How to optimize the boot time for Android

Contents

Stable: 15.04.2021 - 15:55 / Revision: 15.04.2021 - 15:51

Stable: 15.04.2021 - 15:09 | Revision: 15.04.2021 - 15:51

A quality version of this page, approved on *15 April 2021*, was based off this revision.

The purpose of this document is to provide information on how to measure and improve the boot time of a typical STM32MP15 distribution for Android™. This article does not provide an exhaustive list of possible optimizations since those that are considered insufficiently reliable for industrial use are intentionally omitted.

Contents

1 Overview

1.1 BSP stage

On a typical STM32MP1 Linux system, the first stages of the boot process are performed in order by the ROM code, TF-A, U-Boot and the Linux kernel. All these components, except for the ROM code, can be modified and thus configured to start more quickly (please refer to How to optimize the boot time page to get some recommendations).

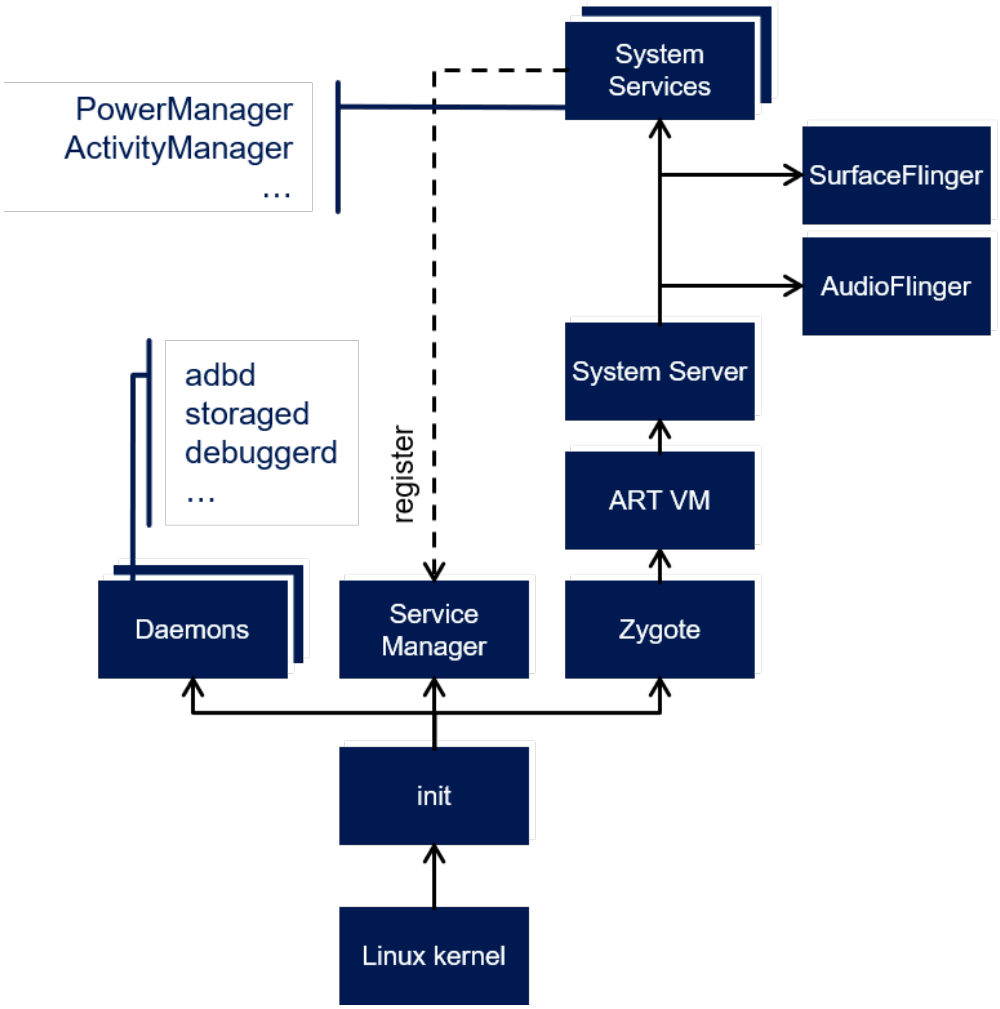
The procedure is the same for all of them: the features that are not required at boot time must be activated (or disabled) after system boot, while the features that improve the boot time must be enabled.

1.2 Android stage

The second stage of the boot process is performed by the the Android™ init.

The main tasks executed as the following:

- initialize the process
  - mount kernel file system (procfs, sysfs, selinuxfs, tmpfs...)
- mount logical read-only partitions (incl. rootfs) if any (system, vendor, product...)
  - load SELinux policy
  - load persisted properties
- initialize based on .rc files
  - parse initialized .rc files
  - start daemons (ueventd, apexd, healthd, ashmed, lmkd, installd, statsd, usbd, storaged, wificond...)
  - start media processes (audioserver, cameraserver, media, mediaextractor, media. swcodec, mediadrm, mediametrics...)
  - start debug processes (console, traced, traced_probes...)
  - start other processes (keystore, gatekeeperd, surfaceflinger, tombstoned, update_engine...)
  - start HAL layer (boot, keystore, audio, camera, allocator, configstore, dumpstate, light, memtrack oemlock, thermal, usb, wifi...)
  - load required kernel modules (.ko files)



    - mount userdata partition
- start Virtual Machine
- start Zygote (incl. resources and classes preload)
- start Bootanim
- start Services
- parse Packages (.apk files) and start Activities

2 Measuring the boot time

Before optimizing the performance of any piece of software, the time duration of each part must be considered so that the effort can be effectively focused.

2.1 Using a serial console

One of the easiest way to measure the boot time of a Linux system is to observe the traces emitted on a serial console. This can be achieved by using a timing software such as the following *measure-timing.sh* script based on microcom and p2f:

```
#!/bin/bash

echo 'Waiting for board reset...'

p2f-wait 'NOTICE:  CPU:'
t0=$(</proc/uptime)

p2f-wait 'U-Boot 20'
t1=$(</proc/uptime)

p2f-wait 'Starting kernel ...'
t2=$(</proc/uptime)

p2f-wait 'init process'
t3=$(</proc/uptime)

p2f-wait 'init second stage started!'
t4=$(</proc/uptime)

p2f-wait 'BootAnalyze: boot completed'
t5=$(</proc/uptime)

t0=$(echo "$t0" | awk '{print $1}')
t1=$(echo "$t1" | awk '{print $1}')
t2=$(echo "$t2" | awk '{print $1}')
t3=$(echo "$t3" | awk '{print $1}')
t4=$(echo "$t4" | awk '{print $1}')
t5=$(echo "$t5" | awk '{print $1}')

echo ''
echo ''
echo 'Timing results:'
echo "FSBL:  $(echo "scale=2; $t1 - $t0" | bc)s"
echo "SSBL:  $(echo "scale=2; $t2 - $t1" | bc)s"
echo "Linux: $(echo "scale=2; $t3 - $t2" | bc)s"
echo "Init first stage end: $(echo "scale=2; $t4 - $t3" | bc)s"
echo "Boot complete: $(echo "scale=2; $t5 - $t0" | bc)s"
```

prerequisites:

- insure that the log level has been set at least to "8" by modifying `device/stm/<STM32Series>/<BoardId>/BoardConfig.mk`

```
BOARD_KERNEL_CMDLINE += loglevel=8
```

- insure that the different messages expected are available on the boot trace. To do this add the required messages by modifying `device/stm/<STM32Series>/<BoardId>/init.stm.rc` (example with latest message waited by the script)

```
on property:sys.boot_completed=1
    write /dev/kmsg "BootAnalyze: boot completed"
```

Then execute:

```
 PC $> microcom -p /dev/ttyACM0 | bash measure-timing.sh
```

2.2 Using ATRACE

The different actions performed during boot phase can be traced using atrace[1] based on ftrace. Here the objective is not to determine the entire boot time but some subparts, considering that the tracing itself has an impact.

First, enable tracing during the boot phase

- add ftrace configuration in the kernel command line by modifying `device/stm/<STM32Series>/<BoardId>/BoardConfig.mk`

```
BOARD_KERNEL_CMDLINE +=trace_buf_size=64M
BOARD_KERNEL_CMDLINE +=trace_event=sched_process_exit,sched_switch,sched_process_free,
task_newtask,task_rename
```

- enable atrace flags by modifying `device/stm/<STM32Series>/<BoardId>/device.mk`

```
PRODUCT_PROPERTY_OVERRIDES += \
    debug.atrace.tags.enableflags=802922 \
    persist.traced.enable=0
```

- remove atrace disabling during initialization by modifying `frameworks/native/cmds/atrace/atrace.rc`

```
    write /sys/kernel/debug/tracing/tracing_on 1
    write /sys/kernel/tracing/tracing_on 1
```

- disable atrace as soon as the boot has completed by modifying `device/stm/<STM32Series>/<BoardId>/init.stm.rc`

```
on property:sys.boot_completed=1
    write /d/tracing/tracing_on 0
    write /d/tracing/events/ext4/enable 0
    write /d/tracing/events/f2fs/enable 0
    write /d/tracing/events/block/enable 0
```

Then rebuild the distribution and flash again the device (see How to build STM32MPU distribution for Android).

You can restart the device and get back the trace:

```
PC $> adb shell cat /d/tracing/trace > boot_trace
```

The boot_trace file can be open using Perfetto tool (see Perfetto for more details).

3 Optimizing boot time

3.1 BSP stage

Refer to How to optimize the boot time for information on how to optimize the first stage.

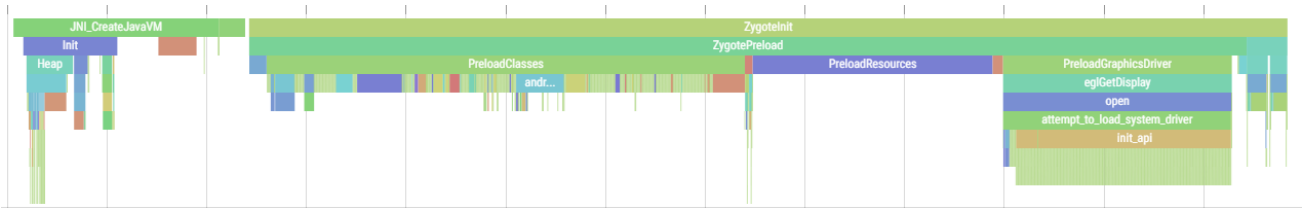
3.2 Android stage

First refer to the Android™ source page dedicated to the boot time[2].

Since I/O performance has an important impact on the boot time, it is key to select an efficient storage solution.

To go further for non-certified devices, patch the Android frameworks.

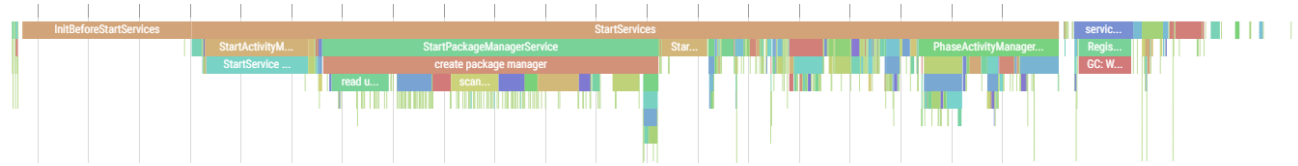
ZygoteInit phase



Unnecessary preloaded classes can be removed (refer to the list available in `frameworks/base/config/preloaded-classes`).

Take care that removing a class that is required later can impact negatively the boot time.

SystemServer phase



Unnecessary service started by the system server can be removed (refer to the list available in `frameworks/base/services/java/com/android/server/SystemServer.java`).

Take care that a removed service can no more be accessed in the frameworks nor in your applications. It is recommended to patch all the calls in the frameworks to *getSystemService(Context.<REMOVED_SERVICE>)*, considering that a null pointer is returned if the service does not exist.

It is also recommended to limit the number of the application packages (.apk) required (that is parsed during boot phase).

3.3 References

- https://source.android.com/devices/tech/debug/ftrace
- https://source.android.com/devices/tech/perf/boot-times

Board support package

Linux® is a registered trademark of Linus Torvalds.

Read Only Memory

Trusted Firmware for Arm® Cortex®-A

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Process File System (See https://en.wikipedia.org/wiki/Procfs for more details)

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Hardware Abstraction Layer

Central processing unit

First Stage Boot Loader

Second Stage Boot Loader

stm32mp1

eval,disco (Generic term used, to complete configuration modules paths depending on used board)