How to update U-Boot

Contents

Stable: 01.02.2021 - 17:27 / Revision: 01.02.2021 - 17:27

Stable: 01.02.2021 - 17:27 / Revision: 01.02.2021 - 17:27

A quality version of this page, approved on *1 February 2021*, was based off this revision.

This page explains how to manually update the U-Boot binaries on an SD card or on the *e*•MMC.

Contents

1 To update an SD card with the Linux dd command

When a Linux console has access to the SD card device partitions:

- on a Linux PC
  - with a card reader of this PC
  - through a USB connection to the target and the `ums` command executed on a U-Boot console
- on target, with Linux console.

The 3 first GPT partitions on the SD card are:

1. FSBL1
2. FSBL2
3. SSBL

See Boot_chain_overview for the bootloader definitions.

You can use the Linux `dd` command to copy the FSBL and SSBL directly to the correct partition:

```
PC $>  dd if=<file> of=/dev/<dev> conv=fdatasync
```

<dev> is:

- mmcblk<X>p<n>: PC-embedded card reader case or target Linux console
- sd<X><n>: USB-connected SD card reader case

where <X> is the ID of the device, and <n> the ID of the partition.

Note: the dd option conv=fdatasync is used to force synchronous copying.

1.1 U-Boot environment

The U-Boot environment is saved at the end of the U-Boot partition, named "ssbl": ID = 3.

To clear this environment, erase the U-Boot partition before any update; for example, by writing 0 to this partition:

```
PC $>  dd if=/dev/zero of=/dev/mmcblk<X>p3 conv=fdatasync
```

```
PC $>  dd if=/dev/zero of=sd<X>3 conv=fdatasync
```

1.2 SD card update example

The internal card reader is /dev/mmcblk0 or for a target Linux console, GPT partition <n> is /dev/mmcblk0p<n>:

```
PC $> dd if=tf-a.stm32 of=/dev/mmcblk0p1 conv=fdatasync
PC $> dd if=tf-a.stm32 of=/dev/mmcblk0p2 conv=fdatasync
PC $> dd if=/dev/zero of=/dev/mmcblk0p3 conv=fdatasync
PC $> dd if=u-boot.stm32 of=/dev/mmcblk0p3 conv=fdatasync
```

Alternatively, with U-Boot console, dev = 0 (SD card device on ST Microelectronics board), GPT partition <n> is /dev/sda<n>:

```
Board $> mmc dev 0
Board $> ums 0 mmc 0

PC $> dd if=tf-a.stm32 of=/dev/sda1 conv=fdatasync
PC $> dd if=tf-a.stm32 of=/dev/sda2 conv=fdatasync
PC $> dd if=/dev/zero of=/dev/sda3 conv=fdatasync
PC $> dd if=u-boot.stm32 of=/dev/sda3 conv=fdatasync
```

1.3 SD card update example with SPL as FSBL

The USB card reader is /dev/sdb, GPT partition <n> is /dev/sdb<n>:

```
PC $> dd if=u-boot-spl.stm32 of=/dev/sdb1 conv=fdatasync
PC $> dd if=u-boot-spl.stm32 of=/dev/sdb2 conv=fdatasync
PC $> dd if=/dev/zero of=/dev/sdb3 conv=fdatasync
PC $> dd if=u-boot.img of=/dev/sdb3 conv=fdatasync
```

2 Update of *e*•MMC with the Linux dd command

The same command, `dd`, can be used to update *e*•MMC memory mapping:

- SSBL U-Boot is the first GPT partition in the *e*•MMC user area
- FSBL = TF-A (or SPL) is saved at the beginning of the *e*•MMC boot partition

The user needs to select the *e*•MMC hardware partition to update: user data, boot1, or boot2.

2.1 On a Linux console

If dev = **mmcblk1** for *e*•MMC device (default on ST Microelectronics board)

The boot partitions are available in /dev/**mmcblk1**boot0 and /dev/**mmcblk1**boot1 [1].

The user perhaps needs to allow access, for example with:

```
Board $>  echo 0 > /sys/class/block/mmcblk1boot0/force_ro
```

The mmc tools allow the boot partition to be selected [2].

The STM32MP15x ROM code requires:

- <send_ack> =1
- the eMMC boot configuration is: 1 wire configuration and 25 MHz, it is done with the command:

```
Board $>  mmc bootbus set  single_backward x1 x1 dev/mmcblk1
```

To update TF-A in boot1 and select this boot partition:

```
Board $> dd if=tf-a.stm32 of=/dev/mmcblk1boot0 conv=fdatasync
Board $> mmc bootpart enable 1 1 /dev/mmcblk1
```

To update TF-A in boot2 and select this boot partition:

```
Board $> dd if=tf-a.stm32 of=/dev/mmcblk1boot1 conv=fdatasync
Board $> mmc bootpart enable 2 1 /dev/mmcblk1
```

To update U-Boot in the first GPT partition:

```
Board $> dd if=/dev/zero of=/dev/mmcblk1p1 conv=fdatasync
Board $> dd if=u-boot.stm32 of=/dev/mmcblk1p1 conv=fdatasync
```

See also [3].

2.2 On a U-Boot console

The *e*•MMC update is done with the `ums` command and the targeted *e*•MMC HW partition is selected in U-Boot by the third parameter partition_access of command **`mmc partconf`**:

- 0: user data partition
- 1: boot partition 1
- 2: boot partition 2

```
Board $>  help mmc
...
mmc bootbus dev boot_bus_width reset_boot_bus_width boot_mode
- Set the BOOT_BUS_WIDTH field of the specified device
mmc bootpart-resize <dev> <boot part size MB> <RPMB part size MB>
- Change sizes of boot and RPMB partitions of specified device
mmc partconf dev [boot_ack boot_partition partition_access]
- Show or change the bits of the PARTITION_CONFIG field of the specified device
```

For example:

```
* dev = 1 (e•MMC device on ST Microelectronics board)
* boot_ack=1 (Boot Acknowledge is needed by ROM code)
* boot_partition = 1 (Boot partition 1 enabled for boot)
* partition_access =  0 (No access to boot partition - default)
* or partition_access =  1 (R/W boot partition 1)
```

And on a PC, the mass storage is mounted as /dev/sda

To update FSBL=TF-A, select the boot1 HW partition and come back to the user area at the end.

```
Board $> mmc dev 1
Board $> mmc partconf 1 1 1 1
Board $> ums 0 mmc 1

PC $> dd if=tf-a.stm32 of=/dev/sda  conv=fdatasync

Board $> mmc partconf 1 1 1 0
```

To update SSBL = U-Boot in the first GPT partition in the user partition

```
Board $> mmc dev 1
Board $> ums 0 mmc 1

PC $> dd if=u-boot.stm32 of=/dev/sda1 conv=fdatasync
```

Before the first boot, select the *e*•MMC correct boot configuration (1 wire, 25 MHz) with the command:

```
Board $> mmc bootbus 1 0 0 0
```

3 References

Please refer to the following links for additional information:

- https://www.kernel.org/doc/Documentation/mmc/mmc-dev-parts.txt
- https://manpages.debian.org/buster/mmc-utils/mmc.1.en.html
- https://www.emcraft.com/som/stm32mp1/booting-linux-from-emmc

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

MultimediaCard

Linux® is a registered trademark of Linus Torvalds.

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

Second Stage Boot Loader

First Stage Boot Loader

GUID Partition Table

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

Trusted Firmware for Arm Cortex-A

Read Only

Read Only Memory

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

Stable: 10.11.2020 - 07:59 / Revision: 10.11.2020 - 07:58

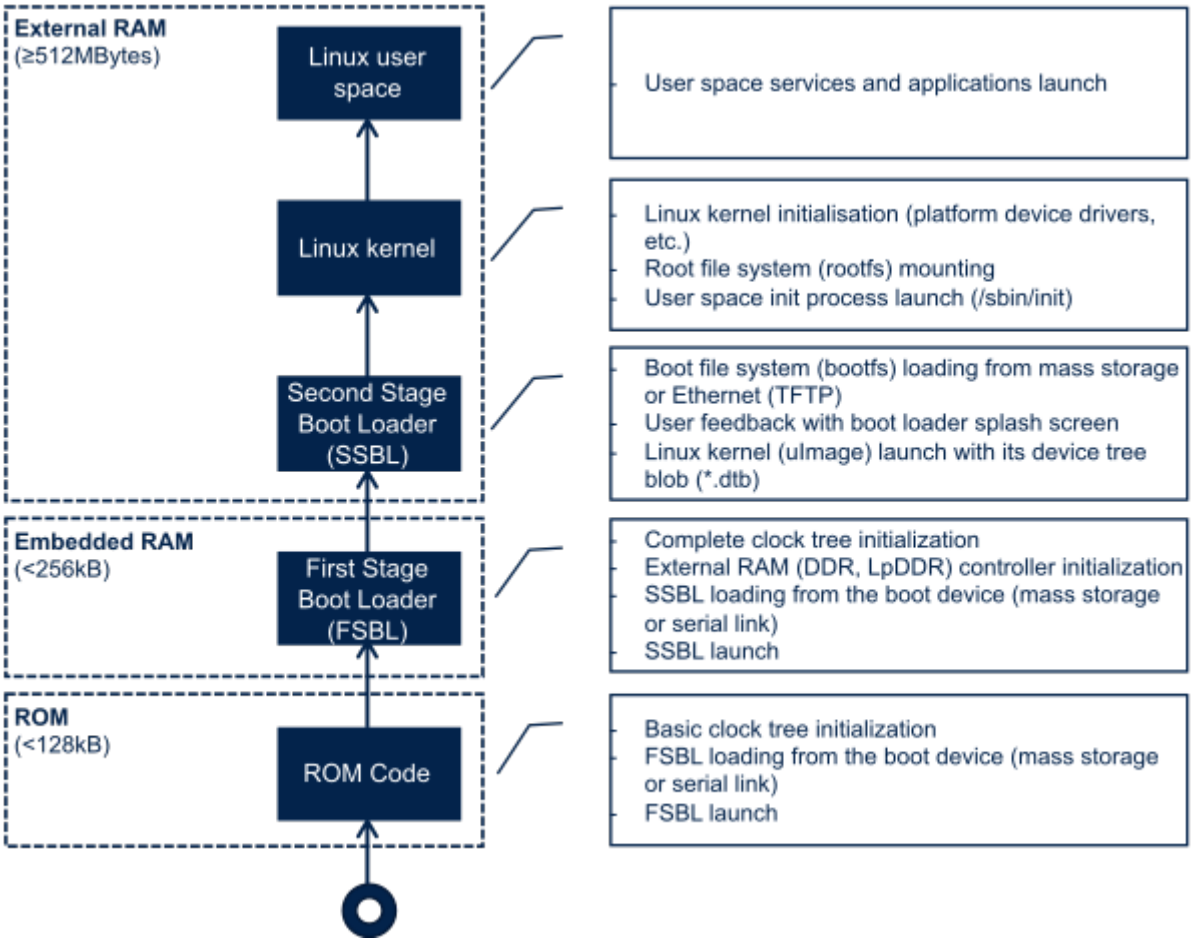
A quality version of this page, approved on *10 November 2020*, was based off this revision.

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

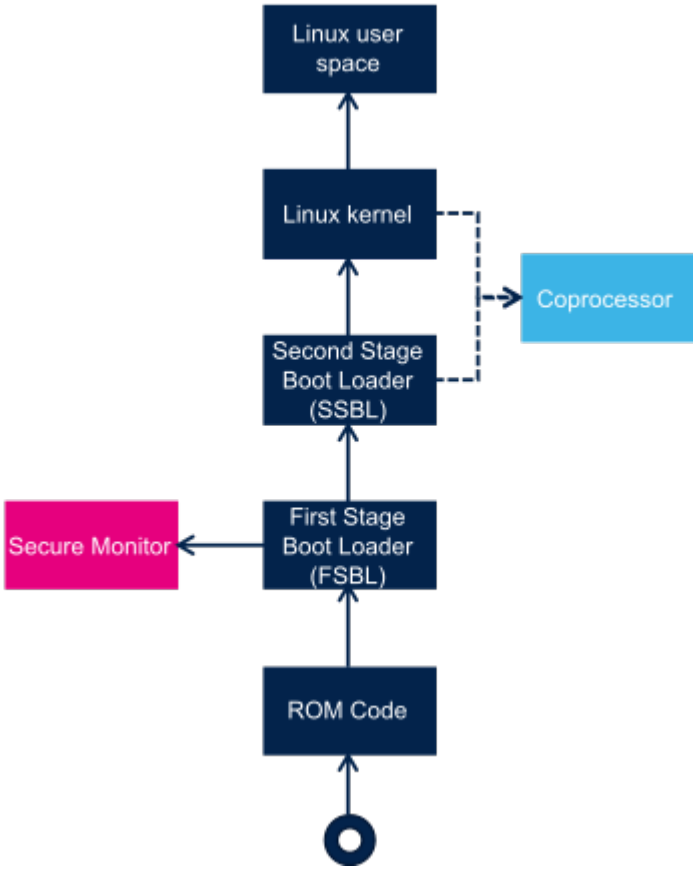
In addition to Linux startup, the boot chain also installs the secure monitor and may support coprocessor firmware loading.

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

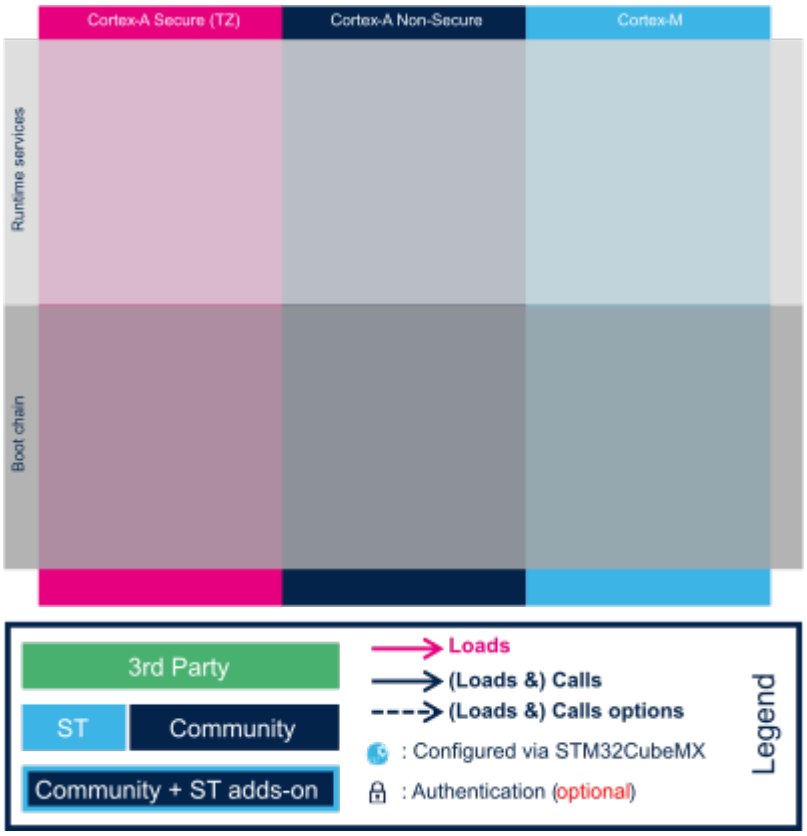
2.1 Diagram frames and legend

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**



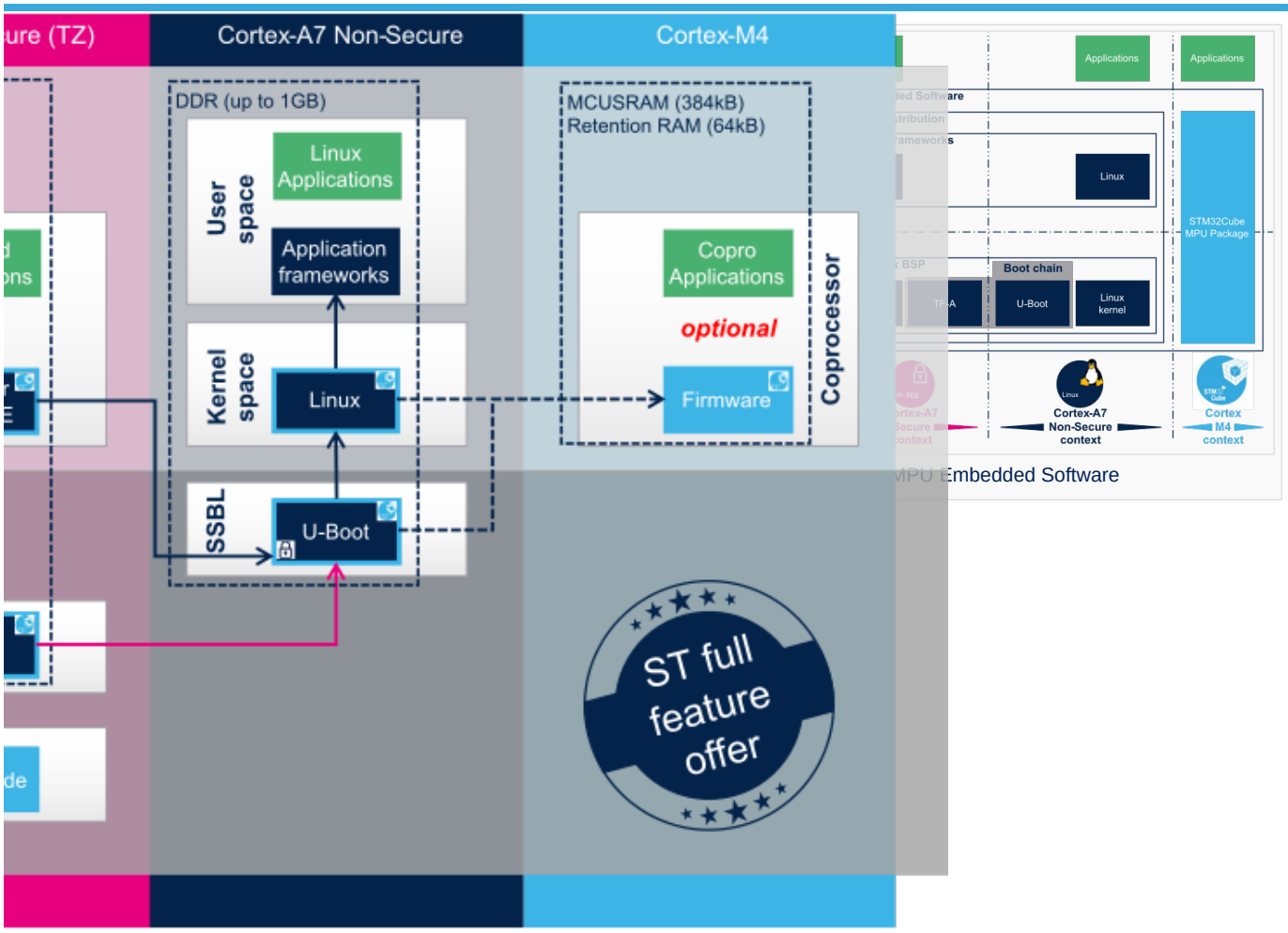
The legend on the right shows how information about the various components shown in the frames, and which are involved in the boot process, is highlighted:

- 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.
Trusted Firmware-A (TF-A) is the FSBL used on the 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 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)

Operating System

Stable: 24.09.2020 - 07:45 / Revision: 24.09.2020 - 07:40

A quality version of this page, approved on *24 September 2020*, was based off this revision.

This page explains how to use the U-Boot command "ums" to update an SD card or eMMC on the device.

1 ums command

In U-Boot, you can directly export the available block devices (sd/mmc/usb) as USB mass storage devices with ums command:

```
Board $> help ums
ums - Use the UMS [USB Mass Storage]

Usage:
ums <USB_controller> [<devtype>] <dev[:part]>  e.g. ums 0 mmc 0
    devtype defaults to mmc
```

This U-Boot command "ums" is infinite (a loop in USB treatments), and the U-Boot console is blocked until user enters a Ctrl-C.

2 Exporting a block device

On ST boards, the OTG USB controller device index is 0, SD card = "mmc 0" and, when available, eMMC = "mmc 1".
You can check the device connected on an SDMMC with the U-Boot command "mmc info".

You can also export a USB device connected to the USB host controller (USBH) = "usb 0".

Then execute one of the following commands:

```
                                Comments
Board $> ums 0 mmc 0 --> start ums on SD card
Ctrl-C
```

```
                                Comments
Board $> ums 0 mmc 1 --> start ums on eMMC
Ctrl-C
```

```
                                Comments
Board $> usb start   --> start USB host controller
Board $> ums 0 usb 0 --> start ums on USB device 0 (USB key for example)
Ctrl-C
Board $> usb stop    --> stop USB host controller
```

After a delay (of up to 15 seconds), the host sees the exported block device and you can use any command on the PC to access the partitions of the exported memory (dd, mount, cp, rsync). A Ctrl-C is needed to stop the command.

See also How to update U-Boot.

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

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

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

User-space Mode Setting

Stable: 20.11.2020 - 17:03 / Revision: 20.11.2020 - 17:02

A quality version of this page, approved on *20 November 2020*, was based off this revision.

Contents

1 Supported Flash memory technologies

STM32MP15 boards support the following types of Flash memory:

- SD card on the SDMMC interface present on EVAL and DISCO boards
- *e*•MMC on the SDMMC interface present on EVAL board only
- Serial NOR Flash memory on the Dual QSPI interface present on EVAL board only
- NAND Flash memory on the FMC interface present on EVAL board only.

The next section lists all partitions used on STM32MP15 boards (size, name, and content), and the following sections show how they are mapped on the different types of Flash memory.

2 Flash partitions

The tables below list the partitions defined for STMP32MP15 boards.

2.1 Minimal

Size	Component	Comment
Remaining area	userfs	The user file system contains user data and examples
768 Mbytes	rootfs	Linux root file system contains all user space binaries (executable, libraries, and so on), and kernel modules
16 Mbytes	vendorfs	This partition is preferred to the rootfs for third-party proprietary binaries, and ensures that they are not contaminated by any open source licence, such as GPL v3
64 Mbytes	bootfs	The boot file system contains: • (option) the init RAM file system, which can be copied to the external RAM and used by Linux before mounting a fatter rootfs • Linux kernel device tree (can be in a Flattened Image Tree - FIT) • Linux kernel U-Boot image (can be in a Flattened Image Tree - FIT) • For all flashes except for NOR: the boot loader splash screen image, displayed by U-Boot • U-Boot distro config file *extlinux.conf* (can be in a Flattened Image Tree – FIT)
2 Mbytes	ssbl	The second stage boot loader (SSBL) is U-Boot, with its device tree blob (dtb) appended at the end
256 Kbytes to 512 Kbytes (*)	fsbl	The first stage boot loader is Arm Trusted Firmware (TF-A) or U-Boot Secondary Program Loader (SPL), with its device tree blob (dtb) appended at the end. At least two copies are embedded. Note: due to ROM code RAM needs, the FSBL payload is limited to 247 Kbytes.

(*): The partition size depends on the Flash technology, to be aligned to the block erase size of the Flash memory present on the board: NOR (256 Kbytes) / NAND (512 Kbytes).

Information

Some boards can be equipped with multiple Flash devices, like the EVAL board, where all of the Flash devices can be programmed with STM32CubeProgrammer. However, caution must be taken for the serial NOR/NAND and SLC NAND because a **static bootable MTD partitioning** is defined in U-Boot include/configs/stm32mp1.h (look for *STM32MP_MTDPARTS*), with the consequence that up to 6 Mbytes of space is lost at the beginning of each such device, **even those which are not bootable**.

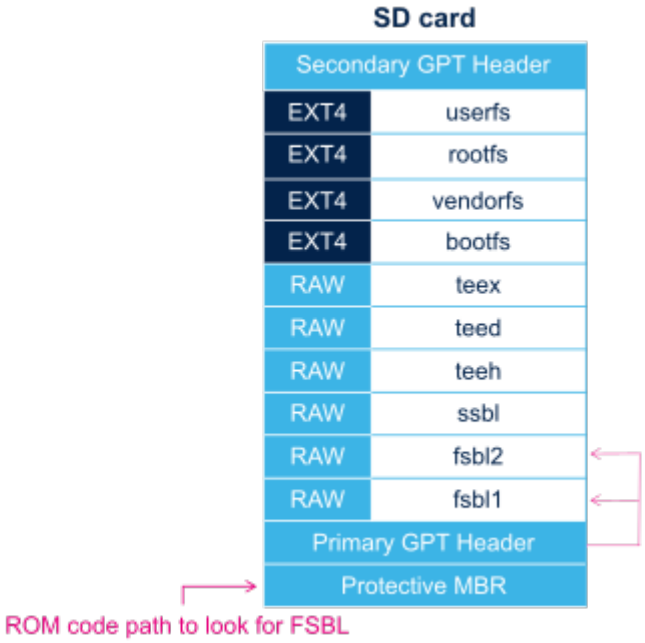
2.2 Optional

Size	Component	Comment
2 * 256 Kbytes (*)	env	This partition is used to store the U-Boot environment while booting from NOR Flash. The information is stored twice, for redundancy. For all other Flash devices, the U-Boot environment is stored either in an EXT4 bootfs partition (*e*•MMC, SD card), or UBI volumes (NAND).
256 Kbytes to 512 Kbytes (*)	teeh	OP-TEE header
256 Kbytes to 512 Kbytes (*)	teed	OP-TEE pageable code and data
256 Kbytes to 512 Kbytes (*)	tex	OP-TEE pager

(*): The partition size depends on the Flash technology, as it should be aligned to the block erase size of the Flash device present on the board (256 Kbytes for NOR and 512 Kbytes for NAND).

3 SD card memory mapping

The SD card has to be partitioned with GPT format in order to be recognized by the STM32MP15. The easiest way to achieve this is to use STM32CubeProgrammer.
The ROM code looks for the GPT entries whose name begins with "fsbl": fsbl1 and fsbl2 for example.
Note: The SD card can be unplugged from the board and inserted into a Linux host computer for direct partitioning with Linux utilities and access to the **bootfs**, **rootfs** and **userfs** partitions. The file system is Linux EXT4.



4 e•MMC memory mapping

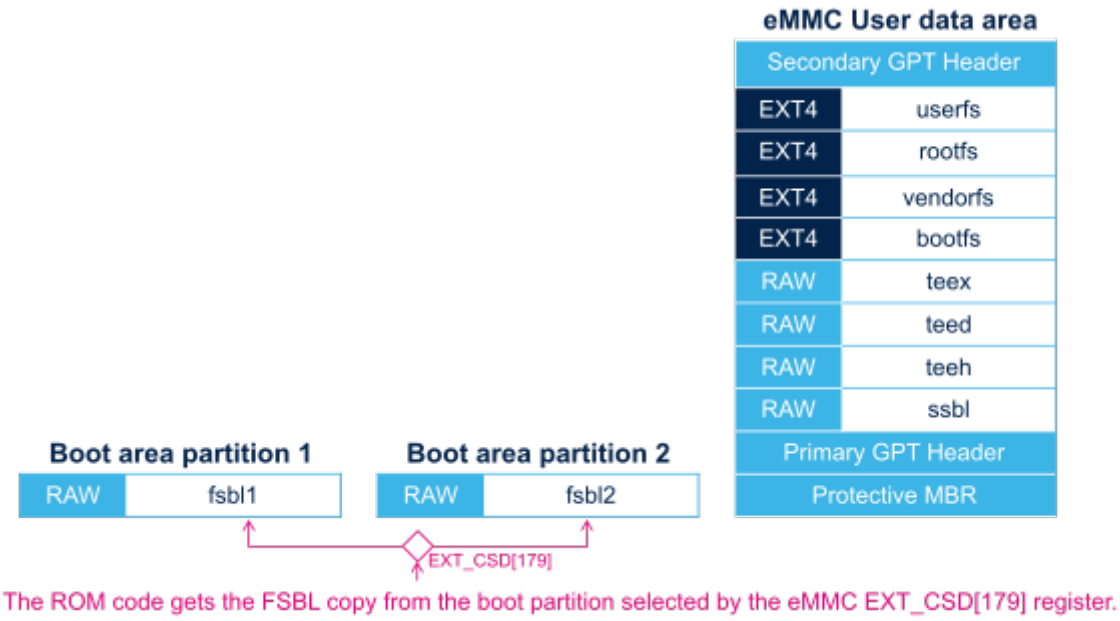
The e•MMC embeds four physical partitions:

- Boot area partition 1: one copy of the FSBL
- Boot area partition 2: one copy of the FSBL
- User data area: formatted with GPT partitioning and used to store all remaining partitions
- Replay Protected Memory Block (RPMB): not shown in the figure below, since not involved in the current boot chain.

STM32CubeProgrammer has to be used to prepare the e•MMC with the layout shown below, and to populate each partition.

Information

The boot area partition used by the e•MMC boot sequence is selected via the EXT_CSD[179] register in the e•MMC. The STM32CubeProgrammer execution is concluded with the selection of the last written partition from the flashlayout file, typically partition 2. The other copy is never used as long as the user does not explicitly change the e•MMC EXT_CSD[179] register to select it.



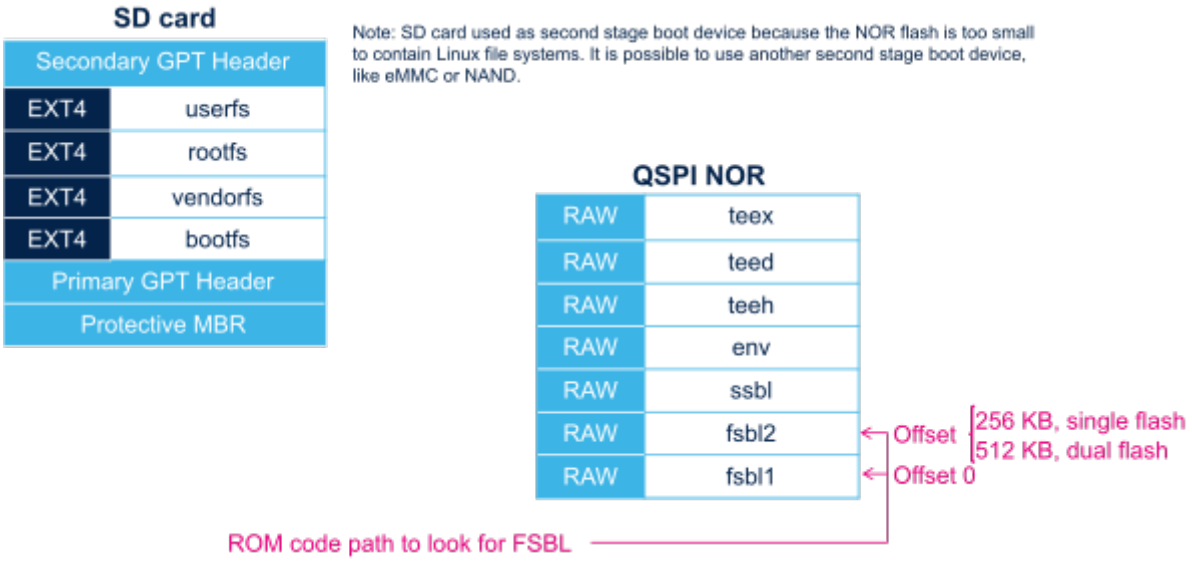
The ROM code gets the FSBL copy from the boot partition selected by the eMMC EXT_CSD[179] register.

5 NOR memory mapping

As NOR Flash memory is expensive, its size is usually limited to the minimum needed to store only the bootloaders. The system files (bootfs, rootfs and userfs) are usually stored in another Flash memory, such as the SD card in OpenSTLinux distribution.
STM32CubeProgrammer must be used to prepare the NOR Flash and the SD card with the layout shown below, and to populate each partition.

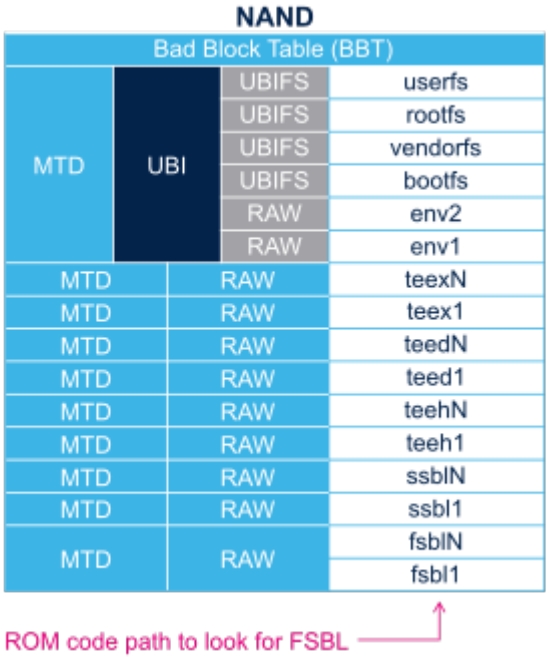
It is possible to use an *e*•MMC card or NAND as second-level Flash memory, rather than an SD card. This requires the following aspects to be changed:

- The Flash memory layout, using STM32CubeProgrammer in order to write the rootfs and userfs to the targeted Flash memory
- The Linux kernel parameters, using U-Boot, in order to indicate where the rootfs and userfs have to be mounted.



6 NAND memory mapping

STM32CubeProgrammer has to be used to prepare the NAND Flash memory with the layout shown below, and to populate each partition.



Notes:
- the MTD partition contains one UBI partition with multiple volumes (UBIFS and RAW)
- U-Boot env is stored with redundancy (env1, env2), on two different volumes
- in the MTD/RAW area, a skip bad block policy is applied so the number of copies and the margins have to defined in STM32CubeProgrammer flash layout, depending on the product expectedlife time and firmware update strategy

Warning

In the **RAW/MTD area**, the number of copies to embed and the number of blocks to reserve at the end of each partition, for future bad blocks replacement, are a critical part of NAND based product dimensioning !

The strategy has to take into account many parameters such as:
- the binaries sizes,
- the read accesses to those partitions during product life, that may generate read disturb effect,
- the capability of the product to refresh the partitions content when erroneous bits are detected,
- the number of software updates estimated on this partition,
- the selected NAND flash characteristics

ST set foundations in the STM32MP15 device in order to allow the integration of NAND flash memories but the product defintion remains the customer responsibility. Please, contact your memory provider for further advice.

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.

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

MultimediaCard

Linux® is a registered trademark of Linus Torvalds.

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.)

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

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

Second Stage Boot Loader

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

Trusted Firmware for Arm Cortex-A

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

Read Only Memory

First Stage Boot Loader

Single-Level Cell is a kind of NAND flash

Memory Technology Device

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

Open Portable Trusted Execution Environment

GUID Partition Table

Stable: 01.03.2021 - 10:54 / Revision: 01.03.2021 - 10:53

A quality version of this page, approved on *1 March 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:

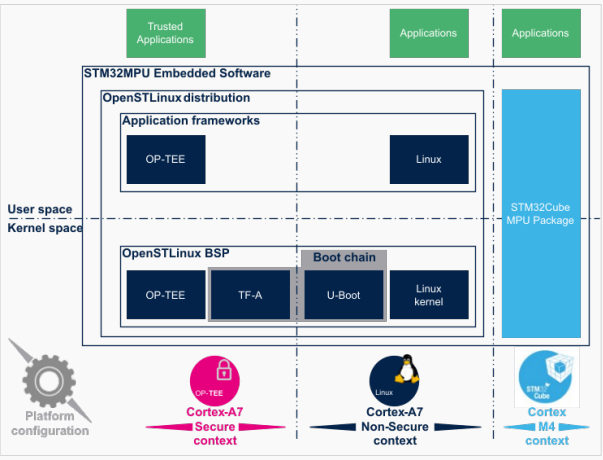
- source file tree structure
- description of CONFIG defines
- instructions for building U-Boot
- brief description of the Hush shell
- list of common environment variables

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

Warning

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

Warning

SPL cannot be used for product development.

SPL is provided only as an example of the simplest SSBL 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:

- 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.

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 (partition named "ssbl" for ST boards).
- 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
```

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:

- bootargs
- kernel + device tree + ramdisk files (optional)
- FIT image

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

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).

The U-Boot generated files when TF-A is used as FSBL, with or without OP-TEE:

- **u-boot.stm32** : U-Boot binary with STM32 image header, loaded by TF-A

The STM32 image format (*.stm32) is managed by mkimage U-Boot tools and Signing_tool. It is requested by ROM code and TF-A (see STM32 header for binary files for details).

The files used to debug with gdb are

- u-boot : elf file for U-Boot

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)

Serial Peripheral Interface

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