# Linux Quick Boot

ELC 2014

May 1st, 2014

Tristan Lelong

Senior Embedded Software Engineer

# TABLE OF CONTENTS

1. Introduction
2. Linux boot process
3. Boot time optimization
4. How to keep a full featured system
5. Conclusion

## Introduction

## ABOUT THE PRESENTER

- Tristan Lelong
- Embedded software engineer @ **Adeneo Embedded**
- French, living in the Pacific northwest
- Embedded software, free software, and Linux kernel enthusiast.

## ARE WE TALKING ABOUT FASTBOOT?

This presentation is about quick boot.

- Fastboot is the protocol used by Android to communicate with the bootloader in order to reflash a system on the device.
- Type *fastboot* into google: 90% hits within first 10 pages are about android.

**Warning**

Fastboot is not the subject here!

## SPEED DOES MATTER

- Critical systems
  - Automotive: need to handle CAN message within a specific time frame after boot (typically x100ms)
  - Monitoring: need to reboot ASAP in case of failure
  - Health: Defibrillator
  - Deeply embedded computer: needs to stay up for a long time

## SPEED DOES MATTER

Consumer are used to previous generation of non-smart devices that starts immediately.

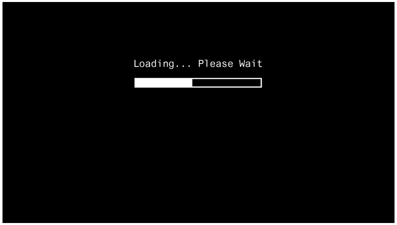
- Consumer products
  - TV, cameras: turn on quickly whenever pressing on the power button
  - Phones: deep sleep mode allowing to have a aggressive power management policy while keeping responsiveness

## SPEED DOES MATTER

It is still possible to cheat.

- Display splash screen early in the bootloader (x100ms)
- Animate the splash screen while booting.

Those tricks give the end user the illusion of a quick boot, but this could not apply to critical systems (BTCS?).



**Set your goals**

The need can be different depending on the targeted market. Consumer wont see much difference below 1 second.

## GOAL OF THE PRESENTATION

This presentation aims at:

- Showing different existing techniques to optimize boot time
- Explaining how to integrate those techniques
- Giving leads on how to optimize boot time for a custom project
- Keeping an almost full-featured Linux system running

## GOAL OF THE PRESENTATION

**This presentation will**

- Focus on standard components of a Linux system
- Focus on ARM architecture

**This presentation will not**

- List all the known methods for quick boot optimizations
- Explain how to achieve the best boot time while sacrificing many features

## INTUITION

The first things that comes in mind are usually right:

- Smaller size
  - Loads faster (bandwidth of the storage mdevice)
  - Loads from faster storage (NOR, MMC class 10)
- Remove uneeded features
  - Smaller system (see above)
  - Not wasting time

## INTUITION

But it is not always 100% accurate.

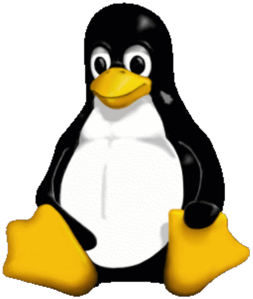
- Higher frequencies
  - Better memory bandwith
  - May require initialization
- More processing power
  - Runs critical code faster
  - High end SOC usually comes with more peripherals

# Linux boot process

## OVERVIEW

The different steps in booting an embedded system.

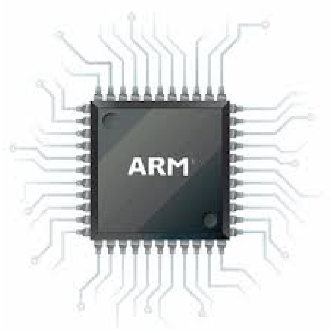
- ROM code
- Secondary Program Loader
- Bootloader
- Kernel
- Userland



## ROM CODE

The ROM code is the first program that is run by the CPU.

- Located in a ROM on the SoC
- Low level initialization
- Cannot be modified



## SECONDARY PROGRAM LOADER

The ROM code only does basic initializations, and the main RAM may not be available yet.

- Located in persistent storage (NOR, NAND, MMC)
- Low level initializations continued
- Small enough to run from internal RAM
- Can be modified (not even mandatory on some systems)

## BOOTLOADER

Once the external RAM is initialized, it is time to take care of everything else.

- Located in persistent storage (NOR, NAND, MMC)
- Low level initialization continued
- High level functions
- Can be modified

## KERNEL

The bootloader finally retrieved the kernel from any media, prepared the system and jumped onto the kernel entry point.

- Kernel may uncompress itself
- Kernel may relocate itself
- Kernel will initialize all its modules
- Kernel will mount the root filesystem
- Kernel will start the init process
- Can be modified

## USERLAND

The operating system is now up, we can start the main services and applications.

- `init` will read configuration and starts all services
- Service are traditionnaly launched by shell scripts (`inittab`, `rcS`, `SXX`, `KXX`)
- Services are started sequentially
- Can be modified

## Boot time optimization

## CONFIGURE THE ENVIRONMENT: BUILDSYSTEM

Before anything else, setup a build system.

- Small modification
- Many rebuilds
- No dummy error
- Time better spent optimizing

## CONFIGURE THE ENVIRONMENT: BUILDSYSTEM

- Buildroot: http://buildroot.uclibc.org
- Yocto: https://www.yoctoproject.org
- Custom scripts: do-it-yourself

**Toolchain recommendation**

Build systems provide the option to get a prebuilt toolchain for a quick start, or to build your own when fine tuning every bit of the system.

## CONFIGURE THE ENVIRONMENT: SCM

A source control manager is really important

- Track your modifications
- Revert a modification that is not working
- Revert a modification that is not worth the gain in time

## CONFIGURE THE ENVIRONMENT: HARDWARE

Analyzing and measuring has to be done throughout the process.

- Oscilloscope
- Identify GPIOs on the platform, LEDs, buttons
- Serial port
- JTAG [optional]
- Camera

## REQUIREMENTS ASSESSMENT

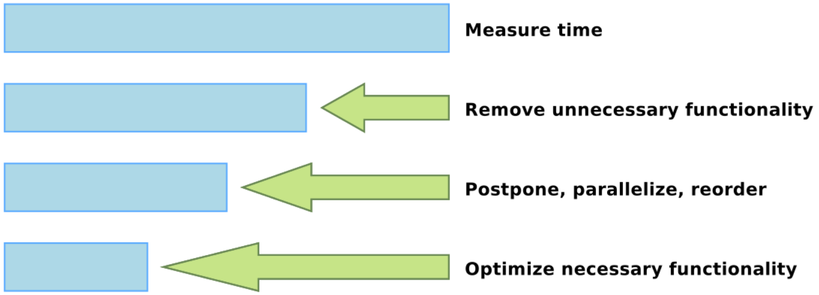
All the products do not share the same requirement.

- Boot in a few seconds
- Give some feedback within a second
- Boot in a about second (= immediate for the user)
- Acknowledge messages and basic processing in less than a second (critical system)

## REQUIREMENTS ASSESSMENT

- What are the required features
- What are the critical features
- What are the optional features

## METHODOLOGY


Measure time
Remove unnecessary functionality
Postpone, parallelize, reorder
Optimize necessary functionality

Free Electron

## MEASURE AND ANALYZE

Measuring allow to target optimization and not to waste time

- Measure the 5 boot steps to *know where to start*
- Keep the goal in mind to *know when to stop*
- Decide whether a feature has to be removed or can stay

## MEASURE AND ANALYZE

What are the performances of an embedded system "out of the box" running on the i.MX6Q Nitrogen using Freescale kernel 3.0.35.

- Running buildroot default image
  - System size: 2MB (initramfs)
  - Kernel size: 4.6MB
  - Boot time: 4.466984 seconds
- Running yocto default image
  - System size: 4.6MB (ext3)
  - Kernel size: 3.6MB
  - Boot time: 7.095716 seconds

## MEASURING TOOLS

Several opensource tools can be used to measure the different steps in the boot time.

- Printk time (kernel): this will print a timestamp in front of every `printk` done by the kernel
- Grabserial (http://eLinux.org/Grabserial): will read the serial output and can display timestamps for each line received
- Bootgraph (`kernel/scripts`): uses the `dmesg` output to generate a kernel startup graph `initcall_debug` + `CONFIG_PRINTK_TIME` + `CONFIG_KALLSYMS`
- Bootchart (http://www.bootchart.org): analyze the boot sequence in userland using `/proc` information

## MEASURING TOOLS

- Gpio: toogle a GPIO at well known phases boundaries

```
/* C code to set GPIO2_2 */
*(volatile unsigned long*)0x20a0004 = 0x00000002;
*(volatile unsigned long*)0x20a0000 = 0xc000f07f;

/* ARM Assembly to clear GPIO2_2 */
ldr     r1, =0x20a0000 @ GPIO2 base register
ldr     r5, =0x2       @ gpio2_2 as output
str     r5, [r1, #4]   @ set gpio2_2 direction
ldr     r5, =0xc000f07d @ gpio2_2 cleared
str     r5, [r1, #0]   @ clear gpio2_2
```

- Camera: record with high speed camera the boot process

## MEASURING TOOLS

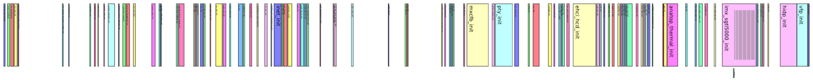


Figure: Full bootgraph

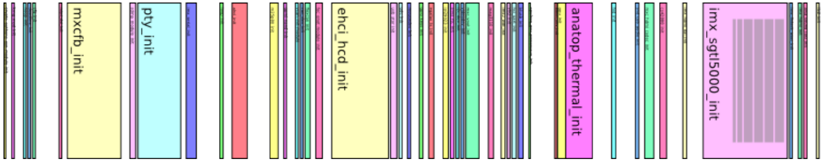
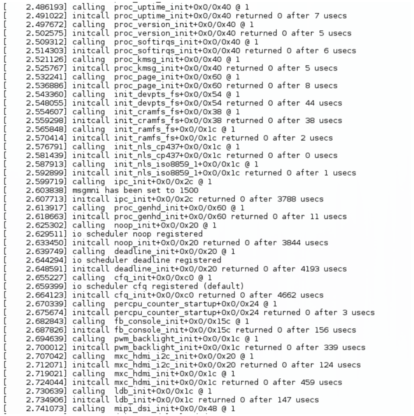


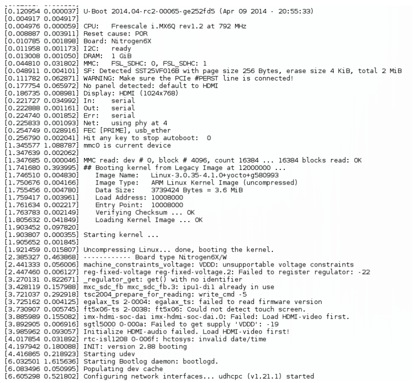
Figure: Zoomed bootgraph

## MEASURING TOOLS



```
[    2.486193] calling  proc_uptime_init+0x0/0x40 @ 1
[    2.491022] initcall proc_uptime_init+0x0/0x40 returned 0 after 7 usecs
[    2.497672] calling  proc_version_init+0x0/0x40 @ 1
[    2.502575] initcall proc_version_init+0x0/0x40 returned 0 after 5 usecs
[    2.509312] calling  proc_softirqs_init+0x0/0x40 @ 1
[    2.514303] initcall proc_softirqs_init+0x0/0x40 returned 0 after 6 usecs
[    2.521126] calling  proc_kmsg_init+0x0/0x40 @ 1
[    2.525767] initcall proc_kmsg_init+0x0/0x40 returned 0 after 5 usecs
[    2.532241] calling  proc_page_init+0x0/0x60 @ 1
[    2.536886] initcall proc_page_init+0x0/0x60 returned 0 after 8 usecs
[    2.543360] calling  init_devpts_fs+0x0/0x54 @ 1
[    2.548055] initcall init_devpts_fs+0x0/0x54 returned 0 after 44 usecs
[    2.554607] calling  init_cramfs_fs+0x0/0x38 @ 1
[    2.559298] initcall init_cramfs_fs+0x0/0x38 returned 0 after 38 usecs
[    2.565848] calling  init_ramfs_fs+0x0/0x1c @ 1
[    2.570414] initcall init_ramfs_fs+0x0/0x1c returned 0 after 2 usecs
[    2.576791] calling  init_nls_cp437+0x0/0x1c @ 1
[    2.581439] initcall init_nls_cp437+0x0/0x1c returned 0 after 0 usecs
[    2.587913] calling  init_nls_iso8859_1+0x0/0x1c @ 1
[    2.592899] initcall init_nls_iso8859_1+0x0/0x1c returned 0 after 1 usecs
[    2.599719] calling  ipc_init+0x0/0x2c @ 1
[    2.603838] msgmni has been set to 1500
[    2.607713] initcall ipc_init+0x0/0x2c returned 0 after 3788 usecs
[    2.613917] calling  proc_genhd_init+0x0/0x60 @ 1
[    2.618663] initcall proc_genhd_init+0x0/0x60 returned 0 after 11 usecs
[    2.625302] calling  noop_init+0x0/0x20 @ 1
[    2.629511] io scheduler noop registered
[    2.633450] initcall noop_init+0x0/0x20 returned 0 after 3844 usecs
[    2.639749] calling  deadline_init+0x0/0x20 @ 1
[    2.644294] io scheduler deadline registered
[    2.648591] initcall deadline_init+0x0/0x20 returned 0 after 4193 usecs
[    2.655227] calling  cfq_init+0x0/0xc0 @ 1
[    2.659399] io scheduler cfq registered (default)
[    2.664123] initcall cfq_init+0x0/0xc0 returned 0 after 4662 usecs
[    2.670339] calling  percpu_counter_startup+0x0/0x24 @ 1
[    2.675674] initcall percpu_counter_startup+0x0/0x24 returned 0 after 3 usecs
[    2.682843] calling  fb_console_init+0x0/0x15c @ 1
[    2.687826] initcall fb_console_init+0x0/0x15c returned 0 after 156 usecs
[    2.694639] calling  pwm_backlight_init+0x0/0x1c @ 1
[    2.700012] initcall pwm_backlight_init+0x0/0x1c returned 0 after 339 usecs
[    2.707042] calling  mxc_hdmi_i2c_init+0x0/0x20 @ 1
[    2.712071] initcall mxc_hdmi_i2c_init+0x0/0x20 returned 0 after 124 usecs
[    2.719021] calling  mxc_hdmi_init+0x0/0x1c @ 1
[    2.724044] initcall mxc_hdmi_init+0x0/0x1c returned 0 after 459 usecs
[    2.730639] calling  ldb_init+0x0/0x1c @ 1
[    2.734906] initcall ldb_init+0x0/0x1c returned 0 after 147 usecs
[    2.741073] calling  mipi_dsi_init+0x0/0x48 @ 1
```

Figure: Bootgraph dmesg

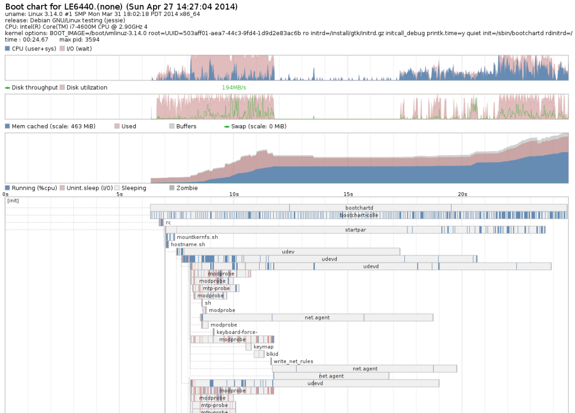


```
[0.120954 0.000037] U-Boot 2014.04-rc2-00065-ge252fd5 (Apr 09 2014 - 20:55:33)
[0.004917 0.004917]
[0.004976 0.000059] CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
[0.008887 0.003911] Reset cause: POR
[0.010785 0.001898] Board: Nitrogen6X
[0.011958 0.001173] I2C:   ready
[0.013008 0.001050] DRAM:  1 GiB
[0.044810 0.031802] MMC:   FSL_SDHC: 0, FSL_SDHC: 1
[0.048911 0.004101] SF: Detected SST25VF016B with page size 256 Bytes, erase size 4 KiB, total 2 MiB
[0.111782 0.062871] WARNING: Make sure the PCIe #PERST line is connected!
[0.177754 0.065972] No panel detected: default to HDMI
[0.186735 0.008981] Display: HDMI (1024x768)
[0.221727 0.034992] In:    serial
[0.222888 0.001161] Out:   serial
[0.224740 0.001852] Err:   serial
[0.225833 0.001093] Net:   using phy at 4
[0.254749 0.028916] FEC [PRIME], usb_ether
[0.256790 0.002041] Hit any key to stop autoboot:  0
[1.345577 1.088787] mmc0 is current device
[1.347639 0.002062]
[1.347685 0.000046] MMC read: dev # 0, block # 4096, count 16384 ... 16384 blocks read: OK
[1.741680 0.393995] ## Booting kernel from Legacy Image at 12000000 ...
[1.746510 0.004830]    Image Name:   Linux-3.0.35-4.1.0+yocto+g580993
[1.750676 0.004166]    Image Type:   ARM Linux Kernel Image (uncompressed)
[1.755456 0.004780]    Data Size:    3739424 Bytes = 3.6 MiB
[1.759417 0.003961]    Load Address: 10008000
[1.761634 0.002217]    Entry Point:  10008000
[1.763783 0.002149]    Verifying Checksum ... OK
[1.805632 0.041849]    Loading Kernel Image ... OK
[1.903452 0.097820]
[1.903807 0.000355] Starting kernel ...
[1.905652 0.001845]
[1.921459 0.015807] Uncompressing Linux... done, booting the kernel.
[2.385327 0.463868] ------------ Board type Nitrogen6X/W
[2.441333 0.056006] machine_constraints_voltage: VDDD: unsupportable voltage constraints
[2.447460 0.006127] reg-fixed-voltage reg-fixed-voltage.2: Failed to register regulator: -22
[3.270131 0.822671] _regulator_get: get() with no identifier
[3.428119 0.157988] mxc_sdc_fb mxc_sdc_fb.3: ipu1-di1 already in use
[3.721037 0.292918] tsc2004_prepare_for_reading: write_cmd -5
[3.725162 0.004125] egalax_ts 2-0004: egalax_ts: failed to read firmware version
[3.730907 0.005745] ft5x06-ts 2-0038: ft5x06: Could not detect touch screen.
[3.885989 0.155082] imx-hdmi-soc-dai imx-hdmi-soc-dai.0: Failed: Load HDMI-video first.
[3.892905 0.006916] sgtl5000 0-000a: Failed to get supply 'VDDD': -19
[3.985962 0.093057] Initialize HDMI-audio failed. Load HDMI-video first!
[4.017854 0.031892] rtc-isl1208 0-006f: hctosys: invalid date/time
[4.197942 0.180088] INIT: version 2.88 booting
[4.416865 0.218923] Starting udev
[6.032501 1.615636] Starting Bootlog daemon: bootlogd.
[6.083496 0.050995] Populating dev cache
[6.605298 0.521802] Configuring network interfaces... udhcpc (v1.21.1) started
```

Figure: Grabserial output

## MEASURING TOOLS

- bootcmd `initcall_debug printk.time=y quiet init=/sbin/bootchartd`
- requires the main script `/sbin/bootchart`



Boot chart for LE6440.(none) (Sun Apr 27 14:27:04 2014)

```
uname: Linux 3.14.0 #1 SMP Mon Mar 31 18:02:18 PDT 2014 x86_64
release: Debian GNU/Linux testing (jessie)
CPU: Intel(R) Core(TM) i7-4600M CPU @ 2.90GHz 4
kernel options: BOOT_IMAGE=/boot/vmlinuz-3.14.0 root=UUID=503aff01-aea7-44c3-9fd4-1d9d2e83ac6b ro initrd=/install/gtk/initrd.gz initcall_debug printk.time=y quiet init=/sbin/bootchartd rdinitrd=/
time : 00:24.67      max pid: 3594
```

## MEASURING TOOLS

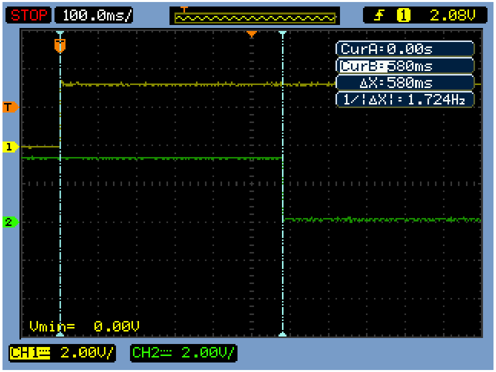


Figure: GPIO: measuring the ROM Code duration

- Yellow trace: `GPIO2_2`: default to high on reset
- Green trace: Reset Button: low when pressed

## SERIAL OUTPUT

The serial output is usually configured to 115200 bauds

- A standard u-boot output is about 500 char: 4ms
- A standard kernel output is about 30000 char: 260ms
- Really, the gap is above 1 second

Adding `quiet` to the kernel command line or by simply removing printk support.

**Removing the printk**

Removing printk will increase the boot speed in two ways.

- No data on serial line
- Smaller kernel (500kB uncompressed)

## PRESET LPJ

During the early boot process, the Linux kernel calculate the `loop_per_jiffy`.

- CPU in a loop for up to 250ms on some system
- On i.MX6 it takes about 80ms
- Could be preset using configuration entry or bootargs

## SMP

Booting an SMP system require a big amount of time.

- Booting 1 CPU is around 80ms
- Bootargs `maxcpus=1`
- Init script

  ```
  echo 1 > /sys/devices/system/cpu/cpu[123]/online
  ```

## U-BOOT TIMEOUT

One easy but sometimes forgotten optimization is to remove the timeout in u-boot. It is typically between 1 and 10 seconds.

```
/* nitrogen6x.h */
#define CONFIG_BOOTDELAY          1
```

| Bootdelay (sec) | Percentage |
|---|---|
| 0 | 8% |
| 1 | 30% |
| 2 | 3% |
| 3 | 25% |
| 5 | 30% |
| 10 | 4% |

Table: Use of the autoboot timeout

## KERNEL SIZE

The loading time of the kernel from the permanent storage to RAM takes a non neglectable time. Dividing its size by two means saving some precious milliseconds.

The kernel size can be reduced using two methods.

- Compression
- Configuration

## KERNEL COMPRESSION

The kernel can be compressed using different algorithms. Each having different caracteristics.

- Compression speed
- Decompression speed
- Compression ratio

### Need for compression

The need for compression depends on the CPU and the memory bandwidth. By measuring different configurations, one will be able to detect the bootle-neck and select the proper solution accordingly.

## KERNEL COMPRESSION

- None: no decompression needed but big image.
- GZIP: Standard ratio, Standard decompression/compression speed
- LZMA: Best compression ratio, but slow to decompression/compress
- XZ (LZMA2): close to LZMA
- LZO: Bad ratio, but fast decompression/compression

## KERNEL CONFIGURATION

The kernel provides a configuration mechanism allowing it to be a match for deeply embedded system up to super computers.

Carefully enabling only the required option is the best way to reduce kernel size as well as the number ofcode that will be ran.

**Warning**

- Removing some option may prevent the system to boot.
- Enabling a feature that was previously disabled doesn't simply work.

Always proceed one chunk of option at a time and commit this change so that you will be able to revert it.

## KERNEL CONFIGURATION CONTINUED

- Mtd support
  - Device Drivers -> Memory Technology Device (MTD) support
  - saves about **700kB**
- Block support
  - Device Drivers -> Enable the block layer
  - saves about **1.2MB**
- Sound support
  - Device Drivers -> Sound card support
  - saves about **300kB**
- Misc drivers
  - Device Drivers
  - USB, SATA, Network, MMC, Staging

## KERNEL CONFIGURATION CONTINUED

- Networking stack
  - Networking support
  - saves about **2MB**
- Kernel .config support
  - General setup -> Kernel .config support
  - saves about **80kB**
- Optimize for size
  - General setup -> Optimize for size
  - saves about **500kB**

## KERNEL CONFIGURATION CONTINUED

- Printk support
  - General setup -> Configure standard kernel features -> Enable support for printk
  - saves about **500kB**
- BUG() support
  - General setup -> Configure standard kernel features -> BUG() support
  - saves about **100kB**
- Debug Filesystem
  - Kernel hacking -> Compile-time checks and compiler options -> Debug Filesystem
  - saves about **80kB**
- Debug symbols
  - General setup -> Configure standard kernel features -> Load all symbols for debugging/ksymoops
  - saves about **700kB**

## INITRAMFS SIZE

The `initramfs` is the initial rootfs that will be loaded with the kernel at boot time.

It will be used as a `tmpfs` (file system in RAM), therefore the `init` process will run faster.

It has to contain the minimum in order run the critical services. Other files can be mounted from persistent storage in a second time.

**initramfs compression**

Do not compress initramfs if you plan on appending in to the Linux kernel since, it will be included before compressing the kernel.

## INITRAMFS SIZE

To keep a small initramfs size, busybox is a perfect match.

- http://www.busybox.net
- http://wiki.musl-libc.org/wiki/Alternative_libraries

## MKLIBS [STEP 1]

To make sure that the initramfs is as small as possible, there is a tool that will help automatize the process: `mklibs`

- Analyze ELF binaries to detect symbols and dependencies
- Copy only required library to satisfy dependencies
- Doesn't detect dlopen

There are 2 versions.

- Debian: python
- Gentoo: shell

The use of mklibs can be a little bit tricky, therefore, it is recommended to use a buildroot `BR2_ROOTFS_POST_BUILD_SCRIPT` for instance.

## MKLIBS [STEP 1]

Example:

```
MKLIBS=$(which mklibs)
SYSROOT="-L $BASE_DIR/target.full/lib $BASE_DIR/target.full/usr/
    lib"
OUTPUT="$BASE_DIR/target/lib/"
BIN="$BASE_DIR/target/bin/*"

export OBJDUMP=arm-buildroot-Linux-uclibcgnueabi-objdump
export OBJCOPY=arm-buildroot-Linux-uclibcgnueabi-objcopy
export GCC=arm-buildroot-Linux-uclibcgnueabi-gcc

$MKLIBS $SYSROOT -d $OUTPUT $BINS
```

## INIT SCRIPT CUSTOMIZATION

Buildsystem / distributions provide a init binary that will read a configuration file and start services in a specific order.

It is a good idea to start critical services early in the boot process and make sure that they depend on as few other services as possible.

## INIT SCRIPT CUSTOMIZATION

sysV `init` with its `inittab` and `rcS` is used a lot in embedded systems.

- init
  - `init` is available in `busybox`
  - `init` is an ELF binary
- inittab
  - `inittab` is a configuration file
  - `inittab` offers a respawn function
  - `inittab` eventually loads `rcS`
- rcS
  - It is a shell script
  - It fork/exec all user services

## INIT SCRIPT CUSTOMIZATION

```
# Startup the system
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mkdir -p /dev/pts
null::sysinit:/bin/mkdir -p /dev/shm
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# Put a getty on the serial port
ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100

# Stuff to do for the 3-finger salute
::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
null::shutdown:/etc/init.d/rcK
null::shutdown:/bin/umount -a -r
null::shutdown:/sbin/swapoff -a
```

## GOING FURTHER

Going further

## U-BOOT FALCON MODE

The i.MX6 SoC family doesn't need a SPL thanks to the IVT header and DCD table.

- Bootloader entry point
- Device Configuration Data

When the bootloader starts, the external memory and major clocks are initialized.

## U-BOOT FALCON MODE

SPL has a secondary feature named *Falcon mode* that allows skipping the bootloader step and directly run the Operating System.

```
/* SPL target boot image */
#define CONFIG_CMD_SPL
#define CONFIG_SPL_OS_BOOT          /* falcon mode */
#define CONFIG_SYS_SPL_ARGS_ADDR    0x4f542000

/* SPL Support for MMC */
#define CONFIG_SPL_MMC_SUPPORT
#define CONFIG_SPL_GPIO_SUPPORT
#define CONFIG_SPL_MMC_MAX_CLK 198000000
#define CONFIG_SPL_BOOT_DEVICE BOOT_DEVICE_MMC1
#define CONFIG_SPL_BOOT_MODE MMCSD_MODE_RAW
#define CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR 2
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTOR 0x400
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTORS 1
#define CONFIG_SYS_MMCSD_RAW_MODE_KERNEL_SECTOR 0x800
```

## U-BOOT FALCON MODE

The SPL has to be implemented for every board / SOC / CPU if not already done (TI boards: MLO).

- Generic
  - Common/spl/spl.c
  - Common/spl/spl_fat.c
  - Common/spl/spl_mmc.c
  - Common/spl/spl_nand.c
  - Common/spl/spl_nor.c
  - ...
- ARM specific
  - Arch/arm/lib/spl.c
- SOC specific
  - Arch/arm/cpu/armv7/mx6/spl.c
  - Arch/arm/cpu/armv7/omap-common/boot-common.c
- Board specific
  - Board/freescale/p1022ds/spl.c

## U-BOOT FALCON MODE

Handful of functions need to be implemented.

- `board_init_f`: SPL
- `spl_board_init`: SPL
- `spl_boot_device`: SPL
- `spl_boot_mode`: SPL
- `spl_start_uboot`: Falcon

## U-BOOT FALCON MODE

One command to generate the context:

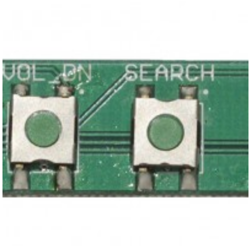
```
#define CONFIG_CMD_SPL
```

- Run bootcmd
- Read bootargs
- Generate ATAGS
- Generate FDT

```
U-boot> spl export atags ${loadaddr}
## Booting kernel from Legacy Image at 12000000 ...
  Image Name: Linux-3.0.35
  Image Type: ARM Linux Kernel Image (uncompressed)
  Data Size:  4865520 Bytes = 4.6 MiB
  Load Address: 10008000
  Entry Point: 10008000
  Verifying Checksum ... OK
  Loading Kernel Image ... OK
  Argument image is now in RAM at: 0x10000100
```

## U-BOOT FALCON MODE

The `spl_start_uboot` function can be static, but can also read a GPIO to decide at run time to boot the *OS* or jump to *u-boot*.



## CUSTOM INIT

Using our custom application as `init`

- Starts all dependencies for the critical application
- Avoid forking: this takes time (100us)
- Run the application code

**Main app running as PID 1**

Here, there is no management of the background processes and the application should never die!

## DEFERRED INITCALLS

Deferred initcall is a way to initialize kernel feature in phases.

1. Run all `initcalls` but the deferred ones
2. Run `init` process
3. `init` process signal the kernel to run remaining initcalls

### deferred initcall patch

Support for deferred initcalls is not integrated into Linux mainline and requires a patch to be applied (available from http://eLinux.org/Deferred_Initcalls):

- 2.6.26, 2.6.27: patch by Tim Bird
- 2.6.28: patch by Simonas Leleiva
- 3.10: patch by Alexandre Belloni

## DEFERRED INITCALLS

`initcall` is a macro that puts the initialization function of every module in a specific section of the ELF binary.

```
module_init(my_module_init);
module_exit(my_module_exit);
```

At boot time, the kernel runs all the `initcall` functions before even starting the `init` process.

Some kernel modules are not required for critical application. Using a small code modification, it is possible to tell the kernel to skip those by putting them in a new section.

```
#define deferred_initcall(fn) \
        static initcall_t __initcall_##fn \
        __used __section(.deferred_initcall.init) = fn
```

## DEFERRED INITCALLS

To identify which module should be deffered, there is no configuration. Each modules needs to be specifically modified.

```
deferred_module_init(my_module_init);
module_exit(my_module_exit);
```

And the init process needs to notify the kernel whenever it is ok to intialize those modules.

```
root@target:~# cat /proc/deferred_initcalls
```

## CUSTOM TOOLCHAIN

Not all toolchains are created equal. Changing toolchains, will usually make you gain a few hundred of ms.

- Prebuilt toolchains are generic and support many SOCs
- Can use the `-mcpu= -march= -mtune=` options

The c library also matters.

- Uclibc: http://www.uclibc.org/
- Musl libc: http://www.musl-libc.org
- Dietlibc: http://www.fefe.de/dietlibc
- Newlib: http://sourceware.org/newlib

## MKLIBS [STEP 2]

Mklibs has a specific feature to strip uneeded symbols from libraries.

- Requires a specific toolchain with *libxxxx_pic.a* included

```
MKLIBS=$(which mklibs)
SYSROOT="$BASE_DIR/target.full/lib"
OUTPUT="$BASE_DIR/target/lib/"
BIN="$BASE_DIR/target/bin/*"

export OBJDUMP=arm-buildroot-Linux-uclibcgnueabi-objdump
export OBJCOPY=arm-buildroot-Linux-uclibcgnueabi-objcopy
export GCC=arm-buildroot-Linux-uclibcgnueabi-gcc

$MKLIBS -L $SYSROOT -d $OUTPUT $BINS
```

## STATIC /DEV

- udev
- mdev (populating using `mdev -s` can take about 100ms)

If the system doesn't require hotplug, static device nodes or `devtmpfs` are faster.

A Linux system requires:

- `/dev/null`
- `/dev/console`

```
mknod -m 622 /dev/console c 5 1
mknod -m 666 /dev/null c 1 3
```

## IS INITRAMFS REALLY FASTER

`initramfs` runs from RAM and therefore has a better bandwidth.

- It makes the kernel image bigger
- Copies the data twice
- Load the whole data inconditionnally

Some measures show:

- Using a 1MB cpio (500kB cramfs), the gap is 400ms with initramfs being faster.
- Using a 10MB cpio (5MB cramfs), the gap is 100ms with initramfs being faster.
- Using a 18MB cpio (13MB cramfs), the gap is 300ms with cramfs being faster.

## How to keep a full featured system

## TRADEOFF WHEN DOING QUICK BOOT

- Reducing the size of the kernel will cause dropping support for some peripherals
- Reducing the size of the userland will cause losing extra functionnality

This works fine for specialized systems with only one well identified task.

## TRADEOFF WHEN DOING QUICK BOOT

Smart devices requires more and more features.

- Connectivity: WiFi, Bluetooth, NFC
- Rich user interfaces: full blown webserver, graphical toolkit, opengl library

Those devices need a Linux system that is not completely stripped down.

## KERNEL SIDE

When configuring there are usually 3 options.

- Disabled
- Built-in
- Module

The third option is interesting: hotplug kernel features.

**modules**

Not all the entries in the kernel configuration can be built as external modules. Only *tristate* Ex: *Networking support*

## KERNEL SIDE

The goal is to add all the extra features as external kernel modules.

- Build using the `make modules` rule
- Install using the `make modules_install` rule along with `INSTALL_MOD_PATH`=<`rootfs path`>
- Store them in persistent storage (not in uImage)
- Load them after running critical application or on demand using udev/mdev

## USERLAND SIDE

From

- Tiny tmpfs rootfs
- Custom init started
- Main application running

To

- Full featured rootfs
- Standard init running

## USERLAND SIDE

The solution is:

- Mounting a new rootfs
- Preparing its content
- Replace the previous /
- Run `init`

For this 2 alternatives:

- `pivot_root`
- `switch_root`

## PIVOT_ROOT

`pivot_root` is the simple old tool to do the job.

- It only changes the rootfs of the current process
- It switches `new-root` and `old-root`
- Requires chroot to do a complete switch
- Doesn't allow to run a new init

```
root@target:~# mount /dev/mmcblk0p1 /new-root
root@target:~# mount --move /sys /newroot/sys
root@target:~# mount --move /proc /newroot/proc
root@target:~# mount --move /dev /newroot/dev
root@target:~# cd /new-root
root@target:~# pivot_root . old-root
root@target:~# exec chroot . sh <dev/console >dev/console 2>&1
root@target:~# umount /old-root
```

## SWITCH_ROOT

`switch_root` is much more integrated. This is the solution used in `initramfs`.

- It switches rootfs
- It exec chroot and release the old console
- It removes all the files from `old-root` (useful for `tmpfs`)
- It runs the `new-root` filesystem `init`

```
root@target:~# mount /dev/mmcblk0p1 /new-root
root@target:~# mount --move /sys /newroot/sys
root@target:~# mount --move /proc /newroot/proc
root@target:~# mount --move /dev /newroot/dev
root@target:~# exec switch_root /newroot /sbin/init
```

## Conclusion

## RESULTS & DEMO

Progression

- Standard system: **4.466984 seconds**
- bootdelay + bootargs: *2.324392 seconds*
- Falcon mode: *1.737441 seconds*
- Stripped kernel: (down to 1.9MB with initramfs)*1.405953 seconds*
- Stripped initramfs: *1.162203 seconds*
- Deferred initcall: **0.886289 seconds**

## CONCLUSION

Quick boot is **always** a matter of trade-off.

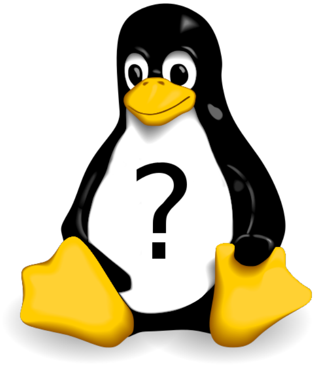
## CONCLUSION

- Quick boot used to be black magic
- Quick boot used to be completely target specific

**BUT**

- Many open source solutions exist
- Several recipes aiming for *"generic"* solutions

## QUESTIONS



## REFERENCES

- http://www.denx.de/wiki/U-Boot
- https://www.kernel.org/
- http://www.etalabs.net/compare_libcs.html
- http://eLinux.org/Deferred_Initcalls
- http://eLinux.org/Boot_Time
- http://free-electrons.com/doc/training/boot-time/slides.pdf