# 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).

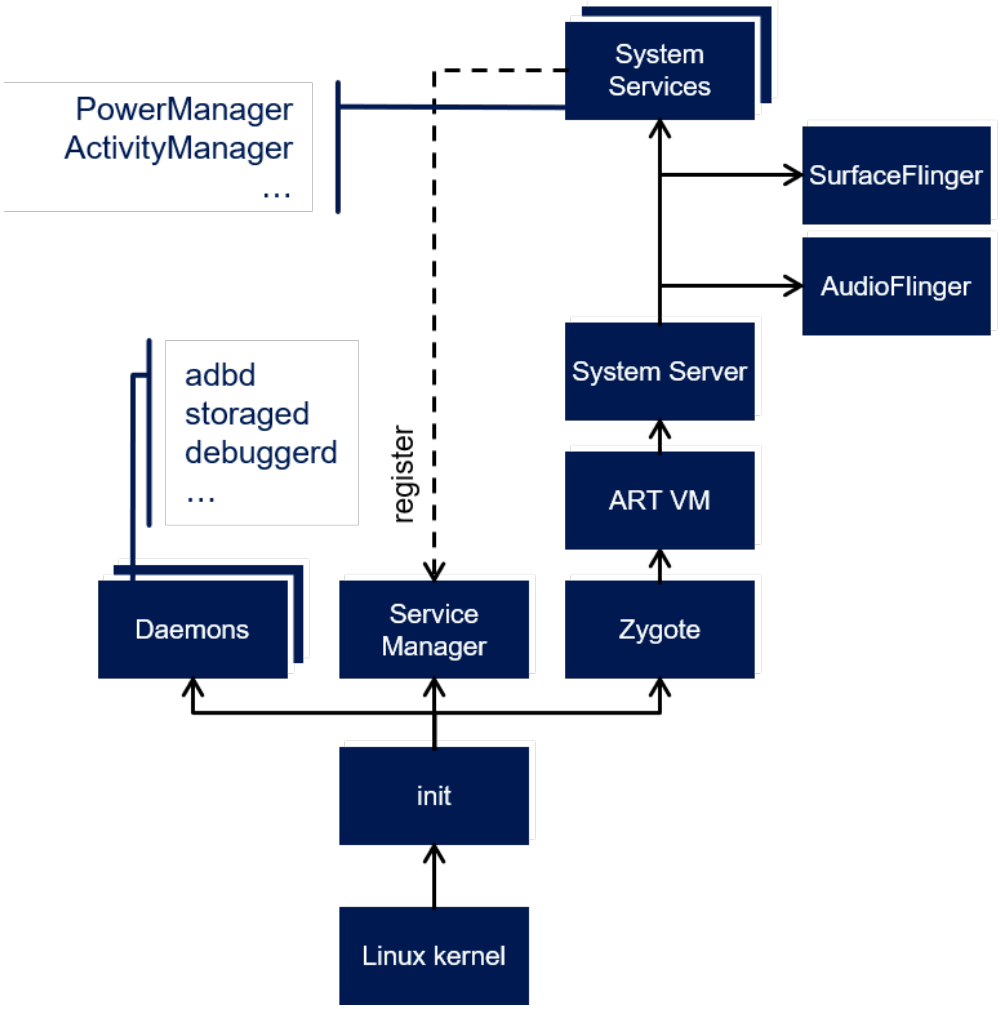
The procedure is the same for all of them: the features that are not required at boot time must be activated (or disabled) after system boot, while the features that improve the boot time must be enabled.

## 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.

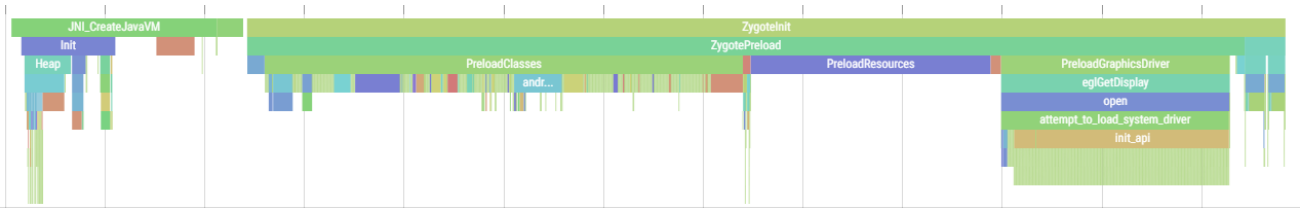
## 3.2 Android stage

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.

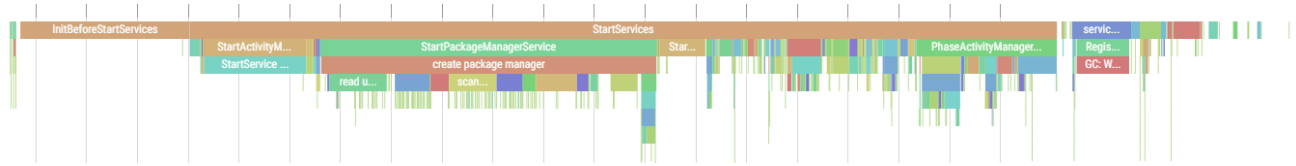
### ZygoteInit phase




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)

Stable: 12.03.2021 - 11:29 / Revision: 12.03.2021 - 11:15

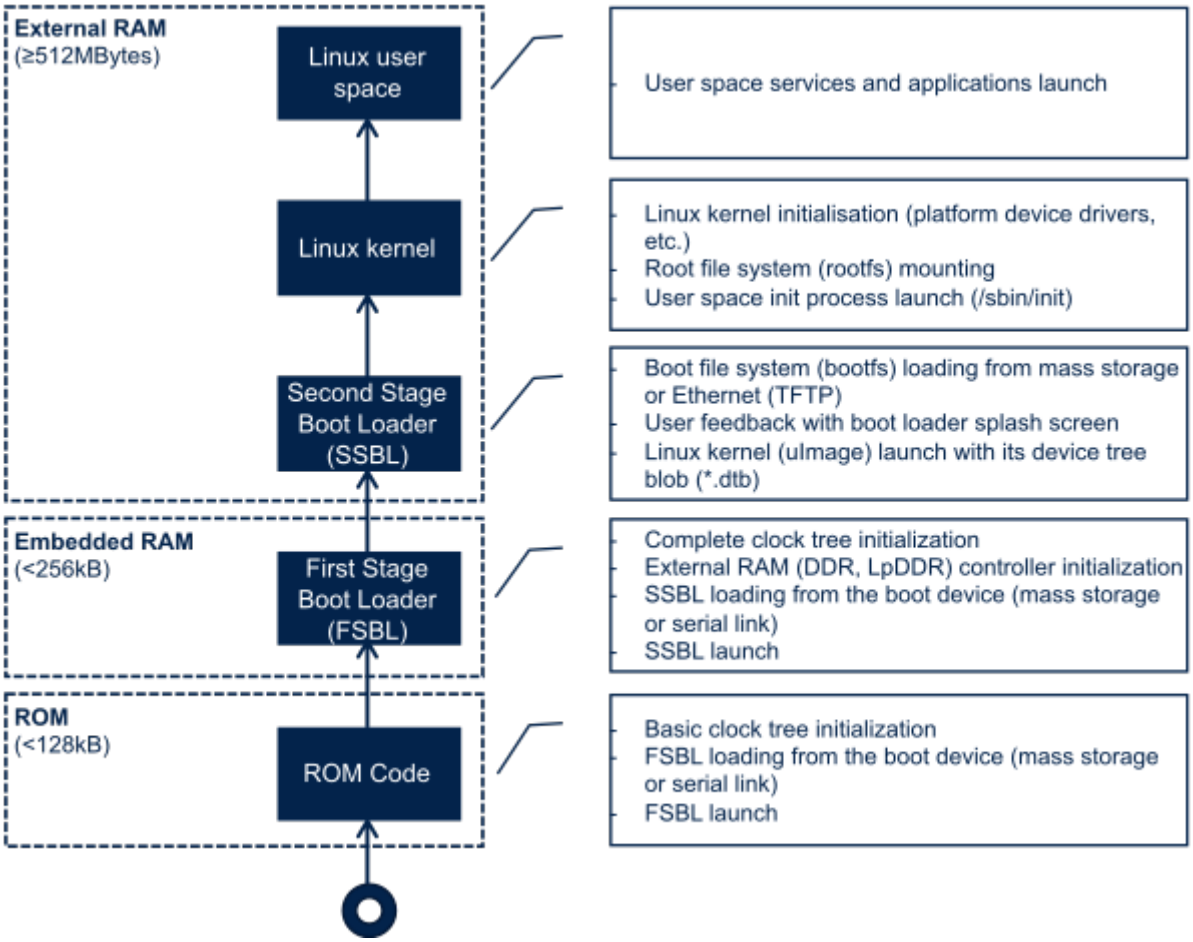
A quality version of this page, approved on *12 March 2021*, was based off this revision.

Contents

# 1 Generic boot sequence

## 1.1 Linux start-up

Starting Linux® on a processor is done in several steps that progressively initialize the platform peripherals and memories. These steps are explained in the following paragraphs and illustrated by the diagram on the right, which also gives typical memory sizes for each stage.




### 1.1.1 ROM code

The ROM code is a piece of software that takes its name from the read only memory (ROM) where it is stored. It fits in a few tens of Kbytes and maps its data in embedded RAM. It is the first code executed by the processor, and it embeds all the logic needed to select the boot device (serial link or Flash) from which the first-stage boot loader (FSBL) is loaded to the embedded RAM.

Most products require to trust the application that is running on the device and the ROM code is the first link in the chain of trust that must be established across all started components: this trust is established by authenticating the FSBL before starting it. In turn, the FSBL and each following component will authenticate the next one, up to a level defined by the product manufacturer.

### 1.1.2 First stage boot loader (FSBL)

Among other things, the first stage boot loader (FSBL) initializes (part of) the clock tree and the external RAM controller. Finally, the FSBL loads the second-stage boot loader (SSBL) into the external RAM and jumps to it.

The Trusted Firmware-A (TF-A) and U-Boot secondary program loader (U-Boot SPL) are two possible FSBLs.

### 1.1.3 Second-stage boot loader (SSBL)

The second-stage boot loader (SSBL) runs in a wide RAM so it can implement complex features (USB, Ethernet, display, and so on), that are very useful to make Linux kernel loading more flexible (from a Flash device, a network, and so on), and user-friendly (by showing a splash screen to the user). U-Boot is commonly used as a Linux bootloader in embedded systems.

### 1.1.4 Linux kernel space

The Linux kernel is started in the external memory and it initializes all the peripheral drivers that are needed on the platform.

### 1.1.5 Linux user space

Finally, the Linux kernel hands control to the user space starting the init process that runs all initialization actions described in the root file system (rootfs), including the application framework that exposes the user interface (UI) to the user.

## 1.2 Other services start-up

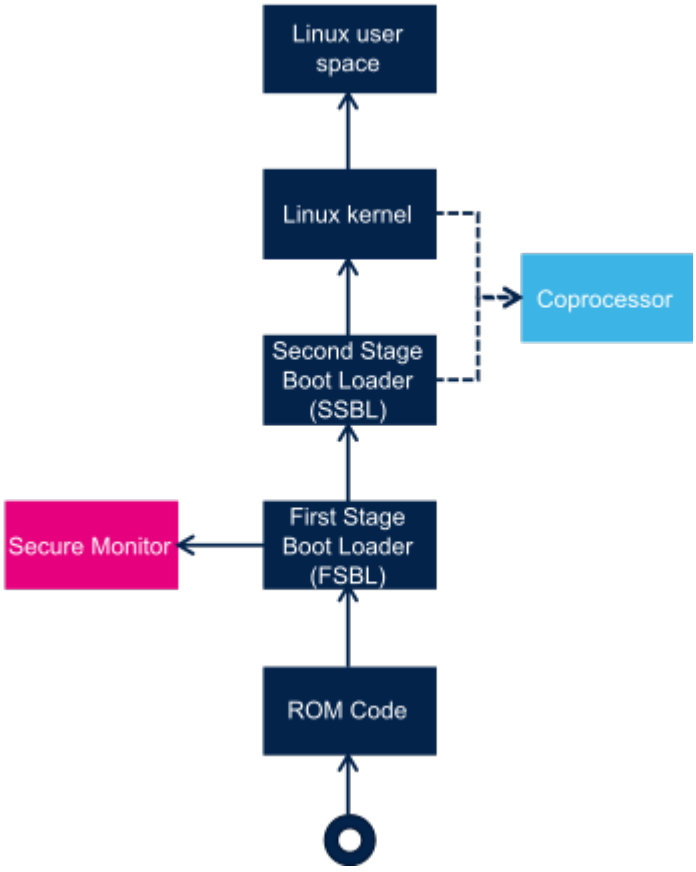
In addition to Linux startup, the boot chain also installs the secure monitor and may support coprocessor firmware loading.

For instance, for the STM32MP15, the boot chain starts:

- the secure monitor, supported by the Arm®Cortex®-A secure context (TrustZone). Examples of use of a secure monitor are: user authentication, key storage, and tampering management.
- the coprocessor firmware, running on the Arm Cortex-M core. This can be used to offload real-time or low-power services.

The dotted lines in the diagram on the right mean that:

- the coprocessor can be started by the **second stage boot loader (SSBL)**, known as “**early boot**”, or **Linux kernel**



# 2 STM32MP boot sequence

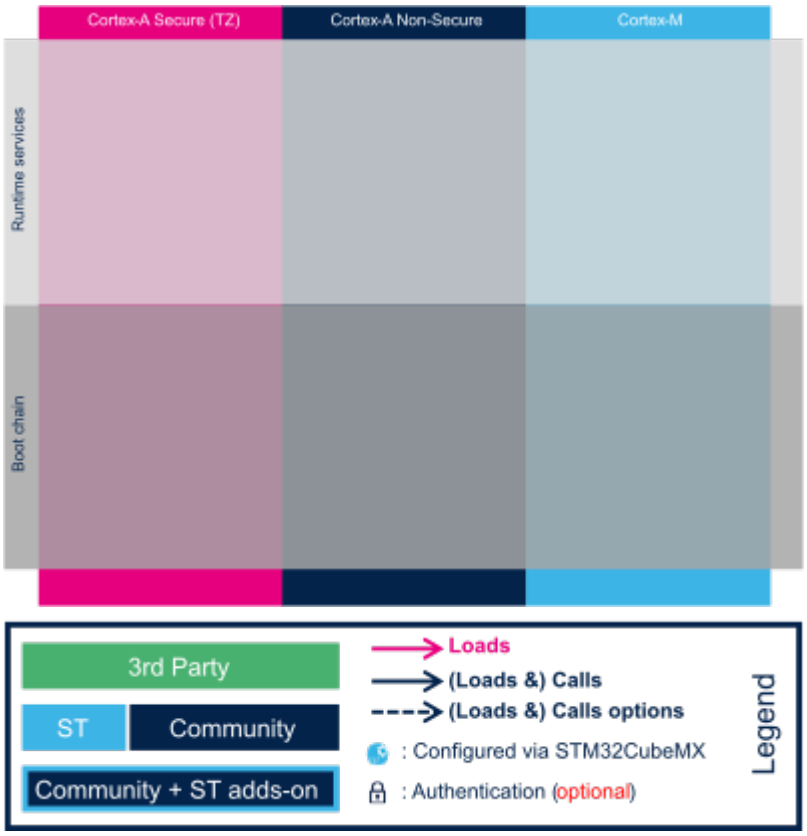
## 2.1 Diagram frames and legend

The hardware execution contexts are shown with vertical frames in the boot diagrams:

- the Arm Cortex-A secure context, in pink
- the Arm Cortex-A non-secure context, in dark blue
- the Arm Cortex-M context, in light blue

The horizontal frame in:

- the bottom part shows the **boot chain**
- the top part shows the **runtime** services, that are installed by the **boot chain**



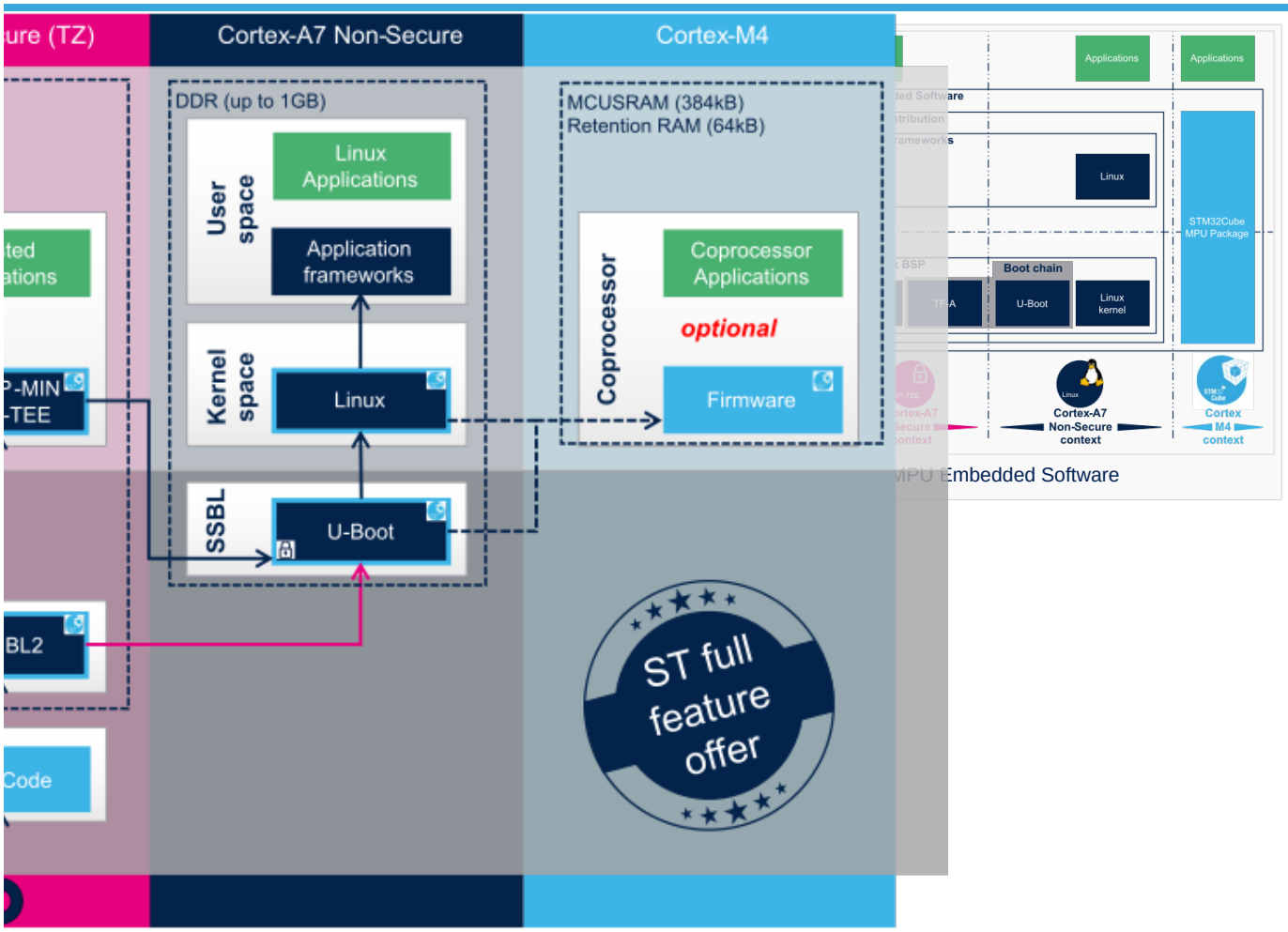

The legend on the right shows how information about the various components shown in the frames, and which are involved in the boot process, is highlighted:

- The box **color** shows the component source code origin
- The **arrows** show the loading and calling actions between the components
- The **Cube** logo is used on the top right corner of components that can be configured via STM32CubeMX
- The **lock** show the components that can be authenticated during the boot process

## 2.2 STM32MP15 boot chain

### 2.2.1 Overview

STM32MP15 boot chain uses Trusted Firmware-A (TF-A) as the FSBL in order to fulfill all the requirements for security-sensitive customers, and it uses U-Boot as the SSBL. Note that the authentication is optional with this boot chain, so it can run on any STM32MP15 device security variant (that is, with or without the Secure boot).
Refer to the security overview for an introduction of the secure features available on STM32MP15, from the secure boot up to trusted applications execution.




Note:

- The STM32MP15 coprocessor can be started at the SSBL level by the U-Boot early boot feature or, later, by the Linux remoteproc framework, depending on the application startup time-targets.

### 2.2.2 ROM code

The ROM code starts the processor in secure mode. It supports the FSBL authentication and offers authentication services to the FSBL.

### 2.2.3 First stage boot loader (FSBL)

The FSBL is executed from the SYSRAM.
Among other things, this boot loader initializes (part of) the clock tree and the DDR controller. Finally, the FSBL loads the second-stage boot loader (SSBL) into the DDR external RAM and jumps to it.
The boot loader stage 2, so called TF-A BL2, is the Trusted Firmware-A (TF-A) binary used as FSBL on STM32MP15.

### 2.2.4 Second stage boot loader (SSBL)

U-Boot is commonly used as a bootloader in embedded software and it is the one used on STM32MP15.

### 2.2.5 Linux

Linux®OS is loaded in DDR by U-Boot and executed in the non-secure context.

### 2.2.6 Secure OS / Secure Monitor

The Cortex-A7 secure world can implement a minimal secure monitor (from TF-A SP-MIN or U-Boot) or a real secure OS, such as OP-TEE.

### 2.2.7 Coprocessor firmware

The coprocessor STM32Cube firmware can be started at the SSBL level by U-Boot with the remoteproc feature (rproc command) or, later, by Linux remoteproc framework, depending on the application startup time-targets.

Linux® is a registered trademark of Linus Torvalds.

Read Only Memory

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Flash memory shortened to gain space in titles, tables and block diagrams

First Stage Boot Loader

Second Stage Boot Loader

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

Secondary Program Loader, *Also known as* ***U-Boot SPL***

User Interface

*Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

Cortex®

TrustZone®

*Arm® and TrustZone® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

Doubledata rate (memory domain)

Trusted Firmware for Arm® Cortex®-A

Boot Loader stage 2

Operating System

Stable: 16.03.2021 - 16:47 / Revision: 16.03.2021 - 15:50

A quality version of this page, approved on *16 March 2021*, was based off this revision.

This article explains how to build the STM32MPU distribution for Android™. It is intended for Distribution Package users.

Contents

# 1 Prerequisites

The environment must be installed using the Distribution Package adapted to your microprocessor device (see the list of Android Distribution Package).

In addition follow the PC prerequisites dedicated to Android to make sure all the packages required to use the environment are present.

# 2 Build

First execute the following command:

```
PC $> source ./build/envsetup.sh
```

It sets several required environment variables for the build and adds several useful scripts in your path. It is mandatory to start with this command as soon as you are using a new terminal.

Use the command below to list some interested aliases created by `envsetup.sh`:

```
PC $> hmm
```

If this is the first time you set up an environment to build Android, continue to read this article. Otherwise directly go to Choosing a build target.

## 2.1 Chip selection

The distribution is built for a default chip version (ex: STM32MP157F for STM32MP1 Series).

It's possible to change the selection, modifying the `SOC_VERSION` value in `device/stm/<STM32Series>/<BoardId>/aosp_<BoardId>.mk` file.

## 2.2 Memory selection

The distribution is built for a default memory type (ex: microSD card 8GiB for STM32MP1 Series).

The memory size and type can be changed. More information refer to How to customize the STM32MPU distribution for Android .

Edit the `device/stm/<STM32Series>/layout/android_layout.config` file.

Locate the two lines below:

```
PART_MEMORY_TYPE
PART_MEMORY_SIZE
```

Current options are :

- PART_MEMORY_TYPE sd and PART_MEMORY_SIZE 8GiB
- PART_MEMORY_TYPE sd and PART_MEMORY_SIZE 4GiB
- PART_MEMORY_TYPE emmc and PART_MEMORY_SIZE 4GiB

When the file has been modified, relaunch the previous command:

```
PC $> source ./build/envsetup.sh
```

## 2.3 Setup

Execute this <STM32Series> setup only once, for example for STM32MP1 Series:

```
PC $> stm32mp1setup
```

It applies specific patches related to your <STM32Series> to customize Android and load the necessary libraries and modules.

## 2.4 Choosing a build target

To choose your target device, execute the command below:

```
PC $> lunch aosp_<BoardId>-<build_type>
```

The available `build_type` values are the following:

- `user`: to generate an end-user production image;
- `userdebug`: similar to an `user` build but with root access and debug capabilities;
- `eng`: development configuration with additional debugging tools.

## 2.5 Generating the image

**In case you selected `user` build, it's required to rebuild the kernel as the prebuilt images are not compatible (see How to build kernel for Android)**

You are now ready to build. To do this, just launch the command:

```
PC $> make -j
```

Depending on your computer settings, several hours might be required to execute this command on the first build.

The result can be found in the `out` folder. The generated partition images are located in `out/target/product/<BoardId>`.

To flash images, refer to Flashing the built image.

## 2.6 Tips

If you encountered the following error during the build:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

It's possible to increase the maximum heap space, executing the following command (can be added in your ~/.bashrc file):

```
PC $> export JAVA_TOOL_OPTIONS="-Xmx4g"
```

stm32mp1

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

Stable: 25.09.2020 - 09:24 / Revision: 25.09.2020 - 09:21

A quality version of this page, approved on *25 September 2020*, was based off this revision.

Contents

# 1 Article purpose

The purpose of this document is to provide information on how to measure and improve the boot-time[1] of a typical STM32MP15 Linux system. This article is not an exhaustive list of possible optimizations, and those that are considered insufficiently reliable for industrial use are intentionally omitted.

# 2 Overview

On a typical STM32MP1 Linux system, the boot-chain is performed, respectively, by: the ROM code, TF-A, U-Boot, the Linux kernel, and the user-land[2]. All these components except the ROM code can be modified, and thus configured to start more quickly. For each of them, the procedure is the same: features that are not required at boot-time must be initiated after system boot (or disabled), and features that improve the boot time must be enabled.

# 3 Measuring 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 focused effectively.

## 3.1 Using a serial console

One of the easiest way sto measure the boot-time of a Linux system is to observe traces emitted on a serial console using timing software, such as serialgrab or like the script File:Measure-timing.txt based on microcom and p2f:

```
PC $> microcom -p /dev/ttyACM0 | bash Measure-timing.txt
Waiting for board reset...
NOTICE:  Model: STMicroelectronics STM32MP157C eval daughter on eval mother
NOTICE:  Board: MB1263 Var1 Rev.C-01
…
U-Boot 2018.11-stm32mp-r2 (Nov 14 2018 - 16:10:06 +0000)

CPU: STM32MP157AAA Rev.B
Model: STMicroelectronics STM32MP157C eval daughter on eval mother
…
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
…
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
…
ST OpenSTLinux - Weston - (A Yocto Project Based Distro) 2.6-openstlinux-4.19-thud-mp1-
19-02-20 stm32mp1 ttySTM0

stm32mp1 login:

Timing results:
FSBL:  1.23s
SSBL:  2.34s
Linux: 2.34s
init:  1.23s
total: 7.14s
```

Note: these are illustrative figures since the boot-time depends on too many parameters to provide meaningful figures here.

## 3.2 Using hardware timers

When measuring traces from a serial console is not precise enough, or when traces are not available, it is possible to use hardware timers. Some components of the boot chain provide timing information based on hardware timers; for instance with U-Boot, the following options can be enabled:

```
CONFIG_BOOTSTAGE=y
CONFIG_BOOTSTAGE_REPORT=y
```

This prints on the serial console — just before booting the OS — something similar to the output below (illustrative figures):

```
Starting kernel ...

Timer summary in microseconds (11 records):
       Mark    Elapsed  Stage
          0          0  reset
    123,456    123,456  board_init_f
  2,345,678  2,222,222  board_init_r
  3,456,789  1,111,111  id=64
  3,567,890    111,101  id=65
  3,678,901    111,011  main_loop
  4,567,890    888,989  bootm_start
  4,678,901    111,011  id=15
  4,789,012    110,111  start_kernel

Accumulated time:
                 23,456  dm_r
                567,890  dm_f
Booting Linux on physical CPU 0x0
```

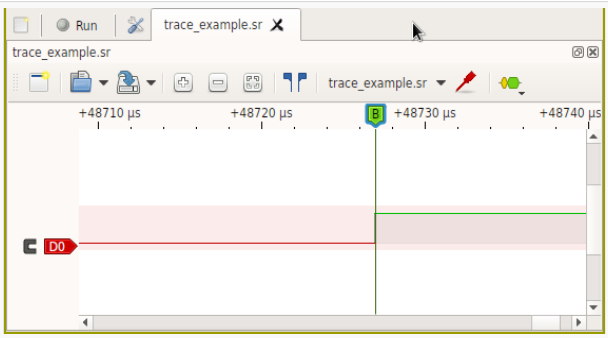
If the serial console is not available, this timing information can also be stored in the device-tree or in memory:

```
CONFIG_BOOTSTAGE_FDT=y
CONFIG_BOOTSTAGE_STASH=y
```

It is possible to add this feature to boot-chain components that do not implement it by default, like TF-A. See the file `arch/arm/cpu/armv7/arch_timer.c` from U-Boot for an example.

## 3.3 Using GPIOs

When the console is disabled, the boot-chain components can be modified to trigger events on GPIOs according to the current stage of the boot-process. Then a logical analyzer can be used to measure the time between each such event.




Example of trace captured by Sigrok and displayed by PulseView.

# 4 Optimizing boot-time

## 4.1 TF-A

### 4.1.1 Configuration

The TF-A execution time can be noticeably reduced by disabling features that are not required. To achieve this, the right options have to be specified when building the TF-A; for instance with a system that boots exclusively from NAND logic connected through the FMC:

```
PC $> make STM32MP1_DEBUG_ENABLE=0    \
          STM32MP1_UART_PROGRAMMER=0 \
          STM32MP1_USB=0             \
          STM32MP1_QSPI_NOR=0        \
          STM32MP1_QSPI_NAND=0       \
          STM32MP_FMC_NAND=1         \
          STM32MP_EMMC=0             \
          STM32MP_SDMMC=0            \
          …
```

These build options must of course be adjusted according to the expected usage and the TF-A version.

### 4.1.2 I2C timing

I2C driver computes device timing at init, in order to guarantee correct data hold and setup times. The execution time of the computation algorithm is not so important, but with all associated device tree accesses, it can last several 10ms. It has to be done for each instance, so it can happen several times in the boot sequence.

For specific needs, a hard-coded timing (and its related frequency) can be used in order to optimize boot performances. Dedicated fields can be added in the I2C instance related handle structure. Then, once retrieved, these boot values can be configured by the client, this avoids to compute timings (and to do several device tree accesses) on first init for each instance.

## 4.2 U-Boot

### 4.2.1 Delay

Before loading the Linux kernel and its device tree, U-Boot, by default, waits one second for a potential user input. This behavior — undesirable when boot time is a concern — can be easily removed by specifying `bootdelay=0` in `include/configs/stm32mp1.h`.

### 4.2.2 Configuration and device tree

More broadly, removing support for all unused devices from U-Boot configuration and unused nodes from the device-tree drastically reduces the U-Boot execution time, since this eliminates the initialization time of those devices and the time to parse the device-tree.

There are also a couple of U-Boot features that are specially designed to improve the boot-time, for example:

```
CONFIG_MTD_UBI_FASTMAP=y
CONFIG_SYS_MALLOC_CLEAR_ON_INIT=n
```

Regarding support for UBI fastmap (for NOR and NAND storage media), please see the "Linux" section below for more information.

### 4.2.3 Configuration access

By default U-Boot searches a boot configuration file `extlinux.conf` or a boot script `boot.scr.uimg` from a bootable partition ("bootfs" for OpenSTLinux), see U-Boot_overview#Generic_Distro_configuration for details. Such access to a file system on storage media can take a short of time, but fortunately it is possible to embed a boot command into the U-Boot binary instead. To do this, the whole U-Boot environment must be specified from scratch:

```
CONFIG_USE_DEFAULT_ENV_FILE=y
CONFIG_DEFAULT_ENV_FILE="path/to/env.txt"
```

Or the U-Boot "distro" mode must be disabled:

```
CONFIG_DISTRO_DEFAULTS=n
```

In the latter case, be aware that if booting from an ext2/4 partition — typically when booting from an SD card or from eMMC — the following options must be explicitly selected (they were previously selected implicitly by CONFIG_DISTRO_DEFAULTS):

```
CONFIG_CMD_EXT2=y
CONFIG_CMD_EXT4=y
```

In both cases only `CONFIG_ENV_IS_NOWHERE` must be set to `y` (remove all the other CONFIG_ENV_IS...) , and the environment variable `bootcmd` must contain the expected boot command, for example:

```
CONFIG_BOOTCOMMAND="run bootcmd_ubi"
```

```
bootcmd_ubi=env set bootargs ubi.mtd=UBI root=ubi0:boot \
                         rootfstype=ubifs rootwait  \
                         rw console=ttySTM0,115200; \
          ubi part UBI;                             \
          ubifsmount ubi0:boot;                     \
          ubifsload 0xc2000000 /zImage;             \
          ubifsload 0xc4000000 /stm32mp157c-ev1.dtb; \
          bootz 0xc2000000 - 0xc4000000
```

## 4.3 Linux

### 4.3.1 Configuration and device-tree

For U-Boot, one of the most efficient ways to decrease the Linux boot time is to remove support for all unused devices from its configuration and device-tree, since this eliminates the time taken to initialize those devices. Also, support for devices and features that are required but not mandatory at boot-time can be compiled as modules, and then loaded by the user-land once the boot-process is done (see the "Init" subsection below).

### 4.3.2 Traces

Linux boot traces have a size of about 2 Kbytes, so they take about 2 seconds to transfer on a serial link configured at 128 Kbit /s. If this is an issue, the following kernel parameters (`bootargs` variable in U-Boot) can be used to remove these traces:

```
quiet loglevel=0
```

This is automatically done by U-Boot when silent mode is required (see CONFIG_SILENT_CONSOLE and CONFIG_SILENT_U_BOOT_ONLY): silent=1 in used U-Boot environment.

### 4.3.3 UBI volume

When partitions such as "bootfs" and "rootfs" are stored on a UBI volume, the following actions are highly recommended to greatly reduce the volume attachment time at boot-time, both by U-Boot and by the Linux kernel:

1. Decrease the size of the UBI volume

```
and
```

2. Enable the fastmap feature. For this, the `CONFIG_MTD_UBI_FASTMAP` option must be set to `y` both for Linux and U-Boot, and `ubi.fm_autoconvert=1` has to be added to the kernel boot parameters. Note that the very first boot is used to create the fastmap information, thus this one does not get faster.

A lot of other volume/file-systems exist for Flash media, such as JFFS, LogFS, F2FS, and so on. Depending on the use-case, some may be faster than others.

## 4.4 User-land

### 4.4.1 Init

The very first user-land process launched by the Linux kernel is `init`; it is in charge of launching other processes in the right order. As a consequence, removing all unnecessary services can greatly speed up this part of the boot-process. The way services can be removed highly depends on the system in use (systemd, OpenRC and so on), so please refer to its documentation.

Ultimately, when no services are required, the `init` system can be replaced by the final application itself; either by storing its binary in `/sbin/init`, or by passing the following Linux boot parameter:

```
init=/path/to/application/binary
```

### 4.4.2 Framework

For graphical applications, the choice of display framework can seriously impact the startup time of the application itself. For instance, one could use the Linux framebuffer instead of Weston or Xorg, since the former is set up faster than the latter.

Likewise for video applications; if the boot-time is a concern, direct use of the V4L2 interface instead of higher-level interfaces such as GStreamer is recommended.

# 5 Conclusion

There is no universal recipe to improve the boot-time of a Linux system, since it depends on the contraints of the final use-case. However the universal rule is: **always benchmark**, before and after optimizing.

# 6 Further reading

- https://www.e-consystems.com/Articles/Product-Design/Linux-Boot-Time-Optimization-Techniques.asp
- https://www.toradex.com/blog/embedded-linux-boot-time-optimization
- https://bootlin.com/blog/tag/boot-time
- https://bootlin.com/doc/training/boot-time/boot-time-slides.pdf

# 7 References

- Sometimes referred to as startup-time.
- Sometimes referred to as user-space.

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)

Central processing unit

The Linux Foundation® and Yocto Project® are registered trademarks of the Linux Foundation. Linux® is a registered trademark of Linux Torvalds

First Stage Boot Loader

Second Stage Boot Loader

Operating System

Universal Asynchronous Receiver/Transmitter

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

Memory Technology Device

SD memory card (https://www.sdcard.org)

former spelling for e•MMC ('e' in italic)

Stable: 23.06.2020 - 13:54 / Revision: 02.06.2020 - 15:39

A quality version of this page, approved on *23 June 2020*, was based off this revision.

This article provides the basic information required to start using the **perfetto** [1] Android™ tool.

# 1 Introduction

The following table provides a brief description of the tool, as well as its availability depending on the software packages:

✓: this tool is either present (ready to use or to be activated), or can be integrated and activated on the software package.

✗: this tool is not present and cannot be integrated, or it is present but cannot be activated on the software package.

| Tool | | | STM32MPU Embedded Software distribution | | | STM32MPU Embedded Software distribution for Android™ | | |
|---|---|---|---|---|---|---|---|---|
| Name | Category | Purpose | Starter Package | Developer Package | Distribution Package | Starter Package | Developer Package | Distribution Package |
| perfetto | Tracing tools | **perfetto**[1] is a performance instrumentation and tracing tool for Android™.<br>• Open-source project for platform tracing<br>• Trace processing and analysis<br>• Web-based trace viewer UI | ✗ | ✗ | ✗ | ✓* | ✓* | ✓ |

# 2 Getting started with perfetto

Perfetto instructions can be generated using the recording tool[2].

- Create record settings required
- Copy generated instructions
- Execute the copied instructions on a opened terminal with the device connected through USB, having ADB installed.

At the end of the trace execution, you can get back the trace:

```
adb pull /data/misc/perfetto-traces/trace
```

From this stage you can directly open the trace through the web viewer[3].

# 3 References

- [1.01.1] https://perfetto.dev/
- https://ui.perfetto.dev/#!/record
- https://ui.perfetto.dev

User Interface
Stable: 19.10.2021 - 13:54 / Revision: 19.10.2021 - 13:54

A quality version of this page, approved on *19 October 2021*, was based off this revision.

Contents

# 1 Das U-Boot

Das U-Boot ("the Universal Boot Loader" or U-Boot) is an open-source bootloader that can be used on ST boards to initialize the platform and load the Linux® kernel.

- Official website: https://www.denx.de/wiki/U-Boot
- Official manual: U-Boot project documentation and https://www.denx.de/wiki/DULG/Manual
- Official **source code** is available under git repository at [1]

Read the README file before starting using U-Boot. It covers the following topics:

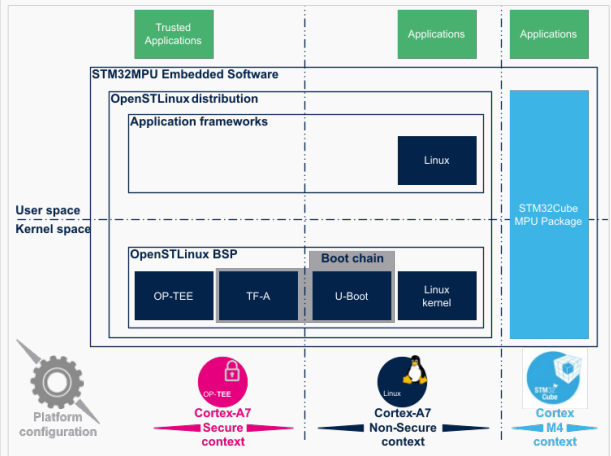
- source file tree structure
- description of CONFIG defines
- instructions for building U-Boot
- brief description of the Hush shell
- list of common environment variables

Do go further, read the documentations available in doc/ and the documentation generated by `make htmldocs` [1].

# 2 U-Boot overview

The STM32 MPU boot chain uses Trusted Firmware-A (TF-A) as FSBL and U-Boot as SSBL.

The same U-Boot source can also generate an alternate FSBL named SPL. The boot chain becomes: SPL as FSBL and U-Boot as SSBL.




Zoom out to STM32MPU Embedded Software

**This alternate boot chain with SPL cannot be used for product development.**

## 2.1 SPL: alternate FSBL

### 2.1.1 SPL description

The **U-Boot SPL** or **SPL** is an alternate first stage bootloader (FSBL).
It is a small binary (bootstrap utility) generated from the U-Boot source and stored in the internal limited-size embedded RAM.
SPL main features are the following:

- It is loaded by the ROM code.
- It performs the initial CPU and board configuration (clocks and DDR memory).
- It loads the SSBL (U-Boot) into the DDR memory.

### 2.1.2 SPL restrictions

**SPL cannot be used for product development.**

SPL is provided only as an example of the simplest FSBL with the objective to support upstream U-Boot development. However, several known limitations have been identified when SPL is used in conjunction with the minimal secure monitor provided within U-Boot for basic boot chain. These limitations apply to:

- power management
- secure access to registers
- limited features (STM32CubeProgrammer / boot from NAND Flash memory)
- SCMI support for clock and reset (not compatible with latest Linux kernel device tree)

There is no workaround for these limitations.

### 2.1.3 SPL execution sequence

**SPL** executes the following main steps in SYSRAM:

- **board_init_f()**: driver initialization including DDR initialization (mininimal stack and heap: CONFIG_SPL_STACK_R_MALLOC_SIMPLE_LEN)
- configuration of heap in DDR memory (CONFIG_SPL_SYS_MALLOC_F_LEN)
- **board_init_r()**: initialization of the other drivers activated in the SPL device tree
- loading and execution of U-Boot (or Kernel in Falcon mode[2]: README.falcon ).

## 2.2 U-Boot: SSBL

### 2.2.1 U-Boot description

**U-Boot** is the second-stage bootloader (SSBL) of boot chain for STM32 MPU platforms.

SSBL main features are the following:

- It is configurable and expendable.
- It features a simple command line interface (CLI), allowing users to interact over a serial port console.
- It provides scripting capabilities.
- It loads the kernel into RAM and gives control to the kernel.
- It manages several internal and external devices such as NAND and NOR Flash memories, Ethernet and USB.
- It supports the following features and commands:
  - File systems: FAT, UBI/UBIFS, JFFS
  - IP stack: FTP
  - Display: LCD, HDMI, BMP for splashcreen
  - USB: host (mass storage) or device (DFU stack)

### 2.2.2 U-Boot execution sequence

**U-Boot** executes the following main steps in DDR memory:

- **Pre-relocation** initialization (common/board_f.c): minimal initialization (such as CPU, clock, reset, DDR and console) running at the CONFIG_SYS_TEXT_BASE load address.
- **Relocation**: copy of the code to the end of DDR memory.
- **Post-relocation initialization**:(common/board_r.c): initialization of all the drivers.
- **Command execution** through autoboot (CONFIG_AUTOBOOT) or console shell.
  - Execution of the boot command (by default bootcmd=CONFIG_BOOTCOMMAND):

  for example, execution of the command `bootm` to:
    - load and check images (such as kernel, device tree and ramdisk)
    - fixup the kernel device tree
    - install the secure monitor (optional) or
    - pass the control to the Linux kernel (or to another target application)

# 3 U-Boot configuration

The U-Boot binary configuration is based on

- **Kbuild infrastructure** (as in Linux Kernel, you can use `make menuconfig` in U-Boot)

The configurations are based on:

  - options defined in Kconfig files (CONFIG_ compilation flags)
  - the selected configuration file: configs/stm32mp*_defconfig

- **other compilation flags** defined in include/configs/stm32mp*.h (these flags are progressively migrated to Kconfig)

The file name is configured through CONFIG_SYS_CONFIG_NAME.

For STM32MP15x lines, the include/configs/stm32mp1.h file is used.

- DeviceTree: U-Boot binaries include a device tree blob that is parsed at runtime

All the configuration flags (prefixed by CONFIG_) are described in the source code, either in the README file or in the documentation directory .

For example, CONFIG_SPL activates the SPL compilation.

Hence to compile U-Boot, select the <target> and the device tree for the board in order to choose a predefined configuration.

Refer to #U-Boot_build for examples.

## 3.1 Kbuild

Like the kernel, the U-Boot build system is based on configuration symbols (defined in Kconfig files). The selected values are stored in a **.config** file located in the build directory, with the same makefile target. .

Proceed as follows:

- Select a predefined configuration (defconfig file in configs directory ) and generate the first **.config**:

```
PC $> make <config>_defconfig.
```

- Change the U-Boot compile configuration (modify .config) by using one of the following five `make` commands:

```
PC $> make menuconfig --> menu based program
PC $> make config  --> line-oriented configuration
PC $> make xconfig --> QT program[3]
PC $> make gconfig --> GTK program
PC $> make nconfig --> ncurse menu based program
```

You can then compile U-Boot with the updated .config.

Warning: the modification is performed locally in the build directory. It will be lost after a `make distclean`.

Save your configuration to be able to use it as a defconfig file:

```
PC $> make savedefconfig
```

This target saves the current config as a defconfig file in the build directory. It can then be compared with the predefined configuration (configs/stm32mp*defconfig).

The other makefile targets are the following:

```
 PC $> make help
....
Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based
                    program
  menuconfig          - Update current config utilising a menu based program
  xconfig          - Update current config utilising a Qt based front-end
  gconfig          - Update current config utilising a GTK+ based front-end
  oldconfig          - Update current config utilising a provided .config as base
  localmodconfig  - Update current config disabling modules not loaded
  localyesconfig  - Update current config converting local mods to core
  defconfig          - New config with default from ARCH supplied defconfig
  savedefconfig   - Save current config as ./defconfig (minimal config)
  allnoconfig           - New config where all options are answered with no
  allyesconfig           - New config where all options are accepted with yes
  allmodconfig           - New config selecting modules when possible
  alldefconfig    - New config with all symbols set to default
  randconfig          - New config with random answer to all options
  listnewconfig   - List new options
  olddefconfig           - Same as oldconfig but sets new symbols to their
                   default value without prompting
```

## 3.2 Device tree

Refer to doc/README.fdt-control for details.

The board device tree has the same binding as the kernel. It is integrated within the U-Boot binaries: u-boot.bin

- By default, it is appended at the end of the code (CONFIG_OF_SEPARATE).
- It can be embedded in the U-Boot binary (CONFIG_OF_EMBED). This is particularly useful for debugging since it enables easy .elf file loading.

The U-Boot device tree (u-boot.dtb) can be also provided as external file loaded by FSBL when U-Boot code is started (u-boot-nodtb.bin: code without device tree): device tree address is provided as boot parameter (in r2 register).

A default device tree is available in the defconfig file (by setting CONFIG_DEFAULT_DEVICE_TREE).

You can either select another supported device tree using the DEVICE_TREE make flag. For stm32mp boards, the corresponding file is <dts-file-name>.dts in arch/arm/dts/stm32mp*.dts , with <dts-file-name> set to the full name of the board:

```
PC $> make DEVICE_TREE=<dts-file-name>
```

or provide a device tree blob (dtb file) resulting from the dts file compilation, by using the EXT_DTB option:

```
PC $> make EXT_DTB=boot/<dts-file-name>.dtb
```

The SPL device tree is also generated from this device tree. However to reduce its size, the U-Boot makefile uses the fdtgrep tool to parse the full U-Boot DTB and identify all the drivers required by SPL.

To do this, U-Boot uses specific device-tree flags to determine if the associated driver is initialized prior to U-Boot relocation and /or if the associated node is present in SPL :

- **u-boot,dm-pre-reloc** => present in SPL, initialized before relocation in U-Boot
- **u-boot,dm-pre-proper** => initialized before relocation in U-Boot
- **u-boot,dm-spl** => present in SPL

In the device tree used by U-Boot, these flags **need to be added in all the nodes** used in SPL or in U-Boot before relocation, and for all used handles (clock, reset, pincontrol).

To obtain a device tree file **<dts-file-name>.dts** that is identical to the Linux kernel one, these U-Boot properties are only added for ST boards in the add-on file **<dts-file-name>-u-boot.dtsi**. This file is automatically included in **<dts-file-name>.dts** during device tree compilation (this is a generic U-Boot Makefile behavior).

# 4 U-Boot command line interface (CLI)

Refer to U-Boot Command Line Interface.

If CONFIG_AUTOBOOT is activated, you have CONFIG_BOOTDELAY seconds (2s by default, 1s for ST configuration) to enter the console by pressing any key, after the line below is displayed and bootcmd is executed (CONFIG_BOOTCOMMAND):

```
Hit any key to stop autoboot:  2
```

## 4.1 Commands

The commands are defined in cmd/*.c . They are activated through the corresponding **CONFIG_CMD_*** configuration flag.

Use the `help` command in the U-Boot shell to list the commands available on your device:

```
Board $> help
```

Below the list of all commands extracted from U-Boot Manual (**not-exhaustive**):

- Information Commands
  - bdinfo - prints Board Info structure
  - coninfo - prints console devices and information
  - flinfo - prints Flash memory information
  - iminfo - prints header information for application image
  - help - prints online help
- Memory Commands
  - base - prints or sets the address offset
  - crc32 - checksum calculation
  - cmp - memory compare
  - cp - memory copy
  - md - memory display
  - mm - memory modify (auto-incrementing)
  - mtest - simple RAM test
  - mw - memory write (fill)
  - nm - memory modify (constant address)
  - loop - infinite loop on address range
- Flash Memory Commands
  - cp - memory copy
  - flinfo - prints Flash memory information
  - erase - erases Flash memory
  - protect - enables or disables Flash memory write protection
  - mtdparts - defines a Linux compatible MTD partition scheme
- Execution Control Commands
  - source - runs a script from memory
  - bootm - boots application image from memory

- go - starts application at address 'addr'

- Download Commands
  - bootp - boots image via network using BOOTP/TFTP protocol
  - dhcp - invokes DHCP client to obtain IP/boot params
  - loadb - loads binary file over serial line (kermit mode)
  - loads - loads S-Record file over serial line
  - rarpboot- boots image via network using RARP/TFTP protocol
  - tftpboot- boots image via network using TFTP protocol
- Environment Variables Commands
  - printenv- prints environment variables
  - saveenv - saves environment variables to persistent storage
  - setenv - sets environment variables
  - run - runs commands in an environment variable
  - bootd - default boot, that is run 'bootcmd'
- Flattened Device Tree support
  - fdt addr - selects the FDT to work on
  - fdt list - prints one level
  - fdt print - recursive printing
  - fdt mknode - creates new nodes
  - fdt set - sets node properties
  - fdt rm - removes nodes or properties
  - fdt move - moves FDT blob to new address
  - fdt chosen - fixup dynamic information
- Special Commands
  - i2c - I2C sub-system
- Storage devices
- Miscellaneous Commands
  - echo - echoes args to console
  - reset - performs a CPU reset
  - sleep - delays the execution for a predefined time
  - version - prints the monitor version

To add a new command, refer to doc/README.commands .

## 4.2 U-Boot environment variables

The U-Boot behavior is configured through environment variables.

Refer to Manual and README / Environment Variables.

On the first boot, U-Boot uses a default environment embedded in the U-Boot binary. You can modify it by changing the content of CONFIG_EXTRA_ENV_SETTINGS in your configuration file (for example ./include/configs/stm32mp1.h) (see README / - Default Environment).

This environment can be modified and saved in the boot device. When it is present, it is loaded during U-Boot initialization:

- To boot from *e*•MMC/SD card (CONFIG_ENV_IS_IN_MMC): at the end of the partition indicated by config field "u-boot,mmc-env-partition" in device-tree (for ST boards: partition named "fip" in ecosystem release v3.0.0  with FIP support or partition named "ssbl" without FIP support).

- To boot from NAND Flash memory (CONFIG_ENV_IS_IN_UBI): in the two UBI volumes "config" (CONFIG_ENV_UBI_VOLUME) and "config_r" (CONFIG_ENV_UBI_VOLUME_REDUND).
- To boot from NOR Flash memory (CONFIG_ENV_IS_IN_SPI_FLASH): the u-boot_env mtd parttion (at offset CONFIG_ENV_OFFSET).

### 4.2.1 env command

The `env` command allows displaying, modifying and saving the environment in U-Boot console.

```
Board $> help env
env - environment handling commands

Usage:
env default [-f] -a - [forcibly] reset default environment
env default [-f] var [...] - [forcibly] reset variable(s) to their default values
env delete [-f] var [...] - [forcibly] delete variable(s)
env edit name - edit environment variable
env exists name - tests for existence of variable
env print [-a | name ...] - print environment
env print -e [name ...] - print UEFI environment
env run var [...] - run commands in an environment variable
env save - save environment
env set -e name [arg ...] - set UEFI variable; unset if 'arg' not specified
env set [-f] name [arg ...]
```

Example: proceed as follows to restore the default environment and save it. This is useful after a U-Boot upgrade:

```
Board $> env default -a
Board $> env save
```

You can also use the command activated by CONFIG_CMD_ERASEENV:

```
Board $> env erase
```

### 4.2.2 bootcmd

"bootcmd" variable is the autoboot command. It defines the command executed when U-Boot starts (CONFIG_BOOTCOMMAND).

For stm32mp, CONFIG_BOOTCOMMAND="run bootcmd_stm32mp":

```
Board $> env print bootcmd
 bootcmd=run bootcmd_stm32mp
```

"bootcmd_stm32mp" is a script that selects the command to be executed for each boot device (see ./include/configs/stm32mp1.h), based on generic distro scripts:

- To boot from a serial/usb device: execute the `stm32prog` command.
- To boot from an *e*•MMC, SD card: boot only on the same device (bootcmd_mmc...).
- To boot from a NAND Flash memory: boot on ubifs partition on the NAND memory (bootcmd_ubi0).
- To boot from a NOR Flash memory: use the SD card (on SDMMC 0 on ST boards with bootcmd_mmc0)

```
Board $> env print bootcmd_stm32mp
```

You can then change this configuration:

- either permanently in your board file
  - default environment by CONFIG_EXTRA_ENV_SETTINGS (see ./include/configs/stm32mp1.h)
  - change CONFIG_BOOTCOMMAND value in your defconfig

```
CONFIG_BOOTCOMMAND="run bootcmd_mmc0"
```

```
CONFIG_BOOTCOMMAND="run distro_bootcmd"
```

- or temporarily in the saved environment:

```
Board $> env set bootcmd run bootcmd_mmc0
Board $> env save
```

Note: To reset the environment to its default value:

```
Board $> env default bootcmd
Board $> env save
```

## 4.3 Generic Distro configuration

Refer to doc/README.distro for details.

This feature is activated by default on ST boards (CONFIG_DISTRO_DEFAULTS):

- one boot command (bootmcd_xxx) exists for each bootable device.
- U-Boot is independent from the Linux distribution used.
- bootcmd is defined in ./include/config_distro_bootcmd.h

When DISTRO is enabled, the command that is executed by default is include/config_distro_bootcmd.h :

```
bootcmd=run distro_bootcmd
```

This script tries any device found in the 'boot_targets' variable and executes the associated bootcmd.

Example for mmc0, mmc1, mmc2, pxe and ubifs devices:

```
bootcmd_mmc0=setenv devnum 0; run mmc_boot
bootcmd_mmc1=setenv devnum 1; run mmc_boot
bootcmd_mmc2=setenv devnum 2; run mmc_boot
bootcmd_pxe=run boot_net_usb_start; dhcp; if pxe get; then pxe boot; fi
bootcmd_ubifs0=setenv devnum 0; run ubifs_boot
```

U-Boot searches for an **extlinux.conf** configuration file for each bootable device. This file defines the kernel configuration to be used with `bootm` command:

- bootargs
- files to start the OS:
  - kernel (uImage) + device tree + ramdisk files (optional)

- FIT image, including all these needed files (for details see doc/uImage.FIT/howto.tx )

## 4.4 U-Boot scripting capabilities

"Script files" are command sequences that are executed by the U-Boot command interpreter. This feature is particularly useful to configure U-Boot to use a real shell (hush) as command interpreter.

See U-Boot script manual for an example.

# 5 U-Boot build

See U-Boot Documentation.

## 5.1 Prerequisites

- a PC with Linux and tools:
  - see PC_prerequisites
  - #ARM cross compiler
- U-Boot source code
  - the latest STMicroelectronics U-Boot version
    - tar.xz file from Developer Package (for example STM32MP1) or from latest release on ST github [4]
    - from GITHUB[5], with `git` command

```
PC $> git clone https://github.com/STMicroelectronics/u-boot
```

  - from the Mainline U-Boot in official GIT repository [6]

```
PC $> git clone https://source.denx.de/u-boot/u-boot.git
```

## 5.2 ARM cross compiler

A cross compiler [7] must be installed on your Host (X86_64, i686, ...) for the ARM targeted Device architecture. In addition, the $PATH and $CROSS_COMPILE environment variables must be configured in your shell.

You can use gcc for ARM, available in:

- the SDK toolchain (see Cross-compile with OpenSTLinux SDK)

PATH and CROSS_COMPILE are automatically updated.

- an existing package

For example, install gcc-arm-linux-gnueabihf on Ubuntu/Debian: (**PC $>** sudo apt-get.

- an existing toolchain:
  - latest gcc toolchain provided by arm (https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads/)
  - gcc v7 toolchain provided by linaro: (https://www.linaro.org/downloads/)

For example, to use *gcc-arm-9.2-2019.12-x86_64-arm-none-linux-gnueabihf.tar.xz* from arm, extract the toolchain in $HOME and update your environment with:

```
PC $> export PATH=$HOME/gcc-arm-9.2-2019.12-x86_64-arm-none-linux-gnueabihf/bin:$PATH
PC $> export CROSS_COMPILE=arm-none-linux-gnueabihf-
```

For example, to use **gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi.tar.xz**
from https://releases.linaro.org/components/toolchain/binaries/7.2-2017.11/arm-linux-gnueabi/
Unzip the toolchain in $HOME and update your environment with:

```
PC $> export PATH=$HOME/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi/bin:$PATH
PC $> export CROSS_COMPILE=arm-linux-gnueabi-
```

## 5.3 Compilation

In the U-Boot source directory, select the defconfig for the <target> and the <device tree> for your board and then execute the `make all` command:

```
PC $> make <target>_defconfig
PC $> make DEVICE_TREE=<device tree> all
```

Use help to list other targets than all:

```
PC $> make help
```

Optionally

- **KBUILD_OUTPUT** can be used to change the output build directory in order to compile several targets in the source directory. For example:

```
PC $> export KBUILD_OUTPUT=<path>
```

- **DEVICE_TREE** can also be exported to your environment when only one board is supported. For example:

```
PC $> export DEVICE_TREE=<device-tree>
```

The result is the following:

```
PC $> export KBUILD_OUTPUT=<path>
PC $> export DEVICE_TREE=<device tree>
PC $> make <target>_defconfig
PC $> make all
```

Examples from STM32MP15 U-Boot:

The boot chain for STM32MP15x lines use stm32mp15_trusted_defconfig:

```
PC $> make stm32mp15_trusted_defconfig
PC $> make DEVICE_TREE=stm32mp157f-dk2 all
```

```
PC $> export KBUILD_OUTPUT=../build/stm32mp15_trusted
PC $> export DEVICE_TREE=stm32mp157c-ev1
PC $> make stm32mp15_trusted_defconfig
PC $> make all
```

## 5.4 Output files

The resulting U-Boot files are located in your build directory (U-Boot or KBUILD_OUTPUT).

Since ecosystem release v3.0.0 , two U-Boot files are used by ST boards to generate FIP used by FSBL TF-A, with or without OP-TEE support:

- BL33_CFG=u-boot.dtb : the U-Boot device tree, selected by DEVICE_TREE, loaded by TF-A BL2 and amended by secure monitor (SPMIN or OP-TEE)
- BL33=u-boot-nodtb.bin: the U-Boot executable, loaded by TF-A BL2 started by secure monitor with BL33_CFG as parameter

Nota: All the compiled device tree are available in $KBUILD_OUTPUT/arch/arm/dts/*.dtb.
You can select them as BL33_CFG without U-Boot recompilation.

See TF-A_overview for FIP details.

The file used to debug with gdb is

- u-boot : elf file for U-Boot

For ecosystem release v2.1.0 : u-boot.stm32 : U-Boot binary with STM32 image header, including device tree selected by DEVICE_TREE, loaded by TF-A

This behavior can be restored if you activate CONFIG_STM32MP15x_STM32IMAGE in your defconfig of ecosystem release v3.0.0

This temporary option is only introduced to facilitate the FIP migration but it will be removed in the next EcosystemRelease.

The STM32 image format (*.stm32) is managed by mkimage U-Boot tools and Signing_tool. It is requested by ROM code and TF-A without FIP support (see STM32 header for binary files for details).

# 6 References


- https://u-boot.readthedocs.io/en/stable/index.html
- https://www.denx.de/wiki/pub/U-Boot/MiniSummitELCE2013/2013-ELCE-U-Boot-Falcon-Boot.pdf
- https://en.wikipedia.org/wiki/Xconfig
- https://github.com/STMicroelectronics/u-boot/releases
- https://github.com/STMicroelectronics/u-boot
- https://source.denx.de/u-boot/u-boot.git or https://github.com/u-boot/u-boot
- https://en.wikipedia.org/wiki/Cross_compiler


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

Linux® is a registered trademark of Linus Torvalds.

First Stage Boot Loader

Secondary Program Loader, *Also known as* ***U-Boot SPL***

Second Stage Boot Loader

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Read Only Memory

Central processing unit

Doubledata rate (memory domain)

Flash memories combine high density and cost effectiveness of EPROMs with the electrical erasability of EEPROMs. For this reason, the Flash memory market is one of the most exciting areas of the semiconductor industry today and new applications requiring in system reprogramming, such as cellular telephones, automotive engine management systems, hard disk drives, PC BIOS software for Plug & Play, digital TV, set top boxes, fax and other modems, PC cards and multimedia CD-ROMs, offer the prospect of very high volume demand.

System control and management interface

Microprocessor Unit

High-Definition Multimedia Interface (HDMI standard)

Device Firmware Upgrade

Device Tree Binary (or Blob)

Memory Technology Device

Trivial File Transfer Protocol (https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol)

Dynamic Host Configuration Protocol (See https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol for more details)

Inter-Integrated Circuit (Bi-directional 2-wire bus standard for efficient inter-IC control.)

MultimediaCard

SD memory card (https://www.sdcard.org)

Firmware Image Package is a packaging format used by TF-A

Serial Peripheral Interface

Operating System

Flattened uImage Tree is a packaging format used by U-Boot

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

Trusted Firmware for Arm® Cortex®-A

Open Portable Trusted Execution Environment

Boot Loader stage 3-3

Boot Loader stage 2