Application note

# USB DFU/USART protocols used in STM32MP1 Series bootloaders

## Introduction

This application note describes the protocols used by the bootloader programming tools for the STM32MP1 Series microprocessors. It details each USB DFU or USART command supported by the embedded software, and the sequences expected by the STM32CubeProgrammer tool.

# 1 Embedded programming service

## 1.1 Introduction

This document applies to the STM32MP1 Series Arm®-based microprocessors.

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

It defines the protocols used by the STM32MP1 Series bootloaders to provide the programming service, that is, the embedded part needed by the STM32CubeProgrammer.

This service provides a way to program the nonvolatile memories (NVM) namely, external flash memory device or on chip nonvolatile memory (OTP).

This document describes the USB or USART protocols used by both the STM32MP1 embedded software and the STM32CubeProgrammer as well as the expected sequences.

## 1.2 Reference

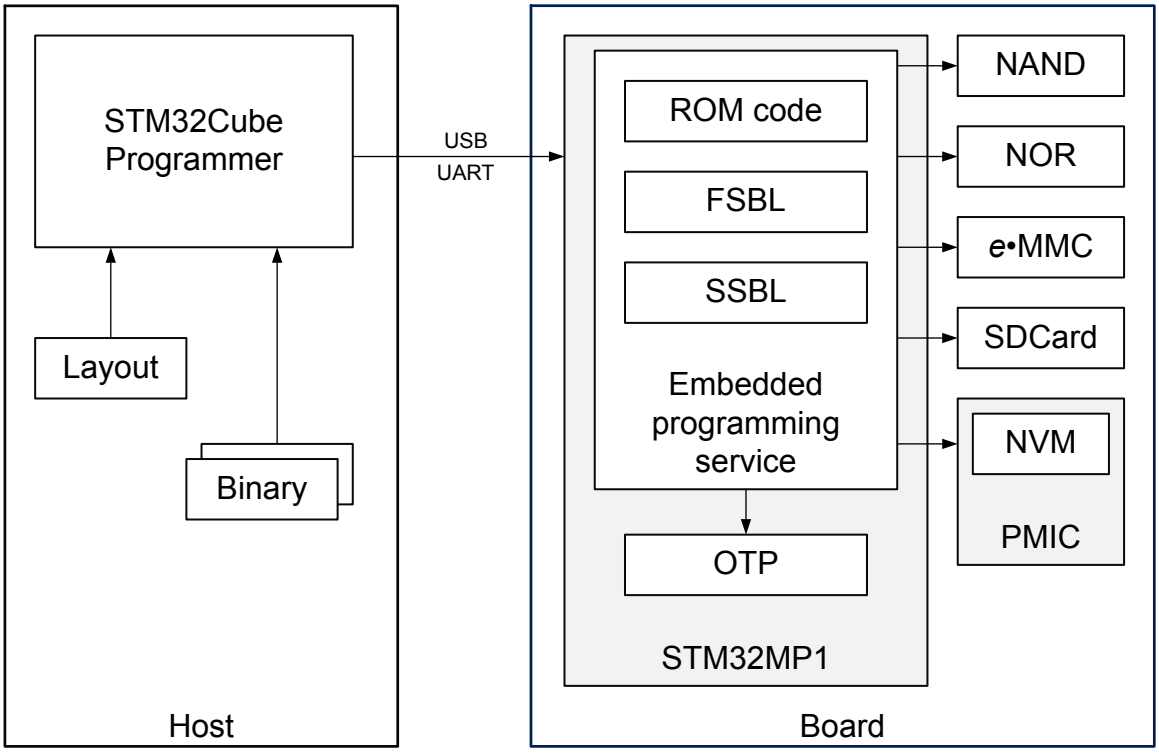
| | |
|---|---|
| [1] | AN3155, Application note, USART protocol used in the STM32 bootloader |
| [2] | AN3156, Application note, USB DFU protocol used in the STM32 bootloader |
| [3] | UM0412, User manual, Getting started with DfuSe USB device firmware upgrade STMicroelectronics extension |
| [4] | UM0424, User manual, STM32 USB-FS-Device development kit |
| [5] | Universal Serial Bus, Device Class Specification for Device Firmware Upgrade, version 1.1 from https://usb.org/(1) |
| [6] | The GUID Partition Table (GPT) is a standard for the layout of partition tables, part of the Unified Extensible Firmware Interface (UEFI) standard (https://uefi.org/)(1) |
| [7] | Signing tool in wiki.st.com/stm32mpu/wiki/Signing_tool |
| [8] | ROM code overview in wiki.st.com/stm32mpu/wiki/Category:ROM_code |
| [9] | FlashLayout in wiki.st.com/stm32mpu/wiki/STM32CubeProgrammer_flashlayout |
| [10] | OTP in wiki.st.com/stm32mpu/wiki/STM32CubeProgrammer_OTP_management |
| [11] | STPMIC1 NVM in wiki.st.com/stm32mpu/wiki/STM32CubeProgrammer_STPMIC1_NVM_management |
| [12] | STM32 header for binary files in wiki.st.com/stm32mpu/wiki/STM32_header_for_binary_files |
| [13] | Chapter Serial boot in ROM code overview in wiki.st.com/stm32mpu/wiki/Category:ROM_code |
| [14] | Firmware image package used by TF-A in wiki.st.com/stm32mpu/wiki/TF-A_overview#FIP |

1. *This URL belongs to a third party. It is active at document publication. However, STMicroelectronics shall not be liable for any change, move, or inactivation of the URL or the referenced material.*

## 1.3 Overview

The programming operation consists in writing binaries initially stored on a host computer, via an interface, to any nonvolatile memory (NVM) on the platform. This process involves the STM32CubeProgrammer tool that communicates with an embedded programming service. This consists of the ROM code, the first stage bootloader (FSBL), and the second stage bootloader (SSBL).

**Figure 1. Programming operation**




A communication protocol is defined for each serial interface (USART, USB) involving a set of commands and some sequences that are as much compatible as possible with existing STM32 MCU devices (see [1] and [2]).

The possible NVMs are:

- An external flash memory device:
  - NAND flash memory
  - *e*•MMC
  - SD card
  - NOR flash memory
- An on-chip nonvolatile memory:
  - STM32MP1 OTP
  - NVM of a PMIC (STPMIC1 for example)

A layout file gives the list of binaries to program their type such as, binary or file system, the targeted NVM, and the position in the NVM.

The eventual signing steps of the binary files with the STM32 header are done previously with the SigningTool (see [7]).

The embedded programming service is based on the ROM code, the FSBL = Arm Trusted Firmware (TF-A) and SSBL = U-Boot. The same protocol is used for both downloading the FSBL and SSBL in RAM and for loading the partition to be programmed in the device NVM.

The ROM code is embedded in the STM32MP device. Its main task is to load, verify, and execute the first stage bootloader (FSBL) in internal RAM through one of the available serial peripherals. In turn, after some initializations (clock and DDR), the FSBL loads the second loader (SSBL) in DDR, verifies the signature and executes it.

## 1.4 Layout file format

The flash memory layout-file is a tab-separated-value (tsv) text file with one line per partition or binary to send to the device. The complete description can be found in ST wiki [9].

## 1.5 Phase ID

A complete description for the phase ID can be found in the ST wiki [9]. The Ids 0x0 and 0xF1 to 0xFD are reserved. For the other values, it is a unique identifier present in the layout file for each download phase-request made to the STM32CubeProgrammer. This request is done either by the ROM code, by the FSBL=TF-A or by the SSBL = U-Boot.

It is used by the embedded programming service to identify the next partition. This is, for example the TF-A and U-Boot to be downloaded in the Get phase command answer.

**Table 1. Phase ID**

| Code | Partition |
|---|---|
| 0x00 | Layout file |
| 0x01 to 0x0F | Boot partitions with FIP header [14] or with STM32 header [12]: SSBL, FSBL, other (trusted execution environment -TEE, Cortex-M4 firmware) |
| 0x01 (reserved) | Image containing FSBL: used by ROM code |
| 0x03 (reserved) | FIP image (such as, embedding SSBL, OP-TEE) used by FSBL=TF-A |
| 0x10 to 0xF0 | User partitions programmed without header (uimage, dtb, rootfs, userfs etc.) |
| 0xF1 to 0xFD | Reserved for internal purpose |
| 0xF1 | Command GetPhase |
| 0xF2 | OTP |
| 0xF3 | Reserved |
| 0xF4 | PMIC NVM (optional) |
| 0xFE | End of operation |
| 0xFF | Reset |

The IDs 0x01 and 0x03 are reserved for image containing FSBL=TF-A or SSBL=U-Boot / FIP. The FSBL is loaded in the RAM by the ROM code, the SSBL is loaded in the RAM by the FSBL.

The Ids 0x0 and 0xF1 to 0xFD are reserved for internal purpose.

A complete description of the special operation for OTP (0xF2) and for PMIC NVM (0xF4) can be found in the ST wikis [10] and [11].

## 1.6 STM32 image header

The details of the STM32 image header can be found in the STM32 MPU wiki [12].

This header is used for the FSBL partition = TF-A loaded by ROM code:

- Version 1 for STM32MP15x devices (size 0x100 bytes = 256, signed or not)
- Version 2 for STM32MP13x devices (size of 0x200 bytes = 512, signed or not)

*Note: This header can be also used by TF-A on STM32MP15x series to load SSBL but the FIP header is recommended.*

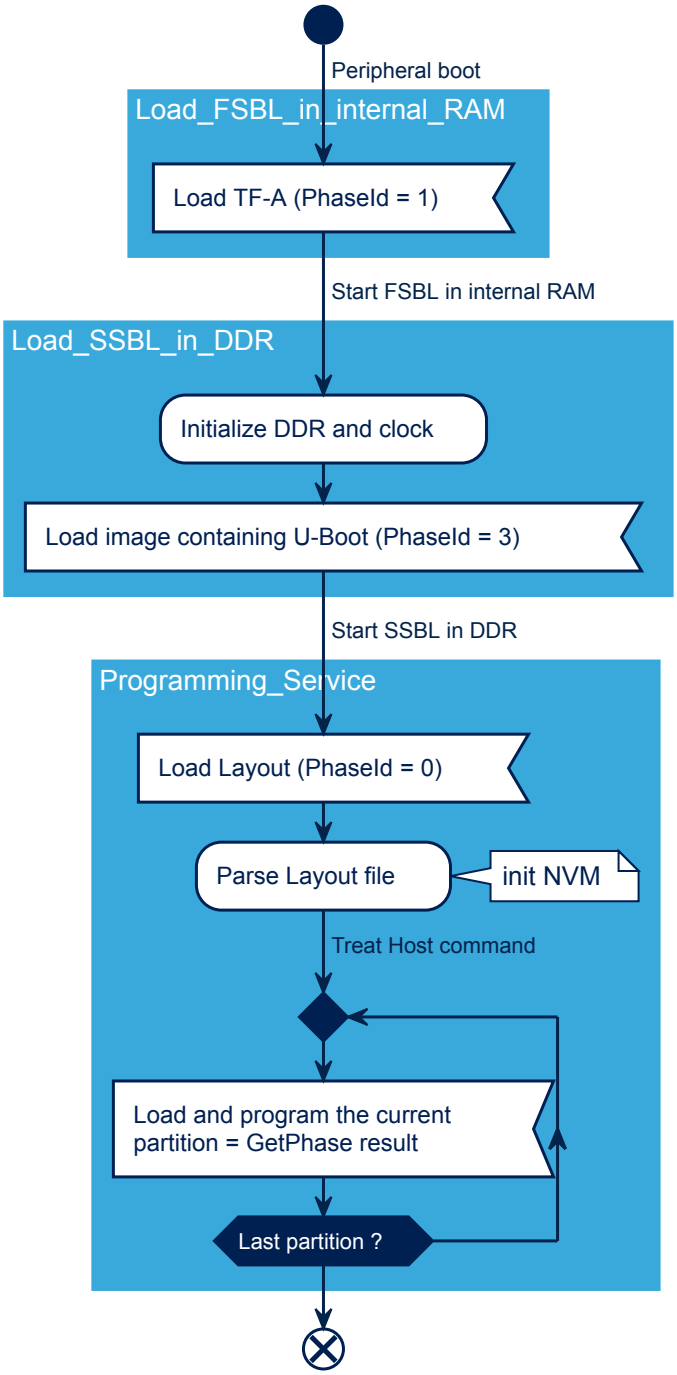
The signature is mandatory only on closed devices. For details on the signature process see signing tools [7], for definition of closed devices see [8].

## 1.7 Programming sequence

### 1.7.1 Case 1–programming from reset

In this normal use case of the STM32CubeProgrammer, the voltage level on the boot pins is used to determine the peripheral used for boot (USB or UART see [13]). The bootloaders are loaded by the STM32CubeProgrammer in RAM and the SSBL U-Boot provides the programming service running in DDR.

**Figure 2. Programing chart**



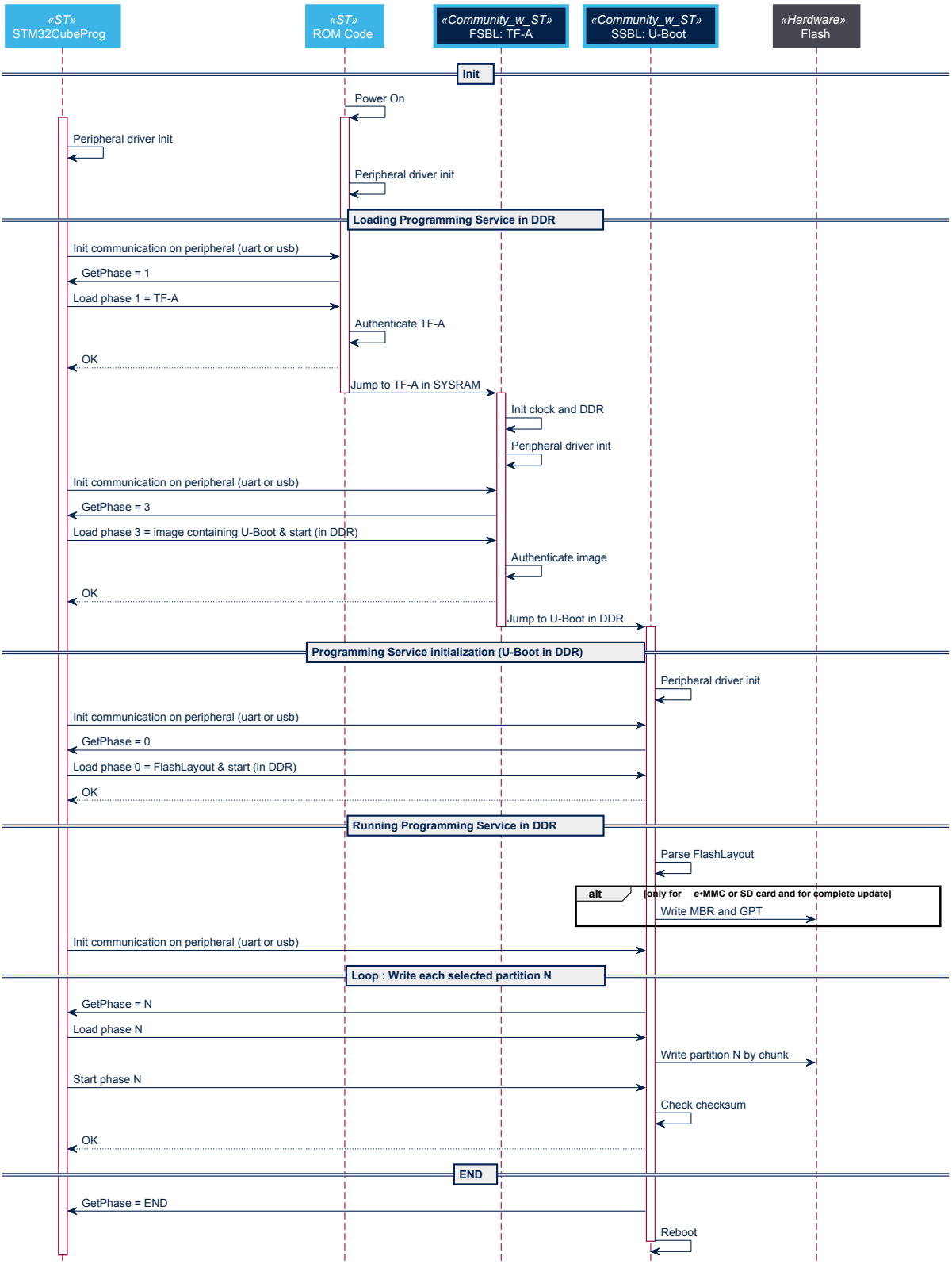
For a full update, the same boot stage partitions (image containing TF-A and U-Boot) is requested twice, once for loading and execution in RAM and once for the NVM update. However, the provided binary is not necessarily the same.

In U-Boot, the data chunks received on UART and on USB are buffered in DDR. The size of this buffer is device dependent.

The buffer is flushed to the NVM when it is full. Therefore, the ACK for each transferred chunk is delayed to when this NVM update is performed.

The diagram below gives an overview of a typical programming sequence.

**Figure 3. Programming sequence overview**



### 1.7.2 Case 2–programming from U-Boot for a programmed device

In this code update use case, the NVM is already programmed with a valid boot chain and selected by the boot pins (see [13]).

The expected sequence is:

1. The user switches on the board
2. The bootloaders are loaded from NVM and executed in RAM
3. The user interrupts the platform boot and starts U-Boot in programmer mode (UART or USB). Several solutions are possible:
   - Key press detection to enter in programming mode for example with the user button on the STMicroelectronics board
   - The user launches the download command in U-Boot console (command stm32prog)
   - Programmer mode requested by Linux via reboot mode
4. The user starts STM32CubeProgrammer on PC

Then the programming service runs in the DDR exactly as in the previous case.

# 2 UART/USART

## 2.1 UART/USART protocol

The UART/USART protocol follows as much as possible the STM32 MCU behavior described in [1].
The figure below highlights the differences and details the expected sequence.

**Figure 4. USART programming sequence description - 1**

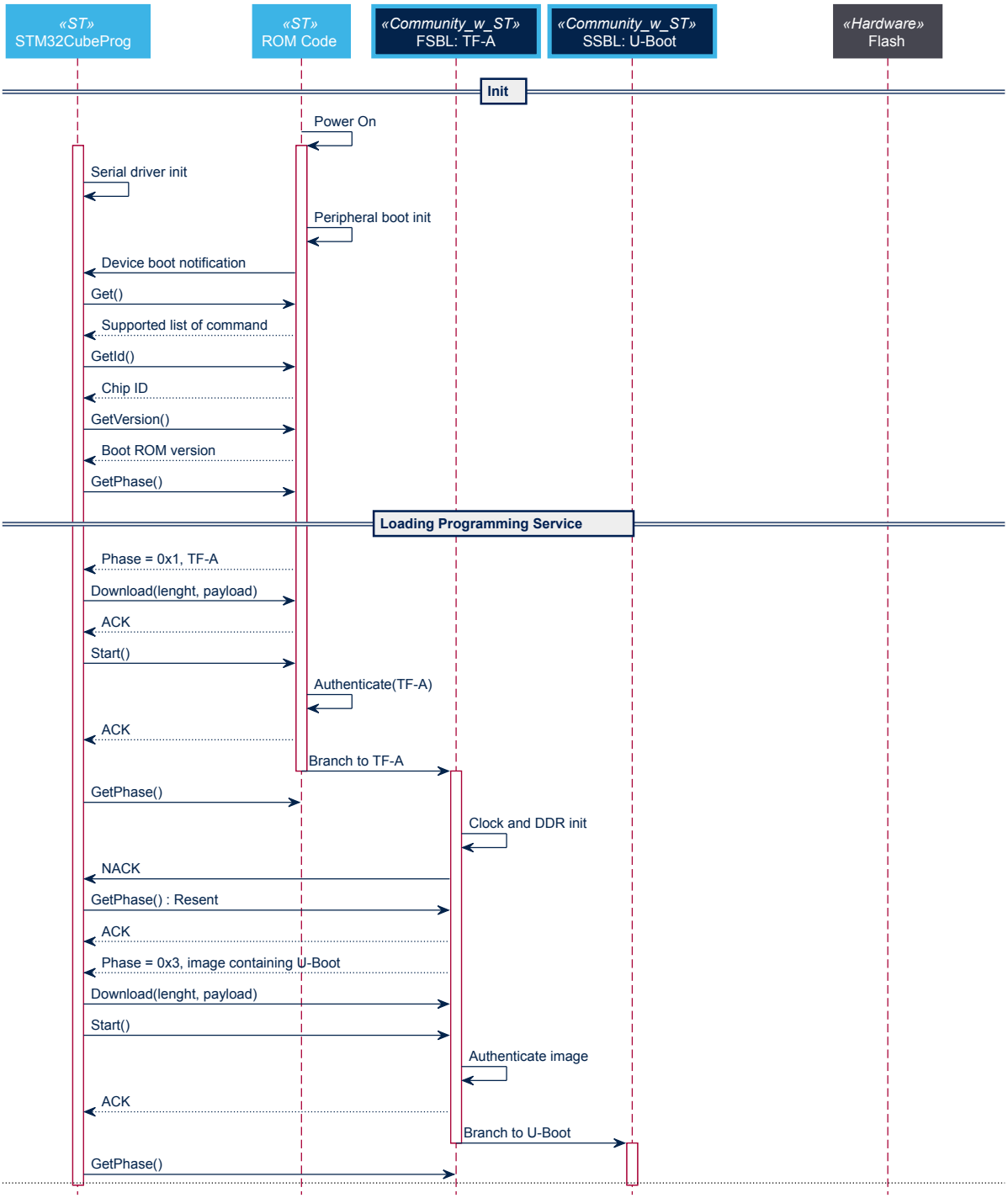
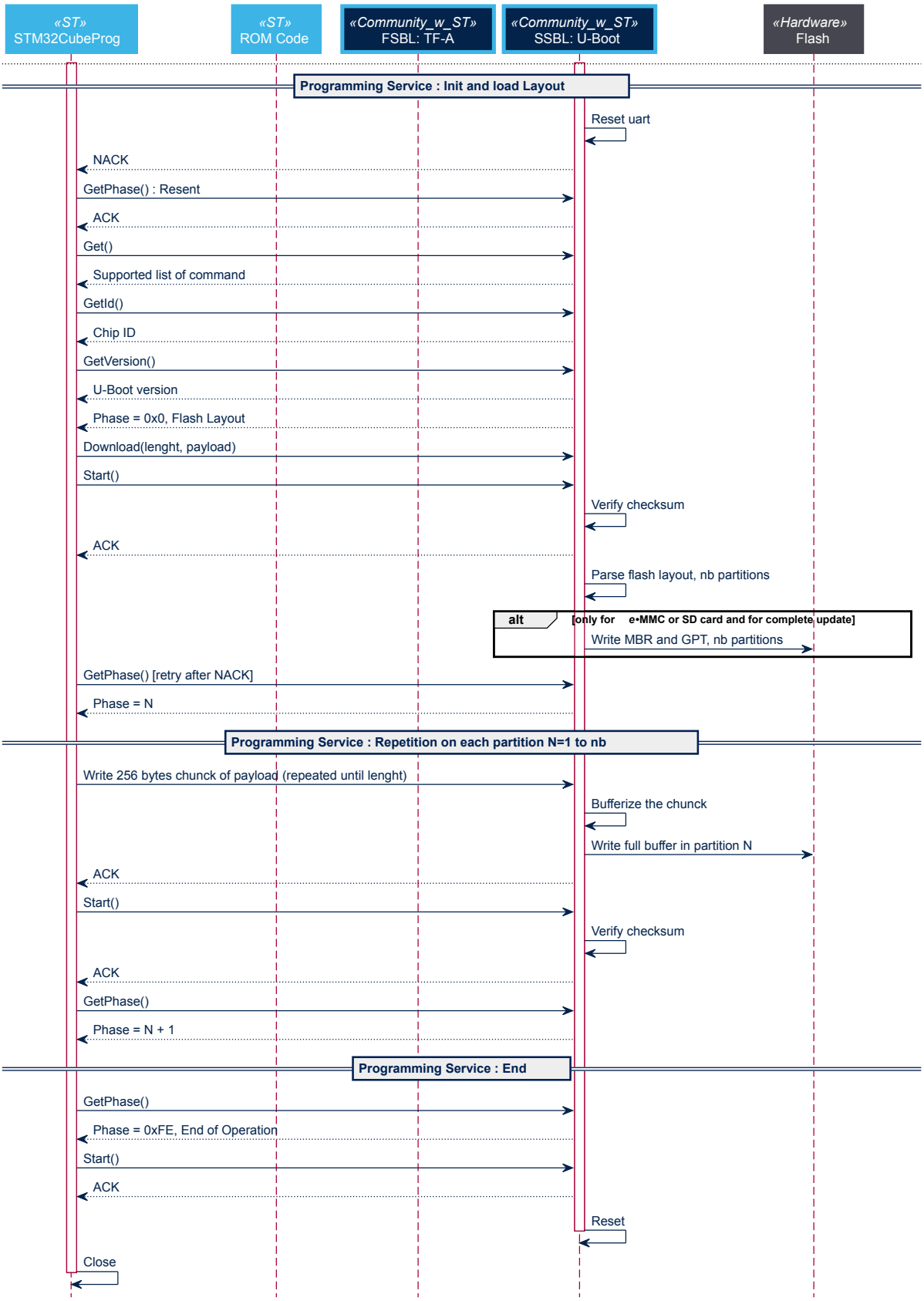


**Figure 5. USART programming sequence description - 2**



## 2.2 UART/USART configuration

The configuration is set by the ROM code ([13]) with the following settings:

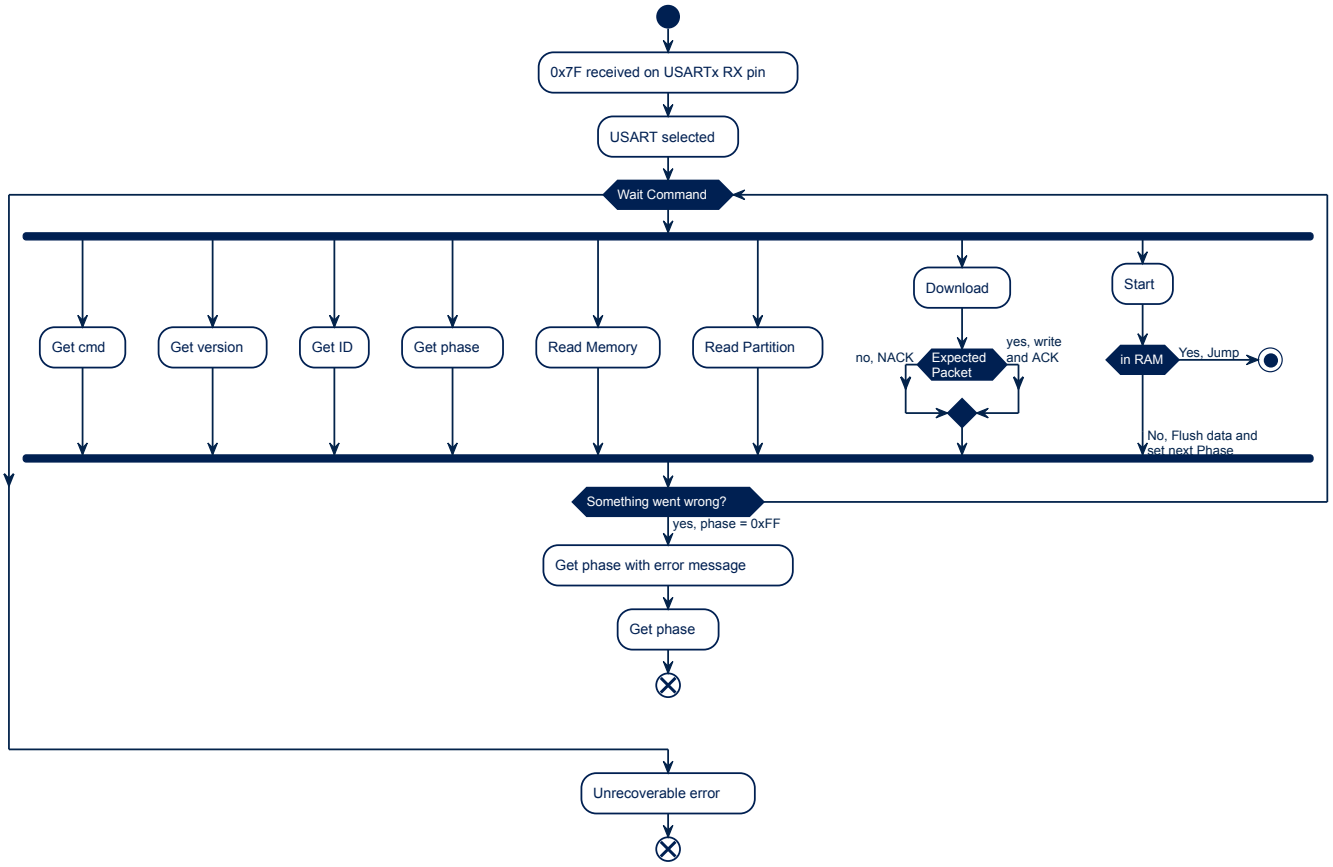
- baudrate = 115200 baud
- 8-bit data
- EVEN parity
- 1 start bit
- 1 stop bit

## 2.3 UART/USART connection

Once the serial boot mode is entered (see boot pin in [13]), all the UART/USART instances are scanned by the ROM code, monitoring for each instance the USARTx_RX line pin, waiting to receive the 0x7F data frame (one start bit, 0x7F data bits, none parity bit and one stop bit).

## 2.4 UART/USART main loop

**Figure 6. Main UART/USART sequence**




The device supports the following commands set:

- **Get cmd** (to get the bootloader version and the allowed commands supported by the current version of the bootloader)
- **Get version** (to get the software version)
- **Get ID** (to get the chip ID)
- **Get phase**(*) (to get the phase ID: the partition that is going to be downloaded)
- **Download** (to download an image into the device)
- **Read Memory** (to read SRAM memory)

- **Read Partition**(*)
- **Start** (to go to user code)

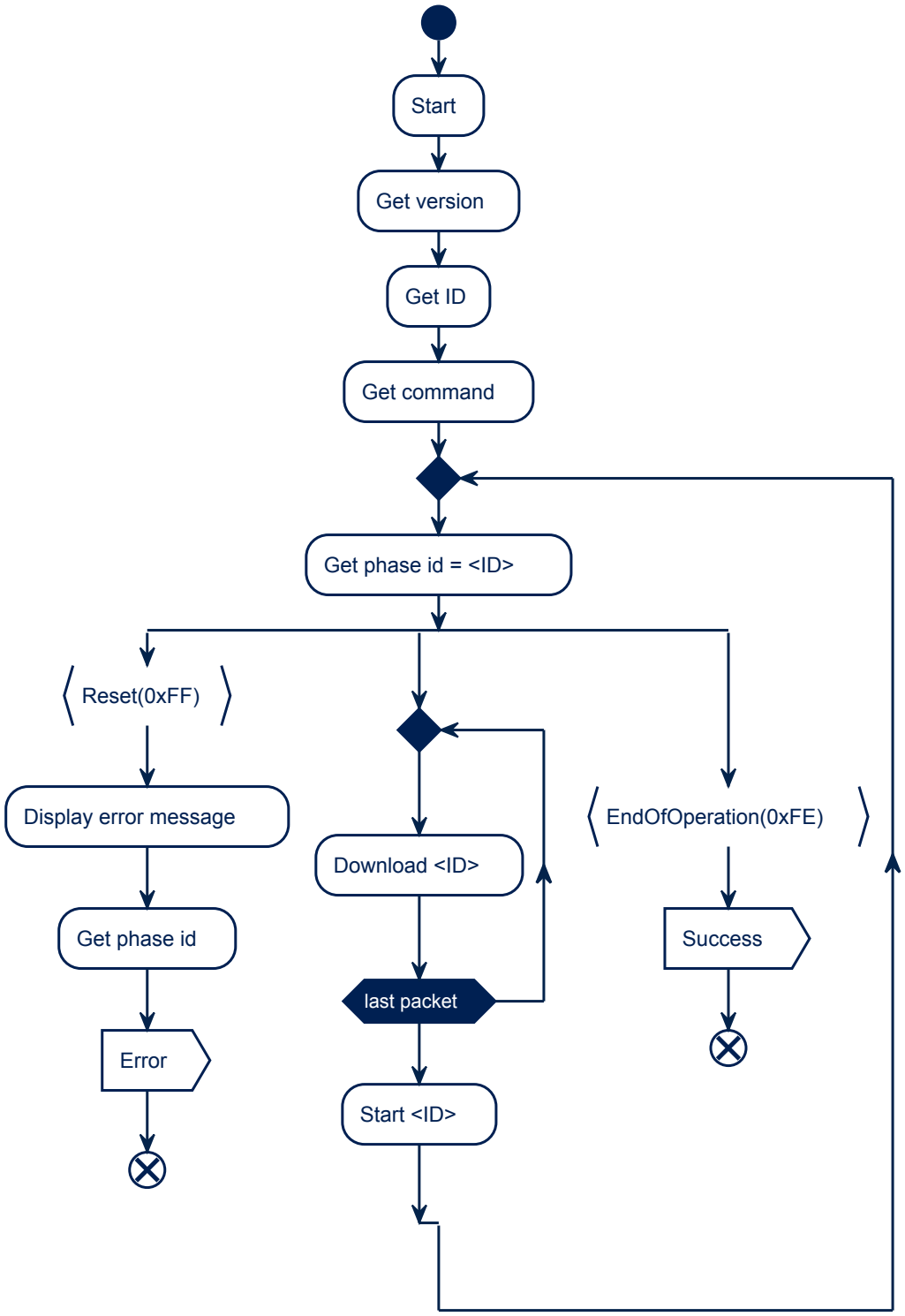
(*) STM32MP1 specific commands

Figure 6 illustrates the device code execution during the different download phases. The host starts by the get cmd command then the get phase command and depend the Phase ID the host sends the adequate data.

- *0: The host sends the data to download the layout file.*
- *1: The host sends the data to download the file containing FSBL (=TF-A) used by the programmer service.*
- *3: The host sends the data to download the image containing the SSBL (=U-Boot) used for the programmer service.*
- *[4, F]: Other boot partition.*
- *[10, F0]: User partition.*
- *F1: Reserved for UART/USART commands.*
- *[F2, FD]: Virtual partition, used for nonvolatile memory access as OTP.*
- *FE: End of operation.*
- *FF: Reset.*

The device returns an error when there is a corruption of data. In case of NACK error the system restarts the operation. In case of ABORT error the system reboots after the next GetPhaseId with the answered value RESET (0xFF).

Figure 7. **STM32CubeProgrammer download sequence**




### Abort response

The abort response is used to inform the host in case of a major issue (error during the signature check or error during writing). The only way to recover is to perform a system reset and to restart communication. The downloaded image is discarded immediately after the Abort response. The device forces a system reset after the next GetPhaseId with the answered value RESET (0xFF).

## 2.5 UART/USART command set

The supported commands are listed in the table below. Each command is further described in the following sections.

**Table 2. USART commands**

| Command | Command code | Command description |
|---|---|---|
| Get | 0x00 | Gets the version of the running element and the allowed commands supported. |
| Get Version | 0x01 | Gets the version |
| Get ID | 0x02 | Gets the device ID |
| Get phase | 0x03 | Gets the phase ID: the identifier of the partition in the layout file that is going to be downloaded |
| Read Memory | 0x11 | Reads up to 256 bytes of memory starting from an address specified by the application |
| Read Partition | 0x12 | Reads up to 256 bytes of partition at offset specified by the application (new in UART/USART bootloader protocol v4.0) |
| Start (Go) | 0x21 | Jumps to the user application located in the RAM or flush the received data in the nonvolatile memory |
| Download (Write Memory) | 0x31 | Download the image |
| Erase | 0x43 | Existing in USART protocol v3, not used in STM32MP1 |
| Extended Erase | 0x44 | Existing in USART protocol v3, not used in STM32MP1 |
| Special | 0x50 | Existing in USART protocol v3, not used in STM32MP1 |
| Extended Special | 0x51 | Existing in USART protocol v3, not used in STM32MP1 |
| Write Protect | 0x63 | Existing in USART protocol v3, not used in STM32MP1 |
| Write Unprotect | 0x73 | Existing in USART protocol v3, not used in STM32MP1 |
| Readout Protect | 0x82 | Existing in USART protocol v3, not used in STM32MP1 |
| Readout Unprotect | 0x92 | Existing in USART protocol v3, not used in STM32MP1 |
| Get Checksum | 0xA1 | Existing in USART protocol v3, not used in STM32MP1 |

#### Communication safety

All communications from STM32CubeProgrammer (PC) to the device are verified as follows:

- The UART/USART even parity is checked.
- For each command the host sends a byte and its complement (XOR = 0x00).
- The device performs a checksum on the sent/received datablocks. A byte containing the computed XOR of all previous bytes is appended at the end of each communication (checksum byte). By XORing all received bytes, data + checksum, the result at the end of the packet must be 0x00. A timeout must be managed in any waiting loop to avoid any blocking situation.

Each command packet is either accepted (ACK answer), discarded (NACK answer) or aborted (unrecoverable error):

- ACK = 0x79
- NACK = 0x1F
- ABORT = 0x5F

### 2.5.1 Get command (0x00)

The Get command returns the bootloader version and the supported commands. When the device receives the Get command, it transmits the version and the supported command codes to the host, as described in Figure 6. The commands not supported are removed from the list.

The device sends the bytes as described in the table below.

**Table 3. Get command response**

| Byte | Description |
|---|---|
| 1 | ACK |
| 2 | N = the number of following bytes – 1 (except current and ACKs) with:<br>• N = 8 for U-Boot, which support all these commands<br>• N = 7 for ROM code STM32MP15x<br>• N = 6 for ROM code STM32MP13x |
| 3 | UART/USART bootloader version (0 < version < 255)<br>example: 0x10 = version 1.0<br>On STM32MP1 the USART protocol version is V4.0, so the value is 0x40 |
| 4 | 0x00: Get command |
| 5 | 0x01: Get version |
| 6 | 0x02: Get ID |
| 7 | 0x03: Get phase ID |
| 8 | 0x31: Download command must be the last byte:11 (cmd ordered in increasing order) |
| 9 | 0x11: Read memory command (not supported by ROM code STM32MP13x) |
| 10 | 0x12: Read partition command (not supported by ROM code) |
| 11 | 0x21: Start command |
| Last | ACK |

### 2.5.2 Get ID command (0x02)

The Get ID command is used to get the version of the device ID (identification). When the device receives the command, it transmits the device ID to the host.

The device sends the bytes as follows:

**Table 4. Get ID command response**

| Byte | Description |
|---|---|
| 1 | ACK |
| 2 | N = 1 = the number of following bytes – 1 (except current and ACKs) |
| 3-4 | Device ID = 0x0500 for STM32MP15x or 0x0501 for STM32MP13x |
| Last | ACK |

### 2.5.3 Get version command (0x01)

The Get version command is used to get the version of the running component. When the device receives the command, it transmits the version to the host.

The device sends the bytes as follows:

**Table 5. Get version command response**

| Byte | Description |
|---|---|
| 1 | ACK |
| 2 | Bootloader version => software version (ROM code/TF-A/U-Boot)<br>(0 < version ≤ 255), example: 0x10 = version 1.0 |
| 3 | Option byte 1: 0x00(1) |
| 4 | Option byte 2: 0x00(1) |
| Last | ACK |

1. *Option byte keeps the compatibility with generic bootloader protocol see [Ref1].*

### 2.5.4 Get phase command (0x03)

*Note: This command is STM32MP1 specific.*

The Get phase command enables the host to get the phase ID, in order to identify the next partition that is going to be downloaded.

When the device receives the Get phase command, it transmits the partition ID to the host as follows:

**Table 6. Get phase command response**

| Byte | Description |
|---|---|
| 1 | ACK |
| 2 | N = the number of following bytes -1 (except current and ACKs, 0 ≤ N ≤ 255) |
| 3 | Phase ID |
| 4-7 | Download address |
| 8 | X = the number of bytes in additional information (X = N-5) |
| X | X bytes of additional information |
| Last | ACK |

The download address, when present, provides the destination address in memory. A value of 0xFFFFFFFF means than the partition is going to be written in NVM.

Phase ID = 0xFF corresponds to an answered value Reset, in this case the information bytes provide the cause of the error in a string just before executing the reset.

#### ROM code

The ROM code sends phase = TF-A

Byte 1: ACK

Byte 2 N = 6

Byte 3: phase ID (file containing FSBL = TF-A, 1)

Byte 4-7: 0x2FFC2400 on STM32MP15x, 0x2FFDFE00 on STM32MP13x

Byte 8: X = 1

Byte 9: 0: reserved

Byte 10: ACK

**TF-A**

The TF-A sends

Byte 1: ACK

Byte 2: N = 5

Byte 3: phase ID (file containing SSBL = U-Boot, 3)

Byte 4-7: load address of FIP in DDR, for example 0xC8000000

Byte 8: X = 0

Byte 9: ACK

**U-Boot**

The U-Boot sends the bytes as follows when no additional information is provided by U-Boot (N = 5, X = 0).

Byte 1: ACK

Byte 2: N = 5

Byte 3: phase ID (next partition to program)

Byte 4-7: 0xFFFFFFFF or load address in DDR

Byte 8: X = 0

Byte 9: ACK. For byte 8: X = 0

Byte 9: ACK

For example, with layout (phase = 0)

Byte 1: ACK

Byte 2: N = 5

Byte 3: phase ID = layout (0)

Byte 4-7: 0xC0000000 (DDR base address = load address of layout file)

Byte 8: X =0

Byte 9: ACK

For example, with TF-A (phase = 1)

Byte 1: ACK

Byte 2: N = 5

Byte 3: phase ID (FSBL1: TF-A =1)

Byte 4-7: 0xFFFFFFFF

Byte 8: X =0

Byte 9: ACK

For the error case, U-Boot sends the bytes as follows:

Byte 1: ACK

Byte 2: N = X-5

Byte 3: phase ID = Reset (0xFF)

Byte 4- 7: 0xFFFFFFFF

Byte 8 X= string size

X bytes: string in ascii (Max 250 bytes): the cause of error

Last byte: ACK

### 2.5.5 Download command (0x31)

The download command is used to download a binary code (image) into the SRAM memory or to write a partition in NVM.

Two types of operations are available:

- **Normal operation**: download current partition binary to the device. For initialization phase the partitions are loaded in SRAM, otherwise for writing phase the partition are written in NVM.
- **Special operation**: download non-signed data to non-executable memory space.

A Start command is necessary to finalize these operations after the download command.

The Packet number is used to specify the type of operation and the number of the current packet. The table below gives the description of the packet number.

**Table 7. Packet number**

| Byte | Value | Description |
|---|---|---|
| 3 | 0x00 | Normal operation: write in current phase |
| | 0xF2 | Special operation: OTP write |
| | 0xF3 | Special operation: Reserved |
| | 0xF4 | Special operation PMIC: NVM write |
| | Others | Reserved |
| 0-2 | - | Packet number, increasing from 0 to 0xFFFFFF (*) |

*Note: Packet number it is not an address as on STM32 MCU with only memory mapped flash, but the index of the received packet. The offset of the packet N the offset in the current partition/phase is N* 256 bytes when only full packets are used.*

- Examples:
  - Packet number = 0x00603102
    Operation = normal operation, packet number = 0x603102
  - Packet number = 0xF200000N: send Nth OTP part
    Operation = OTP write, OTP part number

The host sends the bytes to the device as shown in the table below.

**Table 8. Download command**

| Byte | Description |
|---|---|
| 1 | 0x31 = download (write memory) |
| 2 | 0xCE = XOR of byte 1 |
| - | Wait for ACK or NACK |
| 3-6 | Packet number, as described in Table 7 |
| 7 | Checksum byte: XOR (byte 3 to byte 6) |
| - | Wait for ACK or NACK |
| 8 | Packet size (0 < N < 255) |
| 9-(Last-1) | N+1 data bytes (max 256 bytes) |
| Last | Checksum byte: XOR (byte 8 to Last-1) |
| - | Wait for ACK or NACK |

### 2.5.6 Read memory command (0x11)

The Read memory command is used to read data from any valid memory address in the system memory.

When the device receives the read memory command, it transmits the ACK byte to the application. After the transmission of the ACK byte, the device waits for an address (4 bytes) and a checksum byte, then it checks the received address. If the address is valid and the checksum is correct, the device transmits an ACK byte, otherwise it transmits a NACK byte and aborts the command.

When the address is valid and the checksum is correct, the device waits for N (N = number of bytes to be received -1) and for its complemented byte (checksum). If the checksum is correct the device transmits the needed data (N+1 bytes) to the application, starting from the received address. If the checksum is not correct, it sends a NACK before aborting the command.

To read memory mapped content, the Host sends bytes to the device as follows:

Table 9. **Read memory command**

| Byte | Description |
|---|---|
| 1 | 0x11 = read memory |
| 2 | 0xEE = XOR of byte 1 |
| - | Wait for ACK or NACK |
| 3-6 | Start address |
| 7 | Checksum byte: XOR (byte 3 to byte 6) |
| - | Wait for ACK or NACK |
| 8 | Number of bytes to be received – 1 (N = [0, 255]) |
| 9 | Checksum byte: XOR (byte 8) |
| - | Wait for ACK or NACK |

### 2.5.7 Read partition command (0x12)

*Note: This command is STM32MP1 specific.*

The Read command is used to read data from any valid offset in one partition as the device associated with phase is not memory mapped.

When the device receives the read memory command, it transmits the ACK byte to the application. After the transmission of the ACK byte, the device waits for a partition ID, an offset (4 bytes) and a checksum byte, then it checks the received address. If the address is valid and the checksum is correct, the device transmits an ACK byte, otherwise it transmits a NACK byte and aborts the command.

When the address is valid and the checksum is correct, the device waits for the number of bytes to be transmitted – 1 (N bytes) and for its complemented byte (checksum). If the checksum is correct it then transmits the needed data (N bytes) to the application, starting from the received address. If the checksum is not correct, it sends a NACK before aborting the command.

To read partition in NVM, the Host sends bytes to the device as shown in the table below.

Table 10. **Read Partition command**

| Byte | Description |
|---|---|
| 1 | 0x12 = Read Partition |
| 2 | 0xED = XOR of byte 1 |
| - | Wait for ACK or NACK |
| 3 | partition Id = value |
| 4-7 | offset address |
| 8 | Checksum byte: XOR (byte 3 to byte 7) |
| - | Wait for ACK or NACK |
| 9 | Number of bytes to be received – 1 (N = [0,255]) |
| 10 | Checksum byte: XOR (byte 9) |
| - | Wait for ACK or NACK |

### 2.5.8 Start command (0x21)

The Start command is used:

- To execute the code just downloaded in the memory or any other code by branching to an address specified by the application. When the device receives the Start command, it transmits the ACK byte to the application. If the address is valid the device transmits an ACK byte and jumps to this address, otherwise it transmits a NACK byte and aborts the command.
- To finalize the last download command, when the host indicates the address = 0xFFFFFFFF. The Host sends bytes to the device as shown in the table below.

**Table 11. Start command**

| Byte | Description |
|---|---|
| 1 | 0x21 = start |
| 2 | 0xDE = XOR of byte 1 |
| - | Wait for ACK or NACK |
| 3-6 | Start address or 0xFFFFFFFF |
| 7 | Checksum byte: XOR (byte 3 to byte 6) |
| - | Wait for ACK or NACK |

# 3 USB

## 3.1 DFU protocol

The embedded programming service uses the DFU protocols v1.1 (see [5] for details).

The only difference with DFU v1.1 protocol is: DFU_DETACH is acceptable in dfuIDLE state and get back to runtime mode appIDLE.

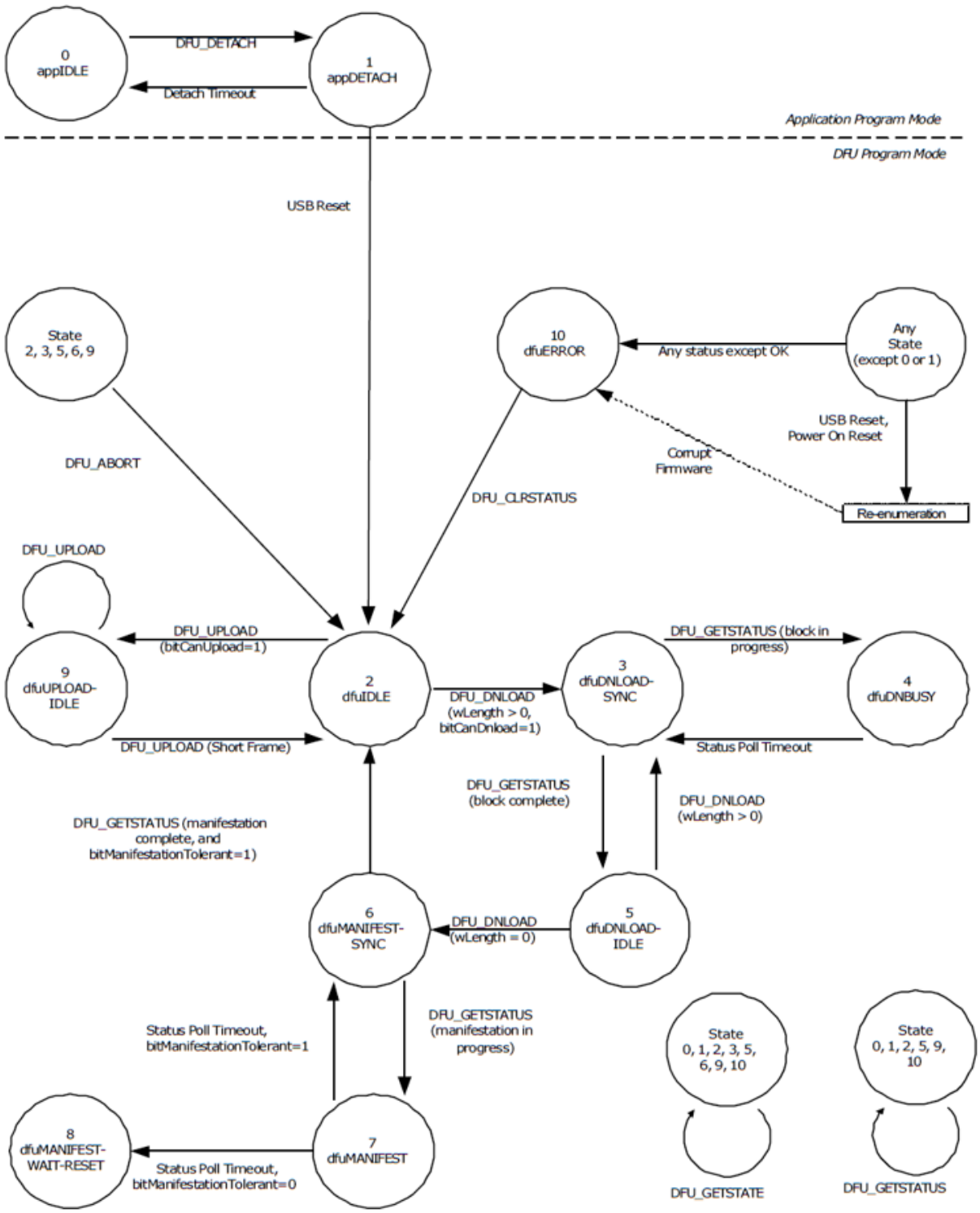
*Note: This behavior is already supported in U-Boot DFU stack, in STM32Programmer (with option '-detach') and in dfu-utils (with -e Option).*

In dfuIDLE state, the device expects a USB reset to continue the execution. In the following sections, only the STM32MP1 specificities are presented.

**Caution:** The STMicroelectronics extensions on DFU standard used in STM32 MCU are not supported in STM32MP1 Series. These extensions are also named DFUSE; the DFU version to v1.1a when these extensions are used (see [2] for details).

The DFU states are described in the interface state transition diagram (see figure below, source: chapter A.1 of [5] ):

**Figure 8. Interface state transition diagram**



## 3.2 USB sequence

Since the USB description is shared by TF-A and by the STM32MP15x ROM code, a new enumeration is not required after TF-A manifestation for STM32MP15x. As a consequence the STM32MP15 ROM code must present the alternate settings used by TF-A, even if they are not supported in the ROM code.

For STM32MP13x, the USB controller is disconnected in ROM code before manifestation on FSBL phase, even if the USB PHY is not reset, a new enumeration is required in FSBL=TF-A.

The DFU application, STM32CubeProgrammer of dfu-util, supports the download of several partitions with alternate settings. Therefore all the DFU stack in ROM code TF-A and U-Boot must indicate it is manifestation-tolerant, (bitMainfestationTolerant = 1 in DFU attributes).

When re-enumeration is required, between ROM code and TF-A on STM32MP13x or between TF-A and U-Boot, a DFU detach is requested with the following sequence:

1. Host request state, device answer is dfuIdle (state 2)
2. Host request DFU_UPLOAD (GetPhase)
3. Device indicates the same phase 0x0 but with the “need detach” flag in additional information bytes.
4. The host sends command DFU_DETACH and DFU is in dfuIdle mode on device:
5. The Host reset the USB

Figure 9 presents the complete USB sequence in more details.

Figure 9. **USB sequence - 1**

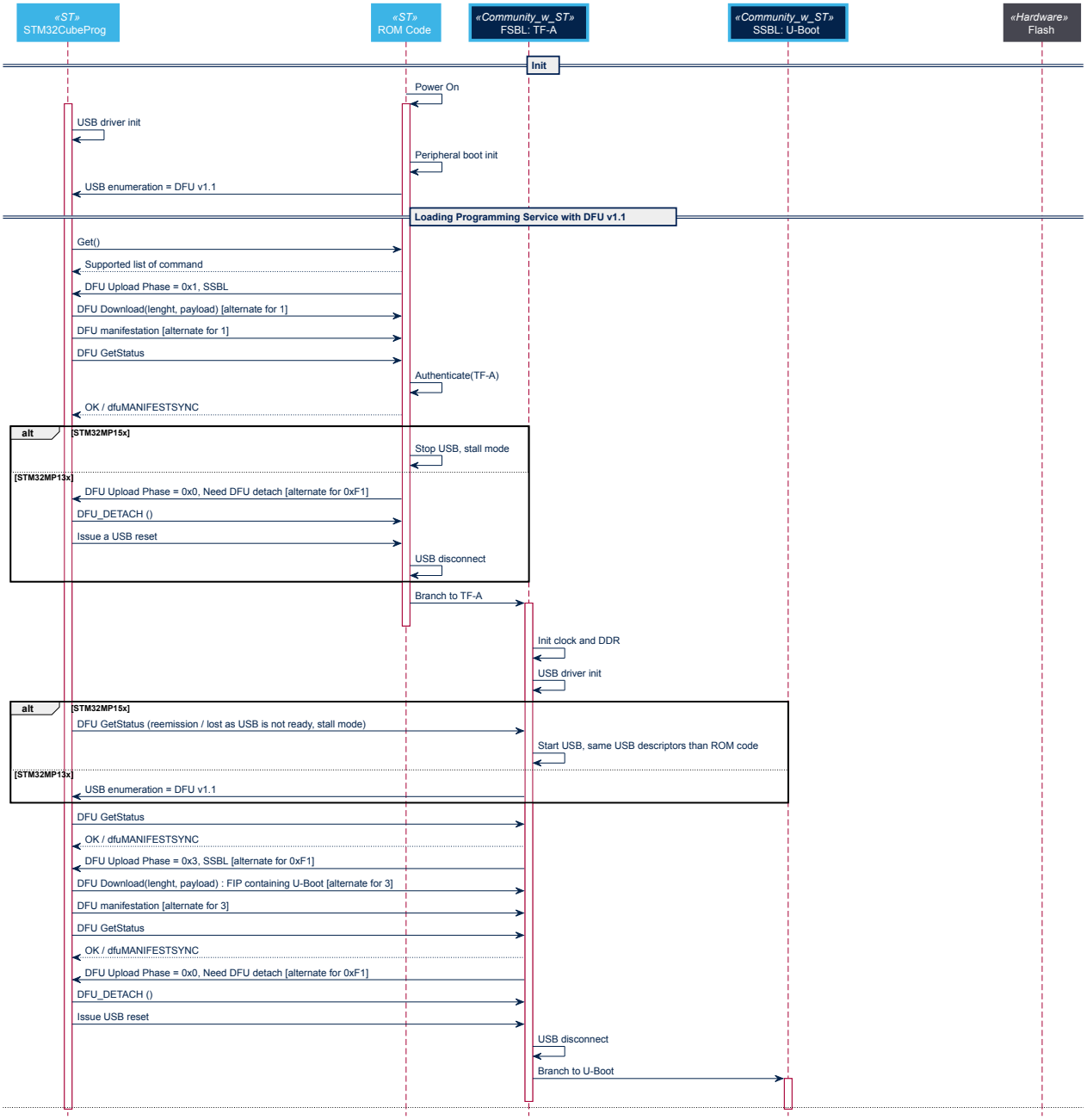
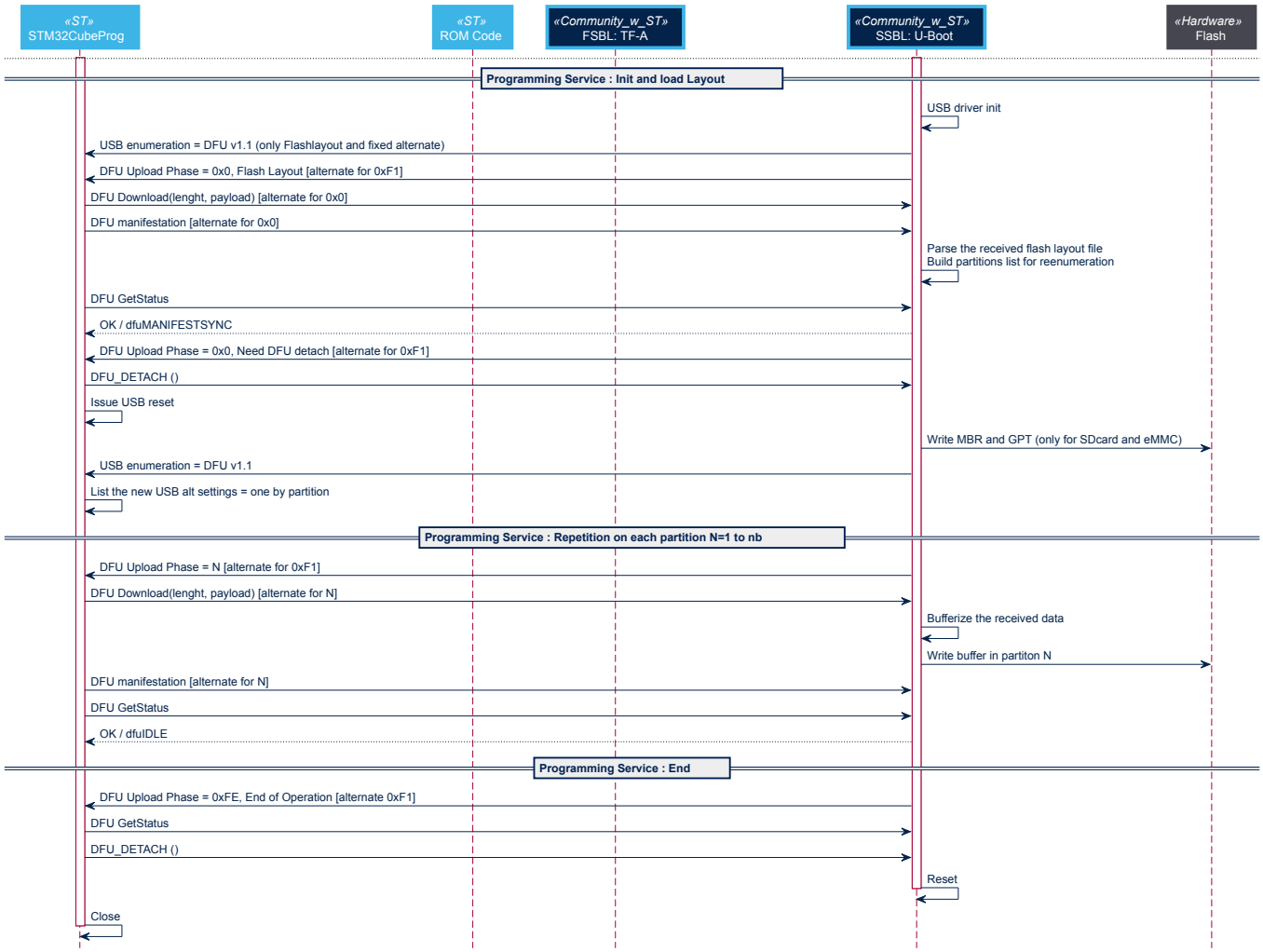


Figure 10. **USB sequence - 2**




## 3.3 DFU enumeration and alternate settings

During the USB enumeration, the STM32 device information is present in the DFU mode device descriptor:

- idVendor = 0x0483 for "STMicroelectronics"
- idProduct = 0xdf11 for "STM Device in DFU Mode"
- iSerial = string build from Unique device ID
- iProduct = the Product string, with STMicroelectronics encoding to provide all the remaining STM32 informations:

```
<Info>@Device ID /<DevId>, @Revision ID /<RevId> [, @Name /<DevName>]
```

with

  - <Info> = information on used USB stack, for example: *DFU in HS Mode* in ROM code or TFA *USB download gadget*in U-Boot
  - <DevId> = Device ID 0x500 for STM32MP15x 0x501 for STM32MP13x
  - <RevId>= Silicon revision
  - <DevName> = the device name build with Device Part Number (RPN), not provided by ROM code

Each partition to load in RAM for ROM code and TF-A or to be program in NVM for U-Boot is available with alternate settings of the DFU profile.

- The Phase ID is aligned with the alternate setting identifier.
- The device support GetPhase with UPLOAD on a specific alternate setting (the last one).

- The name of the alternate setting string descriptor respects the description of [4] chapter 10.

```
@Target Memory Name/Start Address/Sector(1)_Count*Sector(1)_Size
Sector(1)_Type,Sector(2)_Count*Sector(2)_SizeSector(2)_Type,...
...,Sector(n)_Count*Sector(n)_SizeSector(n)_Type
```

Since the partitions for STM32MP1 are not memory mapped, the “Start Address” is replaced by partition IDs (0x + 2 digits for phase ID).

The partitions have only one sector and the size is computed from the layout file. Only two types of partitions are supported for STM32MP1:

- a (0x41): readable for partition not selected in layout file.
- e (0x45): readable and writeable for partition expected to be updated.

Some virtual partitions (not directly linked to memory on NVM) are added and use reserved phase ID:

- 0xF1 = partition ID reserved for command GetPhase.
- 0xF2 = partition ID reserved for OTP
- 0xF4 = partition ID reserved for PMIC NVM

### 3.3.1 The virtual command partitions 0xF1 (GetPhase/SetOffset/Start)

The virtual partition 0xF1 is used for the following commands:

- DFU_UPLOAD: the GetPhase command
- DFU_DOWNLOAD: the SetOffset and Start command (jump to an address in memory)

#### *3.3.1.1 DFU download*

This is only supported in U-Boot. The format of SetOffset command is given below.

**Table 12. DFU SetOffset command**

| Byte | Description |
|---|---|
| 1 | Phase ID: normal partition between 0x00 to 0xF0 |
| 2-5 | Offset in the partition |

When the SetOffset is received, the current Phase ID is updated (for GetPhase) and the offset is used for the next upload/download on the selected partitions

The format of Start command, to execute the code loaded in the memory, is given below.

**Table 13. DFU Start command**

| Byte | Description |
|---|---|
| 1 | 0xFF |
| 2-5 | Memory address to jump in |

#### *3.3.1.2 DFU upload*

The content of the upload on this alternate setting is GetPhase command result:

**Table 14. DFU GetPhase command response**

| Byte | Description |
|---|---|
| 1 | Phase ID |
| 2-5 | Download address |
| 6-9 | Offset |
| 10-Last | Additional information optional, size: 0 up to 250 bytes |

The download address, when present, provides the load address in memory. A value of 0xFFFFFFFF means that the partition identified by phase ID is going to be written in NVM.

The offset field provides the current offset used in partition operation, it is 0 by default.

The content of the additional information is determined by the phase ID value:

- Phase ID = 0xFF, the cause of the error which causes the reset request in a string
- Phase ID = 0x0, for Layout file the information byte is:
  - Byte 1 Need Reset Indication:
    = 1 if DFU_DETACH is requested
    = 0 or absent if DFU_DETACH is not requested

## 3.4 DFU stack in ROM code and TF-A

TF-A and ROM code use a STMicroelectronics DFU v1.1 stack.

### 3.4.1 ROM code first USB enumeration for STM32MP15x device

The STM32MP15x ROM code initializes the USB and the DFU stack, and allows the loading and the execution of TF-A in the internal RAM. Before execution of TF-A, the USB stack is interrupted after the manifestation on TF-A alternate with dfuMANIFESTSYNC state.

In TF-A, the USB device is restored and the DFU stack is resumed (no USB reset, no enumeration). The first request received by TF-A must return dfuIDLE. At the first enumeration ROM code, the alternate settings present the partitions that are used by the ROM code and by FSBL TF-A.

**Table 15. STM32MP15x ROM code USB enumeration**

| Alternate setting | PhaseID | String descriptor | ROM code support | TF-A support |
|---|---|---|---|---|
| **0** | 0 | @Partition0 /0x00/1*256Ke | No | No |
| **1** | 1 | @FSBL /0x01/1*1Me | Yes | No |
| **2** | 2 | @Partition2 /0x02/1*1Me | No | No |
| **3** | 3 | @Partition3 /0x03/1*16Me | No | File containing SSBL (FIP) |
| **4** | 0xF1 | @virtual /0xF1/1*512Ba | GetPhase | GetPhase |

The sizes are chosen to be flexible even if they are fixed in the ROM code:

- Partititon0 = layout file expected less than 256 KB
- FSBL = TF-A size is limited by internal RAM (set max to 1 MB)
- Partition2 = reserved for future use (1 MB)
- Partition3 = SSBL: 16 MB used to be able to load kernel directly
- Partition4 = reserved for future use (16 MB)
- 0xF1 = GetPhase: read-only, limited to 256 bytes normally

### 3.4.2 ROM code first USB enumeration for STM32MP13x device

The STM32MP13x ROM code initializes the USB and the DFU stack, and allows the loading and the execution of TF-A in internal RAM. Before execution of TF-A, the USB is disconnected on the DFU DETACH request and a new USB enumeration is done in TF-A after a USB connect.

At the first enumeration on STM32MP13x, the ROM code alternate settings present the partitions that are used by the ROM code only.

**Table 16. STM32MP13x ROM code USB enumeration**

| Alternate setting | PhaseID | String descriptor |
|---|---|---|
| 0 | 1 | @FSBL /0x01/1*128Ke |
| 1 | 0xF1 | @virtual /0xF1/1*512Ba |

### 3.4.3 TF-A USB enumeration

In STM32MP15x TF-A, as the ROM code USB configuration is re-used without a new USB enumeration, the alternate settings are fixed by the first enumeration done by STM32MP15x in ROM code (see Section 3.4.1 ).

In STM32MP13x TF-A, the second USB enumeration presents the alternate settings for the partitions used to load the file containing the SSBL.

The number and size of partitions are flexible, they can be modified in TF-A code. The table below gives an example.

**Table 17. STM32MP13x TF-A USB enumeration**

| Alternate setting | PhaseID | String descriptor |
|---|---|---|
| 0 | 3 | @SSBL /0x03/1*16Me |
| 1 | 0xF1 | @virtual /0xF1/1*512Ba |

*Note: As shown in the above table Partition3 with maximum size 16 MB can be used for FIP including kernel.*

## 3.5 DFU stack in U-Boot

The existing DFU stack of U-Boot is used.

The USB stack, the controller and the USB PHY must be initialized in DFU mode when the USB download mode is indicated by ROM code or TF-A.

The GetPhase command is available by download command on partition 0xF1.

STM32CubeProgrammer must loop until phase is 0xFE or 0xFF:

- Get current Phase: DFU_UPLOAD (GetPhase = 0xF1)
- Found associated file in field binary of layout file
- Download file in associated alternate settings
- Manifestation and pool manifestation end

For NVM partition, the upload / download operation accesses the NVM and the manifestation flushes the last operation in the device.

For “memory” partition, the manifestation only flushes the current operation and the cache. The start in memory operation is only done with DFU_DOWNLOAD on virtual partition (see 3.3.1).

### 3.5.1 U-Boot first USB enumeration

U-Boot presents the needed alternate settings to load the layout (minimal configuration is, phase ID = 0x0 and 0xF1). Other memory or special region (phase ID > 0xF0) can be also added by U-Boot in other alternates, for example with OTP:

**Table 18. U-Boot first USB enumeration**

| Alternate setting | PhaseID | String descriptor |
|---|---|---|
| 0 | 0 | @Flashlayout /0x00/1*256Ke |
| 1 | 0xF1 | @virtual /0xF1/1*512Be |
| 2 | 0xF2 | @OTP /0xF2/1*776Be |

### 3.5.2 U-Boot second USB enumeration

For its second enumeration, U-Boot presents all the needed alternate settings:

- Each partition in NVM or each device, defined in the Layout file (phase, size, type is writable only if selected), with Address = Phase ID
- Memory mapping region, when it is added in U-Boot
- Special region (address = reserved, phase ID > 0xF0)

It depends on U-Boot settings and Layout file content. An example is shown in the table below.

Table 19. **U-Boot second USB enumeration**

| Alternate setting | PhaseID | String descriptor |
|---|---|---|
| 0 | 0 | **@Flashlayout** /0x00/1*4Ke |
| 1 | 1 | **@fsbl1 /0x01/**1*256Ke |
| 2 | 2 | **@fsbl2 /0x02/**1*256Ke |
| 3 | 3 | **@ssbl /0x03/**1*512Ke |
| 4 | 0x10 | @bootfs /0x10/1*64Me |
| 5 | 0x11 | @rootfs /0x11/1*512Me |
| N | M | Last user partion in Layout (N<0xF0) |
| N+1 | - | @sysram /0x2FFF000/1*256Ke |
| N+2 | - | @mcuram /0x30000000/1*284Ke |
| N+3 | - | @ddr /0xC0000000/1*1Ge |
| Last-1 | 0xF1 | @virtual /0xF1/1*512Ba |
| Last | 0xF2 | @OTP /0xF2/1*512Be |

## Revision history

Table 20. **Document revision history**

| Date | Version | Changes |
|---|---|---|
| 11-Oct-2019 | 1 | Initial release. |
| 17-Mar-2022 | 2 | Updated Section 1.1 Introduction |
| 22-Mar-2022 | 3 | Updated Figure 9. Interface state transition diagram |
| 15-Nov-2022 | 4 | Updated:<br>• Section 1.4 Layout file format<br>• Section 1.6 STM32 image header<br>• Figure 4, Figure 5 and Figure 6<br>• Section 2.5 UART/USART command set |
| 22-Nov-2022 | 5 | Updated:<br>• Section 1.4 Layout file format<br>• Section 1.6 STM32 image header<br>• Figure 4, Figure 5 and Figure 6<br>• Section 2.5 UART/USART command set<br>• Figure 5. USART programming sequence description - 2<br>• Figure 10. USB sequence - 2<br>Corrected figure numbering on:<br>• Figure 4. USART programming sequence description - 1 and Figure 5. USART programming sequence description - 2<br>• Figure 9. USB sequence - 1 and Figure 10. USB sequence - 2 |
| 17-Jan-2023 | 6 | Added:<br>• Section 3.4 DFU stack in ROM code and TF-A<br>• Section 3.4.1 ROM code first USB enumeration for STM32MP15x device<br>• Section 3.4.2 ROM code first USB enumeration for STM32MP13x device<br>• Section 3.4.3 TF-A USB enumeration<br>• Section 3.5 DFU stack in U-Boot<br>• Section 3.5.1 U-Boot first USB enumeration<br>• Section 3.5.1 U-Boot first USB enumeration<br>• Section 3.5.2 U-Boot second USB enumeration<br>Updated:<br>• Section 1.1 Introduction<br>• Section 1.2 Reference<br>• Section 1.3 Overview<br>• Section 1.5 Phase ID<br>– Table 1. Phase ID<br>• Section 1.6 STM32 image header<br>• Figure 2. Programing chart<br>• Figure 3. Programming sequence overview<br>• Section 1.7.2 Case 2–programming from U-Boot for a programmed device<br>• Section 2.1 UART/USART protocol<br>– Figure 4. USART programming sequence description - 1<br>• Section 2.4 UART/USART main loop<br>– Figure 6. Main UART/USART sequence<br>– Figure 7. STM32CubeProgrammer download sequence<br>• Section 2.5 UART/USART command set<br>– Table 2. USART commands<br>• Section 2.5.1 Get command (0x00)<br>– Table 3. Get command response<br>• Table 4. Get ID command response |

| Date | Version | Changes |
|---|---|---|
| | | • Section 2.5.4 Get phase command (0x03)<br>– Table 6. Get phase command response<br>• Section 2.5.5 Download command (0x31)<br>– Table 7. Packet number<br>• Section 2.5.6 Read memory command (0x11)<br>• Section 2.5.6 Read memory command (0x11)<br>• Section 2.5.8 Start command (0x21)<br>• Section 3.1 DFU protocol<br>• Section 3.2 USB sequence<br>– Figure 9. USB sequence - 1<br>• Section 3.3 DFU enumeration and alternate settings<br>• Section 3.3.1 The virtual command partitions 0xF1 (GetPhase/ SetOffset/Start)<br>• Section 3.3.1.1 DFU download<br>– Table 13. DFU Start command<br>• Section 3.4.1 ROM code first USB enumeration for STM32MP15x device<br>Removed:<br>• Figure "Programming sequence for boot from NVM" in Section 1.7.2 Case 2–programming from U-Boot for a programmed device |

# Contents

# List of tables

# List of figures

**IMPORTANT NOTICE – READ CAREFULLY**

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2023 STMicroelectronics – All rights reserved