TLD4020-3ET

Firmware User manual

LITIX™ Interior
RGB LED driver
Z8F80417369

About this document

Scope and purpose

This document provides technical information on the firmware features of the LITIX™ TLD40xx device and technical guidance on how to interact with the device using embedded mechanisms and user utility functions. The subsequent sections provide the necessary information for configuring the device, uploading custom code and for debugging application software.

The bootROM firmware for the TLD40xx provides the following features:

- Startup procedure
- Support for connecting debuggers
- Default Bootstrap Loader (BSL) for NVM programming and diagnostics
- Support for proprietary user BSL
- NVM operations handling, for example, programming, erasing and verifying the NVM

Intended audience

The intended audience are software developers, application system integrators, and debugging tool vendors.

Table of contents

About this document . . . 1
Table of contents . . . 1
1 Firmware architecture . . . 4
1.1 Startup . . . 4
1.2 Default bootstrap loader (BSL) . . . 6
1.3 User bootstrap loader (UBSL) . . . 7
1.4 Utility functions . . . 7
2 Operation modes . . . 8
2.1 Boot modes . . . 8
2.1.1 Boot mode selection . . . 8
2.1.2 Device initialization . . . 9
2.1.2.1 Watchdog configuration . . . 9
2.1.2.2 RAM test and RAM initialization . . . 9
2.1.3 Reset priority . . . 9
2.2 Default BSL mode . . . 10
2.2.1 NAD address . . . 11
2.2.2 NAC time window . . . 11

Table of contents

2.2.3 Default BSL frame format . . . 11
2.2.4 Timing constraints . . . 12
2.2.5 Default BSL interval between frames timeout behavior . . . 12
2.2.6 Default BSL host synchronization . . . 12
2.3 User BSL mode . . . 12
2.4 User mode . . . 13
2.5 User debug mode . . . 14
2.6 Error mode . . . 15
3 Programming model . . . 17
3.1 Debug interface . . . 17
4 API documentation . . . 17
4.1 Modules . . . 18
4.1.1 BSL Commands . . . 18
4.1.1.1 Member summary . . . 19
4.1.1.2 Function details . . . 20
4.1.1.2.1 status_t **bsl_cmd_handle_baudrate_set** (BSL_PROT_t * **pProtocol**) . . . 20
4.1.1.2.2 status_t **bsl_cmd_handle_dev_reset** (const BSL_PROT_t * **pProtocol**) . . . 20
4.1.1.2.3 status_t **bsl_cmd_handle_interface_mgnt** (const BSL_PROT_t * **pProtocol**) . . . 21
4.1.1.2.4 status_t **bsl_cmd_handle_mem_erase** (BSL_PROT_t * **pProtocol**) . . . 22
4.1.1.2.5 status_t **bsl_cmd_handle_mem_exec** (const BSL_PROT_t * **pProtocol**) . . . 22
4.1.1.2.6 status_t **bsl_cmd_handle_mem_read** (BSL_PROT_t * **pProtocol**) . . . 23
4.1.1.2.7 status_t **bsl_cmd_handle_mem_verify** (const BSL_PROT_t * **pProtocol**) . . . 23
4.1.1.2.8 status_t **bsl_cmd_handle_mem_write** (BSL_PROT_t * **pProtocol**) . . . 24
4.1.1.2.9 status_t **bsl_cmd_handle_sfr_read** (BSL_PROT_t * **pProtocol**) . . . 25
4.1.1.2.10 status_t **bsl_cmd_handle_sfr_write** (const BSL_PROT_t * **pProtocol**) . . . 25
4.1.2 User Data Types . . . 26
4.1.2.1 Member summary . . . 26
4.1.2.2 Macro details . . . 26
4.1.2.2.1 **NVM_OPTIONS_NONE** . . . 26
4.1.2.2.2 **NVM_OPTIONS_RETRY_MASK** . . . 26
4.1.2.3 Enumeration details . . . 26
4.1.2.3.1 enum **cmd_id_t** . . . 26
4.1.2.3.2 enum **erase_type_t** . . . 27
4.1.2.3.3 enum **interface_op_t** . . . 27
4.1.2.3.4 enum **interface_t** . . . 27
4.1.2.3.5 enum **status_t** . . . 27
4.1.2.4 srv_1000tp_write_t struct . . . 28
4.1.2.4.1 Detailed description . . . 28
4.1.2.4.2 Variable details . . . 29
4.1.2.5 srv_1000tp_erase_t struct . . . 29
4.1.2.5.1 Detailed description . . . 29

Table of contents

4.1.2.5.2 Variable details . . . 29
4.1.2.6 srv_ifmgnt_t struct . . . 29
4.1.2.6.1 Detailed description . . . 29
4.1.2.6.2 Variable details . . . 30
4.1.3 BootROM API . . . 30
4.1.3.1 Member summary . . . 30
4.1.3.2 Function details . . . 31
4.1.3.2.1 status_t **feature_analog_trimming** (void) . . . 31
4.1.3.2.2 void **feature_core_sleep** (void) . . . 31
4.1.3.2.3 status_t **feature_device_reset** (reset_t **reset**) . . . 31
4.1.3.2.4 void **feature_endless_loop** (void) . . . 31
4.1.3.2.5 status_t **feature_interface_mgnt** (interface_t **interface**, interface_op_t **operation**) . . . 31
4.1.3.2.6 status_t **feature_nvm_cs_1000tp_erase** (uint32_t **address**) . . . 32
4.1.3.2.7 status_t **feature_nvm_cs_1000tp_write** (uint32_t **address**, const uint8_t * **data**, uint32_t **length**) . . . 32
4.1.3.2.8 status_t **feature_nvm_cs_write** (uint32_t **address**, const uint8_t * **data**, uint32_t **length**) . . . 32
4.1.3.2.9 status_t **feature_nvm_ecc_addr_get** (uint32_t * **nvm_ecc_addr_ptr**) . . . 32
4.1.3.2.10 status_t **feature_nvm_ecc_check** (uint32_t **address**, uint32_t **length**) . . . 33
4.1.3.2.11 status_t **feature_nvm_erase** (uint32_t **address**, erase_type_t **erase_type**) . . . 33
4.1.3.2.12 status_t **feature_nvm_erase_module** (void) . . . 33
4.1.3.2.13 status_t **feature_nvm_read** (uint32_t **address**, uint8_t * **data**, uint32_t **length**) . . . 34
4.1.3.2.14 status_t **feature_nvm_verify** (uint32_t **address**, uint32_t **length**, uint32_t **checksum**) . . . 34
4.1.3.2.15 status_t **feature_nvm_write** (uint32_t **address**, const uint8_t * **data**, uint32_t **length**, uint32_t **prog_flag**) . . . 34
4.1.3.2.16 status_t **feature_patch0** (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**) . . . 35
4.1.3.2.17 status_t **feature_patch1** (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**) . . . 35
4.1.3.2.18 status_t **feature_patch2** (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**) . . . 35
4.1.3.2.19 status_t **feature_ram_execute** (uint32_t **address**) . . . 35
4.1.3.2.20 status_t **feature_ram_read** (uint32_t **address**, uint8_t * **data**, uint32_t **length**) . . . 35
4.1.3.2.21 status_t **feature_ram_write** (uint32_t **address**, const uint8_t * **data**, uint32_t **length**) . . . 36
4.1.3.2.22 status_t **feature_rom_signature** (void) . . . 36
4.1.3.2.23 uint32_t **feature_sfr_read** (uint32_t **address**) . . . 36
4.1.3.2.24 void **feature_sfr_write** (uint32_t **address**, uint32_t **value**) . . . 36
4.2 Data Structures . . . 37
4.2.1 srv_1000tp_erase_t struct . . . 37
4.2.1.1 Detailed description . . . 37
4.2.1.2 Variable details . . . 37
4.2.1.2.1 uint32_t **address** . . . 37
4.2.2 srv_1000tp_write_t struct . . . 37

4.2.2.1 Detailed description 37
4.2.2.2 Variable details 38
4.2.2.2.1 uint32_t **address** 38
4.2.2.2.2 uint8_t **data**[NVM_PAGE_SIZE] 38
4.2.2.2.3 uint32_t **length** 38
4.2.3 srv_ifmgnt_t struct 38
4.2.3.1 Detailed description 38
4.2.3.2 Variable details 38
4.2.3.2.1 interface_t **interface** 38
4.2.3.2.2 interface_op_t **interface_op** 38
4.2.3.2.3 uint16_t **reserved** 38
5 List of abbreviations 39
Revision history 40
Disclaimer 41

1 Firmware architecture

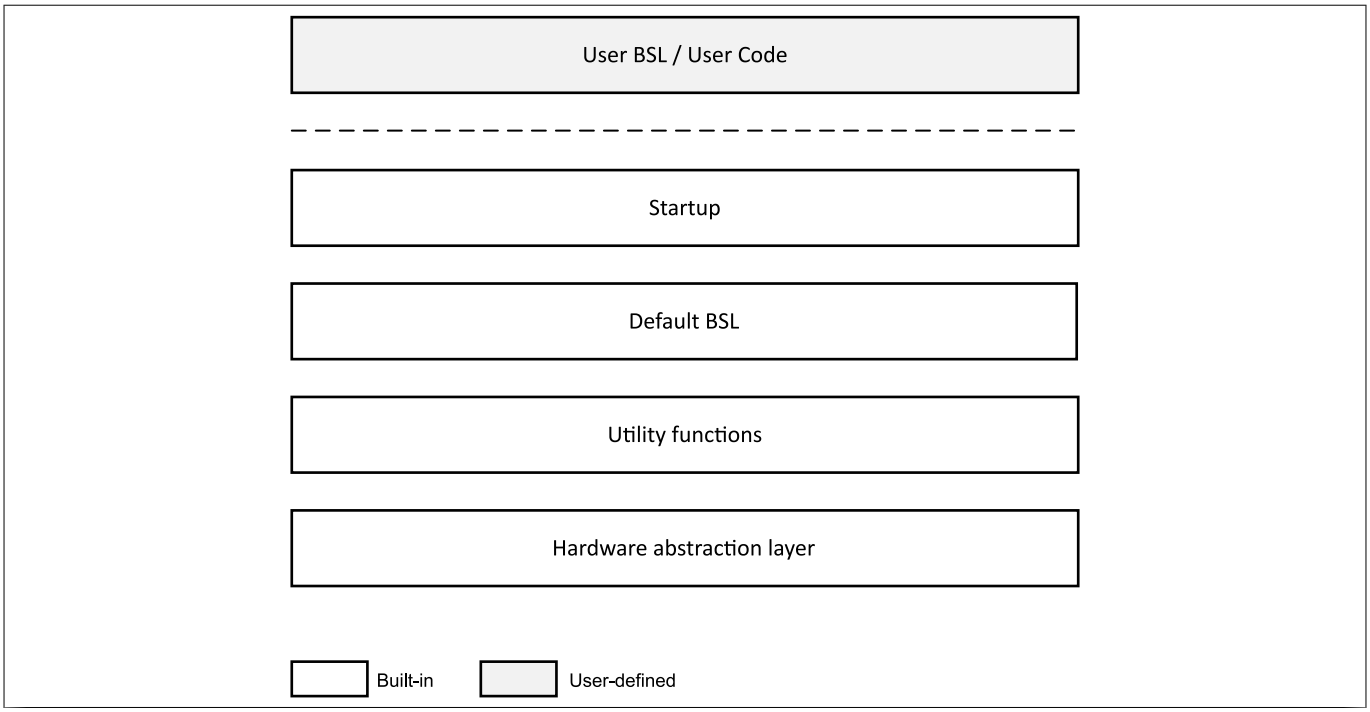

Figure 1 Firmware architecture

1.1 Startup

The Startup module includes these features:

- It executes the first software-controlled operation in the bootROM that is automatically executed after every reset
- It performs different device initialization steps and enters the operation mode determined by the provided configuration (for details, refer to Operation modes)

- Executed with the highest privilege level
- It uses the various routines of the lower abstraction levels

Vector Table

At the end of the startup sequence, the bootROM firmware hands over execution to the user code by writing VT_ADDR into the Vector Table Offset Register (VTOR) and jumping to the Reset vector entry.
The initial stack pointer value is the first entry of the vector table.
Refer to Arm® Cortex®[1]-M23 Devices Generic User Guide for information regarding the vector table.

Table 1 Startup memory map

Address	Content	Description
12000000_H	VT_ADDR	Vector Table Address
$120000C0_H$	SCP_ADDR	Startup configuration page address

Startup configuration overview

The Startup module expects the startup configuration, see table below, at address SCP_ADDR.
It contains the Default BSL configuration and Flash segment sizes information.

Table 2 Startup configuration page at SCP_ADDR

Byte offset	0	1	2	3 ... 7	8	9	10 ... 63
Item	NAC	NAD	FTO	Reserved	UBSL_SIZE	UCODE_SIZE	Reserved
Description	No-activity counter	Node address	Frame TimeOut		User BSL size	User CODE size	

No-activity counter (NAC) configuration

A no-activity counter (NAC) timeout value is stored in the startup configuration page.
In user mode, this parameter is read from the startup configuration. See Table 2.
A changed NAC value takes effect only after the next reset.

Table 3 NAC configuration values

NAC value(s)	NAC window	Behavior
0	0 ms	Timeout is immediately achieved (i. e. no waiting time)
1, …, 28	5 ms * NAC	Timeout is achieved at the end of NAC window
> 28	Endless	Timeout is never achieved (i. e. endless waiting)

Node address for diagnostics (NAD) configuration

The value of the responder node address for diagnostics (NAD) is stored in the startup configuration page. See Table 2.

Table 4 NAD address range

NAD value	Description
00_H to FE_H	Valid responder address range for valid responder addressing.

(table continues...)

[1] Arm and Cortex are registered trademarks of Arm Limited, UK

Table 4 (continued) NAD address range

NAD value	Description
FF_H	Default address if NAD value is not programmed. This NAD can be used to address any device regardless of the NAD configured in its startup page. Commands with this NAD are processed as if the actual NAD in the startup page was sent.

BSL Frame Timeout (FTO) configuration

The value of the frame timeout (FTO) is stored in the startup configuration page. See Table 2.

FTO defines a time window to receive a media frame. After completion of host synchronization, firmware starts polling for the incoming bytes, if a valid frame received before frame timeout, firmware continues parsing and handling the BSL command. If frame timeout happens, firmware clears the receive buffer, and re-starts a time window to receive a new media frame.

The frame timeout has a granularity of 5 ms. For example, frame timeout value is set to 56, which results in a timeout of 280 ms (56 * 5 ms).

User BSL size configuration

The value of the user BSL size (UBSL_SIZE) is stored in the startup configuration page. See Table 2.

User BSL size can be configured by writing the corresponding byte in the startup configuration page. Its size is expressed in sectors, that is 1 sector corresponds to 2 kB. UBSL_SIZE can assume a value in the range between 1 and 15.

User CODE sizes configuration

The value of the user CODE size (UCODE_SIZE) is stored in the startup configuration page. See Table 2.

User CODE size can be configured by writing the corresponding byte in startup configuration page. Its size is expressed in sectors, that is 1 sector corresponds to 2 kB. UCODE_SIZE shall be calculated and written as in (1), where NVM_SIZE is 16.

$$\text{UCODE\_SIZE} = \text{NVM\_SIZE} - \text{UBSL\_SIZE} - 1 \tag{1}$$

1.2 Default bootstrap loader (BSL)

The Default Bootstrap Loader (BSL) module supports uploading user application programs into the NVM using message-based command request-and-response communication over the selected communication interface.

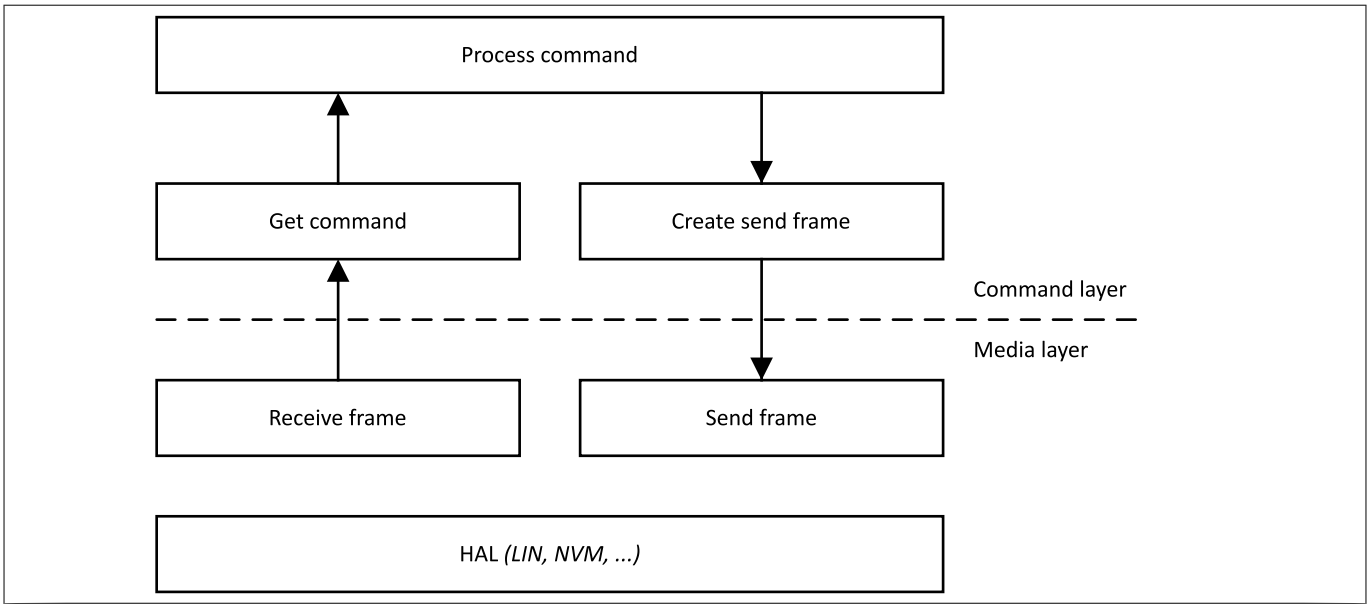

Figure 2 BSL architecture

BSL command layer

This layer is responsible for parsing and processing the command. If the command is valid, it will perform the requested operation, then prepare the reply frame, which can be either a data frame or a status frame.

BSL media layer

This layer is responsible for receiving data over the communication interface and assembling it into a complete frame. Intraframe timeout measurement is used to keep track of frame reception. Only correct frames received within the intraframe time window will be further transported to the command layer.

A media frame is used to transmit data to the device or to receive a response from the device.

1.3 User bootstrap loader (UBSL)

User can implements his own version of bootstrap loader, here called User Bootstrap Loader (UBSL), to support uploading user application programs into the NVM using proprietary user-defined protocols and flows. Firmware functions available via User Firmware API are available for execution of the required low level operation for NVM programming and other essential routines.

The UBSL is part of the user code, and so it is not a part of the bootROM code.

1.4 Utility functions

The bootROM exposes some library functions to the user software. These library functions allow configuration of the device boot parameters and access the NVM.

The main features of the utility functions are the following:

- Reading and writing the various 1000TP pages inside the NVM
- Writing and erasing the NVM pages and sectors
- Checking for single and double ECC errors in the NVM

2 Operation modes

2.1 Boot modes

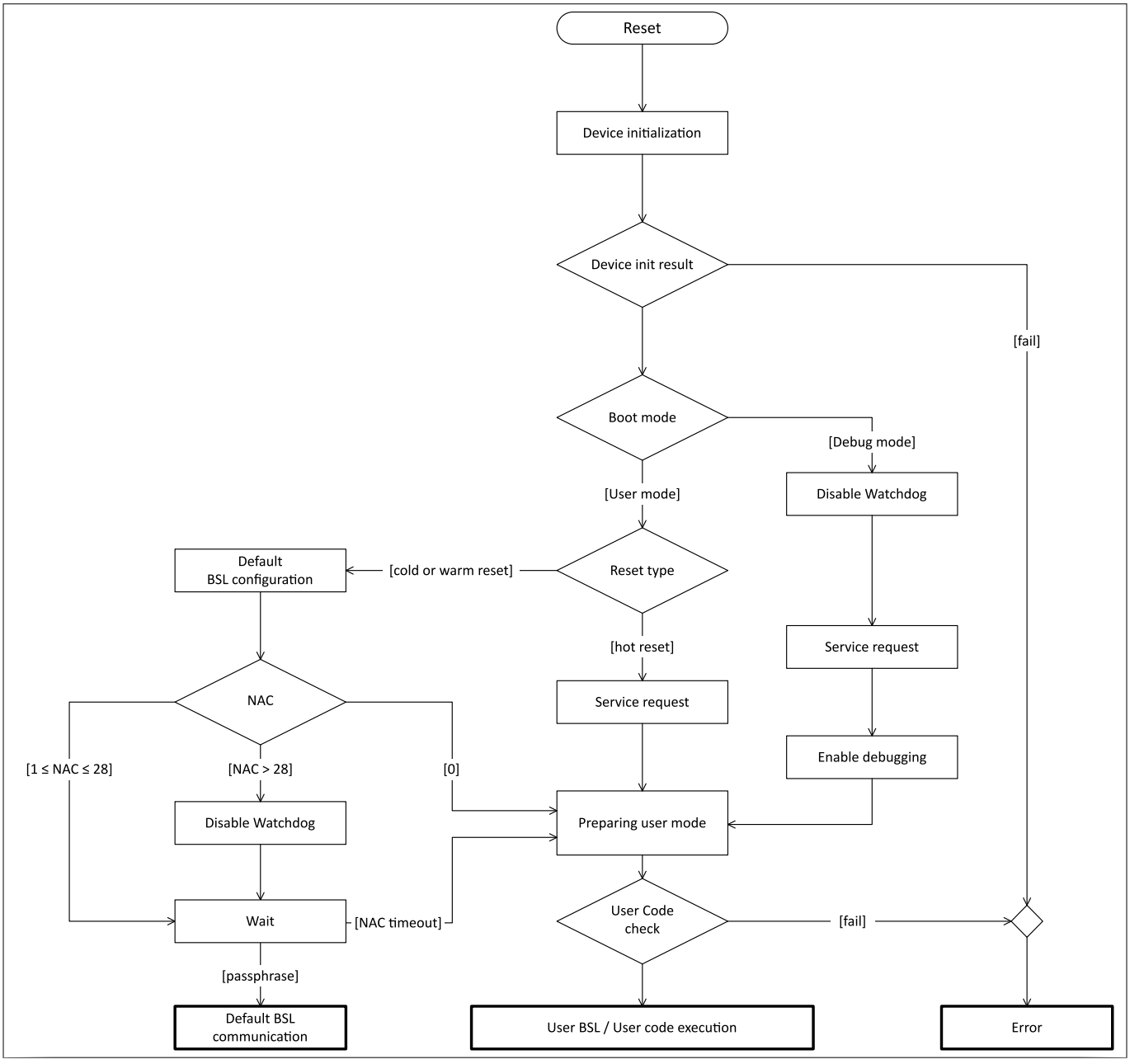

Figure 3 Boot process

2.1.1 Boot mode selection

Table 5 User mode pin configuration

SWDIO/TMS	GPIO1/SWCLK	Mode/comment
0	X	**User mode**
1	1	**User debug mode**: debug support mode with SWD interface

2.1.2 Device initialization

Table 6 Device initialization procedures

Block	Description	Reference
Watchdog disable	WDT configuration	Watchdog configuration
RAM test	RAM MBIST test	RAM test and RAM initialization
RAM initialization	RAM initialization	
Analog module trimming	Analog NVM module trimming	
Start NAC timer	Start a timer which indicates how long there has been no activity in Default BSL mode	
BSL	Default BSL communications	Default BSL mode

2.1.2.1 Watchdog configuration

After a reset, the watchdog WDT starts with a long open window. WDT continues to run while waiting for the first BSL frame. If host synchronization is completed during the BSL waiting time (defined by NAC), WDT is disabled and its status is frozen.

WDT is disabled in debug mode to prevent WDT timeout from interrupting the debugging session.

For all reset types, the WDT is enabled automatically by the hardware before jumping to user code.

2.1.2.2 RAM test and RAM initialization

Depending on the reset type, the hardware tests and initializes the RAM:

- Cold reset: by default, the hardware performs the RAM MBIST and initializes the RAM
- Warm reset: the hardware performs the RAM MBIST and initializes the RAM
- Hot reset: no RAM or initialization is performed by default

Specific user-controllable configurations can modify the default behavior.

At the beginning of the firmware startup routine execution, the firmware checks the result of the RAM test and initialization performed by the hardware.

The RAM MBIST consist of a linear write/read algorithm using alternating data on data and parity fields, which destroys the content and the parity integrity of the tested RAM. For this reason, the RAM is initialized after each test.

The RAM is initialized by writing zeros into all memory locations, with the proper parity to prevent an ECC error during user code execution.

If an error is detected during the RAM MBIST or RAM initialization, the appropriate error status is captured and the device enters an infinite loop. As the watchdog is enabled when entering the infinite error loop after booting in user or debug mode, a WDT cold reset is performed after the timeout expires and the RAM MBIST or RAM initialization is re-executed.

After five consecutive watchdog resets, the device enters sleep mode (through a hardware function).

Note: The standard RAM interface is disabled while the MBIST is executed.

Note: When a RAM MBIST or initialization error occurs, the MEM.MEMSTS register remains untouched.

2.1.3 Reset priority

If more than one reset event occurs, the post-reset initialization procedure with the highest priority type is executed. The priority is evaluated according to this priority order (where items that come first have a higher priority):

- Cold reset
- Warm reset
- Hot reset

***Attention**: The reset source is read from the PMU Reset Status Register (PMU.RESETSTS). Clearing PMU.RESETSTS is strongly recommended in the user startup code because uncleared bits can cause a wrong reset source interpretation in the bootROM firmware after the next reset (for example, handling a warm reset as a cold reset).*

2.2 Default BSL mode

In the startup process, the Default BSL mode refers to a sequence to hand over the control to the BSL communication module of the firmware code (see Figure 4).

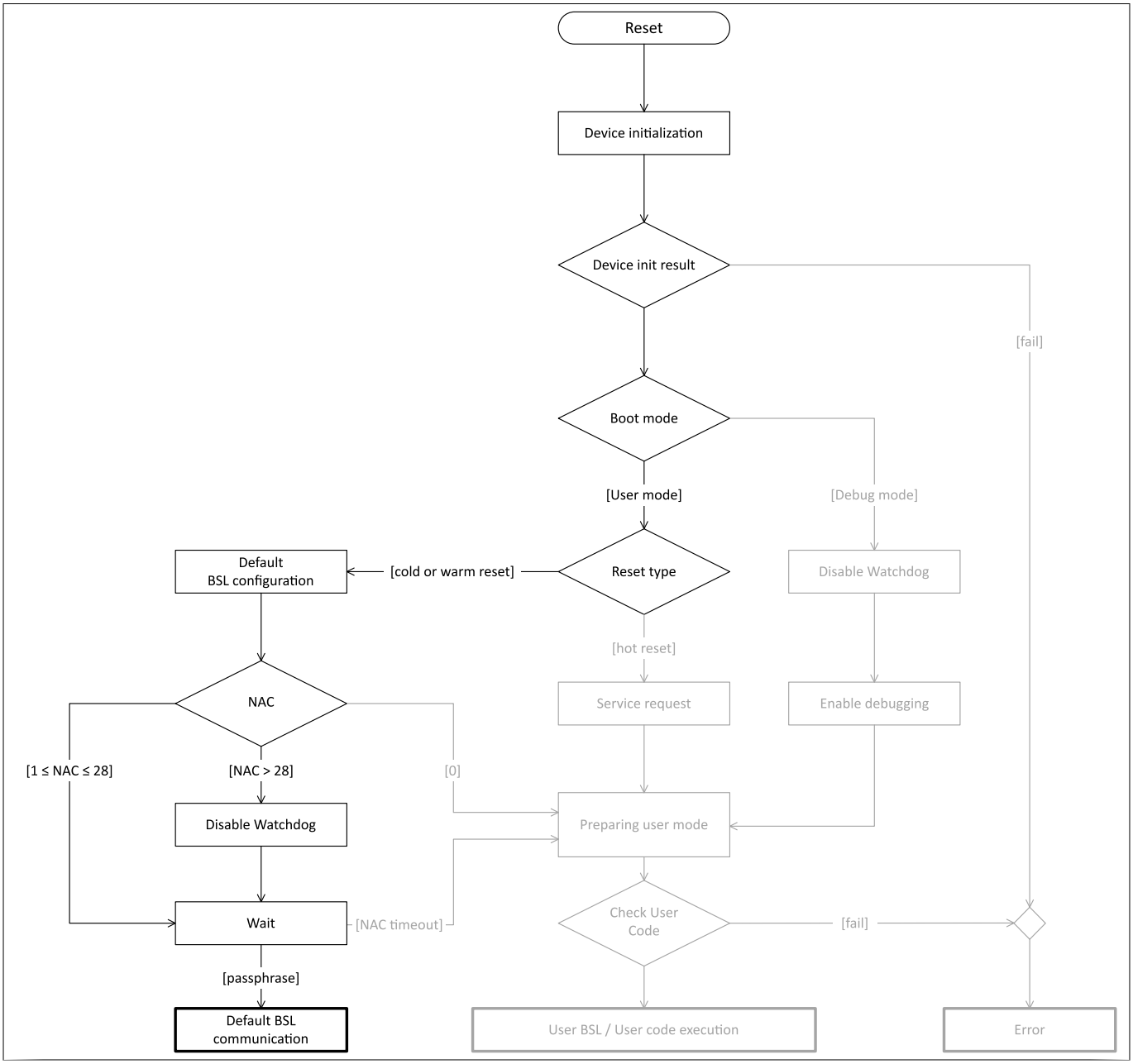

Figure 4 Booting in Default BSL mode

Default BSL communications are activated when correct NAD, passphrase, and checksum values are received in the passphrase frame within the configured NAC timeout window. For details of the passphrase frame, refer to Table 7. Otherwise, Default BSL communications remain inactive and all other received frames are ignored.

A BSL passphrase consists of a single BSL frame with the format [length] + [NAD] + [broadcast] + "PASSPHRASE" + [chk].

"PASSPHRASE" is a byte string with the byte values 50_H, 41_H, 53_H, 53_H, 50_H, 48_H, 52_H, 41_H, 53_H and 45_H.

Table 7 Passphrase frame format

0	1	2	3	4	5	6	7	8	9	10	11	12	13
Length	NAD	broad cast	50_H "P"	41_H "A"	53_H "S"	53_H "S"	50_H "P"	48_H "H"	52_H "R"	41_H "A"	53_H "S"	45_H "E"	Chk

The NAD address is set as a BSL parameter when the device is programmed. The NAD value of FF_H indicates an anycast message, and the Default BSL responds to this message regardless of the value stored in the startup configuration page (for details see Table 2 in Startup)

If no valid startup configuration is installed then the device enters Default BSL and waits indefinitely for a valid passphrase.

Upon successful reception of a valid passphrase frame, the device sends back single acknowledge byte 55_H and is ready for receiving BSL commands. If broadcast mode is enabled, the acknowledge byte is not sent back.

2.2.1 NAD address

The NAD field specifies the address of the active responder node (only responder nodes have NAD addresses). Table 4 lists NAD address ranges supported by the bootROM.

2.2.2 NAC time window

The No-Activity Counter (NAC) value defines the time window, with a granularity of 5 ms, after reset release, within which the firmware is able to receive a BSL passphrase. If no BSL passphrase is received before the NAC timeout expires, the firmware code proceeds to executing user code.

In case of a read error of the NAC value in the NAC location, the NAC value is set to 0 by firmware skipping the Default BSL module.

The maximum NAC timeout is 140 ms, corresponding to NAC = $1C_H$. Firmware reads the NAC from non-volatile memory and sets the NAC time window. The translation from NAC value to NAC time window is explained in Table 3.

2.2.3 Default BSL frame format

The frame uses the general [length] [message] [chk] format, regardless of the communication interface.

Media frames are used to send data to the device or to receive a response from the device.

- [length] denotes the number of successive bytes in the frame
- [message] has the format [message type] [arguments]. The arguments depend on the specific message type. Messages sent to the device are called "command frame" and messages sent by the device are called "response frame".
  - [message type] is a CMD_ID (BSL command number) or a RESP_ID (BSL response number)
  - [arguments] are optional with a maximum length of 69 bytes containing the arguments required for a specific BSL command or BSL response
- [chk] is the frame checksum and it is calculated over the length byte and the message bytes

Checksum calculation over the data bytes only is called classic checksum. The checksum contains the inverted eight bit sum with carry over all data bytes. Eight bit sum with carry is equivalent to sum all values and subtract 255 every time the sum is greater or equal to 256.

2.2.4 Timing constraints

The host must add a delay after each sent Default BSL command header and EOT message before sending the next one.

The bootROM also requires an additional waiting time to process the complete received Default BSL command. During this period of time, no response message can be provided by the bootROM and therefore the host cannot send new commands. The host must wait this length of time before sending a new command.

To give the bootROM time to process each byte in a CMD or EOT frame, the byte and frame timing must comply with the values shown in the table below.

Table 8 Default BSL byte and frame timing limits and highest transfer rates

Delay type	Minimum interval duration [µs]
Between bytes	3.7
Host waiting time after reception of a response before a new frame can be sent	20

Certain Default BSL commands involving NVM write or erase operations need longer processing times. The host waiting time is longer before a command response can be requested or before a result is sent back.

Changing a value in an already programmed NVM page (which happens if a setting is being changed) requires the following NVM steps:

- Read the full page into the hardware assembly buffer
- Update the hardware buffer with new data
- Program the page from the hardware assembly buffer
- Erase the old page

Total time: 8 ms

The processing time must always be taken into account.

2.2.5 Default BSL interval between frames timeout behavior

To keep track of Default BSL frame transmission violations, interval between frames timeouts are used. This section summarizes the different use cases where Default BSL frame timeouts are applied.

Default BSL frame transmission timeouts are handled differently depending on the circumstances:

- If the Default BSL has not yet received any valid host synchronization information, the NAC timeout value is used for all timeout calculations. If a timeout is reached, this means that the NAC timer has expired
- After the host synchronization is completed, the firmware starts polling the incoming bytes. If a valid frame is received before the frame timeout, the firmware continues parsing and handling the BSL command

  When a frame timeout occurs, the firmware clears the receive buffer and re-starts a new time window for receiving a new media frame.

2.2.6 Default BSL host synchronization

The host synchronization is completed when the full passphrase has been received before the NAC timeout expires.

2.3 User BSL mode

The User BSL mode (UBSL) of the startup process refers to program sequence leading to handing over the control to the UBSL. User BSL is a user code that can optionally be installed and implemented in order to support, for example the preparation for a user application code execution or to perform a user software update.

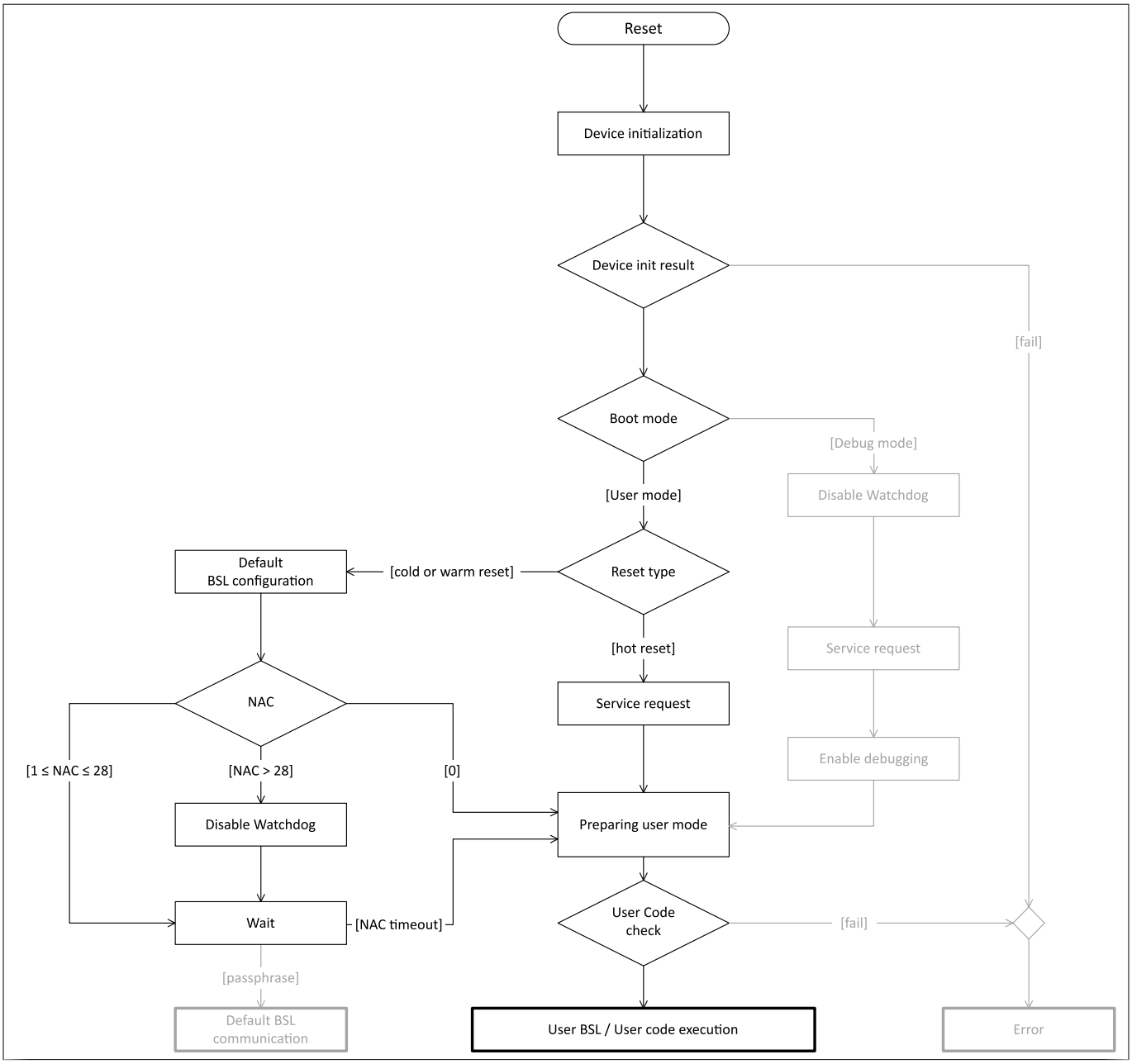

Figure 5 Booting in User BSL mode

2.4 User mode

In the startup process, the user mode refers to a sequence to hand over the control to the user application code (see Figure 6). To select this mode, use the pin configuration shown in Chapter 2.1.1.

At the end of the startup sequence, the bootROM firmware hands over execution to the user code by writing VT_ADDR into the Vector Table Offset Register (VTOR) and jumping to the Reset vector entry.

The initial stack pointer value is the first entry of the vector table.

Refer to Arm® Cortex®[2]-M23 Devices Generic User Guide for information regarding the vector table.

2 Arm and Cortex are registered trademarks of Arm Limited, UK

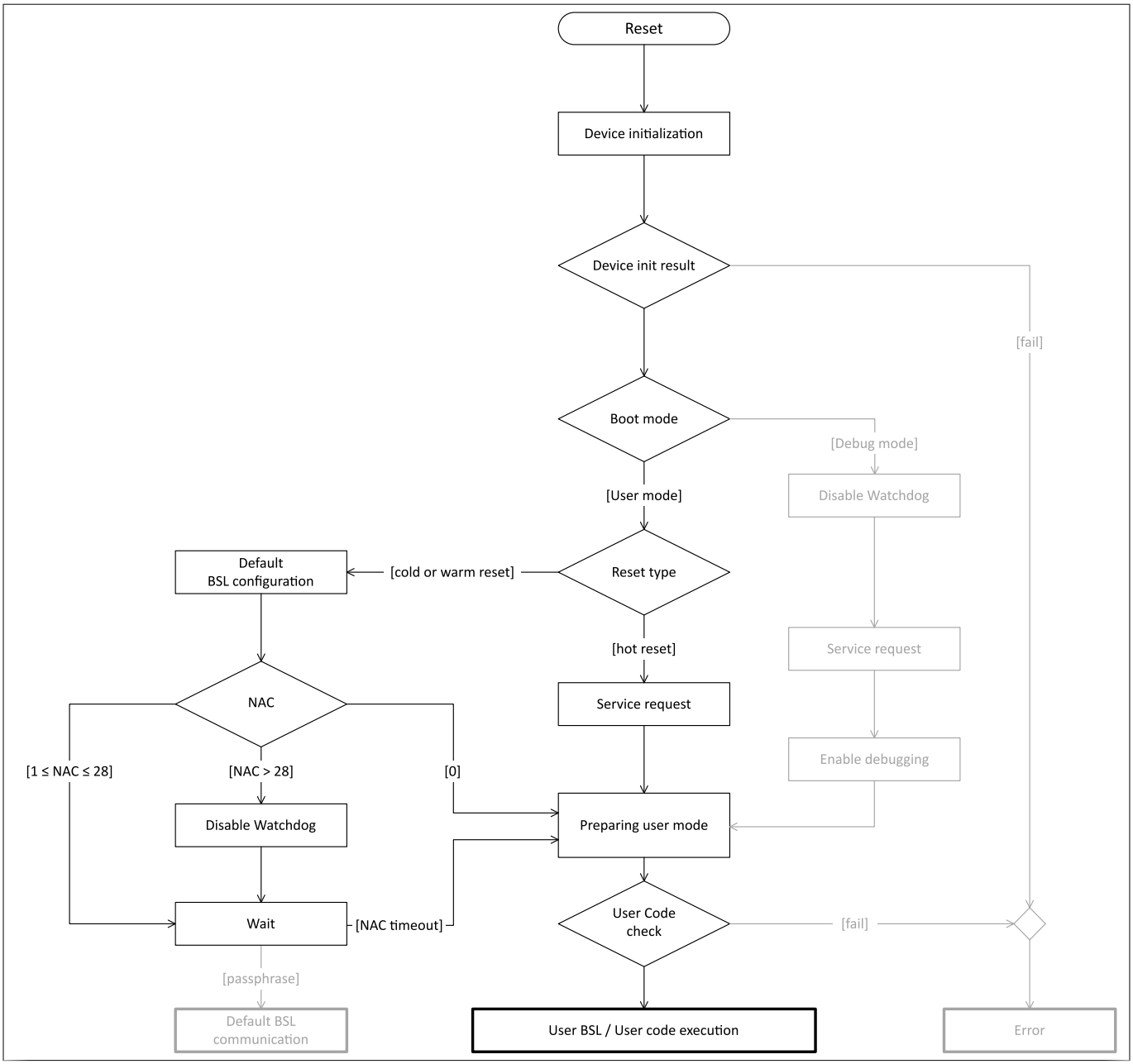

Figure 6 Booting in user mode

2.5 User debug mode

The user debug mode is an extension of the user mode and allows the user to debug the user code using SWD interface. Booting into user debug mode is similar to booting into user mode, but it excludes the option to enter the BSL mode (see Chapter 2.3).

The user can select the user debug mode using the pin configuration as shown in Table 5.

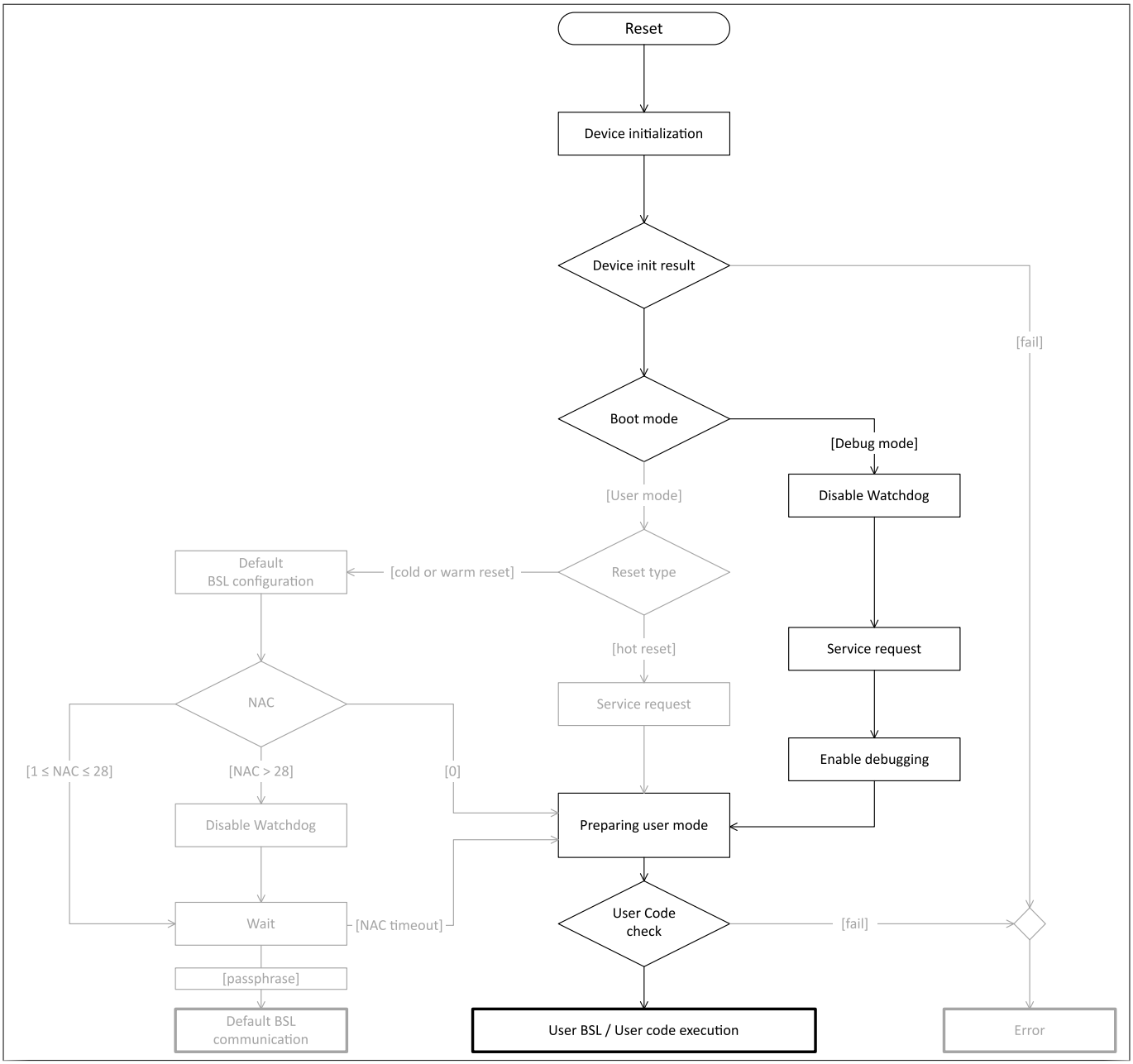

Figure 7 Booting in user debug mode

2.6 Error mode

To ensure that the device is properly booted, error checking and error handling are added to the startup procedure.

The system enters an endless loop when an invalid user code address error (stack pointer or reset handler addresses not written by the user code), RAM MBIST error, or NVM CS pages checksum error occurs, regardless of the boot mode.

If a startup error occurs the device is put into a fail-safe mode with limited access to hardware resources: the device reboots after the WDT timeout has expired and if the errors persist after five WDT-triggered timeouts, the device enters fail-sleep power mode.

If the boot mode is the User debug mode and User code check fails, the device remains in an endless loop state without entering the fails-sleep power mode since the WDT is disabled. This allows the debugger connection to check the device state.

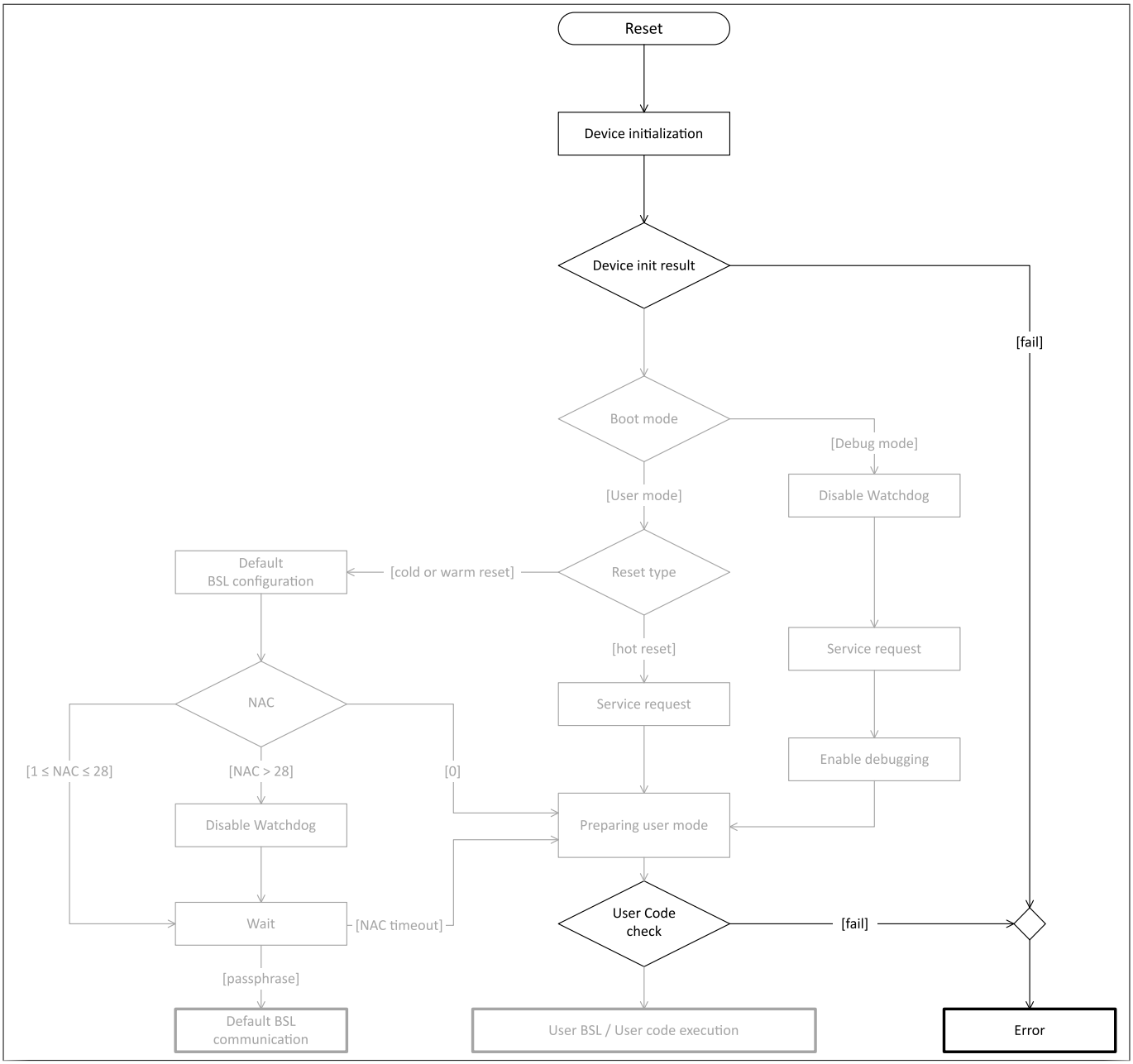

Figure 8 Booting in error mode

3 Programming model

3.1 Debug interface

The debug support mode is available for the serial wire debug (SWD) interface. The bootROM starts the overall device initialization, as described in Chapter 2.5.

The debug interface supports the following features:

- Regular Arm® Cortex®-M23 debug features
- Firmware API calls supported by the utility functions

The watchdog is always disabled in debug support mode, except when the debug error loop is entered after a boot error.

The debug interface can be disabled via BSL command and/or via service request, in order to prevent debug access to the device. It is possible to restore debug access using a service request, in which case the content of the user code and user data segments are deleted.

4 API documentation

4.1 Modules

4.1.1 BSL Commands

BSL commands support flash access and other low-level operations.

This chapter intends to describe the format and functionalities of BSL commands. The functions are only callable by BSL protocol.

Note: A valid BSL passphrase frame to required to start BSL communication:

```
+--------+-----+-----------+-----+-----+-----+-----+-----+
| Length | NAD | broadcast | 'P' | 'A' | 'S' | 'S' | 'P' |  ...
+--------+-----+-----------+-----+-----+-----+-----+-----+
      +-----+-----+-----+-----+-----+----------+
 ...  | 'H' | 'R' | 'A' | 'S' | 'E' | checksum |
      +-----+-----+-----+-----+-----+----------+
```

Remarks

The passphrase frame must be received by target device within NAC time window.

Parameters

length	Number of bytes, from *MAD* to *checksum*, equal to 0xC
NAD	Node address • 0x0-0xFE: Only the matched node in network is selected for BSL communication. • 0xFF: All the nodes in network are selected for BSL communication.
broadcast	Configuration of broadcast • #BSL_CMD_BRAODCAST_MASK: enables broadcast mode, which suppress acknowledge byte. • other value: disables broadcast mode
checksum	The LIN classic checksum over all the preceding bytes.

Returns

Upon successful BSL connection, the target acknowledges with byte 0x55 or none. User can then send functional command to the target.

Note: A data response frame is responded by target if the host requested data from target successfully. e.g. it applies to command: memory read, SFR read

```
+--------+--------+----+----+-----+----+----------+
| length | cmd_id | D0 | D1 | ... | Dn | checksum |
+--------+--------+----+----+-----+----+----------+
```

Parameters

length	Number of bytes, from *cmd_id* to *checksum*

Parameters	
cmd_id	Data response identifier, equal to CMD_EOT
Dn	The nth data that read back, n =[0, 63], D0 represents least significant byte.
checksum	The LIN classic checksum over all the preceding bytes.

Note: A status response frame is responded by target indicating the operation status. exception: When the target is required to send data response frame, no status frame will be sent.

```
+--------+--------+-----+--------+----------+
| length | cmd_id | res | status | checksum |
+--------+--------+-----+--------+----------+
```

Parameters	
length	Number of bytes, from *cmd_id* to *checksum*, equal to 0x4
cmd_id	Status response identifier, equal to CMD_RESPONSE
res	Reserved byte, equal to 0x0
status	Status of the operation, 1-byte, refer to status_t
checksum	The LIN classic checksum over all the preceding bytes.

4.1.1.1 Member summary

Functions

status_t bsl_cmd_handle_baudrate_set (BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_dev_reset (const BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_interface_mgnt (const BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_mem_erase (BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_mem_exec (const BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_mem_read (BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_mem_verify (const BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_mem_write (BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_sfr_read (BSL_PROT_t * **pProtocol**)

status_t bsl_cmd_handle_sfr_write (const BSL_PROT_t * **pProtocol**)

4.1.1.2 Function details

4.1.1.2.1 status_t bsl_cmd_handle_baudrate_set (BSL_PROT_t * pProtocol)

BSL command to configure the baudrate of current BSL session.
Command format:

```
+--------+--------+-----------+-----+----------+----------+
| length | cmd_id | broadcast | res | baudrate | checksum |
+--------+--------+-----------+-----+----------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Remarks

The new baudrate takes effective after this response frame.

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_BAUDRATE_SET, refer to cmd_id_t
broadcast	Configuration of broadcast • #BSL_CMD_BRAODCAST_MASK: enables broadcast mode, which suppress response. • other value: disables broadcast mode
res	Reserved byte, don't care
baudrate	The desired baudrate, 4-byte, little-endian
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or none.

4.1.1.2.2 status_t bsl_cmd_handle_dev_reset (const BSL_PROT_t * pProtocol)

BSL command to trigger device reset.
Command format:

```
+--------+--------+-----+-------+----------+
| length | cmd_id | res | reset | checksum |
+--------+--------+-----+-------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*

Parameters	
cmd_id	CMD_DEVICE_RESET, refer to cmd_id_t
res	Reserved byte, don't care
reset	Select reset type, refer to reset_t • #RESET_WARM • #RESET_HOT
checksum	The LIN classic checksum over all the preceding bytes.

Returns

no response

4.1.1.2.3 status_t bsl_cmd_handle_interface_mgnt (const BSL_PROT_t * pProtocol)

BSL command to manage device's interfaces.
It allows disabling/re-enable of selected interface.
Command format:

```
+--------+--------+-----------+-----------+----------+
| length | cmd_id | interface | operation | checksum |
+--------+--------+-----------+-----------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters	
length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_IF_MGNT, refer to cmd_id_t
interface	Select interface for operation. Refer to interface_t • SWD • BSL
operation	The intended operation. Refer to interface_op_t • IF_DEACTIVATE • IF_ACTIVATE
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame.

4.1.1.2.4 status_t bsl_cmd_handle_mem_erase (BSL_PROT_t * pProtocol)

BSL command to erase memory, NVM only.
Command format:

```
+--------+--------+------------+-----------+---------+----------+
| length | cmd_id | erase_type | broadcast | address | checksum |
+--------+--------+------------+-----------+---------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_MEM_ERASE, refer to cmd_id_t
erase_type	Scope of erase operation. It supports mass erase as well. Refer to erase_type_t
broadcast	• #BSL_CMD_BRAODCAST_MASK: enables broadcast mode, which suppress response. • other value: disables broadcast mode
address	The address of memory, 4-byte, little-endian, e.g. 0x12000123 shall be represented by |0x23|0x01|0x00|0x12|
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or none.

4.1.1.2.5 status_t bsl_cmd_handle_mem_exec (const BSL_PROT_t * pProtocol)

BSL command to execute code from memory, RAM only.
Command format:

```
+--------+--------+------+------+---------+----------+
| length | cmd_id | res0 | res1 | address | checksum |
+--------+--------+------+------+---------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_MEM_EXECUTE, refer to cmd_id_t
res0	Reserved byte, don't care
res1	Reserved byte, don't care

Parameters	
address	The address of memory, 4-byte, little-endian, e.g. 0x12000123 shall be represented by |0x23|0x01|0x00|0x12|
checksum	The LIN classic checksum over all the preceding bytes.

Returns

no response

4.1.1.2.6 status_t bsl_cmd_handle_mem_read (BSL_PROT_t * pProtocol)

BSL command to read from memory, e.g.
RAM, NVM
Command format:

```
+--------+--------+-----+-----+---------+----------+
| length | cmd_id | len | res | address | checksum |
+--------+--------+-----+-----+---------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters	
length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_MEM_READ, refer to cmd_id_t
len	Number of bytes to be read
res	Reserved byte, don't care
address	The address of memory to read from, 4-byte, little-endian, e.g. 0x12000123 shall be represented by |0x23|0x01|0x00|0x12|
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or data frame.

4.1.1.2.7 status_t bsl_cmd_handle_mem_verify (const BSL_PROT_t * pProtocol)

BSL command to verify flash data integrity.
It compares the calculated checksum against given checksum.
Command format:

```
+--------+--------+------+------+---------+-----+-----+----------+
| length | cmd_id | res0 | res1 | address | len | chk | checksum |
+--------+--------+------+------+---------+-----+-----+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_MEM_VERIFY, refer to cmd_id_t
res0	Reserved byte, don't care
res1	Reserved byte, don't care
address	The address of memory, 4-byte, little-endian, e.g. 0x12000123 shall be represented by |0x23|0x01|0x00|0x12|
len	Number of bytes to be verified
chk	The XOR checksum used to verify the flash data integrity, 4-byte, little-endian
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame.

4.1.1.2.8 status_t bsl_cmd_handle_mem_write (BSL_PROT_t * pProtocol)

BSL command to write memory, e.g.
RAM, NVM
Command format:

```
+--------+--------+-----------+-----+---------+----+----+-----+----+----------+
| length | cmd_id | broadcast | res | address | D0 | D1 | ... | Dn | checksum |
+--------+--------+-----------+-----+---------+----+----+-----+----+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_MEM_WRITE, refer to cmd_id_t
broadcast	Configuration of broadcast • #BSL_CMD_BRAODCAST_MASK: enables broadcast mode, which suppress response. • other value: disables broadcast mode
res	Reserved byte, don't care
address	The address of memory to write to, 4-byte, little-endian, e.g. 0x12000123 shall be represented by |0x23|0x01|0x00|0x12|
Dn	The nth data to be written, n =[0, 63]
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or none.

4.1.1.2.9 status_t bsl_cmd_handle_sfr_read (BSL_PROT_t * pProtocol)

BSL command to read value of SFR.
Command format:

```
+--------+--------+------+------+---------+----------+
| length | cmd_id | res0 | res1 | address | checksum |
+--------+--------+------+------+---------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_SFR_READ, refer to cmd_id_t
res0	Reserved byte, don't care
res1	Reserved byte, don't care
address	The address of SFR, 4-byte, little-endian
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or none.

4.1.1.2.10 status_t bsl_cmd_handle_sfr_write (const BSL_PROT_t * pProtocol)

BSL command to write value to SFR.
Command format:

```
+--------+--------+------+------+---------+-------+----------+
| length | cmd_id | res0 | res1 | address | value | checksum |
+--------+--------+------+------+---------+-------+----------+
```

Remarks

The serial frame shall be sent to target starting from *length* byte

Parameters

length	Number of bytes, from *cmd_id* to *checksum*
cmd_id	CMD_SFR_WRITE, refer to cmd_id_t
res0	Reserved byte, don't care
res1	Reserved byte, don't care
address	The address of SFR, 4-byte, little-endian
value	The value to be write, 4-byte, little-endian
checksum	The LIN classic checksum over all the preceding bytes.

Returns

It sends back response frame or none.

4.1.2 User Data Types

The user data types is supposed to be included in user application code.

4.1.2.1 Member summary

Data structures

srv_1000tp_write_t

srv_1000tp_erase_t

srv_ifmgnt_t

Macros

NVM_OPTIONS_NONE (0x00u)

NVM_OPTIONS_RETRY_MASK (0x01u)

Enumerations

enum cmd_id_t { CMD_EOT, CMD_RESPONSE, CMD_MEM_WRITE, CMD_MEM_EXECUTE, CMD_MEM_READ, CMD_MEM_ERASE, CMD_IF_MGNT, CMD_MEM_VERIFY, CMD_DEVICE_RESET, CMD_BAUDRATE_SET, CMD_SFR_WRITE, CMD_SFR_READ }

enum erase_type_t { NVM_ERASE_PAGE, NVM_ERASE_SECTOR, NVM_ERASE_MODULE }

enum interface_op_t { IF_DEACTIVATE, IF_ACTIVATE }

enum interface_t { SWD, BSL }

enum status_t { GENERIC_E_OK, GENERIC_E_NOT_OK, GENERIC_E_BUSY, GENERIC_E_CONFIG_FAILED, GENERIC_E_PROTECTED, GENERIC_E_PARAM_INVALID, NVM_E_INTEGRITY_FAILED, NVM_E_INTEGRITY_FAILED_ECC2, NVM_E_INTEGRITY_FAILED_ECC1, NVM_E_HARDWARE, NVM_E_VIRGIN_PRODUCT, NVM_E_VIRGIN_DEVICE, BSL_E_UNSUPPORTED, BSL_E_INTEGRITY_FAILED, SRV_E_EXEC_FAILED, TRIM_E_MAGIC, TRIM_E_ADDR }

4.1.2.2 Macro details

4.1.2.2.1 NVM_OPTIONS_NONE

4.1.2.2.2 NVM_OPTIONS_RETRY_MASK

4.1.2.3 Enumeration details

4.1.2.3.1 enum cmd_id_t

Enumerator

CMD_EOT = 0x80u	command ID of data frame
CMD_RESPONSE = 0x81u	command ID of status response

Enumerator	
CMD_MEM_WRITE = 0x05u	command ID of memory write
CMD_MEM_EXECUTE = 0x06u	command ID of memory execute
CMD_MEM_READ = 0x07u	command ID of memory read
CMD_MEM_ERASE = 0x08u	command ID of memory erase
CMD_IF_MGNT = 0x09u	command ID of interface management
CMD_MEM_VERIFY = 0x0Cu	command ID of memory verify
CMD_DEVICE_RESET = 0x12u	command ID of device reset
CMD_BAUDRATE_SET = 0x13u	command ID of baudrate set
CMD_SFR_WRITE = 0x7Cu	command ID of SFR write
CMD_SFR_READ = 0x7Bu	command ID of SFR read

4.1.2.3.2 enum erase_type_t

the erase operation scopes.

Enumerator	
NVM_ERASE_PAGE = 0x00u	page Erase
NVM_ERASE_SECTOR = 0x01u	sector Erase
NVM_ERASE_MODULE = 0x02u	module Erase

4.1.2.3.3 enum interface_op_t

The operations of interface management, either activate or deactivate the selected interface.

Enumerator	
IF_DEACTIVATE = 0x55u	deactivate SWD access
IF_ACTIVATE = 0xFFu	activate SWD access

4.1.2.3.4 enum interface_t

interface selection, used by interface management operation.

Enumerator	
SWD = 0x01u	select target interface SWD
BSL = 0x10u	select target interface BSL

4.1.2.3.5 enum status_t

the error code of device operations.

Enumerator	
GENERIC_E_OK = 0x00u	SUCCESS.

Enumerator	
GENERIC_E_NOT_OK = 0x01u	the requested operation failed
GENERIC_E_BUSY = 0x02u	the requested operation failed, because device is busy with other operation
GENERIC_E_CONFIG_FAILED = 0x03u	internal error code of device initialization
GENERIC_E_PROTECTED = 0x04u	the requested operation failed, due to protection
GENERIC_E_PARAM_INVALID = 0x05u	parameter validation failed
NVM_E_INTEGRITY_FAILED = 0x06u	NVM integrity failed.
NVM_E_INTEGRITY_FAILED_ECC2 = 0x07u	NVM integrity failed, due to ECC1 error.
NVM_E_INTEGRITY_FAILED_ECC1 = 0x08u	NVM integrity failed, due to ECC2 error.
NVM_E_HARDWARE = 0x09u	NVM hardware failure.
NVM_E_VIRGIN_PRODUCT = 0x0Au	internal error code of NVM initialization, product is virgin
NVM_E_VIRGIN_DEVICE = 0x0Bu	internal error code of NVM initialization, device is virgin
BSL_E_UNSUPPORTED = 0x0Cu	BSL operation not supported.
BSL_E_INTEGRITY_FAILED = 0x0Du	BSL frame integrity failed.
SRV_E_EXEC_FAILED = 0x0Eu	service request execution failed
TRIM_E_MAGIC = 0x0Fu	internal error code of analog trimming
TRIM_E_ADDR = 0x10u	internal error code of analog trimming

4.1.2.4 srv_1000tp_write_t struct

data structure of service request operation: 1000tp write.

Read more...

```
#include <user_types.h>
```

Public attributes

uint32_t address
uint8_t data[NVM_PAGE_SIZE]
uint32_t length

4.1.2.4.1 Detailed description

data structure of service request operation: 1000tp write.

4.1.2.4.2 Variable details

uint32_t address

uint8_t data[NVM_PAGE_SIZE]

uint32_t length

4.1.2.5 srv_1000tp_erase_t struct

data structure of service request operation: 1000tp erase.
Read more...

```
#include <user_types.h>
```

Public attributes

uint32_t address

4.1.2.5.1 Detailed description

data structure of service request operation: 1000tp erase.

4.1.2.5.2 Variable details

uint32_t address

4.1.2.6 srv_ifmgnt_t struct

data structure of service request operation: interface management.
Read more...

```
#include <user_types.h>
```

Public attributes

interface_t interface
interface_op_t interface_op
uint16_t reserved

4.1.2.6.1 Detailed description

data structure of service request operation: interface management.

4.1.2.6.2 Variable details

interface_t interface

interface_op_t interface_op

uint16_t reserved

4.1.3 BootROM API

BootROM APIs support flash access and other low-level operations.
These routines are exposed by the BootROM to the customer user mode software.

4.1.3.1 Member summary

Functions

status_t feature_analog_trimming (void)
void feature_core_sleep (void)
status_t feature_device_reset (reset_t **reset**)
void feature_endless_loop (void)
status_t feature_interface_mgnt (interface_t **interface**, interface_op_t **operation**)
status_t feature_nvm_cs_1000tp_erase (uint32_t **address**)
status_t feature_nvm_cs_1000tp_write (uint32_t **address**, const uint8_t * **data**, uint32_t **length**)
status_t feature_nvm_cs_write (uint32_t **address**, const uint8_t * **data**, uint32_t **length**)
status_t feature_nvm_ecc_addr_get (uint32_t * **nvm_ecc_addr_ptr**)
status_t feature_nvm_ecc_check (uint32_t **address**, uint32_t **length**)
status_t feature_nvm_erase (uint32_t **address**, erase_type_t **erase_type**)
status_t feature_nvm_erase_module (void)
status_t feature_nvm_read (uint32_t **address**, uint8_t * **data**, uint32_t **length**)
status_t feature_nvm_verify (uint32_t **address**, uint32_t **length**, uint32_t **checksum**)
status_t feature_nvm_write (uint32_t **address**, const uint8_t * **data**, uint32_t **length**, uint32_t **prog_flag**)
status_t feature_patch0 (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**)
status_t feature_patch1 (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**)
status_t feature_patch2 (uint32_t **p1**, uint32_t **p2**, uint32_t **p3**)
status_t feature_ram_execute (uint32_t **address**)
status_t feature_ram_read (uint32_t **address**, uint8_t * **data**, uint32_t **length**)
status_t feature_ram_write (uint32_t **address**, const uint8_t * **data**, uint32_t **length**)
status_t feature_rom_signature (void)
uint32_t feature_sfr_read (uint32_t **address**)
void feature_sfr_write (uint32_t **address**, uint32_t **value**)

4.1.3.2 Function details

4.1.3.2.1 status_t feature_analog_trimming (void)

This function performs analog trimming, which copies trimming data from predefined config sector to trimming registers.

Returns

Function execution status.

4.1.3.2.2 void feature_core_sleep (void)

This function executes WFE instruction to put processor to sleep state.

Returns

none.

4.1.3.2.3 status_t feature_device_reset (reset_t reset)

This function executes image in RAM.

Parameters

reset	Reset type selector. • #RESET_WARM • #RESET_HOT

Returns

Function execution status.

4.1.3.2.4 void feature_endless_loop (void)

This function traps program counter in an endless loop.

Returns

none.

4.1.3.2.5 status_t feature_interface_mgnt (interface_t interface, interface_op_t operation)

This function allows disabling/re-enable of selected interface.

Parameters

interface	Select interface for operation. Refer to interface_t • SWD • BSL
operation	The intended operation. Refer to interface_op_t • IF_DEACTIVATE • IF_ACTIVATE

Returns

Function execution status.

4.1.3.2.6 status_t feature_nvm_cs_1000tp_erase (uint32_t address)

This function erases target flash(1000TP) page.

Parameters

address	Address of the flash memory.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.7 status_t feature_nvm_cs_1000tp_write (uint32_t address, const uint8_t * data, uint32_t length)

This function writes data from the source to the specified flash(1000TP) address *address*.

Parameters

address	Address of the flash memory.
data	Pointer to source data.
length	Number of bytes to write.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.8 status_t feature_nvm_cs_write (uint32_t address, const uint8_t * data, uint32_t length)

This function writes data from the source to the specified flash(CS) address *address*.

Parameters

address	Address of the flash memory.
data	Pointer to source data.
length	Number of bytes to write.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.9 status_t feature_nvm_ecc_addr_get (uint32_t * nvm_ecc_addr_ptr)

This function gets address of last ECC2 event.

Parameters

nvm_ecc_addr_ptr	Pointer to retrieved ECC address.

Returns

Function execution status.

4.1.3.2.10 status_t feature_nvm_ecc_check (uint32_t address, uint32_t length)

This function reads flash memory and reports ECC error if there is.

Parameters

address	Address of the flash memory.
length	Number of bytes to read.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.11 status_t feature_nvm_erase (uint32_t address, erase_type_t erase_type)

This function erases flash in page-wise or in sector-wise.

Parameters

address	Address of the flash memory.
erase_type	Scope of erase operation. Refer to erase_type_t. • NVM_ERASE_PAGE Erase the selected page. • NVM_ERASE_SECTOR Erase the selected sector. • NVM_ERASE_MODULE Not supported in API.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.12 status_t feature_nvm_erase_module (void)

This function erases flash module.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.13 status_t feature_nvm_read (uint32_t address, uint8_t * data, uint32_t length)

This function reads flash memory content to the data buffer in RAM.

Parameters

address	Address of the flash memory.
data	Pointer to destination data buffer.
length	Number of bytes to read.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.14 status_t feature_nvm_verify (uint32_t address, uint32_t length, uint32_t checksum)

This function verifies data in flash memory by comparing calculated checksum against the given checksum.

Parameters

address	Address of the flash memory.
length	Number of bytes to be verified.
checksum	Classic XOR 32-bit checksum.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.15 status_t feature_nvm_write (uint32_t address, const uint8_t * data, uint32_t length, uint32_t prog_flag)

This function writes data from the source to the specified flash address *address*.

Parameters

address	Address of the flash memory.
data	Pointer to source data.
length	Number of bytes to write.
prog_flag	Optional program flag, set #NVM_OPTIONS_RETRY_MASK to enable retry operation, otherwise, set to #NVM_OPTIONS_NONE.

Supported *prog_flag:*

- #NVM_OPTIONS_NONE The default setting: In case of write failure, no retry.
- #NVM_OPTIONS_RETRY_MASK Enables retry write operation if the first write operation verification failed.

Remarks

In an interrupt or multi-threaded environment, this function cannot be called in a re-entrant context.

Returns

Function execution status.

4.1.3.2.16 status_t feature_patch0 (uint32_t p1, uint32_t p2, uint32_t p3)

This function executes potential patch code, if it is enabled.

Returns

Function execution status.

4.1.3.2.17 status_t feature_patch1 (uint32_t p1, uint32_t p2, uint32_t p3)

This function executes potential patch code, if it is enabled.

Returns

Function execution status.

4.1.3.2.18 status_t feature_patch2 (uint32_t p1, uint32_t p2, uint32_t p3)

This function executes potential patch code, if it is enabled.

Returns

Function execution status.

4.1.3.2.19 status_t feature_ram_execute (uint32_t address)

This function executes image in RAM.

Parameters

address	Address of the image in RAM.

Returns

Function execution status.

4.1.3.2.20 status_t feature_ram_read (uint32_t address, uint8_t * data, uint32_t length)

This function reads RAM content to the data buffer in RAM.

Parameters

address	Address of the RAM.
data	Pointer to destination data buffer.
length	Number of bytes to read.

Returns

Function execution status.

4.1.3.2.21 status_t feature_ram_write (uint32_t address, const uint8_t * data, uint32_t length)

This function writes data from the source to the specified RAM address *address*.

Parameters	
address	Address of the RAM.
data	Pointer to source data.
length	Number of bytes to write.

Returns

Function execution status.

4.1.3.2.22 status_t feature_rom_signature (void)

This function validates BootROM signature.

Returns

Function execution status.

4.1.3.2.23 uint32_t feature_sfr_read (uint32_t address)

This function reads SFR value.

Parameters	
address	Address of the SFR to be read.

Returns

returns value of SFR.

4.1.3.2.24 void feature_sfr_write (uint32_t address, uint32_t value)

This function write value to SFR.

Parameters	
address	Address of the SFR.
value	Value to be written.

Returns

Function execution status.

4.2 Data Structures

4.2.1 srv_1000tp_erase_t struct

data structure of service request operation: 1000tp erase.
Read more...

```
#include <user_types.h>
```

Public attributes

uint32_t address

4.2.1.1 Detailed description

data structure of service request operation: 1000tp erase.

4.2.1.2 Variable details

4.2.1.2.1 uint32_t address

4.2.2 srv_1000tp_write_t struct

data structure of service request operation: 1000tp write.
Read more...

```
#include <user_types.h>
```

Public attributes

uint32_t address

uint8_t data[NVM_PAGE_SIZE]

uint32_t length

4.2.2.1 Detailed description

data structure of service request operation: 1000tp write.

4.2.2.2 Variable details

4.2.2.2.1 uint32_t address

4.2.2.2.2 uint8_t data[NVM_PAGE_SIZE]

4.2.2.2.3 uint32_t length

4.2.3 srv_ifmgnt_t struct

data structure of service request operation: interface management.
Read more...

```
#include <user_types.h>
```

Public attributes

interface_t interface
interface_op_t interface_op
uint16_t reserved

4.2.3.1 Detailed description

data structure of service request operation: interface management.

4.2.3.2 Variable details

4.2.3.2.1 interface_t interface

4.2.3.2.2 interface_op_t interface_op

4.2.3.2.3 uint16_t reserved

5 List of abbreviations

1000TP	1000-time-programming
API	Application programming interface
BSL	Bootstrap loader
bootROM	Boot code in ROM
CS	Configuration sector
FS_WDT	Fail-safe watchdog
MBIST	Memory built-in self-test
MCU	Microcontroller unit
NAC	No-activity counter
NAD	Node address
NVM	Non-volatile memory
ROM	Read only memory
SWD	Serial wire debug
UBSL	User bootstrap loader
VTOR	Vector table offset register

Revision history

Document version	Date of release	Description of changes
Rev.0.02	2023-03-31	• Added API document
Rev.0.01	2023-01-27	• Initial release of firmware user manual

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-03-31

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-mmw1674646376628

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.