon-2

Example: booting on Microchip SAMA5D36

You first need to understand how your SoC boots:

Figure 11-2. NVM Bootloader Sequence Diagram

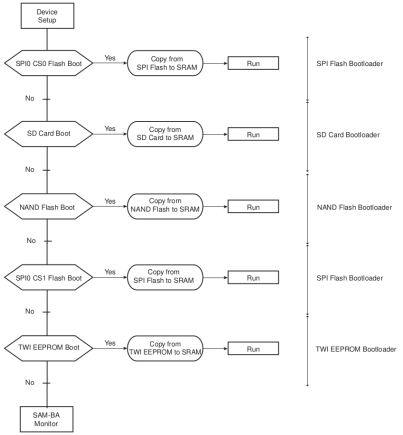
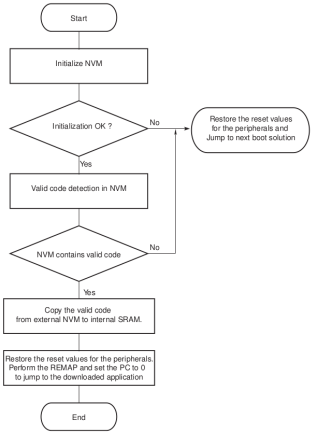
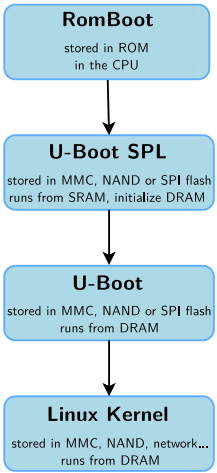


Figure 11-3. NVM Bootloader Program Diagram



Source: Microchip SAMA5D36 datasheet
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet_B.pdf

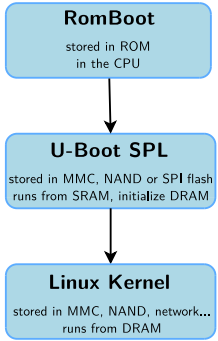
Normal and Falcon boot on Microchip SAMA5D3



Boot process with U-Boot

- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to the SRAM size (here 64 KB).
- **U-Boot SPL** (*Secondary Program Loader*): runs from SRAM (inside the SoC). Initializes the DRAM controller plus storage devices (MMC, NAND), loads the secondary bootloader into DRAM and starts it. Much bigger size limits!
- **U-Boot**: runs from DRAM. Initializes other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to DRAM and starts it.
  This is the part that can be skipped
- **Linux Kernel**: runs from DRAM. Takes over the system completely (the bootloader no longer exists).

This scheme applies to all modern SoCs.



Boot process without U-Boot (*Falcon mode*)

Falcon mode advantages and drawbacks

- Main advantage: since Linux and U-Boot are both loaded to RAM, U-Boot's *Falcon Mode* mainly saves time by directly loading Linux from the SPL instead of loading and executing the full U-Boot first.
- Drawback: you lose the flexibility brought by the full U-Boot. You have to follow a special procedure to update the kernel binary, DTB and kernel command line parameters.
- Advantage: the interactivity offered by the full U-Boot is not necessary on a production device. Falcon boot works in the same way on all SoCs on which U-Boot SPL is supported. This makes it easier to apply this technique on all your projects!

What U-Boot does (1)

U-Boot has multiple ways of preparing the kernel boot:

- *ATAGS* - The old way (before Device Tree)
  U-Boot prepares the Linux kernel command line (`bootargs`), the machine ID and other information for Linux in a tagged list, and passes its address to Linux through a register.
- *Flattened Device Tree* - The standard way
  - U-Boot checks the device tree loaded in RAM or directly provides its own.
  - U-Boot checks the specifics of the hardware (amount and location of RAM, MAC address, present devices...), possibly loads corresponding Device Tree overlays, and modifies (fixes-up) the Device Tree accordingly.
  - U-Boot stores the Linux kernel command line (`bootargs`) in the `chosen` section in the Device Tree.

What U-Boot does (2)

- *FIT Image* - The new way
  - U-Boot loads the kernel(s), device tree(s), initramfs image(s), signature(s) from a single file (*FIT Image*)
  - That's used for secure booting, for booting recovery images, etc.
  - U-Boot also implements Device Tree fix-ups, of course.

Using the `spl export` command in U-Boot, you can do such preparation work ahead of time.

- In this presentation, we just cover standard Device Tree booting.
- U-Boot also has support for FIT Image loading in the SPL, but that may still be a bit experimental, and such code must fit within your maximum allowable size for the SPL.
  See `arch/arm/cpu/armv8/fsl-layerscape/doc/README.falcon`

Falcon mode usage overview (1)

Here are the generic steps you need to go through:

- Recompile U-Boot with support for Falcon Mode (`CONFIG_SPL_OS_BOOT`), with support for `spl export` (`CONFIG_CMD_SPL`), and for the way you want to boot.
- Also make sure that `CONFIG_SPL_SIZE_LIMIT` is set (find the SRAM size for your CPU, `0x10000` for SAMA5D36), otherwise, U-Boot won't complain when the SPL is bigger.
- Build the kernel legacy `uImage` file from `zImage` (see next slides)
- Set the kernel command line (`bootargs` environment variable)
- Load the `uImage`, initramfs (if any) and Device Tree images to RAM as usual.

Falcon mode usage overview (2)

Continued...

- Have U-Boot execute the preprocessing before booting Linux, but stop right before doing it:
  `spl export fdt <kernel-addr> <initramfs-addr> <dtb-addr>`
- Save the exported data (*ARGS*) from RAM to storage, in *Flattened Device Tree* form, so that the SPL can load it and directly pass it to the Linux kernel. The below environment variables will help:
  - `fdtargsaddr`: location of *ARGS* in RAM
  - `fdtargslen`: size of *ARGS* in RAM
- If supported by your board (code explanations given later), set your `boot_os` environment variable to `yes/Yes/true/True/1` to enable direct OS booting.

spl export example output

```
=> fatload mmc 0:1 0x21000000 uImage
5483584 bytes read in 530 ms (9.9 MiB/s)
=> fatload mmc 0:1 0x22000000 dtb
27795 bytes read in 7 ms (3.8 MiB/s)
=> setenv bootargs console=ttyS0,115200
=> spl export fdt 0x21000000 - 0x22000000
## Booting kernel from Legacy Image at 21000000 ...
   Image Name:   Linux-5.12.6
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    5483520 Bytes = 5.2 MiB
   Load Address: 20008000
   Entry Point:  20008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 22000000
   Booting using the fdt blob at 0x22000000
   Loading Kernel Image
   Loading Device Tree to 2fb2c000, end 2fb35c92 ... OK
subcommand not supported
subcommand not supported
   Loading Device Tree to 2fb1f000, end 2fb2bc92 ... OK
Argument image is now in RAM: 0x2fb1f000
```

Image credits:
https://openclipart.org/detail/292953/horus

How to create the uImage file

Microchip SAMA5D3 Xplained board example

- Need to know the loading address that should be used for your board. Usually on ARM32, it's the starting physical address of RAM plus `0x8000`.
- Either generate it from the Linux build system:

```
make LOADADDR=0x20008000 uImage
```

- Or generate it using U-Boot's `mkimage` command:

```
mkimage -A arm -O linux -C none  -T kernel \
-a 0x20008000 -e 0x20008000 \
-n "Linux-5.12.6" \
-d arch/arm/boot/zImage arch/arm/boot/uImage
```

U-Boot code changes to support a new board (1)

Your `board/<vendor>/<board>/<board>.c` file must at least implement the `spl_start_uboot()` function.
Here's the most typical example:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
	if (CONFIG_IS_ENABLED(SPL_SERIAL_SUPPORT) && serial_tstc() && serial_getc() == 'c')
		/* break into full u-boot on 'c' */
		return 1;

	if (CONFIG_IS_ENABLED(SPL_ENV_SUPPORT)) {
		env_init();
		env_load();
		if (env_get_yesno("boot_os") != 1)
			return 1;
	}
	return 0;
}
#endif
```

U-Boot code changes to support a new board (2)

If you cannot fit support for an environment in the SPL, the `spl_start_uboot()` function can be simpler:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
	if (CONFIG_IS_ENABLED(SPL_SERIAL_SUPPORT) && serial_tstc() && serial_getc() == 'c')
		/* break into full u-boot on 'c' */
		return 1;

	return 0;
}
#endif
```

U-Boot code changes to support a new board (3)

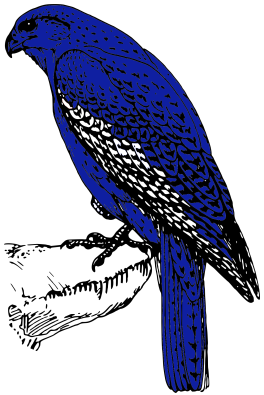
Or even, if reading characters from the serial line doesn't work:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
        return 0;
}
#endif
```

You may also need extra defines to be set, but you will find which ones are missing at compile time.

How to fall back to U-Boot

- If supported by your board, hit the specified key on the serial console and back in U-Boot, disable the `boot_os` environment variable. That's it.
- Otherwise, try to cause OS loading to fail. The easiest way is to erase the kernel binary and if needed the `spl export` output.
- If this doesn't work, re-compile and update the SPL without Falcon mode support, or temporarily modify the `spl_start_uboot()` function to always return `1`. This way, you don't lose your configuration.



Booting from raw MMC - Proposed storage layout

For use on Microchip SAMA5D3 Xplained

Offset (512 b sector)	Offset (bytes)	Contents
0x0	0	MBR (Master Boot Record)
0x100	128 KiB	SPL ARGS
0x200	256 KiB	`u-boot.img`
0x1000	2 MiB	uImage
0x4000	16 MiB	Start of FAT partition

- A FAT partition is required to store the SPL file (`boot.bin`). SAMA5D36 doesn't support an SPL file on raw MMC (unlike i.MX6).
- Caution: partition offsets should be a multiple of the *segment* size, as indicated by the device's `preferred_erase_size` attribute under `/sys/bus/mmc/devices/`.

Booting from raw MMC - Configuration

U-Boot configuration (starting from `sama5d3_xplained_mmc_defconfig`):

```
CONFIG_SPL_OS_BOOT=y
SPL_SIZE_LIMIT=0x10000
CONFIG_SPL_LEGACY_IMAGE_SUPPORT=y
CONFIG_SPL_MMC_SUPPORT=y
CONFIG_CMD_SPL=y
CONFIG_CMD_SPL_WRITE_SIZE=0x7000
CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR=0x200
# CONFIG_SPL_FS_FAT is not set
```

include/configs/sama5d3_xplained.h

```
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTOR 0x100  /* 256 KiB */
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTORS (CONFIG_CMD_SPL_WRITE_SIZE / 512)
#define CONFIG_SYS_MMCSD_RAW_MODE_KERNEL_SECTOR 0x1000 /* 2 MiB */
#define CONFIG_SYS_SPL_ARGS_ADDR 0x22000000
```

Booting from Raw MMC - Writing to raw storage

On your GNU/Linux host:

- Write U-Boot (using the same block size as sector size, to get the same offsets):
  `sudo dd if=u-boot.img of=/dev/sdc bs=512 seek=512 conv=sync`
- Write `uImage`:
  `sudo dd if=uImage of=/dev/sdc bs=512 seek=4096 conv=sync`
- Reminder: in our case (SAMA5D36), the SPL is copied to `boot.bin` in a FAT partition.

On your U-Boot target, after `spl export`:

- Select the right MMC device for `mmc write`:

```
=> mmc list
Atmel mci: 0 (SD)
Atmel mci: 1

=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
```

- Check the size of ARGS

```
=> printenv fdtargslen
```

- Divide it by the sector size (512), and convert it to hexadecimal (round it up), and use the value to save the ARGS to raw MMC:

```
=> mmc write ${fdtargsaddr} 0x100 0x67
```

- **Caution**: the last argument of `mmc write` is a **number of sectors**. If you pass a number of bytes, you'll erase your FAT partition!

Booting from Raw MMC - Results and notes

Reference test

- Loading `zImage` and `dtb` from FAT through `fatload` and using a zero `bootdelay`:
  ```
  setenv bootdelay 0
  setenv bootcmd 'fatload mmc 0:
  1 0x21000000 zImage; fatload mmc 0:
  1 0x22000000; bootz 0x21000000 - 0x22000000'
  ```
- Not completely fair because we have the filesystem overhead, but that's the standard / easiest way on MMC. We could have loaded images from raw MMC, but that's very inconvenient.
- Best result (using `grabserial`):
  ```
  [3.452681 0.000099] Please press Enter to
  activate this console.
  ```

Falcon boot test

- Best result:
  ```
  [3.191228 0.000134] Please press Enter to
  activate this console.
  ```
- We saved 261 ms, but that's disappointing.
- Adding instrumentation to the SPL allowed us to understand why:
  - Time to load the kernel from U-Boot / FAT: 530 ms
  - Time to load the kernel from SPL / raw MMC: 1.010 ms
- Here the specific MMC driver in SPL has poor performance (lack of DMA?)
- We had much better results on different hardware, such as saving 1.2s on i.MX6, and 1.05s on TI AM3358 (Beagle Bone Black, loading from FAT with U-Boot SPL 2022.04).

Booting from raw NAND - Configuration

Proposed NAND layout
For use on Microchip SAMA5D3 Xplained

Offset	Size	Contents
0x0	256 KiB	SPL (`spl/u-boot-spl.bin`)
0x40000	1 MiB	U-Boot (`u-boot.bin`)
0x140000	128 KiB	U-Boot redundant environment
0x160000	128 KiB	U-Boot environment
0x180000	128 KiB	Original DTB or CMD
0x1a0000	6.375 MiB	uImage
0x800000		Other partitions

Notes:

- Only the SPL offset is hardcoded
- All others can be configured differently
- Offsets must be a multiple of the erase block size (128 KiB)

U-Boot configuration

```
CONFIG_SPL_OS_BOOT=y SPL_SIZE_LIMIT=0x10000
CONFIG_ENV_OFFSET=0x160000
CONFIG_ENV_OFFSET_REDUND=0x140000
CONFIG_SPL_LEGACY_IMAGE_SUPPORT=y
CONFIG_SPL_NAND_SUPPORT=y
CONFIG_SPL_NAND_DRIVERS=y
CONFIG_SPL_NAND_BASE=y
CONFIG_CMD_SPL_WRITE_SIZE=0x7000
CONFIG_CMD_SPL_NAND_OFS=0x180000
```

(starting from `sama5d3_xplained_nandflash_defconfig`)

include/configs/sama5d3_xplained.h

```
/* Generic settings */
#define CONFIG_SYS_NAND_U_BOOT_OFFS     0x40000

/* Falcon boot support on raw NAND */
#define CONFIG_SYS_NAND_SPL_KERNEL_OFFS 0x1a0000
```

Booting from raw NAND - Results and notes

- Reference test
  - To be fair, using a zero `bootdelay` and the exact `zImage` and `dtb` size:
    ```
    setenv bootdelay 0
    setenv bootcmd 'nand read 0x21000000 0x1a0000 0x53ac00; nand read
    0x22000000 0x180000 0x6c93; bootz 0x21000000 - 0x22000000'
    ```
  - Best result (using `grabserial`):
    ```
    [4.320618 0.000470] Please press Enter to activate this console.
    ```
- Falcon boot test
  - Best result (using `grabserial`):
    ```
    [3.768543 0.000125] Please press Enter to activate this console.
    ```
  - We saved 552 ms!

- Depending on how you boot, read the corresponding code:
  - `common/spl/spl_mmc.c`
  - `common/spl/spl_nand.c`
  - Other files in `common/spl/`
- If booting doesn't work, the easiest way is to add `puts();` lines to trace strategic functions and check return values. You'll get the messages in the serial console.

Issues and lessons learned (1)

- *SPL storage driver performance*: not on all platforms, but at least here on Microchip SAMA5.
- *Features limited by space*: what can be done with Falcon booting is not limited by U-Boot features, but by how much code can fit in the limited SRAM. This is why I couldn't show Falcon booting from a FAT partition, because adding support for this filesystem and disk partitions to the SPL doesn't fit in the maximum size possible on the particular platform chosen for the demo.
- *U-Boot initializations*: in addition to the FDT fixups without which the Linux kernel may not boot, the Linux kernel may also rely on some initializations performed by U-Boot. Before such dependencies can be removed by updating kernel drivers, you may need to hardcode such initializations in the SPL, provided you have enough space!

Issues and lessons learned (2)

- *Limited automation*: while the `uImage` file can be updated automatically in the storage image, any change in the kernel command line or Device Tree must go through the `spl export` command **on the board**. The FDT fixups done by U-Boot are not trivial to reproduce. This makes it difficult to prepare production images without a manual step in U-Boot.
- *No decompression*: U-Boot currently doesn't seem to support decompression in the SPL. If your architecture doesn't support kernel self-decompression and relies on the bootloader (e.g. arm64 or riscv), Falcon mode won't be available.
- *Side note*: Found that U-Boot's `bootm` was noticeably slower than `bootz` (+170 ms)

Further work

- Improve raw MMC read performance in the SPL on Microchip SAMA5
- Didn't try with what U-Boot calls the *Raw* kernel images yet, supported with `CONFIG_SPL_RAW_IMAGE_SUPPORT`. Assuming this corresponds to the `arch/arm/boot/Image`
- Didn't try FIT Image support in SPL yet. Will try on an SoC with more space for SPL (i.MX)

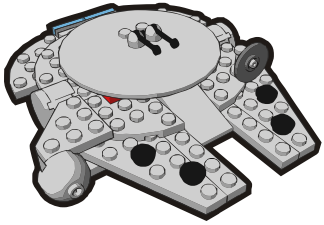
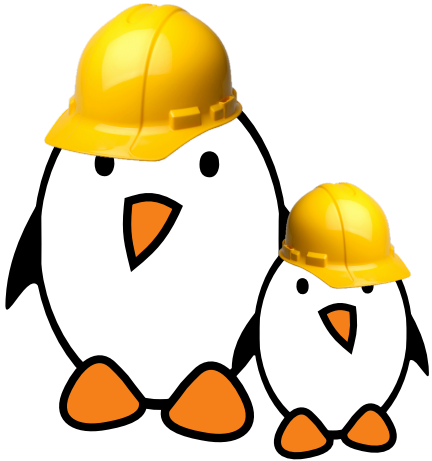


Image credits: https://openclipart.org/detail/224913/clip-is-a-brick-star-wars-millenium-falcon-set-4488

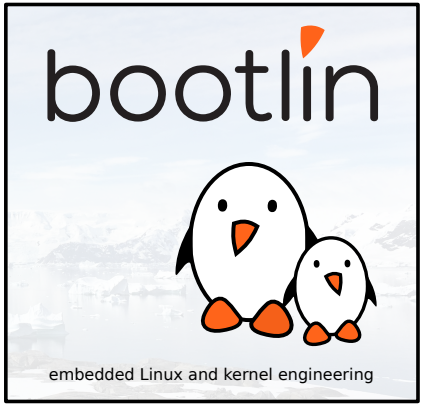
References

- Bootlin's commit to support Falcon boot on SAMA5D3 Xplained in mainline U-Boot: https://source.denx.de/u-boot/u-boot/-/commit/ea83ea5afd18
- U-Boot's doc/README.falcon file
- Linus Walleij: *How the ARM32 kernel decompresses*: https://people.kernel.org/linusw/how-the-arm32-linux-kernel-decompresses



- Experiment with faster storage
- Skipping U-Boot through the *Falcon Mode*, directly booting Linux from U-Boot SPL.
- Measuring the final boot time.

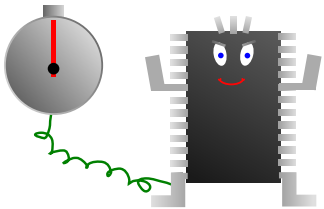
Hardware initialization



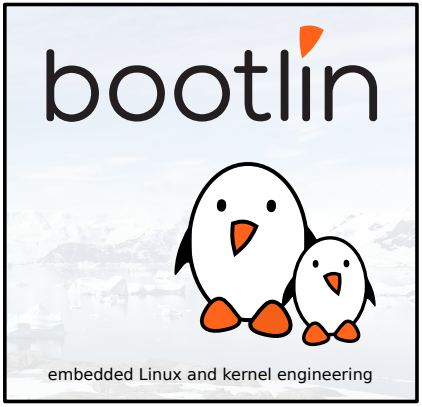
Hardware initialization

The hardware needs time to initialize

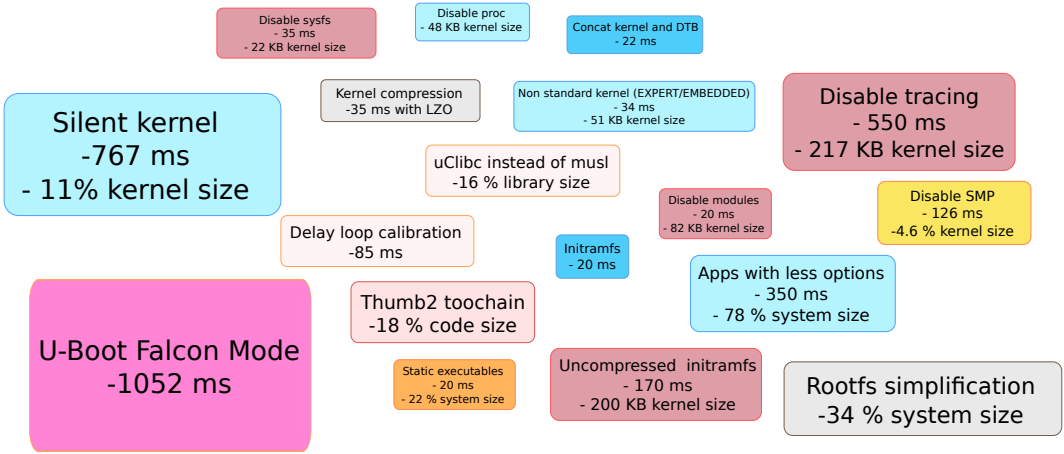
- Voltage regulation, crystal stabilization
- Can be up to 200 ms
- As a software developer, you can't do anything about this part.
- All you can do is measure this time with an oscilloscope and ask the hardware board designers whether the can do anything about this. However, there are delays in the CPU which may not be possible to reduce (see the CPU datasheet).



Conclusions

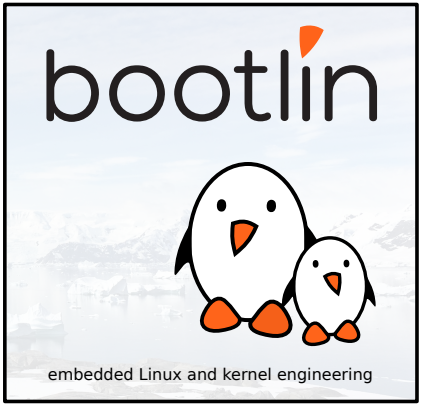


Most successful techniques in our project



Note: "kernel size" is actually "compressed kernel size" with initramfs inside

References



Conference presentations

- Andrew Murray - The Right Approach to Minimal Boot Time (2010)
  Video: https://frama.link/nrf696Hy - Slides: https://frama.link/uCBH9jQM
  Great talk about the methodology.
- Chris Simmonds - A Pragmatic Guide to Boot-Time Optimization (2017)
  Video: https://frama.link/Vnmj5t1m - Slides: https://frama.link/TC0YKM9N
- Jan Altenberg - How to Boot Linux in One Second (2015)
  Video: https://frama.link/BztbLy9T - Slides: https://frama.link/bFkvgLFR
- Michael Opdenacker - U-Boot Falcon Mode and Adding Support for New Boards (2021)
  Video: https://youtu.be/okY9fBEuaoM - Corresponding to the Falcon Mode section in this document.