STM32U3 series UL/CSA/IEC/ 60730-1/60335 -1 self-test library user guide

Introduction

This document applies to the X-CUBE-CLASSB self-test library set for STM32U3 microcontrollers (order code X-CUBE-CLASSB-U3).

The X-CUBE-CLASSB-U3 Expansion Package for STM32Cube includes an application-independent software test library. It is released by ST to implement a relevant subset of safety mechanisms required by the safety concepts applicable to STM32U3 microcontrollers that include the Arm® Cortex®-M33.

Table 1. Applicable product

Part number	Order code
X-CUBE-CLASSB	X-CUBE-CLASSB-U3

1 General information

1.1 Purpose and scope

The X-CUBE-CLASSB-U3 Expansion Package runs on the STM32U3 Arm® Cortex®-M33 based microcontroller.

arm

Note: Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere.

The Arm word and logo are trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved.

The version of the application-independent software library (Self-Test Library), available in the X-CUBE-CLASSB-U3 Expansion Package (and associated to this manual), *STL_Lib.a* is V4.0.0.

Refer to Supported products.

1.2 Reference documents

[1] AN4435, Guidelines for obtaining UL/CSA/IEC 60730-1/60335-1 Class B certification in any STM32 application

[2] ES0647, X-CUBE-CLASSB self test library software errata

2 STM32Cube overview

2.1 What is STM32Cube?

STM32Cube is an STMicroelectronics original initiative to improve designer productivity significantly by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an Eclipse®-based IDE, providing code edition, compilation, programming, and debugging capabilities
  - STM32CubeCLT, an all-in-one command-line development toolset with code compilation, board programming, and debug features
  - STM32CubeIDE for Visual Studio Code (STM32VSCode), a complete IDE based on VS Code® platform
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD), powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real time
  - STM32CubeWiSE (STM32CubeWiSEbe, STM32CubeWiSEce, STM32CubeWiSEcg, STM32CubeWiSEre, STM32CubeWiSE8e), graphical tools designed to evaluate and test the capabilities of RF radios and protocols (Bluetooth® LE, sub-GHz, IEEE 802.15.4)
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeU3 for the STM32U3 series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as ThreadX, FileX, LevelX, NetX Duo, USBX, touch library, Mbed TLS, and OpenBL
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

2.2 How does this software complement STM32Cube?

The software expansion package extends STM32Cube by a middleware component to manage specific software-based diagnostics.

The package provides a generic starting point to help a user to build and finalize application specific safety solutions. It consists of:

- STL: the self-test library. This provides a binary and some source code to manage the execution of generic safety tests for the microcontroller. The STL is a standalone unit, which runs independently from any STM32 software. It collects the self-tests for generic components of the microcontroller.
- User application: This is an STL integration example based on a set of STM32Cube drivers extending the STL by an application specific test. This part is delivered as full source code to be adapted or extended by calling of additional application specific modules defined by end user. The example can be used for the library testing including artificial failing support of all the provided modules.

3 STL overview

The STL is an application-independent software test library released by ST to implement a relevant subset of safety mechanisms required by the safety concepts applicable to STM32U3 microcontrollers. The STL is HAL/LL independent, and is dedicated to STM32U3 microcontrollers. The STL is compilation tool chain-agnostic, so it can be compiled by any standard C-compiler.

The *STL* is an autonomous software, which executes, on application demand, selected tests to detect hardware issues, and reports the outcomes to the application.

The *STL* is delivered partly in object code (for the library itself) and partly in source code for the user interfaces definition and user parameters settings.

3.1 Architecture overview

The *STL* tests the Arm® Cortex®- M33 *CPU* core, the flash memory and the *RAM*.

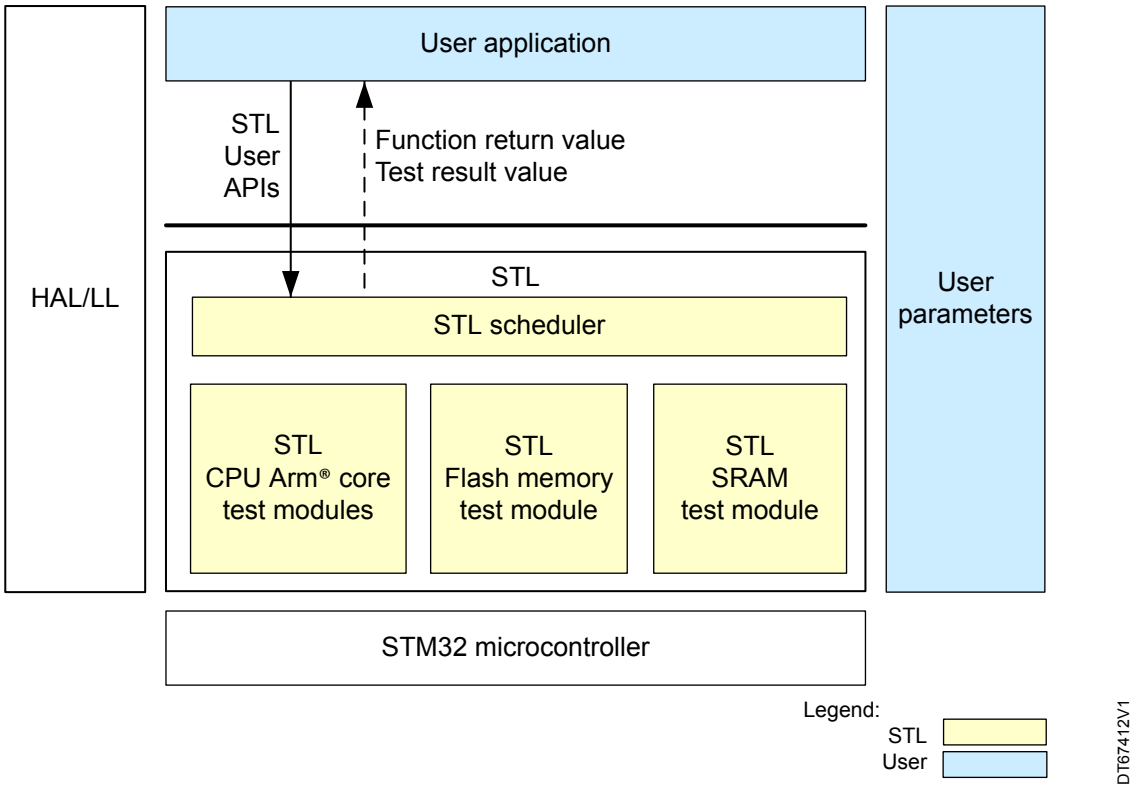
The system architecture of the *STL*, as shown in Figure 1. *STL* architecture, is composed of:

- User application (in light blue)
- User parameters (in light blue)
- *STL* scheduler (in light yellow): directly accessible by the User application via User APIs (not going through *HAL*/*LL*)
- *STL* internal test modules (in light yellow): called by the *STL* scheduler (not visible from the User application).

The *STL* status information returned to the User application at *API* level (see Table 2. STL return information) are:

- The function return values collect the results of internal defensive programming checks (STL_OK, STL_KO)
- The test module result values store the test result information. This partially corresponds to internal status of the module. (STL_PASSED, STL_PARTIAL_PASSED, STL_FAILED, STL_NOT_TESTED, STL_ERROR).

Figure 1. *STL* architecture



The *STL* also allows the developer to:

- Use the artificial-failing feature. The developer can check the application behavior by forcing the *STL* to return a requested test result value. This feature is available through the specific user *API*.

TrustZone®:

When TrustZone® is activated, the *STL* must be executed in Secure state. If the *STL* needs to be executed from non-secure state, then it must be called via the SG (secure gateway) feature.

3.2 Supported products

The STL runs on the following STM32 microcontrollers:

- STM32U3A6xx
- STM32U3B5xx
- STM32U3C5xx
- STM32U335xx
- STM32U345xx
- STM32U356xx
- STM32U366xx
- STM32U375xx
- STM32U385xx
- STM32U396xx

4 STL description

This section collects information about the functionality and performance of the *STL*. It also details constraints and mandatory actions that the end user must respect.

4.1 STL functional description

4.1.1 Scheduler principle

The scheduler is the *API* module needed by the user application to execute the *STL*.

The main scheduler:

- Needs to be initialized before being used
- Manages:
  - Initialization and de-initialization of the applied test modules
  - Configuration of the applied test modules
  - Reset of the applied test modules
- controls the execution of:
  - an applied test sequence (API calls)
- Manages *artificial failing* (for user debug and integration tests)
- Ensures integrity of critical internal data structures via their specific checksums.

The scheduler controls the execution of the following tests, in single testing modes:

- *CPU* tests: no specific initialization or configuration procedures of the CPU test module are required before *CPU* test execution (see Section 7.2: User APIs and Figure 11. State machine diagram - *CPU* test *API*s).
- Flash memory tests over configured subsets (flash memory test module initialization and configuration procedures are mandatory before flash memory test execution, see Section 7.2: User APIs and Section 7.3: State machines).
- *RAM* tests over configured *RAM* subsets (*RAM* test module initialization and *RAM* configuration procedures are mandatory before *RAM* test execution. See Section 7.2: User APIs and Figure 13. State machine diagram - RAM test *API*s)

The *STL*, via the scheduler *API*, is called by the user in polling mode. The *STL* can be called in an interrupt context, but re-entrance is forbidden. In such cases, the *STL* behavior cannot be guaranteed.

The user application has to manage all the information returned from the *STL*. This is provided via a specific predefined data structure that collects status information. See details in Table 2.

Table 2. STL return information

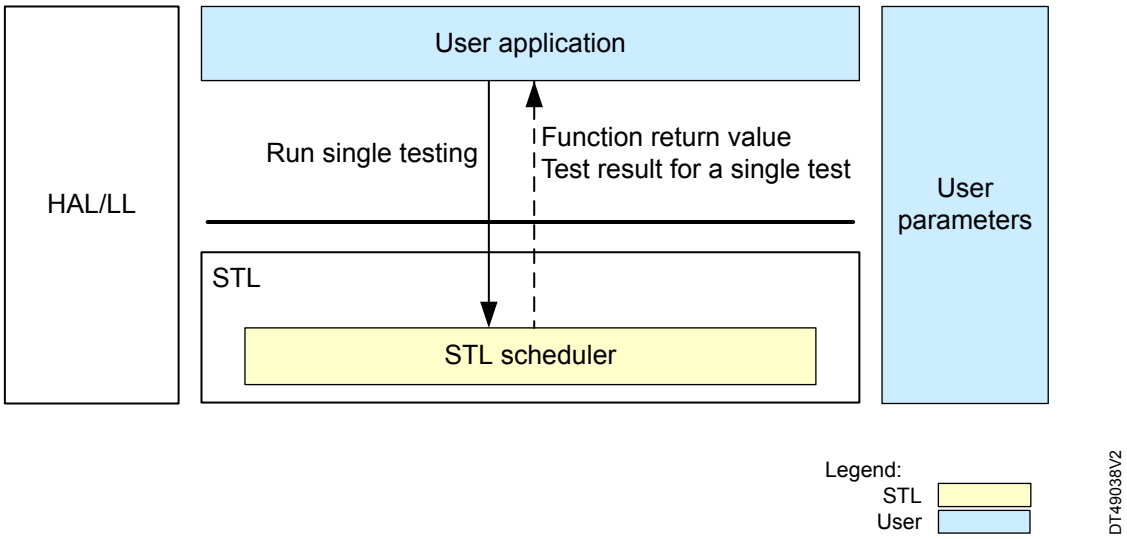
STL information	Value	Description
Function return value(1).	STL_OK	Scheduler function successfully executed.
	STL_KO	Scheduler defensive-programming error (in this case the test result is not relevant).
Test-module result value. (2).	STL_PASSED	Test passed
	STL_PARTIAL_PASSED	Used only for *RAM* and flash memory testing, when test is passed, but end of *RAM*/flash memory configuration not yet reached
	STL_FAILED	Hardware error detection by test module
	STL_NOT_TESTED	Test not executed
	STL_ERROR	Test module defensive-programming error

1. *See STL_Status_t definition in Section 7.1: User structures.*
2. *See Section 7.1: User structures: STL_TmStatus_t for single test*

Single-test mode

The user application applies the call scheme shown in Figure 2. By doing this repeatedly, the user programs a sequence of single-API function calls, and so handles the order of execution of the test modules.

Figure 2. Single-test mode architecture



Scheduler and interrupts

The scheduler is interruptible at any time. Some test modules can temporarily mask interrupts. For more details, refer to Section 4.2.5: STL interrupt masking time and Section 4.3.7: Interrupt management.

4.1.2 CPU tests

The *STL* includes the following *CPU* test modules (*TM*s), together with a generic description (informative only) of the Arm® core test capability:

- TM1L: implements a simplified version for the test of general purpose register
- TM7: implements the test of both stack pointers, MSP and PSP
- TMCB: implements test of the APSR status register

These tests can be run in single test mode.

Caution: *The STL CPU tests are partitioned in separate TMs. This is not intended to allow partial execution of the overall available CPU TMs, but rather as a support feature to allow better CPU test scheduling in end-user applications (for example, timing constraints).*

CPU tests and interrupts

The *CPU TM*s are interruptible at any time.

CPU TM7 masks interrupts during the smallest data granularity time (see Section 4.2.5: STL interrupt masking time).

Refer to Section 4.3.7: Interrupt management for detailed information on *CPU* TM7 interrupt management.

4.1.3 Flash memory tests

Principles

The flash test concerns the embedded flash memory of the STM32 device.

There are three *STL* flash memory area concepts:

- Block: a contiguous area of 4 bytes (FLASH_BLOCK_SIZE). The value is hard coded by the *STL*.
- Section: a contiguous area of 1024 bytes (FLASH_SECTION_SIZE). The value is hard coded by the *STL*. This has no link with the flash memory physical sector. The flash memory is partitioned in sections. The first section starts at the first address of the flash memory, and the next sections are contiguous.
- Binary (named *user program* in Figure 4. Flash memory test: CRC use cases versus program areas): a contiguous area of code provided by the compiler. It starts at the beginning of a section. It usually ends with an incomplete section when the binary area size is not a multiple of the section size. In all cases, the binary must be 32-bit aligned (see ST CRC tool information). When TrustZone® is activated, there is typically one secure binary and one nonsecure binary.
- Subset: a contiguous area of sections with available expected CRC values. The user application can define one subset or several subsets. A subset has to be defined within a binary area. Its start address has to be aligned with a section begin and it can only include sections with a corresponding expected precalculated CRC value. When the last subset section is the last part of the binary, the section may be incomplete, and the user application has to align the end of the subset with the end address of the binary area. If a set of complete sections is tested exclusively, the subset end address has to be aligned with the end of the last tested section.
- Subset size = K * FLASH_SECTION_SIZE + L * FLASH_BLOCK_SIZE, where:
  - K is an integer ≥ 0
  - L is an integer with 0 <= L < 256. (256 corresponds to (FLASH_SECTION_SIZE / FLASH_BLOCK_SIZE).
    Note that L can exist only when the last section of a binary is incomplete (that is, it is smaller than FLASH_SECTION_SIZE).

  The subset is defined by the user application. It is possible to define several subsets.

The *STL* implements a test of the flash memory with the following principles:

- Tests are performed on sections of the subset(s) defined by the user application.
- Tests are performed either in a row (one shot) or partially in a single atomic step for a number of sections defined by the user application.
- Test results are based on a CRC comparison between the computed CRC value (calculated during test execution) and an expected CRC value (calculated before software binary flashing).

The mandatory steps (for the user application) to perform flash memory tests are:

- Test initialization
- Configuration of the subset(s)
- Execution of the test

Once all subsets are tested, the user needs to reset the flash memory test module to perform the test again.

In the case of an STL_ERROR / STL_FAILED test result, the test module is stuck at the failed memory subset. The user needs to deinitialize and then reinitialize the flash memory test module (or reset it) to perform the test again.

Expected CRC precalculation

The flash memory test is based on a built-in hardware CRC unit or software CRC, configurable by compilation flag. The default configuration is with hardware CRC. To use software CRC, the flag STL_SW_CRC (see Section 5.5.3: Steps to build an application from scratch) must be enabled. The CRC is a 32-bit CRC compliant with IEEE 802.3.

Part of the flash memory is reserved for the CRC dedicated area, the size of which depends on the flash memory size. This area has a field format where each flash memory section has sufficient reserved space to store a 32-bit CRC pattern. The user must ensure that valid CRC patterns are calculated and stored in the fields for all the sections to be tested, as shown in Figure 3. Flash memory test: CRC principle.

When TrustZone® is activated, a CRC area must be available in the flash memory corresponding to the tested subsets. If the subsets to be tested are defined in nonsecure flash memory and in secure flash memory, it means that there must be one dedicated CRC area for nonsecure flash memory and one dedicated CRC area for secure flash memory.

One expected CRC value is precalculated for each contiguous section of a binary, from binary start to binary end. This means that the number of testable sections depends on the binary size. Commonly, the binary area is not aligned with section size, in which case the CRC check value of the last incomplete section is precalculated and tested exclusively over the section part that overlays the binary area.

Preconditions:

- The user program areas have to start at the beginning of a section
- The boundaries of the user program areas must be 32-bit aligned.
- Depending on total flash memory size and on user program size, last program data and first CRC data may be both stored in the same flash memory section (without any overlap), in that case, CRC must be computed on the user program data only, see Example 3 in Figure 4. Flash memory test: CRC use cases versus program areas.

CRC tool information

STMicroelectronics provides a CRC precalculation tool, available as a single feature inside the STM32CubeProgrammer (see Section 6.2.2: CRC tool set-up), which automatically fills the binary with padding bits (0x00 pattern) for a 32-bit alignment.

Figure 3. Flash memory test: CRC principle

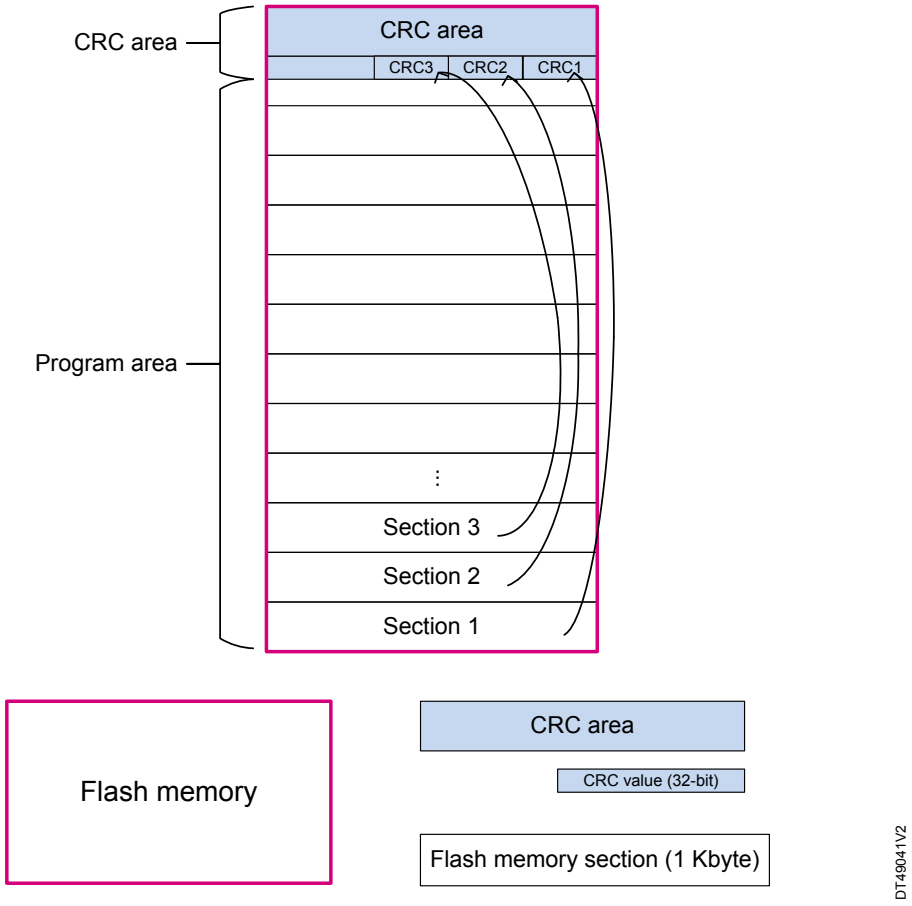
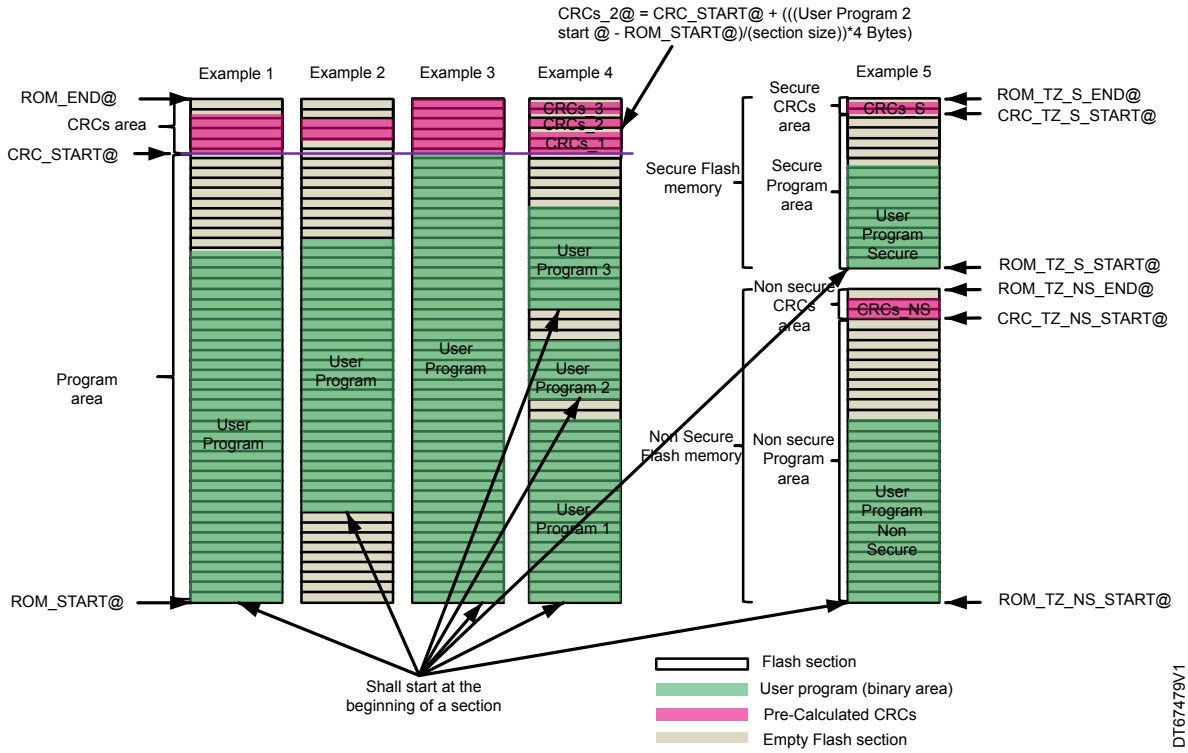


Figure 4. Flash memory test: CRC use cases versus program areas



Use case descriptions:

- Example 1: the user program starts at the ROM_START address, so CRCs are stored from the CRC_START address.
- Example 2: the user program starts at the beginning of a section, but not at ROM_START. Stored CRCs start at the right address of the CRC area.
- Example 3: the user program uses the full program area, so the last program data and the first CRC data are both stored in the same flash memory section (without any overlap).
- Example 4: the user program is defined in three separated areas. This requires three separated areas for the CRC data.
- Example 5: TrustZone® is activated. The user program in flash memory is divided into one secure binary with its dedicated CRC area and one nonsecure binary with its dedicated CRC area.

CRC start address computation:

- Real calculation:
  CRC_START address = (uint32_t *)(ROM_END - 4 * (ROM_END + 1 - ROM_START) / (FLASH_SECTION_SIZE) + 1); with FLASH_SECTION_SIZE = 1024
- Textual translation:
  CRC_START = ROM_END - (CRC size in bytes) * (number of flash memory sections) + 1

Flash memory test and interrupts

Flash memory TM is interruptible at any time.

4.1.4 RAM tests

Principles

The *RAM* test concerns the embedded SRAM memories of the STM32 device.

There are three *RAM* area concepts:

- Block: a contiguous area of 16 bytes (RAM_BLOCK_SIZE). This value is hard coded by the *STL* (there is no link with the memory physical sectors).
- Section: a contiguous area of 128 bytes (RAM_SECTION_SIZE). This value is hard coded by the *STL*.
- Subset: a contiguous area, with the size being a multiple of two blocks and with a 32-bit aligned start address. Then a subset size is not necessarily a multiple of the section size, because the last part of a subset can be less than one section.
- Subset size = N * RAM_SECTION_SIZE + 2 * M * RAM_BLOCK_SIZE, where:
  - N is an integer ≥ 0
  - M is an integer with 0 <= M < 4, where 4 corresponds to RAM_SECTION_SIZE / (2 * RAM_BLOCK_SIZE). Note that the case M > 0 means that the last part of a subset is not aligned with section size.

  The subset is defined by the User application. It is possible to define several subsets.

The *STL* implements a *RAM* memory test with the following principles:

- *RAM* tests are performed on *RAM* blocks defined by the User application
- *RAM* tests are performed either in a row (one shot), or partially in a single atomic step for a number of sections defined by the User application
- The test implementation is based on March C- algorithm

The mandatory steps (for the User application) to perform *RAM* tests are:

- initialization of *RAM* test
- configuration of the *RAM* subset(s)
- execution of the *RAM* test

Once all subsets are tested, the application must reset the *RAM* test module to perform the test again.

In the case of an STL_ERROR / STL_FAILED test result, the test module is stuck at the failed memory subset. It is therefore advisable to deinitialize then initialize the *RAM* test module (or to reset it) before performing the test again.

RAM test and interrupts

The *RAM TM* is interruptible at any time except execution of the smallest data granularity (block - see Section 4.2.5: STL interrupt masking time) when the STM32 interrupts and Cortex® exceptions with configurable priority are temporarily masked by default.

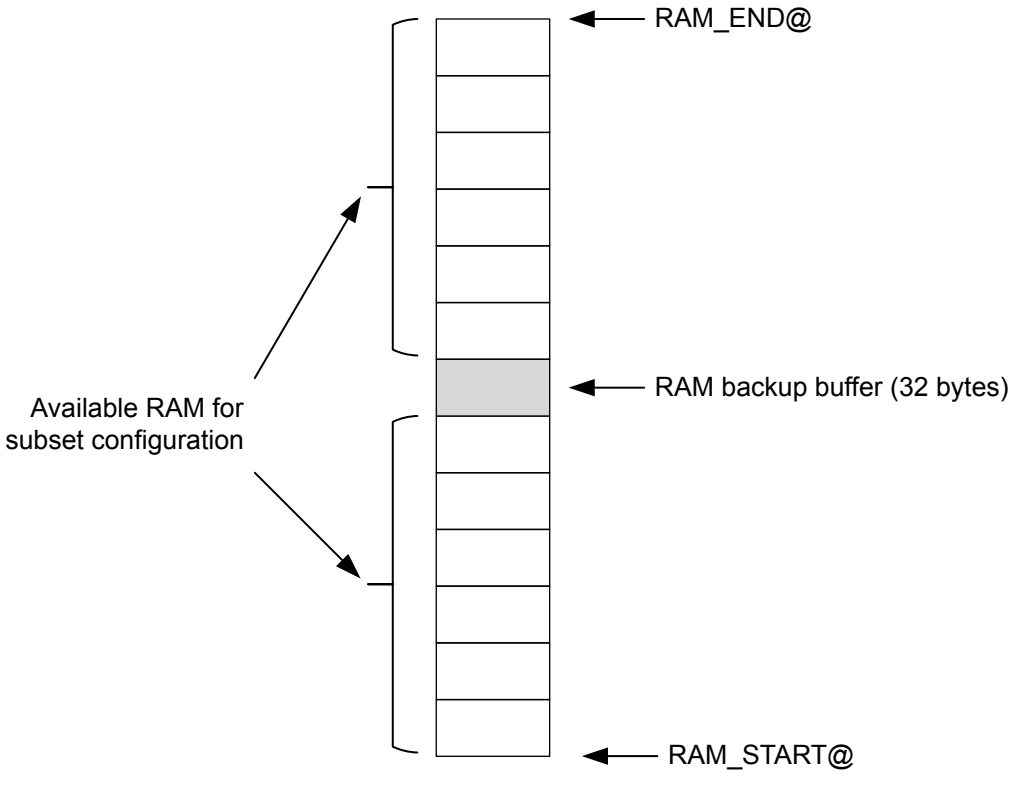
Refer to Section 4.3.7: Interrupt management for detailed information on interrupt and *RAM* March-C tests

March C- test principle

The RAM test is based on a March C- algorithm where memory is overwritten by specific patterns and then read back at specific orders. To restore the initial memory content, a backup process is enabled. User must allocate specific backup area that is also tested by the March C- test. Refer to RAM backup buffer for detailed information on the RAM backup buffer control and allocation. When TrustZone® is activated, the backup buffer must be located in the secure RAM. At this case, the user must allocate a specific backup area that is also tested by the March C- test.

Note: For security reasons, the use of the backup buffer is mandatory in the microcontrollers and not optional as for other devices. This means that the option STL_DISABLE_RAM_BCKUP_BUF is not available and has no effect.

Figure 5. *RAM* test: usage



4.2 STL performance data

The following data are obtained with the following test set-up:

- STL library compilation details, detailed in Section 5.5: Application: compilation process.
- Projects for performance tests are compiled with IAR™ Embedded Workbench for Arm® (EWARM) toolchain v9.60.3
- Compiled SW configuration with:
  - HCLK clock set to 96 MHz
  - Flash latency set to four wait states
  - NUCLEO U3C5ZI-Q
  - ICache disabled

4.2.1 STL execution timing summary

Table 1 contains a summary of the STL execution timings. The measurements for each API are detailed in STL: execution timing details.

Table 3. STL execution timings, clock at 96 MHz

Tested module	Conditions		Result (clock cycles)
CPU	TM1L TM7, TMCB		10918
Flash memory	Default configuration (STL_SW_CRC not enabled):	1 Kbyte tested	6436
		17 Kbytes tested	92308
	STL_SW_CRC enabled:	1 Kbyte tested	24678
		17 Kbytes tested	405062
RAM	Default configuration (STL_ENABLE_IT not enabled):	128 bytes tested	11584
		196 Kbytes tested	15186899

4.2.2 STL code and data size

The STL code and data sizes are detailed in Table 4.

Table 4. STL code size and data size (in bytes)

Configuration	Module	Flash memory code	Flash memory RO-data	R/W data
STL_SW_CRC not enabled, and STL_DISABLE_RAM_BCKUP_BUF not enabled	*stl_user_param_template.o*	-	12	56
	stl_util.o	286	-	8
	stl_lib.a	6356	1478	164
STL_SW_CRC enabled, and STL_DISABLE_RAM_BCKUP_BUF not enabled	*stl_user_param_template.o*	-	12	56
	stl_util.o	150	-	4
	stl_lib.a	6356	1478	164

4.2.3 STL stack usage

The minimum stack available space required by the STL is 200 bytes.

4.2.4 STL heap usage

The *STL* never uses dynamic allocation. Therefore, the heap size is independent of the *STL*.

4.2.5 STL interrupt masking time

STM32 interrupts are masked multiple times by the STL during CPU TM7 and RAM tests. The maximum interrupt time (see Table 1) is obtained for a RAM test, when backup buffer is enabled.

Table 5. STL maximum interrupt masking information

Tested module	Duration (max) in clock cycles	Comments
RAM	1089	Each execution of the STL_SCH_RunRamTM function performs series of interrupt masking during partial steps of the test at the following time durations: • 1083 clock cycles for the RAM backup buffer • 883 clock cycles for the 1st RAM block to be tested • 1089 clock cycles for each middle RAM block to be tested* • 903 clock cycles for the last part RAM block to be tested * Number of RAM blocks (multiple of 2 RAM_BLOCK_SIZE is required) involved at each execution depends on content of user structures (size of defined subset(s) versus atomic step – see Section 4.1.4: RAM tests).
CPU TM7	822	Masked twice with 790 and 822 clock-cycle duration.

4.3 STL user constraints

The end user needs to consider interference between the application and the *STL*. The consequences of ignoring this are possible false *STL* error reporting, and/or application software execution perturbation.

Accordingly, to prevent any interference the application software and the STL integration must comply with each constraint listed in this section.

4.3.1 Function call convention

The STL complies to function call convention in particular on register save and restore. The application must also respect this calling convention to ensure that the STL does not collect false error reporting.

4.3.2 Privileged-level

The CPU TM7 and the RAM TM must be executed with privileged-level, in order to be able to modify certain core registers (for example the PRIMASK register).

4.3.3 RCC resources

During STL execution, the RCC is configured to clock:

- CRC during STL initialization and during TM flash execution

This means that:

- When the STL returns, it restores the user RCC clock setting (enable or disable) for the CRC
- The user application must be careful to save/restore the STL settings when configuring the RCC during STL execution.

4.3.4 *CRC* resources

When the hardware CRC is used, the *STL* relies on the STM32 CRC IP. The *CRC* resources are used during *STL* execution in two different cases:

- During execution of *STL* initialization (function STL_SCH_Init): the hardware CRC is used. The use of hardware CRC in this phase cannot be modified by the application software, so the STL_SW_CRC compilation flag has no impact during execution of the STL_SCH_Init function.
- During execution of flash TM, the application can choose between hardware CRC and software *CRC* by means of the STL_SW_CRC flag. By default, hardware CRC is used (the STL_SW_CRC flag is disabled).

The use of hardware CRC means that:

- Before calling the *STL*, the user application has to save its own hardware CRC configuration, and to restore it after *STL* execution.
- During *STL* execution, the hardware CRC is configured and used for *STL* needs (the user application must take care to save/restore the *STL* settings when using the *CRC* during *STL* execution).

4.3.5 Bit Q of APSR

CPU TMCB execution can cause bit Q of the APSR (sticky bit) to be set. The user application must take this into account when using this bit.

4.3.6 GE bits of APSR

During *CPU* TMCB execution, the GE bits of *APSR* can be set. They are always set to 0 at the end of the *CPU* TMCB.

4.3.7 Interrupt management

Escalation mechanism - Arm Cortex behavior reminder

When the STL disables STM32 interrupts and Cortex exceptions with configurable priority, remember that an Arm Cortex escalation to HardFault might occur. In this case, the HardFault handler is called instead of the fault handler.

Interrupt and CPU TM7

By default, the STM32 interrupts, and Cortex exceptions with configurable priority, are temporarily masked during CPU TM7 execution during smallest data granularity (a few instruction blocks), except when the User application activates the dedicated compilation switch, STL_ENABLE_IT (see Steps to build an application from scratch). If the STL_ENABLE_IT flag is activated, the correct STL CPU TM7 behavior is not warranted. The end user is responsible for managing interferences between the STL and its application software that could possibly lead to false STL error reports, or non-detection of hardware failure.

The CPU test modules unrelated to TM7 are neither affected by masking of STM32 interrupts, nor of Cortex exceptions with configurable priority.

Interrupt and RAM March C- tests

By default, the STM32 interrupts and Cortex exceptions with configurable priority are masked during the RAM March C- tests, except if the user application activates STL_ENABLE_IT (see Steps to build an application from scratch).

If the STL_ENABLE_IT flag is activated:

- The correct STL RAM test behavior is not warranted, as the STL-tested RAM content may be overwritten by the application during its interrupt treatment (this could lead the STL to generate false RAM test error reporting)
- The User application software behavior can be compromised, because wrong data may be read or used from RAM locations being modified by the STL March C- test.

4.3.7.1 How the STL masks the interrupts

For masking the interrupts, STL sets the PRIMASK register PM bitfield to 1. Setting this bit to 1 boosts the current execution priority to 0, masking all exceptions with a programmable priority.

Thus, when the current execution priority is boosted to a particular value, all exceptions with a lower or equal priority are masked.

4.3.8 DMA

The application must manage DMA to avoid unwanted access to RAM bank during the STL March C- test. In this case:

- DMA writes can disturb the STL test causing false error reporting
- DMA reads can return wrong data, due to STL overwrites to DMA dedicated RAM sections.

4.3.9 Supported memories

Only STM32U3 internal memories are supported by the memory tests (Flash TM and RAM TM).

The STL library must be executed from STM32U3 internal memories only.

4.3.10 RAM backup buffer

The backup process is mandatory during *RAM* tests to preserve the *RAM* content. User must reserve a specific area for the *RAM* backup buffer at compilation time. This area must be allocated outside of the *RAM* subset configuration, and must be located in the Secure *RAM* when TrustZone® is activated. There is only one *RAM* backup buffer defined for the test. The *RAM* backup buffer area is also tested by the March C- test each time the *RAM* test function is called (before any subset defined by the user is tested).

4.3.11 Memory mapping

Due to the RAM test module and March C method design, the user must ensure, via proper adaptation of the associated linker file, that the “read only” data of the STL is located in the Flash memory.

The examples below are for EWARM and STM32CubeIDE.

EWARM .icf file adaptation example

```
place in ROM_region { readonly };
```

STM32CubeIDE.ld file adaptation example

```
.rodata :
{
.....
} >ROM
```

Note: Usually the default configuration locates “read only” data in Flash memory.

4.3.12 TrustZone®

When TrustZone is activated, the STL library must be embedded once in secure binary. If the STL needs to be executed from non-secure state, then it must be called via the SG (secure gateway) feature.

4.3.13 Processor mode

The STL CPU TM7 must be executed in thread mode in order to set the active stack pointer to the process stack pointer.

If the STL is not executed in thread mode, CPU TM7 returns STL_ERROR.

4.3.14 Setting the PSPLIM (process stack pointer limit) CPU register

When applying a nonzero value into PSP stack limit special register (PSPLIM or PSPLIM_S when TrustZone® is activated), the caution is advised to prevent any fault exceptions during TM7 test execution. The PSP stack pointer is temporary redirected to a virtual stack area allocated within the STL volatile (RW) data section in RAM. To prevent raising any fault exception, the user must ensure that the PSP stack limit is below this area during testing. This can be achieved by either:

- Adjusting the linker script file to allocate the PSP stack area below the STL volatile data
- Temporarily modifying the content of the PSP special register for the duration of the TM7 execution (for example, by storing a default zero value in the register for that time)

Note: To modify the PSP limit register, predefined CMSIS intrinsic functions can be used, `__get_PSPLIM` and `set_PSPLIM` (or `__TZ_get_PSPLIM` and `__TZ_set_PSPLIM` if TrustZone® is activated) to read, modify, and retrieve the original register setting prior to and after the TM7 module run.

4.4 End user integration tests

This section describes the mandatory tests to be executed by the end user during the verification phase, to guarantee that the STL is correctly integrated in the application software.

4.4.1 Test 1: correct STL execution

The end user must use the expected function return value and the expected test module result value (see Section 7.2: User APIs), to check that each planned diagnostic function (for example run *CPU* tests, configure *RAM* tests or reset flash memory tests) has been correctly executed.

4.4.2 Test 2: correct STL error-message processing

The end user must check that any error information produced by the *STL* function-return and test-module-result values (that is, values different to the expected value, see Section 7.2: User APIs), is correctly interpreted as unexpected behavior, and correctly handled in its application software. During the verification, the artificial-failing feature must be used to emulate the generation of incorrect test-module result values related to associated individual software diagnostics (*CPU* tests, *RAM* test, flash memory test), for each of the individual functions used.

This process cannot be considered as an exhaustive simulation of actual *CPU* failures on real devices. It is rather a testing interface for the implemented *API*s.

5 Package description

This section details the X-CUBE-CLASSB-U3 Expansion Package content and its correct use.

5.1 General description

X-CUBE-CLASSB-U3 is a software Expansion Package for STM32U3 devices.

It provides a complete solution that helps end customers to build a safety application:

- An application-independent software test library is available:
  - partly as object code: STL_Lib.a, the library itself
  - Partly as source file: *stl_user_param_template.c* and *stl_util.c*
  - With three header files: *stl_stm32_hw_config.h*, *stl_user_api.h*, and *stl_util.h*
- A user application example, available as source code

X-CUBE-CLASSB-U3 has been ported on the products listed in Section 3.2: Supported products.

The software Expansion Package includes a sample application that the developer can use to start experimenting with the code. It is provided as a zip archive containing both source code and library.

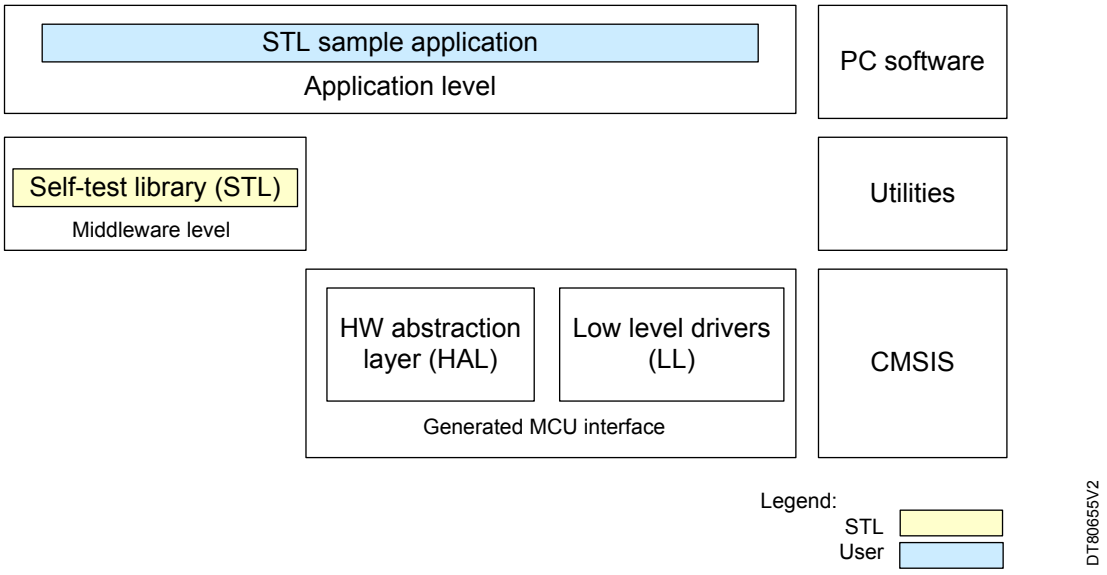
The following integrated development environments are supported:

- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® microcontroller development kit (MDK-ARM)
- STM32CubeIDE

5.2 Architecture

The components of the X-CUBE-CLASSB-U3 Expansion Package are illustrated in Figure 1.

Figure 6. Software architecture overview



5.2.1 STM32Cube HAL

The *HAL* driver layer provides a simple, generic, multi-instance set of APIs (application programming interfaces) to interact with the upper layers (application, libraries and stacks).

It comprises generic and extension *API*s. It is directly built around a generic architecture and allows the layers that are built upon, such as the middleware layer, to implement their functionalities without dependencies on the specific hardware configuration of a given microcontroller.

This structure improves the library code re-usability and guarantees an easy portability to other devices.

5.2.2 STL

The *STL* is available at middleware level. It can mainly (for the *STL* library part) be considered a black box that manages the software-based diagnostic test. The *STL* is independent of the *HAL*, *BSP*, and *CMSIS*, though the *STL* integration examples rely on some *HAL* drivers.

5.2.3 User application

The examples provided show how to:

- use the various *API*s with different test modes and features
- integrate a possible sequence of *STL* test module calls into an application,
- verify the returns of the *API*s and structures
- emulate the failure responses artificially.

The examples are provided solely as a quick demonstration of how to call some *STL API*s when adopting different *IDE*s. They must not be used as representative templates for the integration and use of *STL*s in real safety-application software.

The examples are available with TrustZone enabled and also with TrustZone disabled.

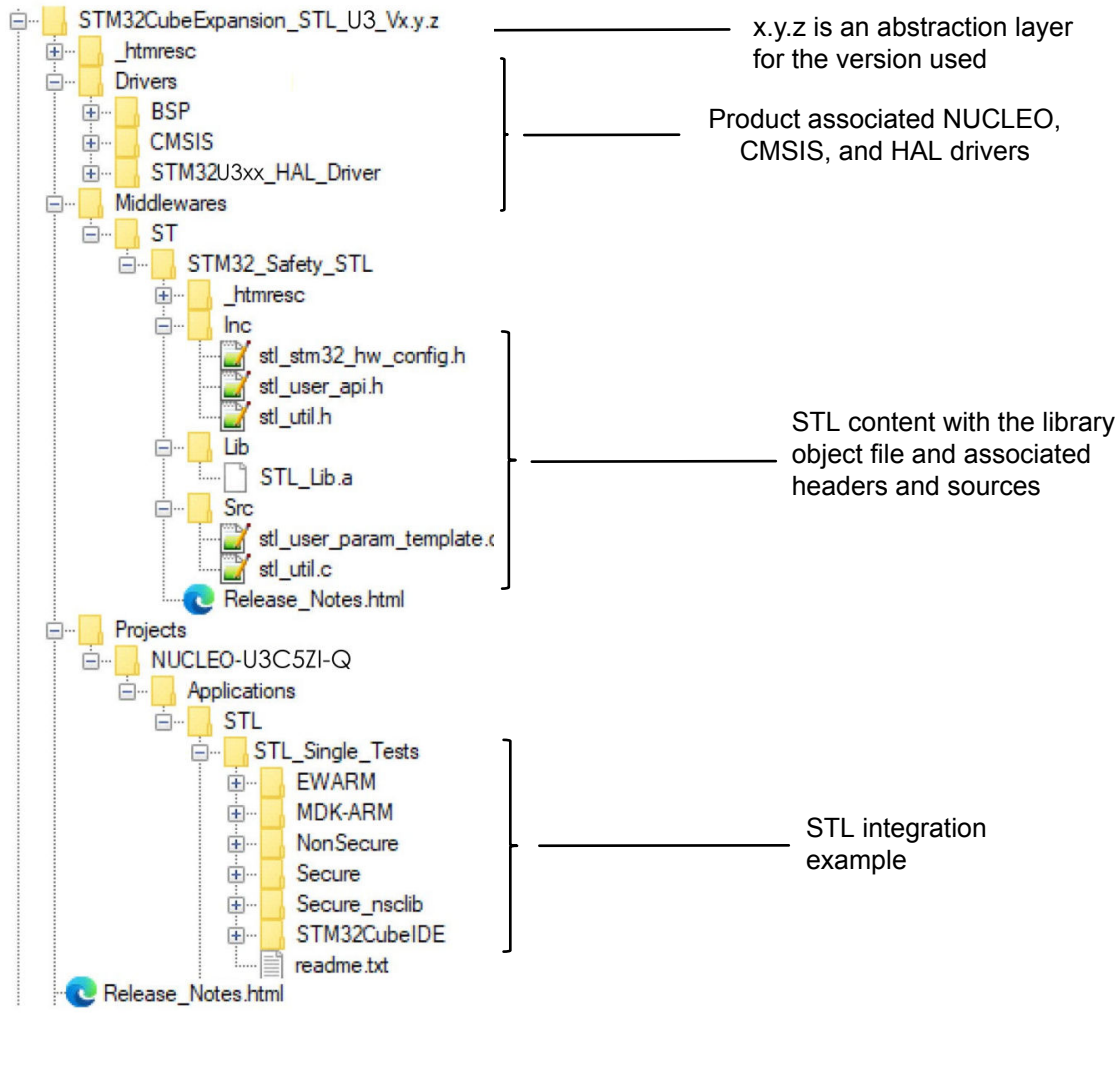
5.2.4 STL integrity

The integrity of the STL content is ensured by hash SHA-256.

5.3 Folder structure

A top-level view of the structure is shown in the following figure.

Figure 7. Project file structure



Note: This is the typical STL folder structure. It might be different.

5.4 APIs

5.4.1 Compliance

Interface compliance

The library part of the *STL*, not delivered in source code, has been compiled with IAR Embedded Workbench® for Arm® (EWARM) v9.20.3 (certified version) with --aeabi and --guard_calls compilation options to fulfill *AEABI* compliance as described in "*AEABI* compliance" of EWARM help section.

Safety guidelines compliance

The library part of the STL, not delivered in source code, has been compiled with IAR Embedded Workbench® for Arm® vxxx EWARMFS (certified version) with **--strict**, **--remarks**, **--require_prototypes**, and **--no_unaligned_access** compilation options to fulfill the safety guideline compliance as described in the IAR Embedded Workbench® safety guide (Advices 2.1-1, 2.2-5, 2.4-1a and 5.4-3) and Keil® safety manual (§4.9.2).

Library compliance

The library part of the *STL* (not delivered in source code) is conformant with C Standard Library ISO C99.

It has been compiled with the IAR™ option . language C dialect = Standard C.

Arm compiler C toolchain vendor/version independency

The *STL* user *API* refers only to the "uint32_t" and "enum" C types:

- "uint32_t" C type is a fixed type size of 32 bits according to C Standard C99
- "enum" C type size, according to C Standard C99, is implementation-defined. It must be capable of representing the values of all the enumeration members. In the *STL* interface, the enum type values are unsigned integers, smaller than or equal to (2^{32} - 1). The user must ensure that enum type value can hold a 32-bit value.

5.4.2 Dependency

- The STL Library calls the memset Standard C library function.
- Furthermore, the IAR EWARM toolchain compiler (used to compile the STL library) may, under some circumstances, call the following Standard C library functions: memcpy, memset and memclr. This behavior is intrinsic to the IAR EWARM toolchain compiler (it is not possible to disable or avoid it).

As a result, when linking the STL Library the user must ensure that these Standard C library functions are defined (use either those provided by the toolchain or the user's own ones).

5.4.3 Details

Detailed technical information about the available *API*s can be found in Section 7.2: User APIs, where the functions and parameters are described.

5.5 Application: compilation process

5.5.1 Steps to build a delivered STL example in an application project without TrustZone

The STL examples are delivered for both TrustZone® enabled and disabled configurations. If TrustZone® is enabled, the STL runs in secure binary and the application runs in nonsecure binary.

Ensure the following steps are completed:

1. Install the ST CRC tool (see Section 6.2.2: CRC tool set-up), used for Flash test. If the customer prefers to use their own CRC tool, some steps specific to ST tools are no longer needed.
2. Project choice. Select a project example and open it
3. Project build. Launch the build that:
   a. compiles a binary
   b. calculates the CRCs (in the case of error, check the CRC tool path), for details see Section 5.5.3: Steps to build an application from scratch
   c. automatically adds the computed CRCs to the compiled binary
4. Load and execute the code example according to the instructions in the readme file available in the associated project folder.

Note: After the compilation/link of a binary, the CRC tool used in the post-build script:

- *Calculates the CRCs (in the case of error, check the CRC tool path), for details see Section 5.5.3: Steps to build an application from scratch.*
- *Automatically adds the computed CRCs to the generated binary.*

5.5.2 Steps to build a delivered STL example with TrustZone enabled

In these delivered STL examples, the STL is running in secure binary and the application is running in non-secure binary.

1. Install the ST CRC tool (see Section 6.2.2: CRC tool set-up), used for Flash test. If the customer prefers to use their own CRC tool, some steps specific to ST tools are no longer needed.
2. Project choice. Select a project example and open it.
3. Project build & Loading: refer to README file of the examples available in the release package. There are few differences to manage the secure and non-secure binary depending on the IDE.
4. Execution. Boot the board, the test result is:
   a. LED ON: test result as expected
   b. LED toggling: error (except for artificial case)

Note: After the compilation/link of a binary, the CRC tool used in the post-build script:

- *Calculates the CRCs (in the case of error, check the CRC tool path). For details see Steps to build an application from scratch.*
- *Automatically adds the computed CRCs to the generated binary.*

5.5.3 Steps to build an application from scratch

1. Install the ST CRC tool (see Section 6.2.2: CRC tool set-up), used for Flash test. If the customer prefers to use their own CRC tool, some steps specific to ST tools are no longer needed.
2. Create new application project with the necessary directory structure and the required packages. Use the STM32CubeMX tool to make it automatically.
3. Copy paste the content of the *...Middleware\ST\STM32_Safety_STL* directory from the delivered STL example into the application project directories structure. Refer to Folder structure. Then modify the project settings while following the next steps:
   a. add all the STL source files located at the SRC directory into the project
   b. assign the INC directory as an additional include one at the project, and
   c. force the linker to include the library object file located at the LIB directory as an additional library.
4. If needed, add the next optional preprocessor compilation switches to the project settings.
   a. Option to enable STL_SW_CRC, if SW CRC is used. If not activated, the HW CRC is used by default.
   b. Option to enable STL_ENABLE_IT, if the User application wants to enable the STM32 interrupts and Cortex exceptions with configurable priority during the CPU TM7 and the RAM test. If not activated, the interrupts are masked during these tests. Note that if the switch is activated, the end user must evaluate potentially related issues (as described in Section 4.1.4: RAM tests).
   c. It is mandatory to enable the STM32 device part number, to select the correct HW CRC configuration and the correct flash memory density.

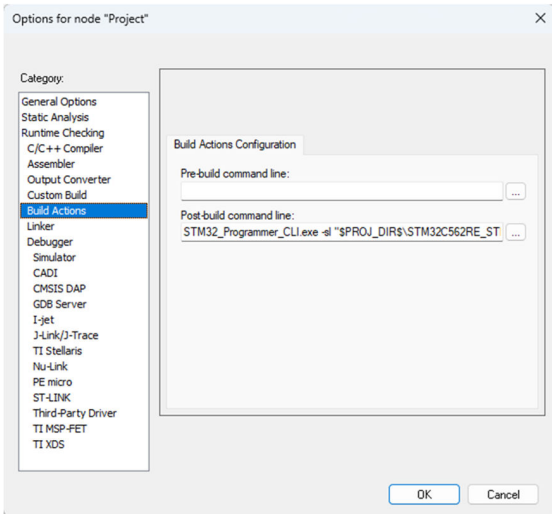
Note: The option STL_DISABLE_RAM_BCKUP_BUF is not available at this STL version. Refer to RAM tests.

5. Check flash memory density configuration. It is mandatory to:
   a. Set the correct range of the flash memory for the project in the *stl_user_param_template.c* file. Ensure coherency especially with the associated linker scatter file and CRC tool script (see step 6).
   b. Check the configuration file stl_user_param_template.c and potentially to update the STL_ROM_END_ADDR to the correct size of the Flash memory (coherency with the associated linker scatter file).
6. Develop the user application by implementing the proper sequence of API calls repeated at periodical cycles, as required by the defined safety task.
   It is mandatory to ensure a proper filling of all the associated user structures to control the memory tests and apply a correct check of the STL return information. Section 7: STL: User APIs and state machines.
7. Apply the CRC tool to build the CRC area content necessary for the CRC calculation. ST CRC calculations are done either by post build (within the IDE), or with a command line:
   a. Post-build example with IAR IDE: In the project "Option", select "Build Actions", and add in post-build the “one line” command line (as shown in Figure 8. IAR post-build action screenshot)
   b. Command line example: Use the same command line shown in Figure 10. CRC tool command line. In the case of an error, check the CRC tool path.
8. Loading. Load the compiled binary.
9. Compile, load, and execute the binary.

Note: When TrustZone® is activated, the STL must be embedded in secure binary. In this case, the steps 3, 4. and 6. apply for configuration of the secure project only. The other steps affect both the secure and nonsecure project configuration. Moreover, if STL is executed from a nonsecure state, the user has to implement a secure gateway feature to access and call all APIs embedded at the secured part.

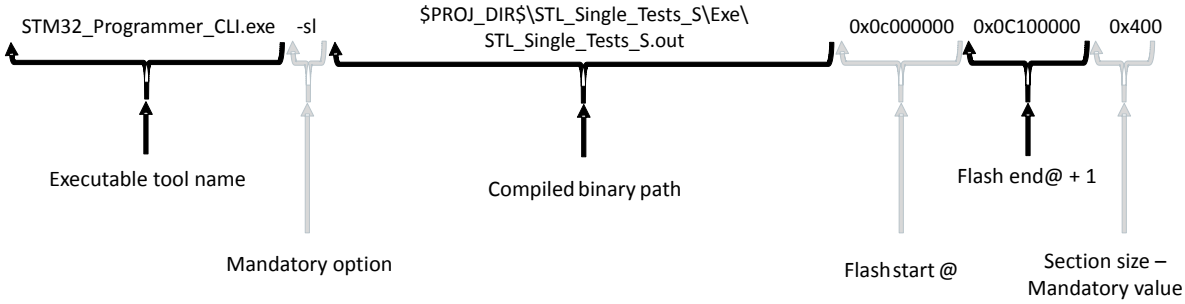
Remember: During the verification, the artificial-failing feature (see Section 7.2.5: Artificial-failing APIs) must be used to emulate the generation of incorrect test-module result values related to associated individual software diagnostics (CPU tests, RAM test, Flash memory test), for each of the individual functions used. See Section 4.4.2: Test 2: correct STL error-message processing.

Figure 8. **IAR post-build action screenshot**



DT80656V1

Figure 9. **CRC tool command line**



Note: The Flash start and end addresses used in the post-build command depend on the project configuration, particularly on whether TrustZone® is enabled or disabled, and whether the project is Secure or Non-Secure. Make sure to check and adjust these addresses according to the actual memory mapping for the STM32U3 project based on the STM32U3 series.

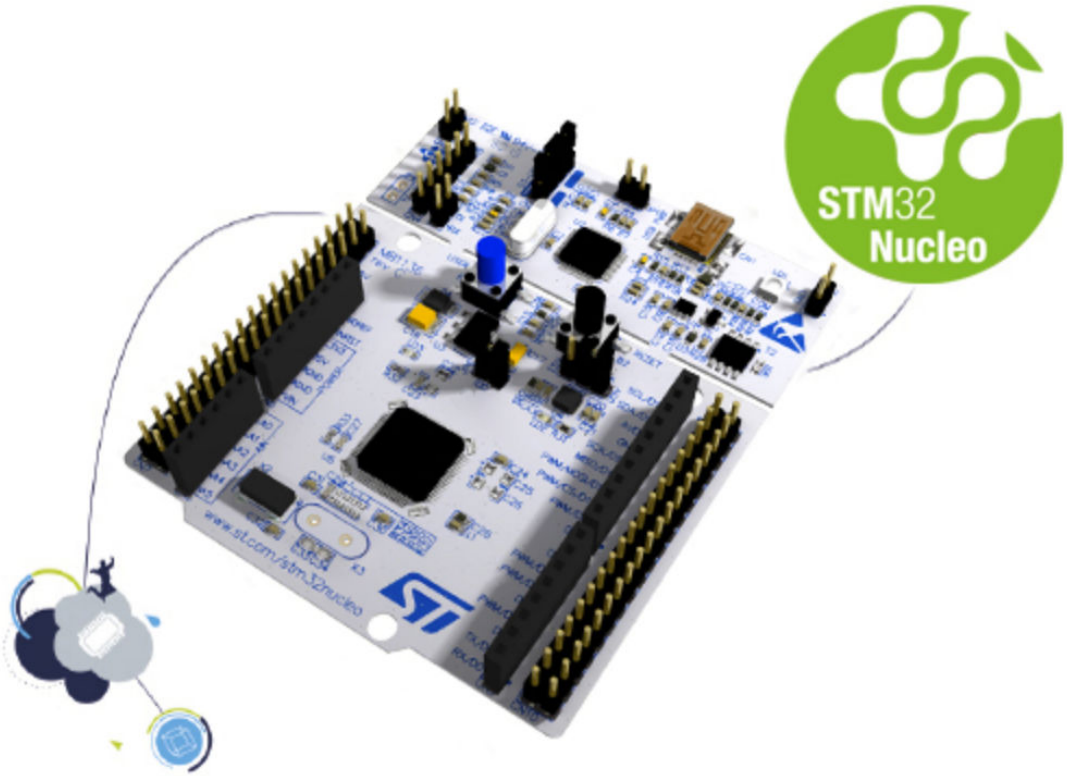
6 Hardware and software environment setup

6.1 Hardware setup

The STM32 Nucleo boards provide an affordable and flexible way for users to try out new ideas and build prototypes with any STM32 device. The ARDUINO® connectivity support and ST morpho headers make it easy to expand the functionality of the STM32 Nucleo open development platform with a wide choice of specialized expansion boards. The STM32 Nucleo board does not require any separate probe as it integrates the ST-LINK/V3 debugger/programmer. The STM32 Nucleo board comes with the STM32 comprehensive software HAL library together with various packaged software examples.

Details about the STM32 Nucleo boards (see Figure 10) are available from the http://www.st.com/stm32nucleo web page.

Figure 10. STM32 Nucleo board example



DT49235V2

Note: This is a picture of a specimen Nucleo board and is not contractual

The following components are needed:

- STM32 NUCLEO-U3C5ZI-Q development board
- USB type C cable to connect the STM32 Nucleo board to the PC

6.2 Software setup

This section lists the minimum requirements for the developer to setup the SDK, run the sample scenario, and customize applications.

6.2.1 Development tool-chains and compilers

Select one of the IDEs supported by the STM32Cube software Expansion Package.

Read the system requirements and setup information provided by the selected IDE provider.

Check the projects *Release_Notes.html* file inside the release package, and refer to the "IDE compatibility" chapter, if it exists.

6.2.2 CRC tool set-up

A *CRC* tool that can be used to generate the *CRC* checksums, needed for X-CUBE-CLASSB Flash memory test, is available as a single feature inside the STM32CubeProgrammer. Other *CRC* tools can be used, provided they fulfill the requirements detailed in Expected CRC precalculation.

Tool installation procedure:

- Select STM32CubeProgrammer on the dedicated web page available on www.st.com.
- Install the package.

The easiest way is to add the tool path inside the environment variable (computer administration rights are required). If not, remember the path, as it is directly added in the project for compilation, in the post-build option.

7 STL: User APIs and state machines

7.1 User structures

The structures are defined in *stl_user_api.h*. It is forbidden to change the content of this file.

Structures detailed hereafter are a copy of the *stl_user_api.h* content:

```
typedef enum
{
  STL_OK = STL_OK_DEF, /* Scheduler function successfully executed */
  STL_KO = STL_KO_DEF /* Scheduler function unsuccessfully executed
                         (defensive programming error, checksum error). In this case
                          the STL_TmStatus_t values are not relevant */
} STL_Status_t; /* Type for the status return value of the STL function execution */
```

```
typedef enum
{
  STL_PASSED = STL_PASSED_DEF,  /* Test passed. For Flash/RAM, test is passed and end of
                                   configuration is also reached */
  STL_PARTIAL_PASSED = STL_PARTIAL_PASSED_DEF, /* Used only for RAM and Flash testing.
                                                  Test passed, But end of Flash/RAM
                                                  configuration not yet reached */
  STL_FAILED = STL_FAILED_DEF, /* Hardware error detection by Test Module */
  STL_NOT_TESTED = STL_NOT_TESTED_DEF, /* Initial value after a SW init, SW config,
                                          SW reset, SW de-init or value when Test Module
                                          not executed */
  STL_ERROR = STL_ERROR_DEF /* Test Module unsuccessfully executed (defensive programing
                               check failed) */
} STL_TmStatus_t; /* Type for the result of a Test Module */
```

```
typedef enum
{
  STL_CPU_TM1L_IDX,     /* CPU Arm Core Test Module 1L index */
  STL_CPU_TM7_IDX,      /* CPU Arm Core Test Module 7 index */
  STL_CPU_TMCB_IDX,     /* CPU Arm Core Test Module TMCB index */

  STL_CPU_TM_MAX        /* Number of CPU Arm Core Test Modules */
} STL_CpuTmxIndex_t;    /* Type for index of CPU Arm Core Test
                           Modules */
```

```
typedef enum
{
  STL_CRC_TABLE_NO_TZ = STL_CRC_TABLE_NO_TZ_DEF,   /* CRC Table idx when Trust Zone is
                                                      disabled */
  STL_CRC_TABLE_TZ_S = STL_CRC_TABLE_TZ_S_DEF,     /* CRC Table idx when Trust Zone is
                                                      enabled and subset is located in
                                                      Secure binary */
  STL_CRC_TABLE_TZ_NS = STL_CRC_TABLE_TZ_NS_DEF    /* CRC Table idx when Trust Zone is enable
d
                                                      and subset is located in non-Secure bin
ary */
} STL_CrcTableIdx_t;
```

```
typedef struct STL_MemSubset_struct
{
  uint32_t StartAddr;             /* start address of Flash or RAM memory subset */
  uint32_t EndAddr;               /* end address of Flash or RAM memory subset */
  STL_CrcTableIdx_t CrcTableIdx; /* Index of the CRC table (used by Flash TM)*/
  struct STL_MemSubset_struct *pNext; /* pointer to the next Flash or RAM memory subset
                                         - to be set to NULL for the last subset */
} STL_MemSubset_t;               /* Type used to define Flash
                                    or RAM subsets to test */
```

```
typedef struct
{
  STL_MemSubset_t *pSubset; /* Pointer to the Flash or RAM subsets to test */
  uint32_t NumSectionsAtomic; /* Number of Flash or RAM sections to be tested
                                 during an atomic test */
} STL_MemConfig_t; /* Type used to fully define Flash or RAM test configuration */
```

```
typedef struct
{
  STL_TmStatus_t aCpuTmStatus[STL_CPU_TM_MAX]; /* Array of forced status value
                                                  for CPU Test Modules */
  STL_TmStatus_t FlashTmStatus; /* Forced status value for Flash Test Module */
  STL_TmStatus_t RamTmStatus; /* Forced status value for RAM Test Module */
} STL_ArtifFailingConfig_t; /* Type used to force Test Modules status to a specific
                               value for each STL Test Module */
```

7.2 User APIs

The following APIs are defined in the *stl_user_api.h* file. It is forbidden to change the content of this file.

Caution: *For pointers defined by the user application and used as STL API parameters, the user application must set valid pointers, maintain pointer availability, and check the pointer integrity. The STL does not copy the pointer content, and directly accesses memory addresses defined by the application.*

This applies during overall STL execution, where, for example, the pointers content for flash and RAM configuration must be maintained because they are still used by the STL_SCH_run_xxx functions, even if these pointers are not part of the input parameters.

For more details about proper sequences of the *API* calls see Section 7.3: State machines and Test examples.

7.2.1 Common APIs (not test mode specific)

7.2.1.1 *STL_Status_t STL_SCH_Init*

Description: initializes the scheduler. It can be used at any time to re-initialize the scheduler (it resets all tests).
Declaration: `STL_Status_t STL_SCH_Init(void)`

Table 6. *STL_SCH_Init* input information

Allowed state(s)	Parameter(s)
CPU TMx: all Flash *TM*: all *RAM TM*: all	-

Table 7. *STL_SCH_Init* output information

STL_Status_t return value		Returned state
Value	Comment(s)	
STL_OK	Function successfully executed	*CPU* TMx: CPU_TMx_CONFIGURED Flash *TM*: FLASH_IDLE *RAM TM*: RAM_IDLE
STL_KO	Source of defensive programming error: • *STL* internal data corrupted	No state change

Additional information: there is no specific *CPU* initialization function for *CPU* test modules.

Note: This function uses hardware CRC as explained in Section 4.3.4: CRC resources.

7.2.2 CPU testing APIs

7.2.2.1 *STL_Status_t STL_SCH_RunCpuTMx*

Description: runs CPU test module x. Used only for single test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_RunCpuTMx(STL_TmStatus_t *pSingleTmStatus)` where TMx can be one of TM1L, TM7, or TMCB.

Table 8. *STL_SCH_RunCpuTMx* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
CPU_TMx_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 9. *STL_SCH_RunCpuTMx* output information

STL_Status_t return value		**pSingleTmStatus output*		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_PASSED	-	CPU_TMx_CONFIGURED
		STL_FAILED	-	
		STL_ERROR	Source of defensive programming error: • STL internal data corrupted • software is not executed with privileged level for CPU TM7 • software is not executed in thread mode for CPU TM7	
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • STL internal data corrupted	Not relevant	Value must not be used	No state change

7.2.3 Flash memory testing APIs

7.2.3.1 *STL_Status_t STL_SCH_InitFlash*

Description: initializes flash memory test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_InitFlash(STL_TmStatus_t *pSingleTmStatus)`

Table 10. *STL_SCH_InitFlash* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
FLASH_IDLE FLASH_INIT FLASH_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 11. *STL_SCH_InitFlash* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	FLASH_INIT
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

7.2.3.2 *STL_Status_t STL_SCH_ConfigureFlash*

Description: configures flash memory test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_ConfigureFlash(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pFlashConfig)`

Table 12. *STL_SCH_ConfigureFlash* input information

<table>
<tr><th rowspan="2">Allowed state(s)</th><th colspan="4">Parameter</th></tr>
<tr><th>Value</th><th colspan="3">Comment(s)</th></tr>
<tr><td rowspan="11">FLASH_INIT</td><td>*pSingleTmStatus</td><td colspan="3">See Caution</td></tr>
<tr><td rowspan="10">*pFlashConfig</td><td colspan="3">• This pointer contains the flash memory configuration
• See Caution</td></tr>
<tr><td>Field</td><td colspan="2">Comment(s)</td></tr>
<tr><td rowspan="5">*pSubset</td><td colspan="2">• Pointer to flash memory subset
• A section cannot overlap with the CRC area
• See Caution</td></tr>
<tr><td>Field</td><td>Comment(s)</td></tr>
<tr><td>StartAddr</td><td>• Start subset address in bytes
• Cannot be lower than ROM_START and higher than CRC_START address
• For further details see Section 4.1.3: Flash memory tests</td></tr>
<tr><td>EndAddr</td><td>• End subset address in bytes
• Cannot be lower than ROM_START and higher than CRC_START address
• Needs to be higher than StartAddr
• For further details see Section 4.1.3: Flash memory tests</td></tr>
<tr><td>*pNext</td><td>• Pointer to next flash memory subset
• Must be set to NULL for the last subset
• See Caution</td></tr>
<tr><td>NumSectionsAtomic</td><td colspan="2">• Number of flash memory sections to be tested during an atomic test
• Set to 1, as minimum (one section per test)
• If the value is higher than the number of sections in all subsets, all flash memory subsets are tested in one pass</td></tr>
</table>

Table 13. *STL_SCH_ConfigureFlash* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	**Comment(s)**	**Value**	**Comment(s)**	
STL_OK	Function successfully executed	STL_NOT_ TESTED	-	FLASH_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • STL internal data corrupted • State not allowed • Wrong configuration detected	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *pFlashConfig* = NULL • STL internal data corrupted	Not relevant	Value must not be used	No state change

*Note: When the return value set to STL_KO or *pSingleTmStatus set to STL_ERROR, the flash memory configuration is not applied.*

7.2.3.3 *STL_Status_t STL_SCH_RunFlashTM*

Description: runs flash memory test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_RunFlashTM(STL_TmStatus_t *pSingleTmStatus)`

Table 14. *STL_SCH_RunFlashTM* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
FLASH_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 15. *STL_SCH_RunFlashTM* output information

STL_Status_t return value		**pSingleTmStatus output*		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_PASSED	-	FLASH_CONFIGURED
		STL_PARTIAL_ PASSED	-	FLASH_CONFIGURED
		STL_FAILED	-	FLASH_CONFIGURED
		STL_NOT_ TESTED	All subsets are already tested	FLASH_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • STL internal data corrupted • State not allowed • Configuration corrupted	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

7.2.3.4 *STL_Status_t STL_SCH_ResetFlash*

Description: resets flash memory test (see Section 7.3: State machines).Declaration: `STL_Status_t STL_SCH_ResetFlash(STL_TmStatus_t *pSingleTmStatus)`

Table 16. *STL_SCH_ResetFlash* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
FLASH_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 17. *STL_SCH_ResetFlash* output information

STL_Status_t return value		**pSingleTmStatus output*		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_NOT_ TESTED	Configuration successfully applied	FLASH_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • *STL* internal data corrupted • State not allowed • Configuration corrupted	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

*Note: Once all subsets are tested, the user needs to reset the test module to perform the Flash memory test again. When a return value is set to STL_KO or *pSingleTmStatus set to STL_ERROR, the Flash reset is not applied.*

7.2.3.5 *STL_Status_t STL_SCH_DeInitFlash*

Description: deinitializes flash memory test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_DeInitFlash(STL_TmStatus_t *pSingleTmStatus)`

Table 18. *STL_SCH_DeInitFlash* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
FLASH_IDLE FLASH_INIT FLASH_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 19. *STL_SCH_DeInitFlash* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	FLASH_IDLE
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • STL internal data corrupted	Not relevant	Value must not be used	No state change

7.2.4 RAM testing APIs

7.2.4.1 *STL_Status_t STL_SCH_InitRam*

STL_Status_t STL_SCH_InitRam(STL_TmStatus_t *pSingleTmStatus)

Description: initializes RAM test (see State machines).

Table 20. *STL_SCH_InitRam* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
RAM_IDLE RAM_INIT RAM_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 21. *STL_SCH_InitRam* output information

STL_Status_t return value		**pSingleTmStatus output*		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_ERROR	-	RAM_IDLE
		STL_NOT_TESTED	-	RAM_INIT
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • STL internal data corrupted	Not relevant	Value must not be used	No state change

7.2.4.2 *STL_Status_t STL_SCH_ConfigureRam*

Description: configures *RAM* test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_ConfigureRam(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pRamConfig)`

Table 22. *STL_SCH_ConfigureRam* input information

Allowed state(s)	Parameter			
	Value	Comment(s)		
RAM_INIT	**pSingleTmStatus*	See **Caution**		
	**pRamConfig*	This pointer contains the *RAM* configuration. See **Caution**		
		Field	Comment(s)	
		**pSubset*	• Pointer to *RAM* subset • A subset cannot overlap with the *RAM* backup buffer if defined • See **Caution**	
			Field	Comment(s)
			StartAddr	• Start subset address in bytes • Start address must be 32-bit aligned • *RAM* subset must be inside *RAM* area • Cannot be lower than RAM_START and higher than RAM_END address
			EndAddr	• End subset address in bytes • Higher than StartAddr • Cannot be lower than RAM_START and higher than RAM_END address • Subset size (EndAddr – StartAddr) needs to be multiple of 2 * RAM_BLOCK_SIZE, 32 bytes • Subset cannot overlap with the *RAM* backup buffer , if defined
			**pNext*	• Pointer to next *RAM* subset • Must be set to NULL for the last subset • See **Caution**
		NumSectionsAtomic	• Number of *RAM* sections to be tested during an atomic test • Set to 1, as minimum (one section per test) • If the value is higher than the number of sections in all subsets, all *RAM* subsets are tested in one pass	

Table 23. ***STL_SCH_ConfigureRam*** **output information**

STL_Status_t **return value**		****pSingleTmStatus*** **output**		**Returned state**
Value	**Comment(s)**	**Value**	**Comment(s)**	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	RAM_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • *STL* internal data corrupted • State not allowed • Wrong configuration detected	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *pRamConfig* = NULL • STL internal data corrupted	Not relevant	Value must not be used	No state change

*Note: When the return value is set to STL_KO or *pSingleTmStatus set to STL_ERROR, the RAM configuration is not applied.*

7.2.4.3 *STL_Status_t STL_SCH_RunRamTM*

Description: runs *RAM* test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_RunRamTM(STL_TmStatus_t *pSingleTmStatus)`

Table 24. *STL_SCH_RunRamTM* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
RAM_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 25. *STL_SCH_RunRamTM* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_PASSED	-	RAM_CONFIGURED
		STL_PARTIAL_PASSED	-	RAM_CONFIGURED
		STL_FAILED	-	RAM_CONFIGURED
		STL_NOT_TESTED	All subsets are already tested	RAM_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • STL internal data corrupted • State not allowed • Configuration corrupted • The software is not executed with privileged level	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

7.2.4.4 *STL_Status_t STL_SCH_ResetRam*

Description: resets *RAM* test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_ResetRam(STL_TmStatus_t *pSingleTmStatus)`

Table 26. *STL_SCH_ResetRam* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
RAM_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 27. *STL_SCH_ResetRam* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_NOT_TESTED	Configuration successfully applied	RAM_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: • STL internal data corrupted • State not allowed • Configuration corrupted	No state change
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

Note: Once all subsets are tested, the user needs to reset the test module to perform the RAM test again.
*When the return value is set to STL_KO or *pSingleTmStatus set to STL_ERROR, the RAM reset is not applied.*

7.2.4.5 *STL_Status_t STL_SCH_DeInitRam*

Description: deinitializes *RAM* test (see Section 7.3: State machines).
Declaration: `STL_Status_t STL_SCH_DeInitRam(STL_TmStatus_t *pSingleTmStatus)`

Table 28. *STL_SCH_DeInitRam* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
RAM_IDLE RAM_INIT RAM_CONFIGURED	**pSingleTmStatus*	See **Caution**

Table 29. *STL_SCH_DeInitRam* output information

STL_Status_t return value		**pSingleTmStatus* output		Returned state
Value	Comment(s)	Value	Comment(s)	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	RAM_IDLE
STL_KO	Possible source of defensive programming error: • *pSingleTmStatus* = NULL • *STL* internal data corrupted	Not relevant	Value must not be used	No state change

7.2.5 Artificial-failing APIs

7.2.5.1 *STL_Status_t STL_SCH_StartArtifFailing*

Description: sets artificial-failing configuration and starts artificial-failing feature.
Declaration: `STL_Status_t STL_SCH_StartArtifFailing(const STL_ArtifFailingConfig_t *pArtifFailingConfig)`

Behavior: The next function calls are executed normally. If the *STL_Status_t* return value is set to STL_OK, the test module status (**pSingleTmStatus, *pTmListStatus*) is forced to a configured value. It is forced for each STL test module, even if not enabled. It applies to each STL internal test module (not user ones).

Table 30. *STL_SCH_StartArtifFailing* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
CPU TMx: • CPU_TMx_CONFIGURED Flash *TM*: • FLASH_IDLE • FLASH_INIT • FLASH_CONFIGURED *RAM TM* • RAM_IDLE • RAM_INIT • RAM_CONFIGURED	**pArtifFailingConfig*	-

Table 31. *STL_SCH_StartArtifFailing* output information

STL_Status_t return value	Comment(s)	Output	Comment(s)
STL_OK	Function successfully executed	No output parameter	No state change
STL_KO	Possible source of defensive programming error: • *pArtifFailingConfig* = NULL • configured values are not set for each test module • *STL* internal data corrupted		

7.2.5.2 *STL_Status_t STL_SCH_StopArtifFailing*

Description: stops the artificial-failing feature.
Declaration: `STL_Status_t STL_SCH_StopArtifFailing(void)`

Table 32. *STL_SCH_StopArtifFailing* input information

Allowed state(s)	Parameter(s)	
	Value	Comment(s)
CPU TMx: • CPU_TMx_CONFIGURED Flash TM: • FLASH_IDLE • FLASH_INIT • FLASH_CONFIGURED RAM TM • RAM_IDLE • RAM_INIT • RAM_CONFIGURED	-	-

Table 33. *STL_SCH_StopArtifFailing* output information

STL_Status_t return value	Comment(s)	Output	Comment(s)
STL_OK	Function successfully executed	No output parameter	No state change
STL_KO	Possible source of defensive programming error: • STL internal data corrupted		

7.3 State machines

Each *CPU* test module has its own state machine diagram linked to the *CPU* test *API*s.

Figure 11. State machine diagram - *CPU* test *API*s

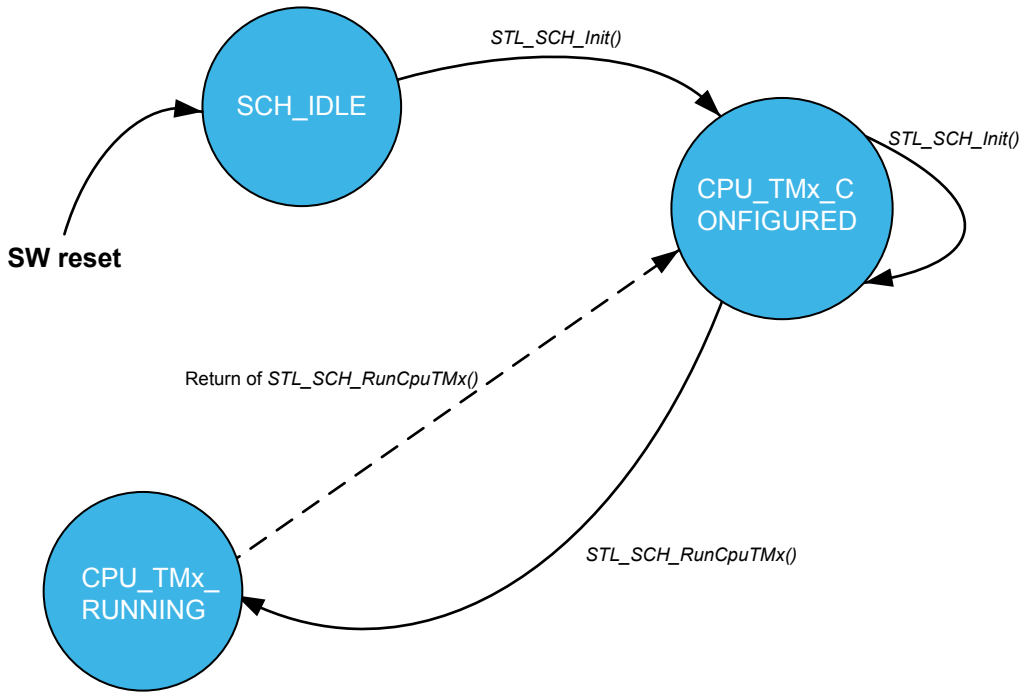
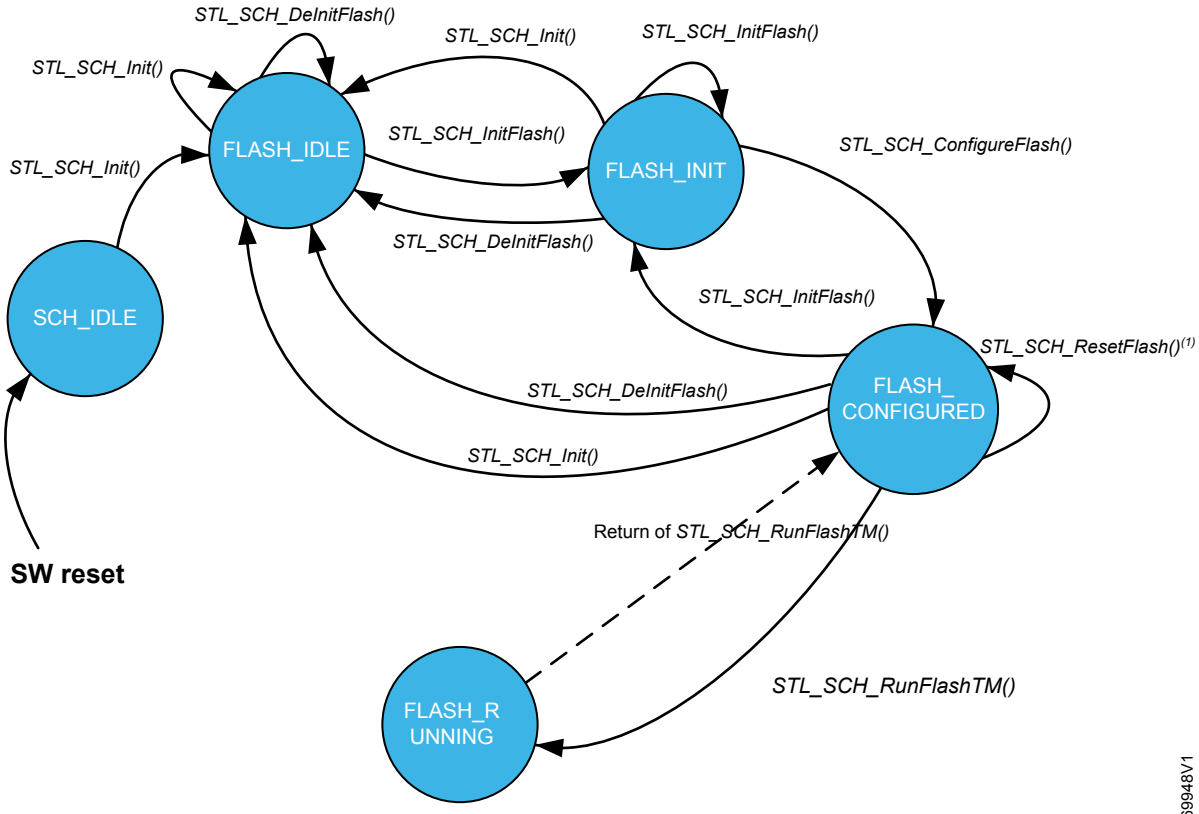
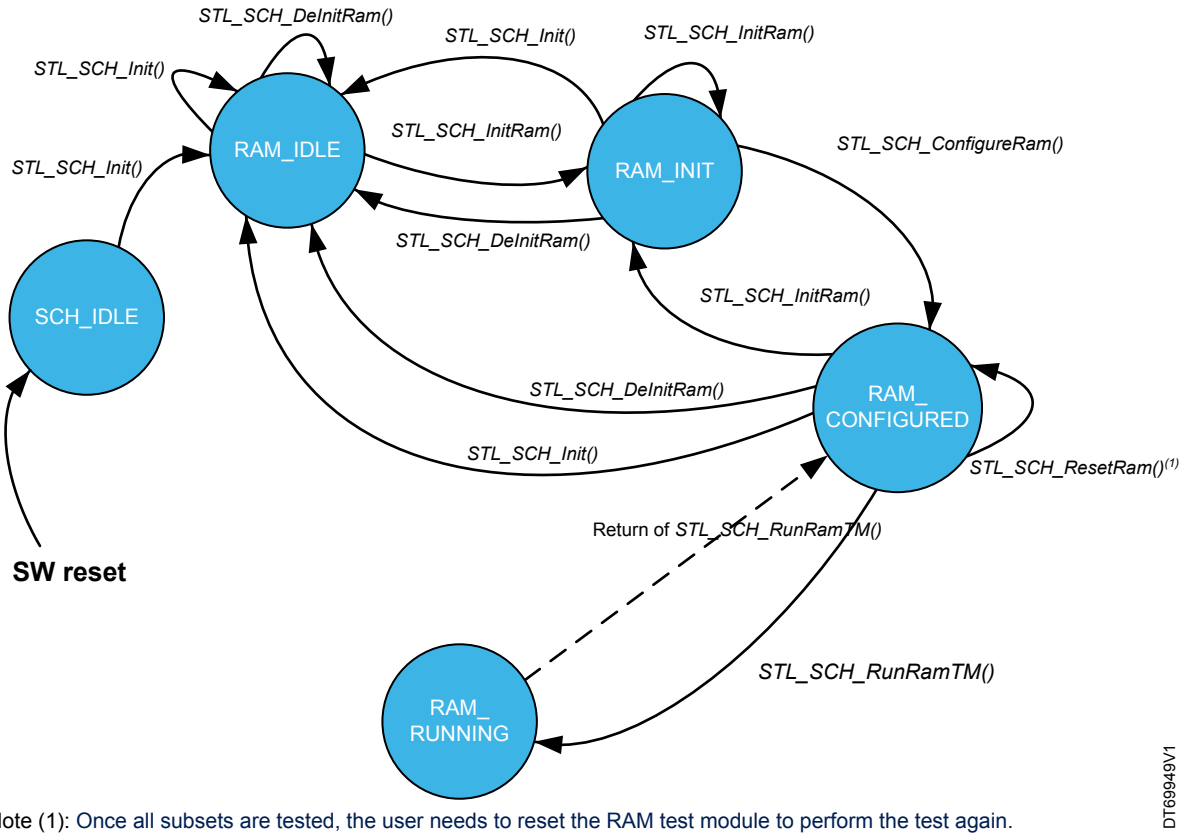


Figure 12. **State machine diagram - flash memory test *API*s**



Note (1): Once all subsets are tested, the user needs to reset the flash memory test module to perform the test again.

Figure 13. **State machine diagram - RAM test *APIs***



Note (1): Once all subsets are tested, the user needs to reset the RAM test module to perform the test again.

7.4 API usage

The user application must:

- Maintain the availability and integrity of pointers passed as parameters during the tests (the *STL* does not copy the pointer content, and directly accesses memory addresses defined by the application)
- Check the status of function return (*STL_Status_t*), before checking the test result (*STL_TmStatus_t* or *STL_TmListStatus_t*). See the example in the delivered applications.

7.5 User parameters

In addition to parameters set directly inside the *APIs*, a few parameters need to be customized in the file *stl_user_param_template.c* .They can be found in the code, with the following comments:

```
/* customisable */
```

Extract from *stl_user_param_template.c*:

```
/* FLASH configuration */
#define STL_ROM_NO_TZ_START_ADDR (0x08000000UL) /* customizable */ /* No TZ */
#define STL_ROM_TZ_S_START_ADDR  (0x0C000000UL) /* customizable */ /* TZ, Secure Flash */
#if defined (STM32U3C5xx)
#define STL_ROM_TZ_NS_START_ADDR (0x08100000UL) /* customizable */ /* TZ, Non-Secure Flash */
#define STL_ROM_NO_TZ_END_ADDR   (0x081FFFFFUL) /* customizable */ /* no TZ: 2 Mbytes */
#define STL_ROM_TZ_S_END_ADDR    (0x0C0FFFFFUL) /* customizable */ /* TZ, Secure: 1 Mbytes */
#define STL_ROM_TZ_NS_END_ADDR   (0x081FFFFFUL) /* customizable */ /* TZ, Non-Secure: 1 Mbyte
s */
#elif defined (STM32U385xx)
#define STL_ROM_TZ_NS_START_ADDR (0x08080000UL) /* customizable */ /* TZ, Non-Secure Flash */
#define STL_ROM_NO_TZ_END_ADDR   (0x080FFFFFUL) /* customizable */ /* no TZ: 1 Mbytes */
#define STL_ROM_TZ_S_END_ADDR    (0x0C07FFFFUL) /* customizable */ /* TZ, Secure: 512K */
#define STL_ROM_TZ_NS_END_ADDR   (0x080FFFFFUL) /* customizable */ /* TZ, Non-Secure:512K */
#elif defined (STM32U366xx)
#define STL_ROM_TZ_NS_START_ADDR (0x08040000UL) /* customizable */ /* TZ, Non-Secure Flash */
#define STL_ROM_NO_TZ_END_ADDR   (0x0807FFFFUL) /* customizable */ /* no TZ: 512K */
#define STL_ROM_TZ_S_END_ADDR    (0x0C03FFFFUL) /* customizable */ /* TZ, Secure: 256K */
#define STL_ROM_TZ_NS_END_ADDR   (0x0807FFFFUL) /* customizable */ /* TZ, Non-Secure: 256K */
#else
#error "please add your device ROM end address"
#endif
```

The customization depends upon the STM32 product and the user choice.

```
/* TM RAM Backup Buffer configuration */
…..
/* User shall locate the buffer in secure RAM */
/* The RAM backup buffer is placed in "backup_buffer_section". */
/* "backup_buffer_section" section is defined in scatter file */
```

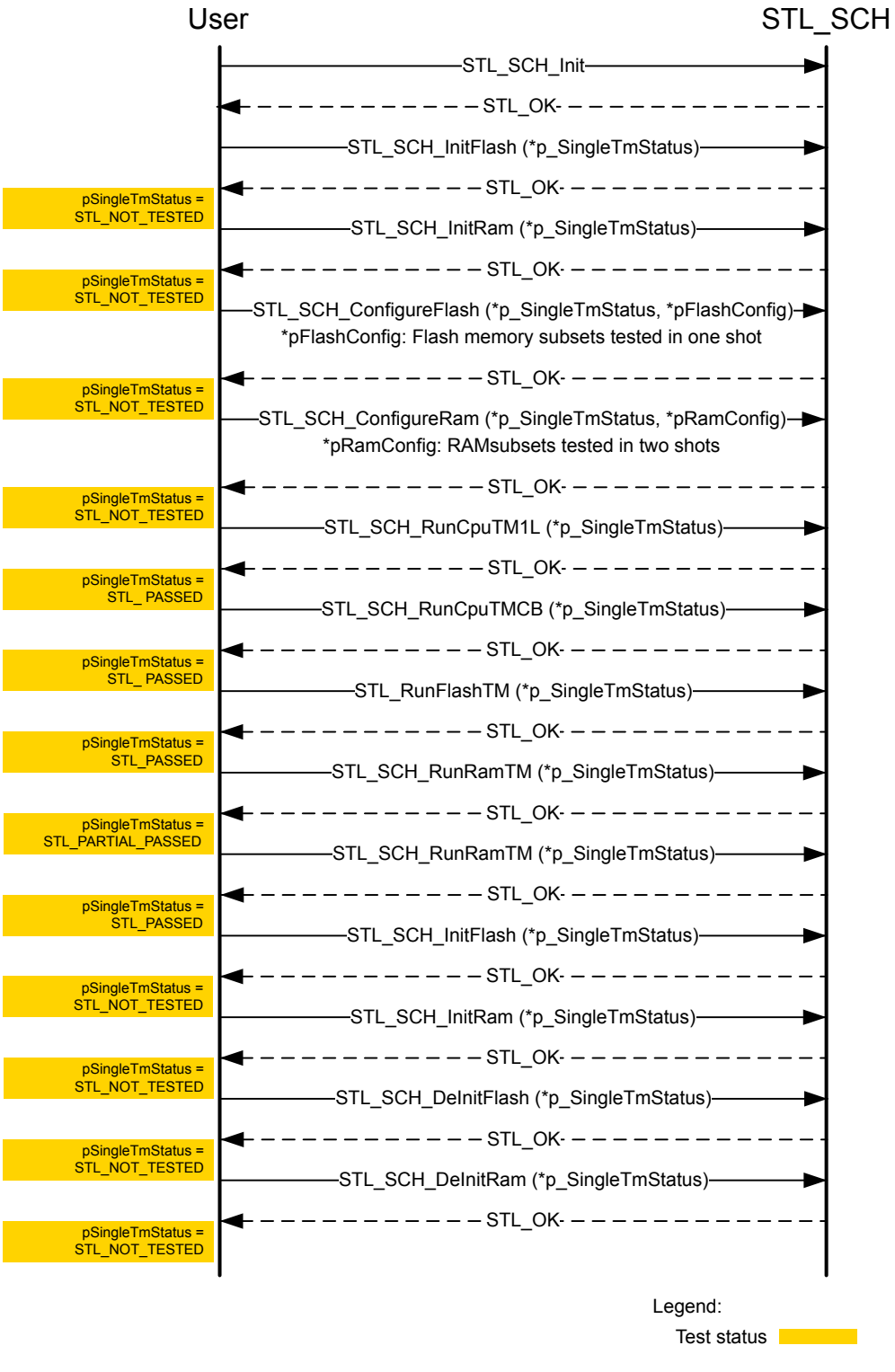
The remaining user parameters are defined by flags, and can be checked inside the files:

- *stl_user_param_template.c*: use of *RAM* backup buffer or not
- *stl_user_param_template.c*: use the locate *RAM* backup buffer in *RAM*. When TrustZone® is enabled, it must be in secure *RAM*.
- *stl_util.c*: use of software or hardware *CRC* computation
- *stl_stm32_hw_config.h*: if *CRC* hardware is used, choose the right CRC IP configuration according to the STM32 device

See Steps to build an application from scratch to check flag configuration.

7.6 Test examples

Figure 14. **Single testing example**



7.7 Details of testing examples

7.7.1 Flash memory single test example

Figure 15 shows a flash memory single test:

- Use of two flash memory subsets
- Use of functions
  - STL_SCH_RunFlashTM → only the flash memory test module is executed
  - STL_SCH_ResetFlash
- Function return value
- Flash memory test module result value: pSingleTmStatus → in this case, it contains the result of the flash memory test

7.7.2 RAM single test example

Figure 16 shows a RAM single test:

- Use of two RAM subsets
- Use of functions:
  - STL_SCH_RunRAMTM → only the RAM test module is executed
  - STL_SCH_ResetRam
- Function return value
- RAM Test Module result value: pSingleTmStatus → in this case, it contains the result of the RAM memory test

Figure 15. **Flash memory single test example**

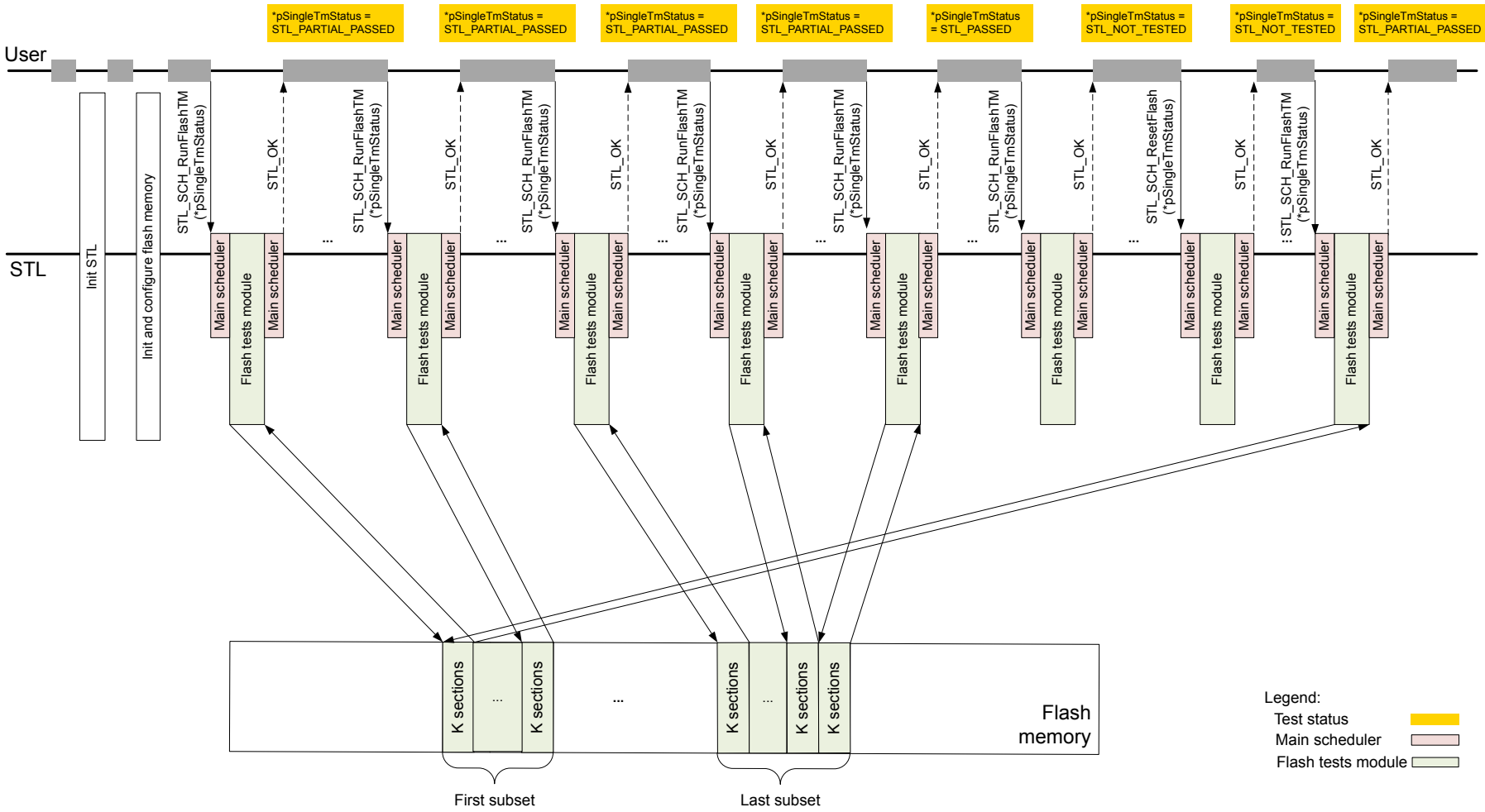
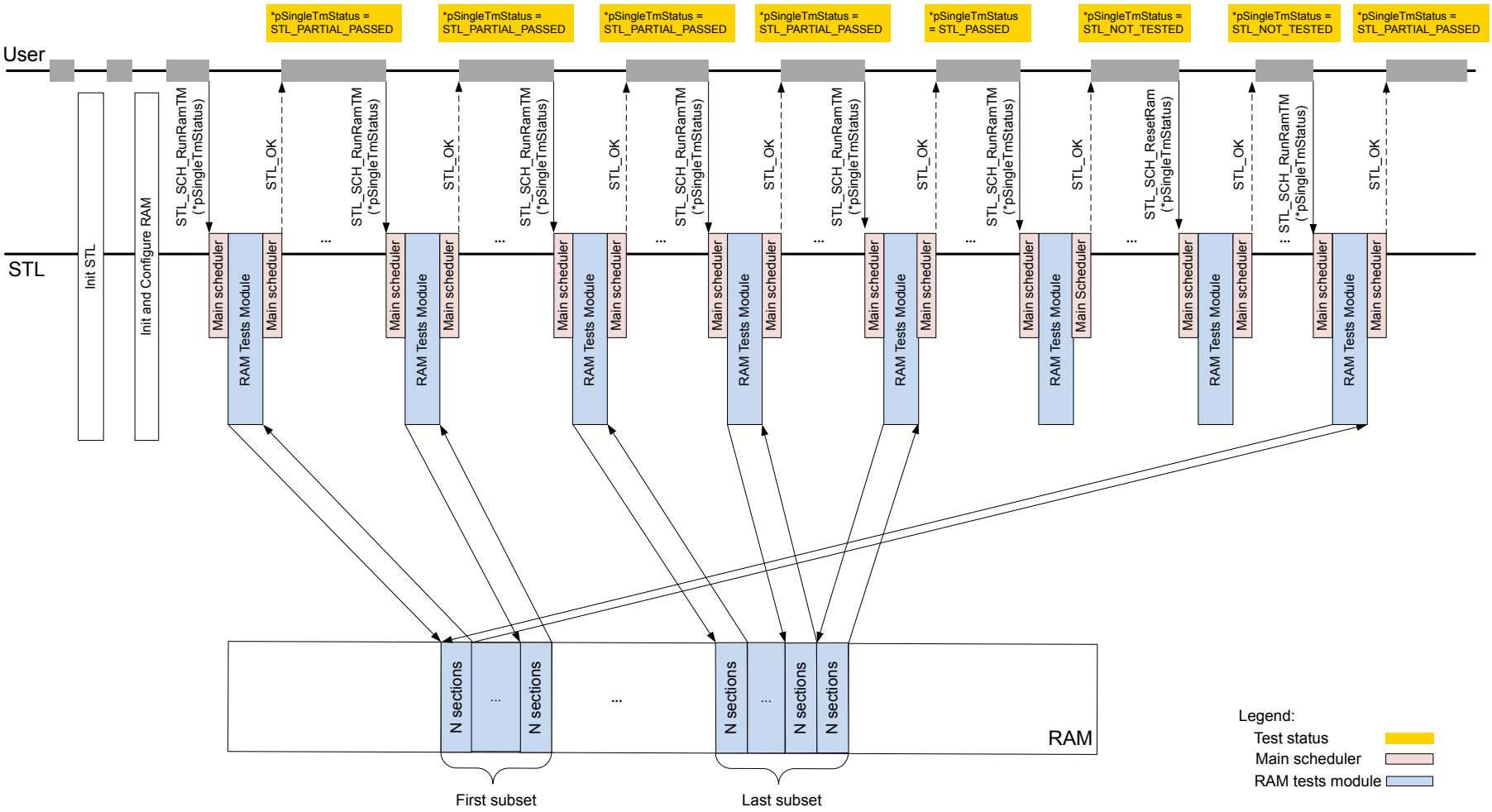


Figure 16. **RAM single test example**



8 STL: execution timing details

Refer also to the information in Section 4.2.1: STL execution timing summary.

The data in Table 34. Integration tests is obtained with the following test set-up:

- STL library compilation details, detailed in Section 5.5: Application: compilation process.
- Projects for performance tests compiled with IAR™ Embedded Workbench for Arm® (EWARM) toolchain v9.60.3
- Compiled SW configuration with:
  - HCLK clock set to 96 MHz
  - Flash latency set to four wait states
  - NUCLEO U3C5ZI-Q
  - ICache disabled

Table 34. Integration tests

Test	Duration (clock cycles)		Tested memory
	HW CRC	SW CRC	
STL_SCH_InitFlash()	686	673	-
STL_SCH_ConfigureFlash()	932	895	-
STL_SCH_RunFlashTM()	67370	258956	10240 bytes
STL_SCH_InitRam()	681	668	-
STL_SCH_ConfigureRam()	984	938	-
STL_SCH_RunRamTM()	11336641	11054125	131039 bytes
STL_SCH_RunCpuTMCB()	893	887	-
STL_SCH_RunCpuTM1L()	7944	7574	-
STL_SCH_RunCpuTM7()	2814	3197	-

9 Application-specific tests not included in ST firmware self-test library

The user must focus on all the remaining required tests covering application specific *MCU* parts not included in the ST firmware library:

- Test of analog parts (ADC/DAC, multiplexer)
- Test of digital I/O
- External addressing
- External communication
- Timing and interrupts
- System clock frequency measurement.

A valid solution for these components is strongly dependent on application and device-peripheral capability. The application must follow as precisely as possible the suggested testing principles from the very early stages of its design.

Very often this method leads to redundancy at both hardware and software levels.

Hardware methods can be based on:

- Multiplication of inputs and/or outputs
- Reference point measurement
- Loop-back read control at analog or digital outputs such as *DAC*, *PWM*, *GPIO*
- Configuration protection.

Software methods can be based on:

- Repetition in time, multiple acquisitions, multiple checks, decisions, or calculations made at different times or performed by different methods
- Data redundancy (data copies, parity check, error correction/detection codes, checksum, protocol)
- Plausibility check (valid range, valid combination, expected change, or trend)
- Periodicity and occurrence checks (flow and occurrence in time controls)
- Periodic checks of correct configuration (for example, read back the configuration registers).

9.1 Analog signals

Measured values must be checked for consistency and verified by measurements performed on other redundant channels. Free channels can be used for reading some reference voltages with testing of analog multiplexers used in the application. The internal reference voltage must also be checked.

Some STM32 microcontroller devices feature two (or even three) independent *ADC* blocks. To ensure the reliability of the results, perform several conversions on the same channel using two different *ADC* blocks for security reasons. The results can be obtained using either:

- Multiple acquisitions from one channel
- Compare redundant channels followed by an averaging operation.

Here are some tips for testing the functionality of analog parts at STM32 microcontroller devices.

ADC input pin disconnection

The ADC input pin disconnection can be tested by applying additional signal source on the tested pin.

- Some STM32 microcontroller devices feature internal pull-down or pull-up resistor activation facilities on the analog input. They can also feature a free pin with *DAC* functionality or a digital *GPIO* output. Any one of these pins can be used as a known reference input to the ADC.
- Some STM32 microcontroller devices feature a routing interface. This interface can be used for internal connection between pins to make:
  - testing loop-back
  - additional signal injection
  - duplicate measurement at some other independent channel.

Note: The user must prevent any critical voltage injection into an analog pin. This can happen when digital and analog signals are combined and different power levels are applied to analog and digital parts ($V_{DD} > V_{DDA}$).

Internal reference voltage and temperature sensor (V_{BAT} for some devices)

- Ratio between these signals can be verified within the allowed ranges.
- Additional testing can be performed where the V_{DD} voltage is known.

ADC clock

Measurement of the *ADC* conversion time (by timers) can be used to test the independent *ADC* clock functionality.

DAC output functionality

Free *ADC* channels can be used to check if the *DAC* output channel is working correctly.

The routing interface can be used when connecting the *ADC* input channel and the *DAC* output channel.

Comparator functionality

Comparison between known voltage and *DAC* output or internal reference voltage can be used for testing comparator output on another comparator input.

Analog signal disconnection can be tested by pull-down or pull-up activation on a tested pin and comparing this signal with the *DAC* voltage as reference on another comparator input.

Operational amplifier

Functionality can be tested forcing (or measuring) a known analog signal to the operational amplifier (*OPAMP*) input pin, and internally measuring the output voltage with the *ADC*. The input signal to the *OPAMP* can be also measured by *ADC* (on another channel).

9.2 Digital I/Os

Class B tests must detect any malfunction on digital I/Os, too. It could be covered by plausibility checks together with some other application parts. For example, change of an analog signal from the temperature sensor must be checked when heating/cooling digital control is switched on/off. Selected port bits can be locked by applying the correct lock sequence to the lock bit in the GPIOx_LCKR register. This action prevents unexpected changes to the port configuration. Reconfiguration is only possible at the next reset sequence in this case. In addition, the bit banding feature can be used for atomic manipulation of the SRAM and peripheral registers.

9.3 Interrupts

Occurrence in time and periodicity of events must be checked. Different methods can be used; one of them uses a set of incremental counters where every interrupt event increments a specific counter. The values in the counters are then cross-checked periodically with other independent time bases. The number of events occurred within the last period depends upon the application requirements.

The configuration lock feature can be used to secure the timer register settings with three levels controlled by the TIMx_BDTR register. Unused interrupt vectors must be diverted into a common error handler. Polling is preferable for non-safety relevant tasks if possible to simplify an application interrupt scheme.

9.4 Communication

Data exchange during communication sessions must be checked while including redundant information in the data packets. Parity, sync signals, CRC check sums, block repetition, or protocol numbering can be used for this purpose. Robust application software protocol stacks like TCP/IP give higher level of protection, if necessary. Periodicity and occurrence in time of the communication events together with protocol error signals has to be checked permanently.

The user can find more information and methods in product-dedicated safety manuals.

9.5 STL library extension capabilities

This framework version features a significantly easier and more flexible implementation than the previous versions of this *STL* library (see Section 1.2: Reference documents) which allows for an easier extension. Even with the new applied format, the framework keeps the same set of self-testing methods to comply with the IEC 60730 standard that is already implemented by previous versions of the library:

- Test of registers at *CPU TM*s
- 32-bit *CRC* calculation compatible with STM32 HW *CRC* unit at Flash memory *TM*
- March C test respecting the physical address order of the RAM *TM*

The main improvements of the new framework version are:

- Module oriented
- Supports partial testing
- Based on configuration and parametrizing structures
- No differentiation between startup and runtime test modules
- *CRC* calculation support based on a format provided by the STM32CubeProgrammer command-line feature
- Precompiled and fixed object code format of key generic modules
- No dependency of the generic modules execution on drivers or compilers
- Error handling includes reporting of defensive programing results
- Artificial failure control feature to verify the proper integration of the modules with no need for additional instrumentation code
- Easy extension by additional application-specific modules.

Note: The user can find examples of the possible extension of the library to older 4.0.0 versions of this firmware applied to former products. Some examples include clock cross-reference testing or automated setting of the end of the flash memory test based on forcing a specific EMPTY PADDING block placed at the end of binary area. In addition, there are testing methods for raising the robustness of the firmware, even though this is not specifically required by the standard (for example, stack boundary checks or functional tests of the embedded system of watchdogs applied to versions 2.x.x).

10 Compliance with IEC, UL, and CSA standards

The pivotal IEC standards are IEC 60730-1 and IEC 60335-1, harmonized with UL/CSA 60730-1 and UL/CSA 60335-1 starting from the 4th edition. Previous UL/CSA editions use references to the UL1998 standard in addition.

The standards are updated at regular intervals. The range of all the regulations collected in the standards is very large; the sections that concern the requirements for software self-tests of generic parts of microcontrollers is very specific. In most cases, the provided updates do not impact these specific parts of the standard at all. Therefore, an obsolete certification can still comply and stay valid for newer editions of the standard.

The relevant detailed conditions required are defined in:

- Annexes Q and R of the IEC 60335-1 norm
- Annex H of the IEC 60730-1 norm.

Three classes are defined by the IEC 60730-1:2010 H.2.22 they are:

- Class A: control functions that are not intended to be relied upon for the safety of the application.
- Class B: control functions that are intended to prevent an unsafe state of the controlled equipment. Failure of the control function does not directly lead to a hazardous situation.
- Class C: control functions that are intended to prevent special hazards such as explosion or which failure could directly cause a hazard in the appliance.

For a programmable electronic component applying a safety protection function, the IEC 60335-1 standard requires incorporation of software measures to control fault and error conditions specified in tables R.1 and R.2:

- Table R.1 summarizes general conditions comparable with requirements given for Class B level
- Table R.2 summarizes specific conditions comparable with requirements given for Class C level.

Requirements for Class B level software, which is the subject of this user manual, are defined to prevent hazards if another fault occurs elsewhere in the appliance. In this case, the self-test software is run on the appliance after a failure. An accidental software fault occurring during a safety-critical routine execution does not necessarily result in a hazard due to another applied redundant software procedure or hardware protection function required at this level.

There is no such hardware protection required in Class C level counting that whatever fault at safety-critical software can result in a potential hazard. To comply with this level, more robust testing is required than the one usually applicable to standard industrial microcontrollers like the STM32. An acceptable solution usually leads to the implementation of specific hardware redundancy at system level, like dual channel structures.

For more information on more stringent test methods, refer to the industrial documentation Section 1.2: Reference documents.

IEC 60730-1 defines the set of applicable architectures acceptable for the design of Class B control functions:

- Single channel with functional test. A single CPU executes the software control functions as required. A functional test is performed as the software starts. It guarantees that all critical features work properly.
- Single channel with periodic self-test. A single CPU executes the software control functions. Embedded periodic tests check the various critical functions of the system without impacting the performance of the planned control tasks.
- Dual channel (homogeneous or diverse) with comparison. The software is designed to execute control functions (identically or differently) on two independent CPUs. Both CPUs compare internal signals for fault detection when executing any safety-critical task.

Note: This structure is recognized to comply with Class C level also. A common principle is that whatever method complies with Class C automatically complies with Class B.

An overview of the methods applied by *STL* and their references to the standards are listed on the table below. The *STL* is focused on generic components of the microcontroller reused at all applications. The test of the other parts is under the end-user responsibility as their testing is mostly application specific and can be achieved effectively at the planning stage of the system design. Refer to Section 9: Application-specific tests not included in ST firmware self-test library for more information on how to handle these application-specific tests.

Table 35. IEC 60335-1 components covered by the X-CUBE-CLASSB library by methods recognized by IEC-60730-1

Component of Table R.1 (IEC 60335-1: Annex R)		Class B	References to IEC 60730-1: Annex H)	Fault/error	Safety method applied at X-CUBE-CLASSB	Note
1. CPU	1.1 CPU registers	X	H.2.16.5 H.2.16.6 H.2.19.6	Stuck at	Periodic run of the *STL* TM1L, TM7, and TMCB CPU test modules	Combination of functional and pattern tests of the *CPU* registers,(general-purpose R0-R12, special-purpose main and process stack pointers R13, program status APSR and CONTROL registers)(1)
	1.2 Instruction decoding and execution	N/A				Not required for Class B
	1.3 Program counter	X	H.2.18.10.2 H.2.18.10.4	Stuck at	N/A End-user responsibility	Logical and time slot program sequence monitoring, implementation of watchdogs
	1.4 Addressing	N/A				Not required for Class B
	1.5 Data path instruction decoding	N/A				Not required for Class B
2. Interrupt handling and execution		X	H.2.18.10.4 H2.18.18	No interrupt or too frequent interrupts	Handshake of results is applied at the interrupt associated with a clock cross-check measurement module	End-user responsibility for the other interrupts implemented at application
3. Clock		X	H.2.18.10.1 H.2.18.10.4	Wrong frequency	N/A End-user responsibility	Clock cross-check measurement between two independent clock sources
4. Memory	4.1 Invariable memory	X	H.2.19.3.1 H.2.19.3.2 H.2.19.8.2	All single bit faults	Periodic execution of the *STL* Flash memory TM test module	ECC enable under end-user responsibility(2)
	4.2. Variable memory	X	H.2.19.6 H.2.19.8.2	DC fault	Periodic execution of the *STL* RamTM test module	ECC or parity enable under end-user responsibility(2)
	4.3 Addressing (relevant for variable and invariable memory)	X	H.2.19.8.2	Stuck at	-	Tested indirectly by execution of the applied memory test modules ECC enable under end-user responsibility(2)
5. Internal data path	5.1 Data	X	H.2.19.8.2	Stuck at	-	
	5.2 Addressing	X	H.2.19.8.2	Wrong address	-	
6. External communication		X	-	-	N/A End-user responsibility	-
7. I/O periphery		X	-	-	N/A End-user responsibility	-
8. Monitoring devices and comparators		N/A				Not required for Class B
9. Custom chips		X	-	-	N/A	-

1. *CPU registers R14 (LR) and R15 (PC) are tested indirectly via defensive programming methods.*
2. *For availability and functionality of concrete embedded hardware safety feature, refer to the product user and safety manual.*

Revision history

Table 36. **Document revision history**

Date	Revision	Changes
14-Sep-2026	1	Initial release.

Contents

List of tables

List of figures

Glossary

ADC Analog-to-digital converter

AEABI Arm® embedded application binary interface

API Application programming interface

APSR CPU status register

BSP Board support package

Class B Middle level of regulations targeting safety for home appliances (UL/CSA/IEC 60730-1/60335-1)

CMSIS Common microcontroller software interface standard

CPU Central processing unit

CRC Cyclic redundancy check

DAC Digital-to-analog conveter

FPU Floating-point unit

GPIO General-purpose input/output

HAL Hardware abstraction layer

ICache Instruction cache of an Arm® core

IDE Integrated development environment - a software application that provides comprehensive facilities to computer programmers for software development

LL Low layer, low-layer

MCU Microcontroller unit

MPU Memory protection unit

MSP Main stack pointer

OPAMP Operational amplifier

PSP Process stack pointer

PWM Pulse-width modulation

RAM Random access memory

SDK Software development kit

STL Self-test library

TM Test module

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.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The 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.

The 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.

If the purchasers identify an ST product that meets their functional and performance requirements, but that is not designated for the purchasers’ market segment, the purchasers shall contact ST for more information.

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.