



## How to use USB mass storage in U-Boot



---

## Contents

---

1. How to use USB mass storage in U-Boot .....	3
2. How to update U-Boot .....	5
3. OTG internal peripheral .....	10
4. SDMMC internal peripheral .....	15
5. U-Boot overview .....	21
6. USBH internal peripheral .....	37



---

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:

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

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

```
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 eMMC ('e' in italic)

Das U-Boot -- the Universal Boot Loader (see [U-Boot\\_overview](#))

User-space Mode Setting

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 eMMC.

### Contents

1 To update an SD card with the Linux dd command .....	6
1.1 U-Boot environment .....	6
1.2 SD card update example .....	6
1.3 SD card update example with SPL as FSBL .....	7
2 Update of eMMC with the Linux dd command .....	8
2.1 On a Linux console .....	8
2.2 On a U-Boot console .....	9
3 References .....	10



## 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 eMMC with the Linux dd command

The same command, dd, can be used to update eMMC memory mapping:

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

The user needs to select the eMMC hardware partition to update: user data, boot1, or boot2.

### 2.1 On a Linux console

If dev = **mmcbk1** for eMMC device (default on ST Microelectronics board)

The boot partitions are available in `/dev/mmcbk1boot0` and `/dev/mmcbk1boot1` <sup>[1]</sup>.

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

```
Board $> echo 0 > /sys/class/block/mmcbk1boot0/force_ro
```

The mmc tools allow the boot partition to be selected <sup>[2]</sup>.

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/mmcbk1
```

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

```
Board $> dd if=tf-a.stm32 of=/dev/mmcbk1boot0 conv=fdatasync
Board $> mmc bootpart enable 1 1 /dev/mmcbk1
```

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

```
Board $> dd if=tf-a.stm32 of=/dev/mmcbk1boot1 conv=fdatasync
Board $> mmc bootpart enable 2 1 /dev/mmcbk1
```

To update U-Boot in the first GPT partition:

```
Board $> dd if=/dev/zero of=/dev/mmcbk1p1 conv=fdatasync
Board $> dd if=u-boot.stm32 of=/dev/mmcbk1p1 conv=fdatasync
```

See also <sup>[3]</sup>.





## 2.2 On a U-Boot console

The eMMC update is done with the `ums` command and the targeted eMMC 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 (eMMC 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 eMMC 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 eMMC. ('e' in *italic*)

Stable: 16.02.2021 - 16:08 / Revision: 16.02.2021 - 15:51

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

### Contents

1 Article purpose .....	11
2 Peripheral overview .....	12
2.1 Features .....	12
2.2 Security support .....	12
3 Peripheral usage and associated software .....	13
3.1 Boot time .....	13
3.2 Runtime .....	13
3.2.1 Overview .....	13
3.2.2 Software frameworks .....	13
3.2.3 Peripheral configuration .....	13
3.2.4 Peripheral assignment .....	13
4 References .....	15



## 1 Article purpose

---

The purpose of this article is to

- briefly introduce the OTG peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how it can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when needed, how to configure the OTG peripheral.



---

## 2 Peripheral overview

---

The **OTG** peripheral is used to interconnect other systems with STM32 MPU devices, using USB standard.

### 2.1 Features

The **OTG** peripheral is a USB Dual-Role Device (DRD) controller that supports both device and host functions.

In Host mode, it supports high-speed (480 Mbit/s), full-speed (12 Mbit/s) and low-speed (1.5 Mbit/s).

In Peripheral mode, high-speed and full-speed are supported, not low-speed.

The **OTG** peripheral embeds a full-speed PHY and supports a UTMI interface connected to internal HS PHY.

The **OTG** peripheral is fully compliant with

- *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification*<sup>[1]</sup>, Revision 2.0, May 8, 2009
- *Universal Serial Bus Revision 2.0 Specification*<sup>[2]</sup>, Revision 2.0, April 27, 2000
- *USB 2.0 Link Power Management Addendum Engineering Change Notice to the USB 2.0 specification*<sup>[3]</sup>, July 16, 2007
- *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*<sup>[4]</sup>, Version 1.05, March 29, 2001
- *UTMI+ Specification*<sup>[5]</sup>, Revision 1.0, February 25, 2004

Refer to [STM32MP15 reference manuals](#) for the complete hardware feature list, and to the software components (introduced below) to know which features are supported.

### 2.2 Security support

The **OTG** peripheral is a **non-secure** peripheral.



## 3 Peripheral usage and associated software

### 3.1 Boot time

The OTG peripheral is used by ROM code, FSBL and SSBL in device mode (DFU) to support serial boot for flash programming with STM32CubeProgrammer.

The SSBL can use it in host mode (mass storage), for instance to boot on a kernel stored on a USB key, or after a kernel panic to perform the crash dump saving to the USB key.

### 3.2 Runtime

#### 3.2.1 Overview

The OTG peripheral can be allocated to the Arm®Cortex®-A7 non-secure core to be used under Linux® with USB framework.

#### 3.2.2 Software frameworks

Domain	Peripheral	Software frameworks		Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)		
High speed interface	OTG (USB OTG)		Linux USB framework	

#### 3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration by itself can be performed via the STM32CubeMX tool for all internal peripherals. It can then be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

For **Linux** kernel configuration, please refer to [OTG device tree configuration](#).

For **U-boot** configuration, please refer to [Configure USB OTG node in U-Boot](#).

#### 3.2.4 Peripheral assignment

**Check boxes** illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

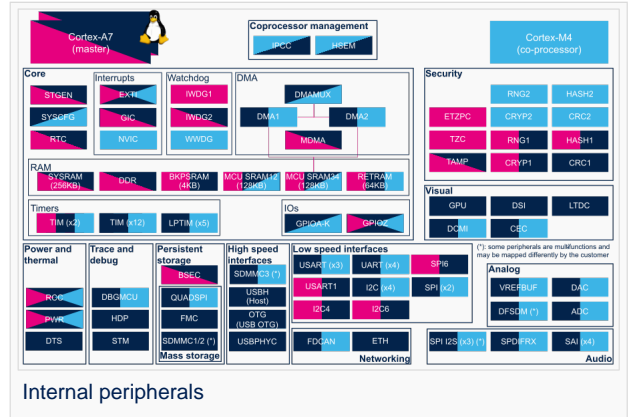
- means that the peripheral can be assigned ( ) to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to [How to assign an internal peripheral to a runtime context](#) for more information on how to assign peripherals manually or via STM32CubeMX.

The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possibilities might be described in STM32MP15 reference manuals.



## How to use USB mass storage in U-Boot



Domain	Periphera	Runtime allocation		Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)	
High speed interface	OTG (USB OTG)	OTG (USB OTG)		



## 4 References

- On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification
- Universal Serial Bus Revision 2.0 Specification
- ECN USB 2.0 Link Power Management Addendum
- USB 2.0 Transceiver Macrocell Interface (UTMI) Specification
- UTMI+ Specification

USB On-The-Go (Capability/type of USB port, acting primarily as USB device, to also act as USB host. Also known as USB OTG.)

Microprocessor Unit

Dual-Role Device (USB standard defines host and device roles. OTG controllers support both roles and can be called Dual-Role Devices controllers.)

USB 2.0 Transceiver Macrocell Interface

Device Firmware Upgrade

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

Cortex<sup>®</sup>

Linux<sup>®</sup> is a registered trademark of Linus Torvalds.

Open Portable Trusted Execution Environment

Stable: 14.05.2020 - 07:13 / Revision: 14.05.2020 - 07:12

A quality version of this page, approved on 14 May 2020, was based off this revision.

### Contents

1 Article purpose .....	16
2 Peripheral overview .....	17
2.1 Features .....	17
2.2 Security support .....	17
3 Peripheral usage and associated software .....	18
3.1 Boot time .....	18
3.2 Runtime .....	18
3.2.1 Overview .....	18
3.2.2 Software frameworks .....	18
3.2.3 Peripheral configuration .....	19
3.2.4 Peripheral assignment .....	19
4 How to go further .....	20
5 References .....	21



## 1 Article purpose

---

The purpose of this article is to

- briefly introduce the SDMMC peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the SDMMC peripheral.





---

## 2 Peripheral overview

---

The **SDMMC** peripheral is used to interconnect STM32 MPU to SD memory cards, SDIO and MMC devices.

### 2.1 Features

Refer to the [STM32MP15 reference manuals](#) for the complete list of features, and to the software components, introduced below, to see which features are implemented.

### 2.2 Security support

SDMMC1/2/3 instances are either **non-secure** or **secure** peripherals (under ETZPC control).

#### Warning

- When an SDMMC instance is secure internal, the DMA cannot be used to perform data transfers.
- STMicroelectronics does not provide secure MMC driver (see below chapter)



## 3 Peripheral usage and associated software

### 3.1 Boot time

SDMMC1/2 instances can be used to support memory boot on SD or MMC Flash devices.

The SDMMC3 is not used at boot time.

#### **i** Information

The SDMMC instances are ordered by address in the [device tree](#) `arch/arm/boot/dts/stm32mp151.dtsi` file:

```
sdmmc3: sdmmc@48004000 {
    ...
sdmmc1: sdmmc@58005000 {
    ...
sdmmc2: sdmmc@58007000 {
```

By default, in [OpenSTLinux](#) distribution, **sdmmc3 is disabled** so the `sdmmc1` (SD card on [Evaluation boards](#) and [Discovery kits](#)) and `sdmmc2` (eMMC on [Evaluation boards](#) and [Wifi on Discovery kits](#)) are respectively aliased to `mmc0` and `mmc1`.

**If you enable `sdmmc3`**, it will take the `mmc0` alias and the aliases above will shift, so don't forget to update the Linux kernel boot command accordingly!

For instance, `'root=/dev/mmcblk0p6'` will become `'root=/dev/mmcblk1p6'` to mount the rootfs from the `sdmmc1` (SD card) when `sdmmc3` is enabled.

### 3.2 Runtime

#### 3.2.1 Overview

SDMMC1/2/3 instances can be allocated to:

- the Arm®Cortex®-A7 non-secure core to be controlled in Linux® by the MMC framework

or

- the Arm®Cortex®-M4 to be controlled in STM32Cube MPU Package by STM32Cube SDMMC driver

Chapter [#Peripheral assignment](#) describes which peripheral instance can be assigned to which context.

#### 3.2.2 Software frameworks

Domain	Peripheral	Software frameworks		Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4  (STM32Cube)		
Mass storage	SDMMC		Linux MMC framework	STM32Cube SDMMC driver



### 3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals), according to the information given in the corresponding software framework article.

For Linux<sup>®</sup> kernel configuration, please refer to SDMMC device tree configuration.

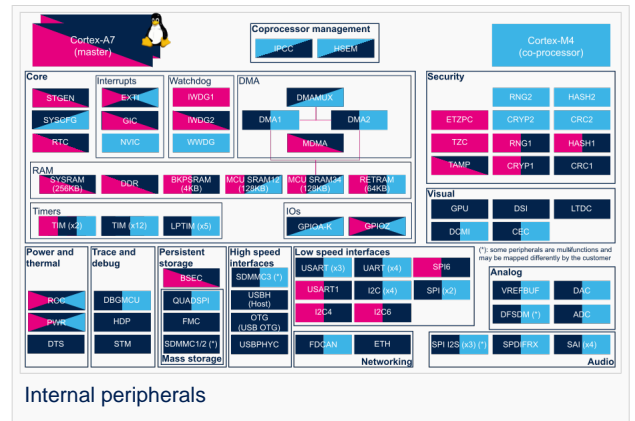
### 3.2.4 Peripheral assignment

**Check boxes** illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

- means that the peripheral can be assigned ( ) to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to How to assign an internal peripheral to a runtime context for more information on how to assign peripherals manually or via STM32CubeMX.

The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possibilities might be described in STM32MP15 reference manuals



Domain	Periphera	Runtime allocation		Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)	
Mass storage	SDMMC	SDMMC1		
		SDMMC2		
		SDMMC3		Assignment (single choice)



## 4 How to go further

---



## 5 References

Microprocessor Unit

MultimediaCard

Direct Memory Access

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

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

Linux<sup>®</sup> is a registered trademark of Linus Torvalds.

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



Cortex<sup>®</sup>

Open Portable Trusted Execution Environment

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 .....	23
2 U-Boot overview .....	24
2.1 SPL: alternate FSBL .....	24
2.1.1 SPL description .....	24
2.1.2 SPL restrictions .....	24
2.1.3 SPL execution sequence .....	25
2.2 U-Boot: SSBL .....	25
2.2.1 U-Boot description .....	25
2.2.2 U-Boot execution sequence .....	25
3 U-Boot configuration .....	26
3.1 Kbuild .....	26
3.2 Device tree .....	27
4 U-Boot command line interface (CLI) .....	29
4.1 Commands .....	29
4.2 U-Boot environment variables .....	30
4.2.1 env command .....	31
4.2.2 bootcmd .....	31
4.3 Generic Distro configuration .....	32
4.4 U-Boot scripting capabilities .....	32
5 U-Boot build .....	33
5.1 Prerequisites .....	33
5.2 ARM cross compiler .....	33



---

5.3 Compilation .....	34
5.4 Output files .....	34
6 References .....	36



---

## 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<sup>®</sup> 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

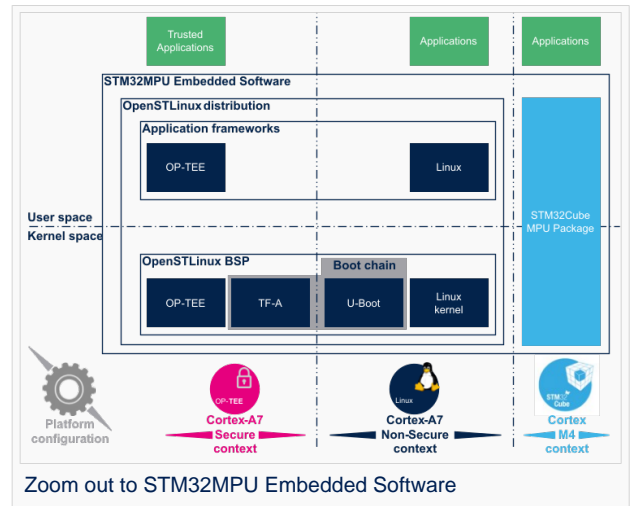
Do go further, read the documentations available in `doc/` and the documentation generated by `make htmldocs` [1].



## 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.



### 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 (minimal 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<sup>[2]</sup>: 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 splashscreen
  - 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 --> ncurses 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
  - `imininfo` - 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 mknod - 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 eMMC/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 partition (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 eMMC, 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 (bootcmd\_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 <sup>[4]</sup>
    - from GITHUB<sup>[5]</sup>, with git command

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

- from the Mainline U-Boot in official GIT repository <sup>[6]</sup>

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

### 5.2 ARM cross compiler

A cross compiler <sup>[7]</sup> 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-gnueabi 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-gnueabi.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-gnueabi/bin:$PATH
PC $> export CROSS_COMPILE=arm-none-linux-gnueabi-
```

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 `make 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](https://en.wikipedia.org/wiki/Cross_compiler)

Das U-Boot -- the Universal Boot Loader (see [U-Boot\\_overview](#))

Linux<sup>®</sup> 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 had serial 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-access semiconductor 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](https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol))

Dynamic Host Configuration Protocol (See [https://en.wikipedia.org/wiki/Dynamic\\_Host\\_Configuration\\_Protocol](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 ulmage 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

Stable: 25.09.2020 - 09:43 / Revision: 25.09.2020 - 09:37

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

## Contents

1 Article purpose .....	38
2 Peripheral overview .....	39
2.1 Features .....	39
2.2 Security support .....	39
3 Peripheral usage and associated software .....	40
3.1 Boot time .....	40
3.2 Runtime .....	40
3.2.1 Overview .....	40
3.2.2 Software frameworks .....	40
3.2.3 Peripheral configuration .....	40
3.2.4 Peripheral assignment .....	40
4 References .....	42



## 1 Article purpose

---

The purpose of this article is to

- briefly introduce the USBH peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how it can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when needed, how to configure the USBH peripheral.



---

## 2 Peripheral overview

---

The **USBH** peripheral is used to interconnect other systems with STM32 MPU devices, using USB standard.

### 2.1 Features

The **USBH** peripheral is a USB Host controller supporting high-speed (480 Mbit/s) using an **EHCI** controller, and full- and low-speeds (12 and 1.5 Mbit/s) through **OHCI** controller.

The **USBH** peripheral has two physical ports providing a **UTMI+** physical interface, mapped on an on-chip 2-port **high-speed UTMI+ PHY**.

It supports the standard registers used for low- and full-speed operation (**OHCI** model) and high-speed operation (**EHCI** model) and the power management feature called Link Power Management (LPM).

The supported standards are:

- *Universal Serial Bus Revision 2.0 Specification*<sup>[1]</sup>, Revision 2.0, April 27, 2000
- *USB 2.0 Link Power Management Addendum Engineering Change Notice to the USB 2.0 specification*<sup>[2]</sup>, July 16, 2007
- *Enhanced Host Controller Interface Specification for Universal Serial Bus*<sup>[3]</sup>, Revision 1.0, March 12, 2002
- *EHCI v1.1 Addendum*<sup>[4]</sup>, August 2008
- *Open Host Controller Interface Specification for USB*<sup>[5]</sup>, Release 1.0a, September 14, 1999
- *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*<sup>[6]</sup>, Version 1.05, March 29, 2001
- *UTMI+ Specification*<sup>[7]</sup>, Revision 1.0, February 25, 2004

Refer to *STM32MP15 reference manuals* for the complete features list, and to the software components, introduced below, to know which features are really implemented.

### 2.2 Security support

The **USBH** peripheral is a **non-secure** peripheral.



### 3 Peripheral usage and associated software

#### 3.1 Boot time

The **USBH** peripheral is usually not used at boot time. But it may be used by the SSBL (see Boot chain overview), for example to boot a kernel stored on a USB key (mass storage).

#### 3.2 Runtime

##### 3.2.1 Overview

The **USBH** peripheral can be allocated to the Arm®Cortex®-A7 non-secure core to be used under Linux® with USB framework.

##### 3.2.2 Software frameworks

Domain	Peripheral	Software frameworks	Comment
Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)	
High speed interface	USBH (USB Host)	Linux USB framework	

##### 3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration by itself can be performed via the **STM32CubeMX** tool for all internal peripherals. It can then be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

For Linux kernel configuration, please refer to **USBH device tree configuration**.

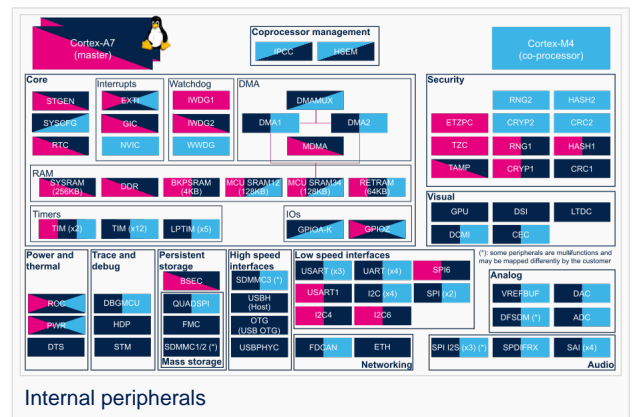
##### 3.2.4 Peripheral assignment

**Check boxes** illustrate the possible peripheral allocations supported by **STM32 MPU Embedded Software**:

- means that the peripheral can be assigned ( ) to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.

Refer to **How to assign an internal peripheral to a runtime context** for more information on how to assign peripherals manually or via **STM32CubeMX**.

The present chapter describes **STMicroelectronics** recommendations or choice of implementation. Additional possibilities might be described in **STM32MP15** reference manuals .







Domain	Periphera	Runtime allocation			Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4  (STM32Cube)		
High speed interface	USBH (USB Host)	USBH (USB Host)			



---

## 4 References

---

- Universal Serial Bus Revision 2.0 Specification
- ECN USB 2.0 Link Power Management Addendum
- Enhanced Host Controller Interface Specification for Universal Serial Bus
- Enhanced Host Controller Interface Specification: Addendum
- Open Host Controller Interface Specification for USB
- USB 2.0 Transceiver Macrocell Interface (UTMI) Specification
- UTMI+ Specification

USB Host (STM32 specific)

Microprocessor Unit

Enhanced Host Controller Interface

Open Host Controller Interface

USB 2.0 Transceiver Macrocell Interface

Second Stage Boot Loader

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

Cortex<sup>®</sup>

Linux<sup>®</sup> is a registered trademark of Linus Torvalds.

Open Portable Trusted Execution Environment