

STM32N6 series 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 the STM32N6 series microcontrollers that include an Arm® Cortex®-M55 core. Order code X-CUBE-CLASSB-N6.

Safety has an essential role in electronic applications. The level of safety requirements for components is steadily increasing and, manufacturers of electronic devices include many new technical solutions in their designs. Techniques for improving safety are continuously evolving, and are regularly incorporated into updated versions of the safety standards.

The current safety recommendations and requirements are specified in worldwide standards issued by various authorities. These include: the international electro-technical commission (IEC), Underwriters laboratories (UL), and the Canadian standards association (CSA) authorities.

Compliance, verification, and certification are the focus of the certification institutes. These include: the German TUV and VDE (mostly operating in Europe), and the UL and the CSA (targeting mainly the USA and Canadian markets).

Standards related to safety requirements have a very wide scope. These safety standards cover many areas such as: classification, methodology, materials, mechanics, labeling, hardware, and software testing. Here, the target is just compliance with the software requirements when testing programmable electronic components, which form a specific part of the safety standards. These requirements are exceptionally subject of any change when a new upgrade of the standard is released. Also, there is significant similarity across commonly oriented safety standards that concern the testing of generic parts of microcontrollers, such as the *CPU* or memories.

The library presented in this document is based on a partial subset of testing modules developed and applied by ST to satisfy the stringent IEC 61508 industrial safety standard requirements. These modules are adapted to fulfill the IEC 60730 standard targeting household safety. That is why this new library adopts a different delivery format to that was used for previous releases. This format is derived from the industrial safety library, which is currently delivered as a black box pre-compiled object with no sources but with a clear outer interface definition. The advantage of this immutable solution is that it is compilation tool-chain agnostic. It is also independent of any other firmware such as *HAL*, *LL*, or *CMSIS* layer. This solution prevents unexpected compilation results when source code files previously verified on older versions of the library are re-compiled later by any newer compiler version or combined with the latest firmware drivers. This is generally a common practice.

Table 1. Applicable product

Part number	Order code
X-CUBE-CLASSB	X-CUBE-CLASSB-N6



1 General information

1.1 Purpose and scope

This X-CUBE-CLASSB-N6 expansion package provides application independent software to comply with the UL/CSA/IEC 60730-1 safety standard. The UL/CSA/IEC 60730-1 safety standard targets the safety of automatic electrical controls used in association with household equipment and similar electronic applications.

The main purpose of this software library is to facilitate and accelerate:

- user software development
- certification processes for applications which are subject to the associated requirements and certifications



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 test library, self-test library, available in the X-CUBE-CLASSB-N6 expansion package (and associated to this manual), `STL_Lib.a` file, is V4.0.0.

1.2 Reference documents

- [1] UM331, STM32N6 series safety manual dedicated for applications targeting industrial safety
- [2] AN4435, Guidelines for obtaining UL/CSA/IEC 60730-1/60335-1 Class B certification in any STM32 application dedicated to older versions of this library

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 \(STM32CubeWiSEbe, STM32CubeWiSEce, STM32CubeWiSEcg, STM32CubeWiSEere, STM32CubeWiSE8e\)](#), 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 STM32CubeN6 for the STM32N6 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, USB PD, video encoder API, and OpenBLRTOS, USB, and graphics
 - 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 STMicroelectronics. The aim is to provide the implementation of a relevant subset of safety mechanisms required by the "Class B" related safety standards applicable to STM32N6 series microcontrollers. The *STL* is an *HAL / LL* independent library, dedicated to these microcontrollers. The *STL* is a compilation tool chain-agnostic, so any standard C-compiler can compile it.

The *STL* is an autonomous software. It 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 interface definitions and the user parameter settings.

3.1 Architecture overview

The *STL* implements tests required by UL/CSA/IEC 60730-1 for the Arm® Cortex®-M55 CPU core, and the volatile and nonvolatile memories embedded in the product.

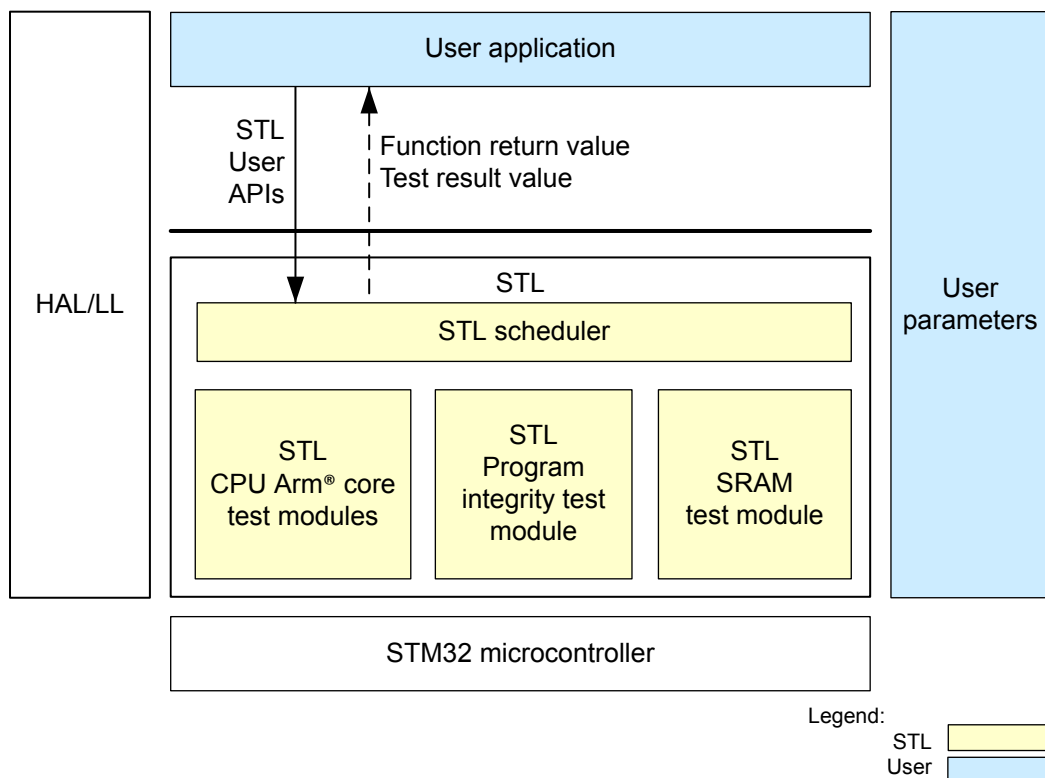
As shown in the figure below, a system architecture with an end-user application integrating the *STL* is composed of:

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

The *STL* status information returned to the user application at API level (summarized in Table 2) is:

- Function return value collects result of internal defensive programming checks.
- Test module result value stores the test result information. This partially corresponds to internal status of the module (see Section 7.3: State machines).

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.

3.2 Supported products

The *STL* runs on the following STM32N6 series microcontrollers:

- STM32N645xx
- STM32N647xx
- STM32N655xx
- STM32N657xx

4 STL description

This section describes basic information on the functionality and performance of the *STL*. The section also summarizes restrictions and mandatory actions to be followed by the end user.

4.1 STL functional description

Some test modules can temporarily mask interrupts. For more details, refer to [Section 4.2.5: STL interrupt masking time](#) and to [Section 4.3.6: Interrupt management](#).

4.1.1 Scheduler principle

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

The main scheduler:

- Must be initialized before being used
- Manages:
 - The initialization and deinitialization of the applied test modules
 - The configuration of the applied test modules
 - The reset of the applied test modules.
- Controls the execution of an applied test sequence (*API* calls)
- Manages "artificial failing" used for user debug and integration tests.
- Ensures the integrity of critical internal data structures via their specific checksums.

The scheduler controls the execution of the following tests:

- *CPU* tests: no specific initialization or configuration procedures of the *CPU* test module are required before any *CPU* test execution (see [Section 7.2: User APIs](#) and [Figure 11](#)).
- Program integrity tests operate on the content of the dedicated user configuration structures defining subsets of the memory to be tested (see [Section 7.1: User structures](#)). These structures must be filled by the end user and the content maintained during both configuration and execution of the program integrity test. The test module initialization and configuration procedures are mandatory before any program integrity test execution, see [Section 7.2: User APIs](#) and [Figure 12](#).
- *RAM* memory tests operate on the content of the dedicated user configuration structures defining subsets of the memory to be tested (see [Section 7.1: User structures](#)). These structures must be filled by the end user and the content maintained during both configuration and execution of the *RAM* test. The test module initialization and configuration procedures are mandatory before *RAM* test execution see and [Figure 13](#)

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

The user application has to consider all the returned information from the *STL*, provided via a specific predefined data structure collecting status information. See details in the following table.

Table 2. STL return information

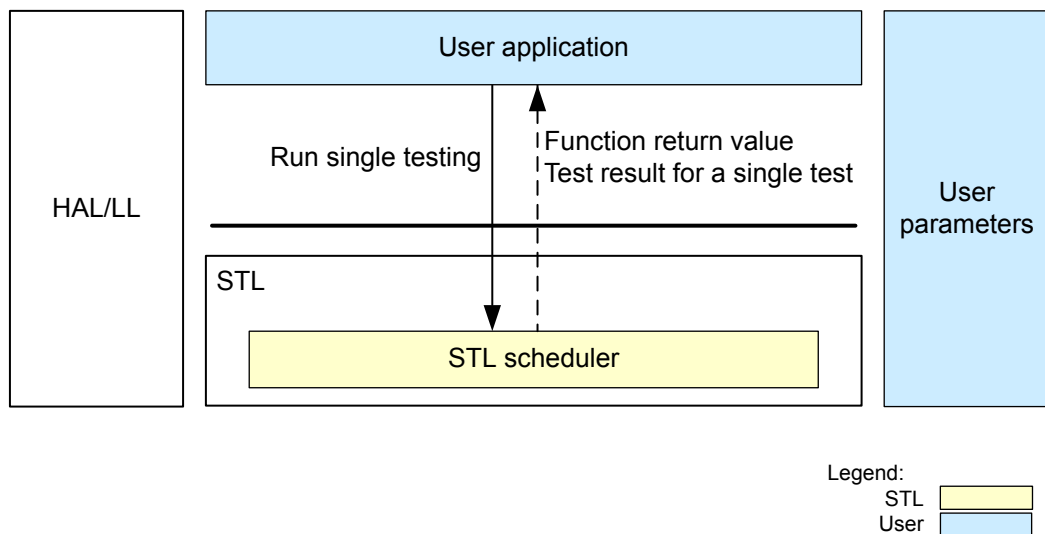
STL information	Value	Description
Function return value ⁽¹⁾	STL_OK	Scheduler function successfully executed
	STL_KO	Scheduler defensive programming error (in this case the test result is not relevant)
Test module result value ⁽²⁾	STL_PASSED	Test passed
	STL_PARTIAL_PASSED	Used only for memory testing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

1. Refer to *STL_Status_t* definition in Section 7.1: User structures.

2. See *STL_TmStatus_t* in Section 7.1: User structures.

The user application repeatedly applies the call control scheme illustrated in the following figure to program a sequence of API function calls and so handle the order of the test modules execution.

Figure 2. Single test control call architecture



Scheduler and interrupts

The scheduler can be interrupted at any time.

4.1.2 CPU Arm® core tests

The STL includes the CPU test modules listed below, together with a generic description (for information only) of the test capability:

- TM1L: implements a light pattern test of general-purpose registers
- TM7: implements the pattern and functional tests of both stack pointers: *MSP*, and *PSP*
- TMCB: implements functional test of the *APSR* status register.

Caution: *The STL CPU tests are partitioned in separated test modules. This is not intended to allow partial execution of the overall available CPU TMs. It is intended as a support feature to allow better CPU test scheduling in the end-user applications, for example timing constraints. By default, all available TMs are assumed to be executed.*

CPU Arm® core tests and interrupts

The CPU test modules are interruptible at any time. The **TM7** one only applies masking interrupt during the smallest data granularity time. Refer to [Section 4.3.6: Interrupt management](#) for detailed information on **CPU TM7** interrupt management.

4.1.3 Program integrity tests

Principles

The purpose of program integrity tests is to test the integrity of the Arm® Cortex® M55 CPU program image stored in SRAM, as read only memory. It is essential that the program binary remains unchanged to ensure the data integrity of the test result.

The following structures must be respected to provide correct configuration of the program integrity test.

- Integrity area: a continuous area of read only code and read only data (see [Figure 2](#)). The STL manages only one integrity area.
- Block: a continuous area of 4 bytes (`INTEGRITY_BLOCK_SIZE`), hard coded by *STL*.
- Section: a continuous area of 1024 bytes (`INTEGRITY_SECTION_SIZE`), hard coded by the *STL*. This has no link with the SRAM memory physical sector. The memory is partitioned in sections. The first section starts at the first address of the memory, and the following sections are contiguous with each other. The user must ensure proper calculation and placement of the *CRC* checksum for each section that is to be checked during the memory integrity test.
- Binary (named 'user program' in [Figure 4](#)): a continuous 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](#) below).
- Subset: a continuous area of contiguous sections defined by the user. 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 the beginning of a section. It can only include sections with the corresponding precalculated *CRC* values. When the last section of a subset is the last part of the binary, the section may be incomplete. 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.

The subset is calculated as follows:

$$\text{Subset size} = K * \text{INTEGRITY_SECTION_SIZE} + L * \text{INTEGRITY_BLOCK_SIZE}$$

where:

- K is an integer ≥ 0 .
- L is an integer, $0 \leq L < (\text{INTEGRITY_SECTION_SIZE} / \text{INTEGRITY_BLOCK_SIZE})$ when $L > 0$ the last section of a binary is incomplete.

The user application defines single or multiple subsets as well as their associated test sequences.

The *STL* implements a test of the program integrity with the following principles (based on actual content of the user configuration structures):

- Tests are performed on sections of one or more subsets 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 image is copied into SRAM).

The mandatory steps (for the user application) to perform the program integrity test are:

- Test initialization
- Configuration of one or more subsets
- Execution of the test.

Once all subsets are tested, the user needs to reset the program integrity 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. In this case, deinitialize, initialize and reconfigure the test module test prior to running it again.

Expected CRC precalculation

The program integrity test is based either on the embedded hardware *CRC* calculation unit or software *CRC* computation algorithm, which is configurable by a flag. The default configuration is with the *CRC* hardware unit. To use the software *CRC*, the flag `STL_SW_CRC` must be enabled as defined in step 3 in [Section 5.5.2: Steps to build an application from scratch](#). The *CRC* is a 32-bit *CRC* compliant with IEEE 802.3.

Part of the integrity area is reserved for the *CRC* dedicated area, the size of which depends on the integrity area size. This area has a field format where each integrity area 1 KB 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. This is shown in [Figure 3](#).

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 the section size. In that 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 end of each user program area must be 32-bit aligned.
- Depending on total integrity area size and on user program size, last program data and first *CRC* data may be both stored in the same integrity area section (without any overlap). In that case, the *CRC* must be computed on the user program data only, see example 3 in [Figure 4](#).

ST CRC tool information

ST provides a *CRC* precalculation tool. This tool is 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. Program integrity test: CRC principle

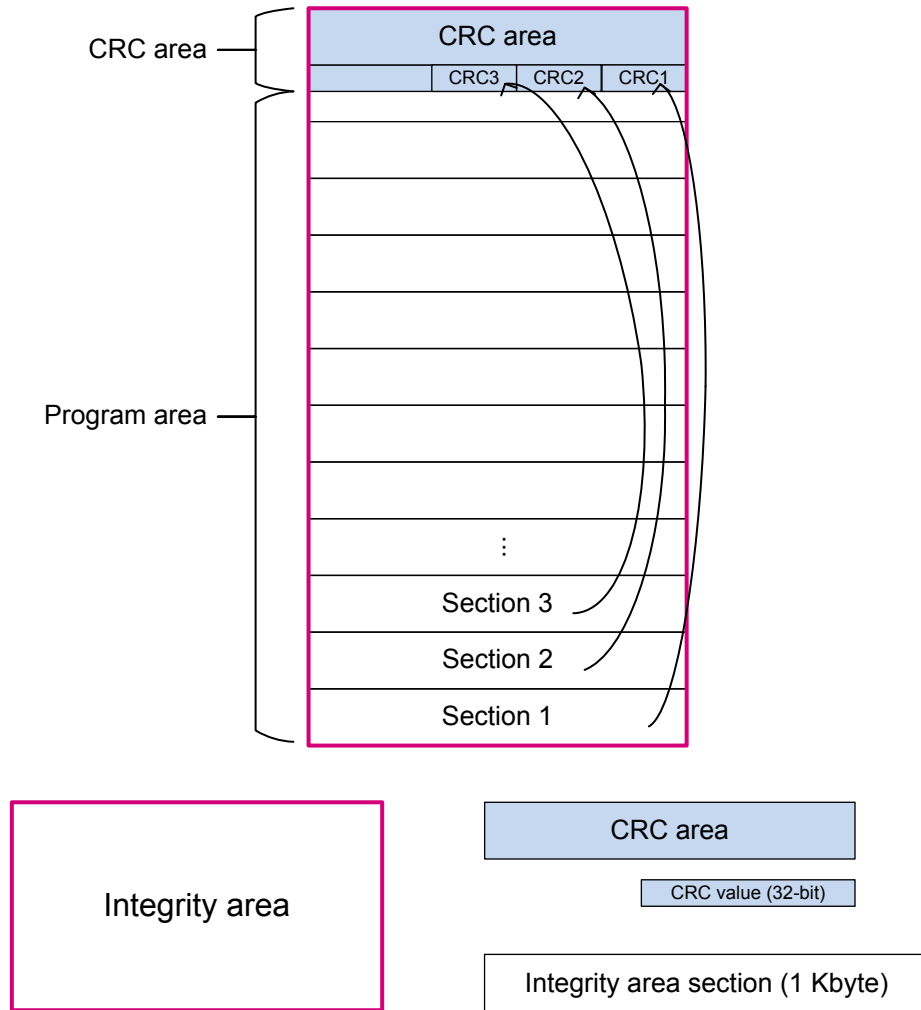
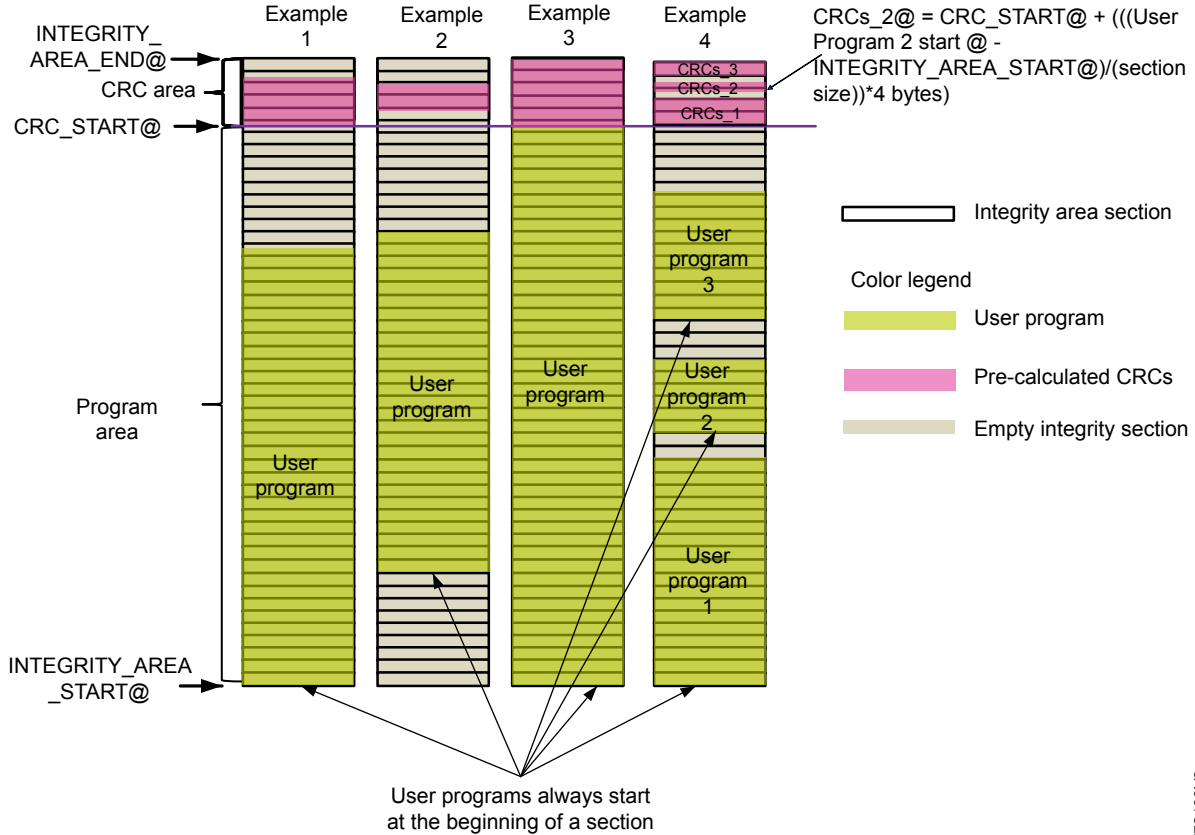


Figure 4. Program integrity test: CRC use cases versus program areas



DT72483V3

Use case descriptions illustrated in Figure 4:

- Example 1: the user program starts at the INTEGRITY_AREA_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 INTEGRITY_AREA_START. The 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 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.

CRC start address computation:

- Real calculation:

$$\text{CRC_START address} = (\text{uint32_t})(\text{INTEGRITY_AREA_END} - 4 * (\text{INTEGRITY_AREA_END} + 1 - \text{INTEGRITY_AREA_START}) / (\text{INTEGRITY_SECTION_SIZE} + 1));$$
 with INTEGRITY_SECTION_SIZE = 1024
- Textual translation:

$$\text{CRC_START} = \text{INTEGRITY_AREA_END} - (\text{CRC size in bytes}) * (\text{number of the memory sections}) + 1$$

Program integrity test and interrupts

Program integrity TM is interruptible at any time.

4.1.4 RAM tests

Principles

The RAM test concerns the embedded SRAM memories of STM32N6 series.

The following structures must be respected to provide correct configuration of the RAM test.

- Block: a continuous area of 16 bytes (RAM_BLOCK_SIZE), hard coded by the STL (no link with the memory physical sectors).
- Section: a continuous area of 128 bytes (RAM_SECTION_SIZE), hard coded by the STL.
- Subset: a continuous area, with the size being a multiple of two blocks and with a 32-bit aligned start address. 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 $0 \leq M < 4$, when $M > 0$, the size of the subset is not aligned with multiple section size.
 The user application defines single or multiple subsets as well as their associated test sequences.

The STL implements a RAM memory test with the following principles (based on actual content of the user configuration structures):

- 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 the March C- algorithm

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

- Initialization of RAM test
- Configuration of one or more RAM subsets
- Execution of the RAM test

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

In the case of an STL_ERROR / STL_FAILED test result, the test module is stuck in the failed memory subset. In this case, deinitialize, initialize and reconfigure the RAM prior to running the test again.

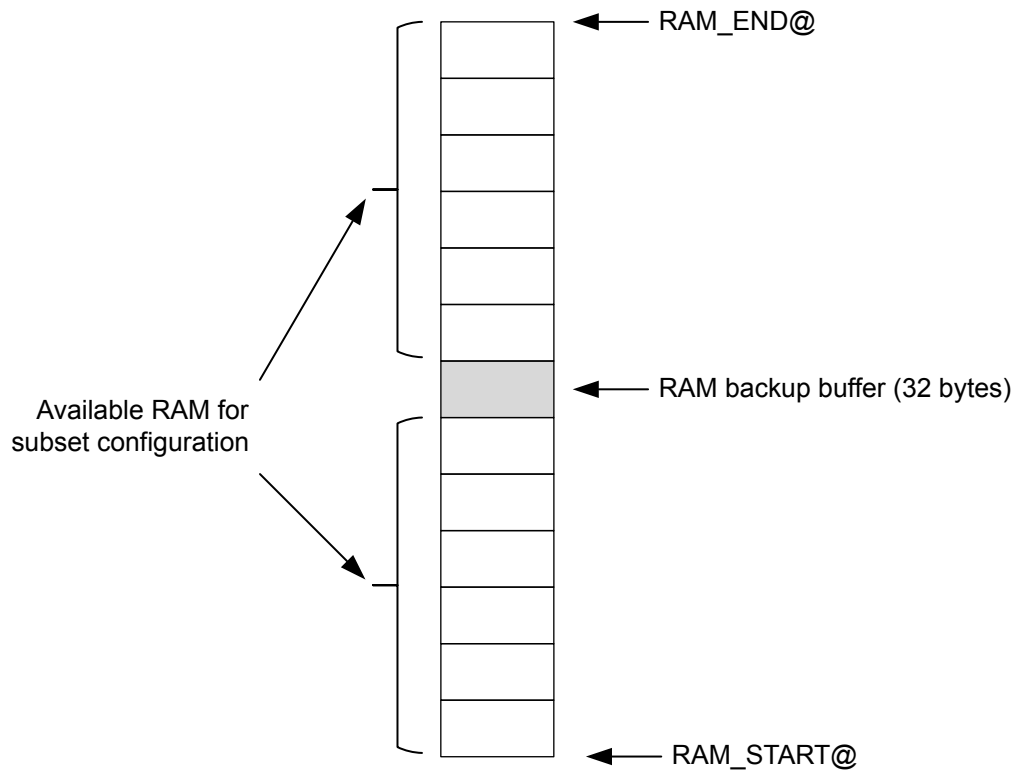
RAM test and interrupts

The RAM TM is interruptible at any time. The RAM TM masks interrupts during the smallest data granularity time (see [Section 4.2.5: STL interrupt masking time](#)). This occurs when the STM32 interrupts and Cortex® exceptions with configurable priority are temporarily masked by default. Refer to [Section 4.3.6: Interrupt management](#) for detailed information on interrupt management during RAM March-C tests.

March C- test principle and memory backup principle

The RAM test is based on a March C- algorithm where memory is overwritten by specific patterns and then read back in specific orders. Due to this principle, the RAM test must not be performed on the parts of the RAM area storing program images otherwise, the overall software behavior becomes unpredictable. To restore the initial memory content, a backup process is enabled and performed by default. The backup process can be optionally disabled if it is not required. Refer to [Section 4.3.12: RAM backup buffer](#) for detailed information on the buffer control and allocation.

Figure 5. RAM test: usage



DT49050V1

4.2 STL performance data

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

- STL library compilation details, described in Application: compilation process.
- Projects for performance tests are compiled with IAR Embedded Workbench® for Arm® (EWARM) toolchain v9.60.2
- Compiled software configuration with:
 - CPU clock set to 600 MHz
 - STM32 NUCLEO-N657X0-Q board MB1940 Rev C

4.2.1 STL execution timings

A summary of the STL execution timings is shown in the following table. The measurements for each API are detailed in Section 9: STL: execution timing details.

Table 3. STL execution timings, clock at 600 MHz

Tested module	Conditions		Result in μ s
CPU	TM1L, TM7, TMCB		9
Program integrity	Default configuration (STL_SW_CRC not enabled):	1 Kbyte tested	8
		17 Kbytes tested	114
	STL_SW_CRC enabled:	1 Kbyte tested	20
		17 Kbytes tested	298
RAM	Default configuration (STL_ENABLE_IT not enabled):	128 bytes tested at AXIM RAM	503
		128 bytes tested at DTCM RAM	9
		4 Kbyte tested at AXIM RAM	2204
		4 Kbyte tested at DTCM RAM	1199

4.2.2 STL code and data size

The STL code and data sizes are detailed in the following table.

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

Configuration	Module	Program integrity code	Program integrity RO-data	R/W data
STL_SW_CRC not enabled	stl_user_param_template.o	-	6	44
	stl_util.o	286 ⁽¹⁾	-	8
	stl_lib.a	5728	1454	164
STL_SW_CRC enabled	stl_user_param_template.o	-	6	44
	stl_util.o	150 ⁽¹⁾	-	4
	stl_lib.a	5728	1454	164

1. When the software CRC calculation is applied, there are less code / functions compiled (the CRC hardware initialization is missing), consequently the object code size is reduced compared to the hardware CRC.

4.2.3 STL stack usage

The minimum stack-available space required by the STL to execute available APIs is 256 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

The STM32 interrupts, and Cortex[®] exceptions with configurable priority, are masked multiple times by the STL during the execution of CPU TM7 and RAM tests. As shown in the following table, the maximum interrupt masking time is obtained for a RAM test.

Table 5. STL maximum interrupt masking information

Tested module	Duration (max) in clock cycles	Comments
DTCM RAM	2864 clock cycles	Each execution of the <code>STL_SCH_RunRamTM</code> function performs a series of interrupt masking during partial steps of the test at the following time durations: <ul style="list-style-type: none"> • 2864 clock cycles for the RAM backup buffer • 1804 clock cycles for the 1st RAM block to be tested • 1790 clock cycles for each middle RAM block to be tested⁽¹⁾ • 2080 clock cycles for the last part RAM block to be tested
AXIM RAM	234417 clock cycles	Each execution of the <code>STL_SCH_RunRamTM</code> function performs a series of interrupt masking during partial steps of the test at the following time durations: <ul style="list-style-type: none"> • 234 417 clock cycles for Disable DCache • 52 091 clock cycles for Enable DCache • 4707 clock cycles for the RAM backup buffer • 3747 clock cycles for the 1st RAM block to be tested • 4078 clock cycles for each middle RAM block to be tested • 4008 clock cycles for the last part RAM block to be tested
CPU TM7	606 clock cycles	Each execution of the function <code>STL_SCH_RunRamTM CPU TM7</code> performs two interrupt masking intervals at the time durations 606 and 457 clock cycles

1. Number of RAM blocks (multiple of two `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](#)).

4.2.6 Data cache performance impact

This section applies in cases where the DCache is enabled by the user and AXI RAM is used, but not when TCM is applied to the test.

By design, the *STL* performs a DCache flush each time the RAM test is invoked, so the related performance loss is linked to the frequency of the atomic execution of the RAM test. This must be taken into account by the end user to define the size of RAM to be tested in an atomic way.

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 issues. Application must respect call convention in particular on register save and restore commonly.

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

4.3.1 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) else these TMs return `STL_ERROR`.

4.3.2 RCC resources

During *STL* execution, the RCC is configured to always provide a clock to the *CRC* hardware unit during *STL* initialization and optionally during *STL* program integrity test module execution. This means that:

- When the *STL* returns, it restores the user RCC clock setting (enabled or disabled) for the *CRC* unit.
- The user application should be careful when configuring the RCC during *STL* execution by saving/restoring the *STL* settings.

4.3.3 CRC resources

The STM32 *CRC* hardware unit is used during *STL* execution in two different cases:

- During execution of *STL* initialization (function `STL_SCH_Init`), the use of the CRC hardware unit in this phase cannot be modified by the application software, so the `STL_SW_CRC` flag has no impact during execution of the `STL_SCH_Init` function.
- During execution of the program integrity test module, the application can choose between hardware and software calculation of the CRC checksums 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 must save any specific CRC unit configuration. The user configuration has to be restored after the *STL* execution.
- During the *STL* execution, the hardware unit is configured and used for *STL* needs (the user application must save/restore the *STL* settings when using the unit while interrupting the *STL* execution).

4.3.4 Bit Q of APSR

CPU TMCB execution sets bit Q (DSP overflow and saturation flag) of the *APSR*. The user application must take this into account when using .

4.3.5 Bits GE of APSR

During the *CPU* TMCB test module execution, the GE bits of the *APSR* can be set. They are always set to 0, in normal circumstances, at the end of this test.

4.3.6 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 the *CPU* TM7 execution within smallest data granularity (a few instruction blocks), except if the user application activates the `STL_ENABLE_IT` compilation switch (see [Section 5.5.2: Steps to build an application from scratch](#)). If the `STL_ENABLE_IT` flag is activated, the correct *STL* CPU TM7 behavior is not guaranteed. The end user is responsible for managing interferences between the *STL* and its application software that could lead the *STL* to generate false test error reporting or not to detect hardware failures.

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 the `STL_ENABLE_IT` compilation switch (see [Section 5.5.2: Steps to build an application from scratch](#)).

If the `STL_ENABLE_IT` flag is activated:

- The correct *STL* *RAM* test behavior is not guaranteed, as the application may overwrite the *STL*-tested *RAM* content during its interrupt treatment. The end user is responsible for managing interferences between the *STL* and its application software that could lead the *STL* to generate false *RAM* test error reporting.
- The behavior of the user application software can be compromised. Wrong data may be read or used from *RAM* locations that are being modified by the *STL* March C- test.

Interrupt and general purpose registers

During *STL* execution, the application must save and restore the general-purpose registers in the STM32 interrupt and Cortex® exception with configurable priority service routine to ensure correct *STL* behavior and prevent any false error reporting.

How STL masks the interrupts

To mask the interrupts, the *STL* sets the PRIMASK register bit to 1. Setting this bit to 1 boosts the current execution priority to 0, preventing the activation of all exceptions with configurable priority. Thus when the current execution priority is boosted to a particular value, all exceptions with a lower or equal priority are masked.

4.3.7 DMA

The application must manage the DMA to avoid unwanted accesses to the 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.

4.3.8 Supported memories

Only internal memories are supported by the memory tests. The *STL* library must be executed from the STM32 internal memory only.

4.3.9 RAM TM

The user must ensure, through proper adaptation of the associated linker file, that the RAM TM subsets do not contain program memory areas.

Naming for EWARM icf file:

- Program memory is named read-only code memory. It is located in the `.text` section.
- Nonprogram (that is, data) memory is named:
 - Read write data memory. It is located in the `.data` section.
 - Read-only data memory. It is located in the `.text` section.

4.3.10 Program integrity TM

The user must ensure, through proper adaptation of the associated linker file, that the program integrity TM is executed only on *read only* memory areas.

Naming for EWARM icf file:

- *read only program memory* areas
- *read only data memory* areas

Otherwise the program integrity TM may report an STL_FAILED or STL_ERROR status

4.3.11 Read-only data

Due to the RAM test module and March C method design, the user must ensure, through proper adaptation of the associated linker file, that the read-only data of the *STL* is located in the dedicated ROM 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 the memory dedicated for storing the program.

4.3.12 RAM backup buffer

The backup process is enabled by default during RAM tests to preserve the RAM content. The user must then reserve a specific area for the RAM backup buffer at compilation time which must be allocated outside RAM subset configuration. There is only one RAM backup buffer defined for the test. The RAM backup buffer area is also tested by the March C- algorithm each time a RAM run test is called (prior to any subset defined by the user is tested).

The backup process can be optionally disabled either permanently by activating the compilation switch `STL_DISABLE_RAM_BCKUP_BUF` (see [Section 5.5.2: Steps to build an application from scratch](#)) or temporarily suppressed by specific control sequence (see note below). In this case, it is the responsibility of the end-user to guarantee that the application software does not consume data destroyed by the March C- test. This option can be used to speed up testing of those RAM areas where users do not need to preserve the memory content.

Note: *For a temporary suppression of the RAM backup buffer, the user must follow a set sequence:*

1. *Change the `STL_pRamTmBckUpBuf` variable value to overwrite it by `NULL`, while keeping a backup of the original value (default value stored by the STL).*
2. *The RAM test must then be restarted. To do so the user can use one of the APIs which force the RAM test into either `RAM_IDLE` or `RAM_INIT` state (see state diagrams in [Section 7.3: State machines](#)).*

To remove the backup suppression, the user must perform the same steps as above while restoring the default value of `STL_pRamTmBckUpBuf` to its original value and reinitialize the RAM test.

4.3.13 Cortex®-M55 cache resources

The TM RAM tests the RAM content, but not the DCache content. To ensure this, in cases where the DCache is enabled by the user, the TM RAM:

- Disables and cleans the DCache before performing the RAM test
- Invalidates and enables the DCache when the RAM test is complete (when the TM RAM returns)

Note: *During these operations, the STM32 interrupts and Cortex® exceptions with configurable priority are temporarily masked.*

The user must not enable the DCache during TM RAM execution, as the DCache must remain disabled during this test execution.

4.3.14 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, the *CPU* TM7 returns `STL_ERROR`.

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

When applying a nonzero value for the PSP stack limit special register (PSPLIM), caution is advised to prevent any fault exceptions during TM7 test execution. The PSP stack pointer is temporarily redirected to access 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 (e.g. 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` 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. These tests 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 has been correctly executed. This concerns both the test modules execution and all their configuration actions.

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 is correctly interpreted as unexpected behavior, and correctly handled in its application software. Error information refers to values different to the expected value, see [Section 7.2: User APIs](#)). 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, for each of the individual functions used.

This process cannot be considered as an exhaustive simulation of actual CPU failures on real devices but rather a testing interface of the implemented APIs.

Note: In some circumstances, experts performing the safety assessment of final systems embedding the STL might require exhaustive simulation to demonstrate STL capability to capture corruption of STM32 registers or memories injected intentionally during debugging the STL code. To perform these tests is practically impossible for any end user due to STL object delivery format. This specific testing was done for all the provided TMs during the STL certification process and its passing is recorded at internal test reports and guaranteed by the valid certificate issued for this ST firmware.

5 Package description

This section details the X-CUBE-CLASSB-N6 expansion package content and its correct use.

5.1 General description

X-CUBE-CLASSB-N6 is a software expansion package for STM32N6 series microcontrollers.

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-N6 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.

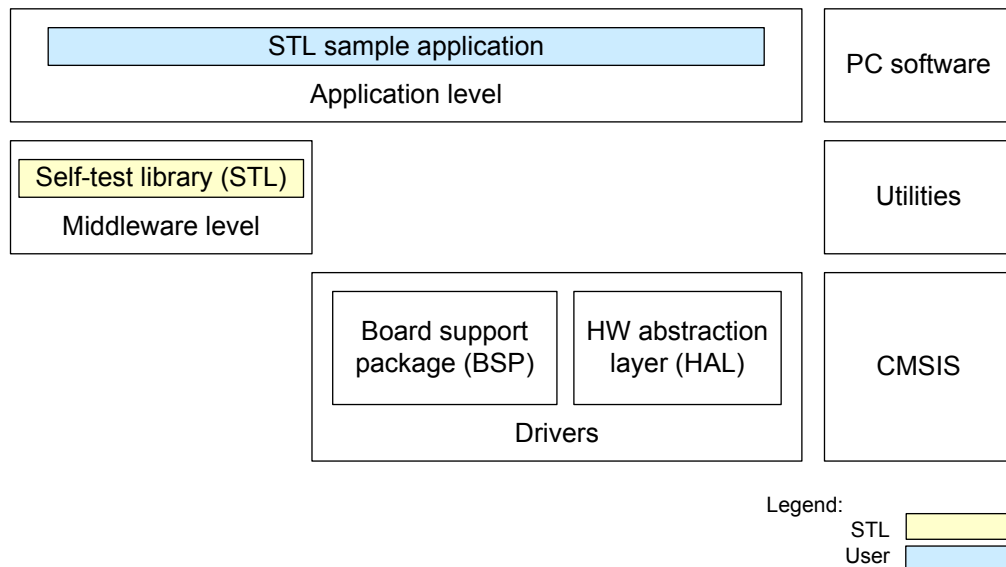
The following integrated development environments are supported:

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

5.2 Architecture

The components of the X-CUBE-CLASSB-N6 expansion package are illustrated in [Figure 6](#).

Figure 6. Software architecture overview



DT49051V1

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 *APIs*. 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 Board support package (BSP)

The software package needs to support the peripherals on the STM32 boards, apart from the MCU. This software is included in the board support package (BSP). This is a limited set of *APIs* that provides a programming interface for some specific board components, such as the LED and the user button.

5.2.3 STL

A significant part of the *STL*, available at middleware level, is a black box that manages the software-based diagnostic test. It is independent from the *HAL*, *BSP*, and *CMSIS*, even if the *STL* integration example relies on some *HAL* drivers.

5.2.4 User application example

Reference projects provided in the example show how to integrate a possible sequence of the *STL* test module calls into an application when adopting different IDEs, verify the returns of the *APIs*, and emulate their failure responses artificially. Additionally, a specific module for testing the clock system by applying a monitoring method compliant with the "Class B" standard requirements is included with a full source code to extend the available library set. It demonstrates how the library can be extended by specific tests or modules entirely defined by the end user.

The example also shows a possible way to differentiate between the overall initial startup test and the sequence of partial tests performed periodically during application runtime.

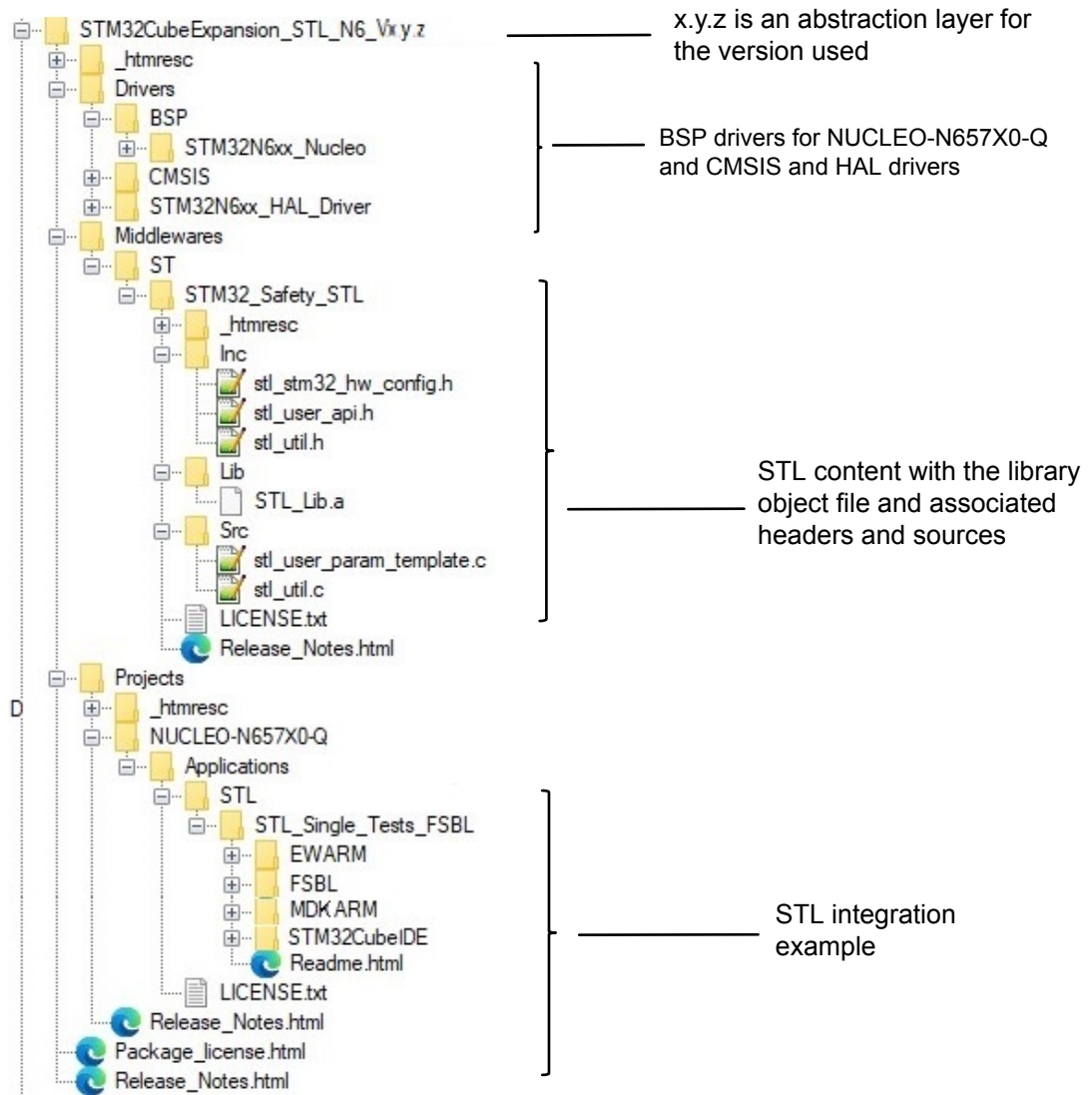
5.2.5 STL integrity

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 Figure 7.

Figure 7. Project file structure



DT80130V1

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 v9.30.1. The compilation is done with `--aeabi` and `--guard_calls` compilation options to fulfill AEABI compliance as described in “AEABI compliance” of the EWARM help section.

X-CUBE-CLASSB can be compiled and linked against the standard version of any C-compiler.

Safety guidelines

To fulfill the safety guidelines compliance as described in the IAR Embedded Workbench® safety guide (advice 2.1-1, 2.2-5, 2.4-1a and 5.4-3) and the Keil® safety manual (§4.9.2), the compliance is done with `--strict`, `--remarks`, `--require_prototypes` and `--no_unaligned_access` compilation options.

Library compliance

The library part of the *STL* (not delivered in source code) is compliant 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 defined by the implementation. It must be able to represent 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 the `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 is used to compile the *STL* library. This compiler 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. The user can use either the functions provided by the toolchain or the user ones.

5.4.3 Details

Detailed technical information about the available *APIs* 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

The X-CUBE-STL-N6 *STL* example project is based on the FSBL (First Stage Boot Loader) project template, which is also the most commonly used template for HAL examples in the STM32CubeN6 MCU package.

For more information, refer to the Getting Started with STM32CubeN6 for STM32N6 series (UM3249).

Ensure the following steps are completed:

1. Install the ST *CRC* tool (see [Section 6.2.2: CRC tool set-up](#)) or other *CRC* tool that generate an adequate structure necessary for proper execution of the program integrity test.
2. Select a project example and open it.
3. Build the project and launch the compiled binary.
4. Execute the example code.

Boot the board and check the result:

- LED toggles quickly and regularly: all the repeated tests provide results as expected.
- LED toggles irregularly: there is an error.

If any test returns a failure result, the LED flashes once every 2 sec. If the *STL* detects a defense programming error, the LED flashes once every 4 sec.

The `FailSafe_Handler` procedure is then called with a parameter keeping the identification code of the failed module

Note: The codes definitions are given in the `stl_user_api.h` file, in the case of a defensive programming failure, the `DEF_PROG_OFFSET` is added to the module code. User can adapt or extend the set of definitions applied by the STL example there.

Note: After the compilation/link of a binary, the CRC tool follows two actions:

- Calculates the CRCs (in the case of error, check the CRC tool path). For details see [Section 5.5.2: Steps to build an application from scratch](#)
- Automatically adds the computed CRCs to the generated binary.

5.5.2 Steps to build an application from scratch

To build an application from scratch, follow the steps listed below:

1. Create new application project with a suitable directory structure and with all the appropriate packages. Use STM32CubeMX tool to make it automatically.
2. Copy and paste the content of the `..Middleware\ST\STM32_Safety_STL` directory from the delivered STL example into the application project directories structure. Refer to [Section 5.3: Folder structure](#). In this case, modify the project setting manually while following the next steps:
 - Add all the STL source files located at the Src directory into the project.
 - Assign the Inc directory as an additional include path to be listed in the project settings.
 - Force the linker to include the library object file located at the Lib directory as an additional library.
3. If needed, add the next optional preprocessor compilation switches at project settings:
 - Option to enable `STL_DISABLE_RAM_BCKUP_BUF`, if the RAM backup buffer is not used (in this case the RAM data of the tested subsets are destroyed). If not activated, the RAM backup buffer is used by default. In such cases, the "backup_buffer_section" section must be defined in the linker scatter file.
 - Option to enable `STL_SW_CRC`: this is where the user application selects the software CRC . If not activated, the hardware CRC calculation is used by default.
 - Option to enable `STL_ENABLE_IT`: this is where the user application enables the STM32 interrupts during the CPU TM7, and RAM test. If not activated, the interrupts are masked during these tests. See [Section 4.3.6: Interrupt management](#) and [Section 4.1.4: RAM tests](#).
4. Check the integrity area size configuration.
It is mandatory to set the correct range of the memory for the project at `stl_user_param_template.c` file. Update the `STL_ROM_END_ADDR` there especially to ensure coherency with the associated linker scatter file and the CRC tool script (see step 6.).
5. Develop the user STL flow control. It is done 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 setup proper ranges of the tested memory areas and apply a correct check of the STL return information. Refer to [Section 7: STL: User APIs and state machines](#).

Note: The ST CRC tool calculates valid CRC checksums for full sections and their parts occupied by binary exclusively, so the range of the integrity test must be always kept inside the binary. This must be ensured by properly defining the dedicated user parameter. The rules for filling the memory subset are defined in [Section 4.1.3: Program integrity tests](#).

6. Apply the CRC tool to build the CRC area content necessary for the CRC calculation. Refer to [Section 6.2.2: CRC tool set-up](#).

Execute a proper command line of the STM32CubeProgrammer. This can be done automatically within the compilation process by invoking the IDE post build feature action as seen in [Figure 8](#) and [Figure 9](#).

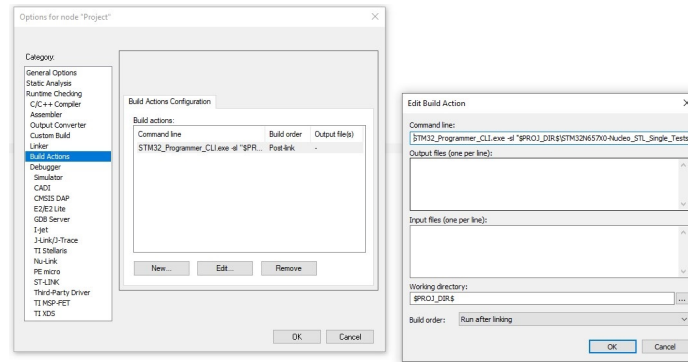
Note: STM32CubeProgrammer supports check sum computation on the single compact area only. To ensure the integrity of the ISR vector table content, the vector table can be either relocated from its default location in retention RAM to SRAM where the rest of the code is stored or the user should apply some other mechanism of integrity check of this separate area content. The ISR table remapping is applied at the STL integration example. For more details see `readme` file in the project directory.

7. Convert the project output binary file using the binary conversion command line. (see the examples below). It is mandatory to do this step after the CRC calculation.
8. Compile, load, and execute the binary.

Artificial failing APIs can be used to debug a correct behavior of the programmed STL flow if the STL detects a hardware failure.

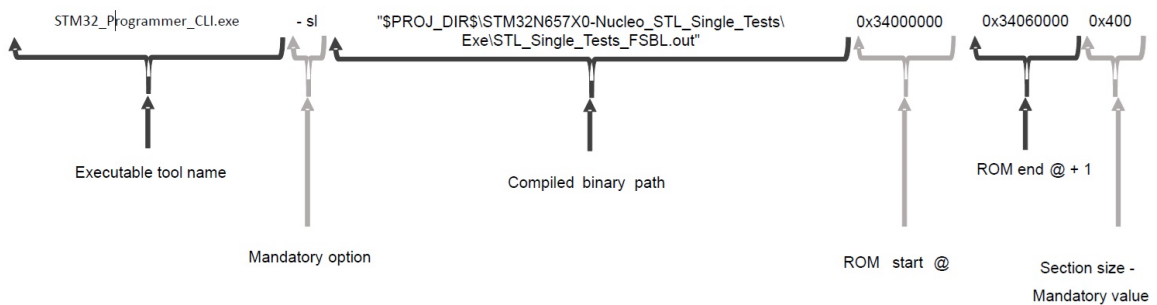
Note: During the verification, the artificial-failing feature (refer to [Section 7.2.5: Artificial-failing APIs](#)) must be used to emulate the generation of incorrect test-module result values related to associated individual software diagnostics (CPU tests, RAM test, Program integrity test), for each of the individual functions used. Also, refer to [Section 4.4.2: Test 2: correct STL error-message processing](#).

Figure 8. IAR™ post-build actions screenshot



DT80131V1

Figure 9. CRC tool command line



DT80132V1

Binary conversion command line

The following list provides the specific command lines for converting compiled projects to IAR Embedded Workbench®, Keil®, and STM32CubeIDE binary format.

- IAR Embedded Workbench® for Arm® (EWARM):

```
ielftool "$PROJ_DIR$\\STM32N657X0Nucleo_STL_Single_Tests\\Exe\\STL_Single_Tests_FSBL.out" --bin "$PROJ_DIR$\\STM32N657X0-Nucleo_STL_Single_Tests\\Exe\\STL_Single_Tests_FSBL.bin"
```

- Keil® microcontroller development kit (MDK-ARM):

```
fromelf --bincombined -o "$H@H.bin" "$H@H.axf"
```

- STM32CubeIDE:

```
arm-none-eabi-objcopy -O binary "${BuildArtifactFileName}.elf" "${BuildArtifactFileName}.bin"
```

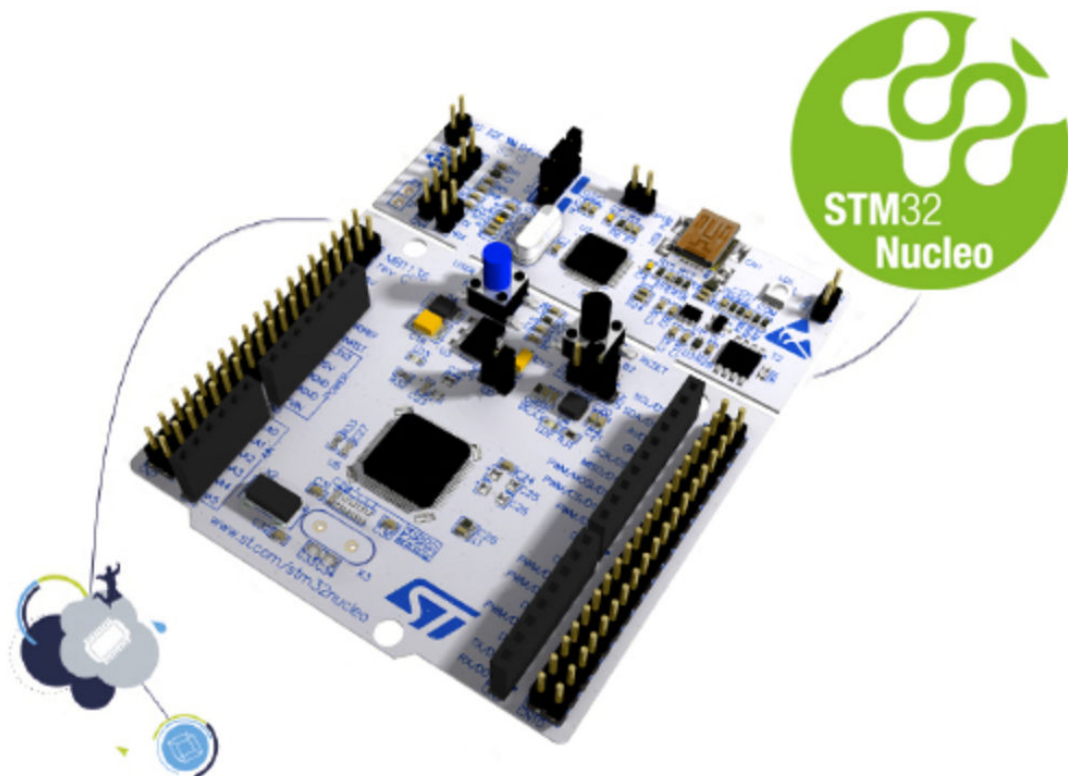
6 Hardware and software environment setup

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 microcontroller lines. 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/V2-1 debugger/programmer. The STM32 Nucleo boards come with the STM32 comprehensive software HAL library together with various packaged-software examples.

Details about the STM32 Nucleo boards are available from the <http://www.st.com/stm32nucleo> web page.

Figure 10. STM32 Nucleo board example



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

The following components are needed:

- STM32 NUCLEO-N657X0-Q development board
- USB type C to Mini-B USB cable to connect the development board to the PC.

6.2 Software setup

This section lists the minimum requirements for the developer to set up the SDK, to run the sample scenario, and to 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 chapter *IDE* compatibility, if it exists.

6.2.2 CRC tool set-up

ST provides a *CRC* tool, available as a single feature inside the STM32CubeProgrammer, used for program integrity testing. Other *CRC* tools can be used, provided they fulfill the requirements detailed in [Expected CRC precalculation](#).

Tool installation procedure:

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

The easiest way is to add the tool path in the environment variable (computer administration rights are required). If not, the path must be added directly 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 copies 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 Integrity/RAM, test is passed and end of
                                configuration is also reached */
    STL_PARTIAL_PASSED = STL_PARTIAL_PASSED_DEF, /* Used only for RAM and Integrity testing.
                                                Test passed, But end of Integrity/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 programming
                              check failed) */
} STL_TmStatus_t; /* Type for the result of a Test Module */
```

```
typedef enum
{
    STL_CPU_TM1L_IDX = 0U, /* 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 Class B 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 struct STL_MemSubset_struct
{
    uint32_t StartAddr; /* start address of Integrity or RAM memory subset */
    uint32_t EndAddr; /* end address of Integrity or RAM memory subset */
    STL_CrcTableIdx_t CrcTableIdx; /* Index of the CRC table (used by Integrity TM)*/

    struct STL_MemSubset_struct *pNext; /* pointer to the next Integrity or RAM memory subset
                                         - to be set to NULL for the last subset */
} STL_MemSubset_t; /* Type used to define Integrity
                  or RAM subsets to test */
```

```
typedef struct
{
    STL_MemSubset_t *pSubset; /* Pointer to the Integrity or RAM subsets to test */
    uint32_t NumSectionsAtomic; /* Number of Integrity or RAM sections to be tested
                                during an atomic test */
} STL_MemConfig_t; /* Type used to fully define Integrity 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 IntegrityTmStatus; /* Forced status value for Integrity 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 declared in the file `stl_user_api.h`. 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 accesses directly to the memory addresses defined by the application.

This applies during the overall STL execution. For example, the pointers to access the content of structures that keep the configuration of the memory tests must be maintained. They are still used by the `STL_SCH_run_xxx` functions, even if they are not always part of the input parameter list when an API associated with these tests is called.

For more details about proper API sequence calls see [Section 7.3: State machines](#) and [Section 8: Test examples](#).

7.2.1 Common API

The following sections present details on common APIs.

7.2.1.1 STL_SCH_Init

Description: initializes the scheduler. It can be used at any time to reinitialize the scheduler (it resets all tests).

Declaration: `STL_Status_t STL_SCH_Init(void)`.

Table 6. STL_SCH_Init input information

Allowed states	Parameters
CPU TMx: all Integrity TM: all RAM TM: all	-

Table 7. STL_SCH_Init output information

STL_Status_t return value		Returned state
Value	Comments	
STL_OK	Function successfully executed	CPU TMx: CPU_TMx_CONFIGURED Integrity TM: INTEGRITY_IDLE RAM TM: RAM_IDLE
STL_KO	Source of defensive programming error: <ul style="list-style-type: none"> 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 4.3.3: CRC resources](#).

7.2.2 CPU Arm® core testing APIs

7.2.2.1 STL_SCH_RunCpuTMx

Description: runs one of the CPU test modules.

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 states	Parameters	
	Value	Comments
CPU_TMx_CONFIGURED	*pSingleTmStatus	See Caution

Table 9. STL_SCH_RunCpuTMx output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_PASSED	-	CPU_TMx_CONFIGURED
		STL_FAILED	-	
		STL_ERROR	Source of defensive programming error: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> pSingleTmStatus = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

7.2.3 Program integrity testing APIs

7.2.3.1 STL_SCH_InitIntegrity

Description: initializes program integrity test.

Declaration: STL_Status_t STL_SCH_InitIntegrity(STL_TmStatus_t *pSingleTmStatus)

Table 10. STL_SCH_InitIntegrity input information

Allowed states	Parameters	
	Value	Comments
INTEGRITY_IDLE INTEGRITY_INIT INTEGRITY_CONFIGURED	*pSingleTmStatus	Caution

Table 11. STL_SCH_InitIntegrity output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	INTEGRITY_INIT
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> pSingleTmStatus = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

7.2.3.2 STL_SCH_ConfigureIntegrity

Description: configures the program integrity test.

Declaration: STL_Status_t STL_SCH_ConfigureIntegrity(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pIntegrityConfig)

Table 12. STL_SCH_ConfigureIntegrity input information

Allowed states	Parameter			
	Value	Comments		
INTEGRITY_INIT	*pSingleTmStatus	See Caution		
	*pIntegrityConfig	Pointer to the program integrity configuration. See Caution .		
		Field	Comments	
		*pSubset	<ul style="list-style-type: none"> Pointer to program integrity subset. See Caution A section cannot overlap with the CRC area 	
			Field	Comments
			StartAddr	<ul style="list-style-type: none"> Start subset address in bytes Cannot be lower than INTEGRITY_AREA_START and higher than CRC_START address
EndAddr	<ul style="list-style-type: none"> End subset address in bytes Cannot be lower than INTEGRITY_AREA_START and higher than CRC_START address Needs to be higher than StartAddr 			
*pNext	<ul style="list-style-type: none"> Pointer to next program integrity subset. See Caution Must be set to NULL for the last subset 			
	NumSectionsAtomic	<ul style="list-style-type: none"> Number of program integrity 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 program integrity subsets are tested in one pass 		

Table 13. STL_SCH_ConfigureIntegrity output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	INTEGRITY_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: <ul style="list-style-type: none"> • State not allowed • Wrong configuration detected • STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> • pSingleTmStatus = NULL • pIntegrityConfig = NULL • STL internal data corrupted 	Not relevant	Value must not be used	No state change

Additional information: in the case of a return value set to STL_KO or *pSingleTmStatus set to STL_ERROR, the program integrity configuration is not applied.

7.2.3.3

STL_SCH_RunIntegrityTM

Description: runs program integrity test.

Declaration: STL_Status_t STL_SCH_RunIntegrityTM(STL_TmStatus_t *pSingleTmStatus)

Table 14. STL_SCH_RunIntegrityTM input information

Allowed states	Parameters	
	Value	Comments
INTEGRITY_CONFIGURED	*pSingleTmStatus	See Caution

Table 15. STL_SCH_RunIntegrityTM output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_PASSED	-	INTEGRITY_CONFIGURED
		STL_PARTIAL_PASSED	-	INTEGRITY_CONFIGURED
		STL_FAILED	-	INTEGRITY_CONFIGURED
		STL_NOT_TESTED	All subsets are already tested	INTEGRITY_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: <ul style="list-style-type: none"> State not allowed Configuration corrupted STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> pSingleTmStatus = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

7.2.3.4
STL_SCH_ResetIntegrity

Description: resets program integrity test.

Declaration: STL_Status_t STL_SCH_ResetIntegrity(STL_TmStatus_t *pSingleTmStatus)

Table 16. STL_SCH_ResetIntegrity input information

Allowed states	Parameters	
	Value	Comments
INTEGRITY_CONFIGURED	*pSingleTmStatus	See Caution

Table 17. STL_SCH_ResetIntegrity output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	Configuration successfully applied	INTEGRITY_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: <ul style="list-style-type: none"> State not allowed Configuration corrupted STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> pSingleTmStatus = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

Additional information

- Once all subsets are tested, the user needs to reset the test module to perform the program integrity test again.
- In the case of a return value set to STL_KO or *pSingleTmStatus set to STL_ERROR, the program integrity reset is not applied.

7.2.3.5

STL_SCH_DeInitIntegrity

Description: deinitializes program integrity test.

Declaration: STL_Status_t STL_SCH_DeInitIntegrity(STL_TmStatus_t *pSingleTmStatus)

Table 18. STL_SCH_DeInitIntegrity input information

Allowed states	Parameters	
	Value	Comments
INTEGRITY_IDLE INTEGRITY_INIT INTEGRITY_CONFIGURED	*pSingleTmStatus	See Caution

Table 19. STL_SCH_DeInitIntegrity output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	INTEGRITY_IDLE
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> • 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_SCH_InitRam

Description: initializes the RAM test.

Declaration: STL_Status_t STL_SCH_InitRam(STL_TmStatus_t *pSingleTmStatus).

Table 20. STL_SCH_InitRam input information

Allowed states	Parameters	
	Value	Comments
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	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	RAM_INIT
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> 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: Description: configures the RAM test.

Declaration: STL_Status_t STL_SCH_ConfigureRam(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pRamConfig)

Table 22. STL_SCH_ConfigureRam input information

Allowed states	Parameter		
	Value	Comments	
RAM_INIT	*pSingleTmStatus	See Caution	
	*pRamConfig	This pointer contains the RAM configuration. See Caution	
		Field	Comments
			<ul style="list-style-type: none"> Pointer to RAM subset. See Caution A subset cannot overlap with the RAM backup buffer if defined
		Field	Comments
		StartAddr	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 		
	*pNext	<ul style="list-style-type: none"> Pointer to next RAM subset. See Caution Must be set to NULL for the last subset 	
	NumSectionsAtomic	<ul style="list-style-type: none"> 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	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	RAM_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: <ul style="list-style-type: none"> State not allowed Wrong configuration detected STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> pSingleTmStatus = NULL pRamConfig = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

Additional information: in the case of a return value set to STL_KO or *pSingleTmStatus set to STL_ERROR, the RAM configuration is not applied.

7.2.4.3

STL_SCH_RunRamTM

Description: runs the RAM test.

Declaration: STL_Status_t STL_SCH_RunRamTM(STL_TmStatus_t *pSingleTmStatus)

Table 24. STL_SCH_RunRamTM input information

Allowed states	Parameters	
	Value	Comments
RAM_CONFIGURED	*pSingleTmStatus	See Caution

Table 25. STL_SCH_RunRamTM output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
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: <ul style="list-style-type: none"> State not allowed Configuration corrupted STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> 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 the RAM test.

Declaration: STL_Status_t STL_SCH_ResetRam(STL_TmStatus_t *pSingleTmStatus)

Table 26. STL_SCH_ResetRam input information

Allowed states	Parameters	
	Value	Comments
RAM_CONFIGURED	*pSingleTmStatus	See Caution

Table 27. STL_SCH_ResetRam output information

STL_Status_t return value		*pSingleTmStatus output		Returned state
Value	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	Configuration successfully applied	RAM_CONFIGURED
		STL_ERROR	Possible source of defensive programming error: <ul style="list-style-type: none"> State not allowed Configuration corrupted STL internal data corrupted 	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> pSingleTmStatus = NULL STL internal data corrupted 	Not relevant	Value must not be used	No state change

Additional information

- Once all subsets are tested, the user needs to reset the test module to perform the RAM test again.
- In the case of a return value set to STL_KO or *pSingleTmStatus set to STL_ERROR, the RAM reset is not applied.

7.2.4.5
STL_SCH_DeInitRam

Description: deinitializes the RAM test.

Declaration: STL_Status_t STL_SCH_DeInitRam(STL_TmStatus_t *pSingleTmStatus)

Table 28. STL_SCH_DeInitRam input information

Allowed states	Parameters	
	Value	Comments
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	Comments	Value	Comments	
STL_OK	Function successfully executed	STL_NOT_TESTED	-	RAM_IDLE
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> • 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_SCH_StartArtifFailing

Description: sets artificial-failing configuration and starts artificial-failing feature.

Declaration: STL_Status_t STL_SCH_StartArtifFailing(const STL_ArtifFailingConfig_t *pArtifFailingConfig)

Table 30. STL_SCH_StartArtifFailing input information

Allowed states	Parameters	
	Value	Comments
CPU TMx: <ul style="list-style-type: none"> • CPU_TMx_CONFIGURED Integrity TM: <ul style="list-style-type: none"> • INTEGRITY_IDLE • INTEGRITY_INIT • INTEGRITY_CONFIGURED RAM TM <ul style="list-style-type: none"> • RAM_IDLE • RAM_INIT • RAM_CONFIGURED 	*pArtifFailingConfig	No state change

Table 31. STL_SCH_StartArtifFailing output information

STL_Status_t return value	Comments	Output	Comments
STL_OK	Function successfully executed	No output parameter	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> • pArtifFailingConfig = NULL • configured values are not set for each test module • STL internal data corrupted 		

Additional information: All the following *API* calls are executed normally except if the *STL_Status_t* return value is set to STL_OK, the test module status (*pSingleTmStatus, *pTmListStatus) is forced to a configured value.

7.2.5.2
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 states	Parameters	
	Value	Comments
CPU TMx: <ul style="list-style-type: none"> • CPU_TMx_CONFIGURED Integrity TM: <ul style="list-style-type: none"> • INTEGRITY_IDLE • INTEGRITY_INIT • INTEGRITY_CONFIGURED RAM TM <ul style="list-style-type: none"> • RAM_IDLE • RAM_INIT • RAM_CONFIGURED 	No input parameter	No state change

Table 33. STL_SCH_StopArtifFailing output information

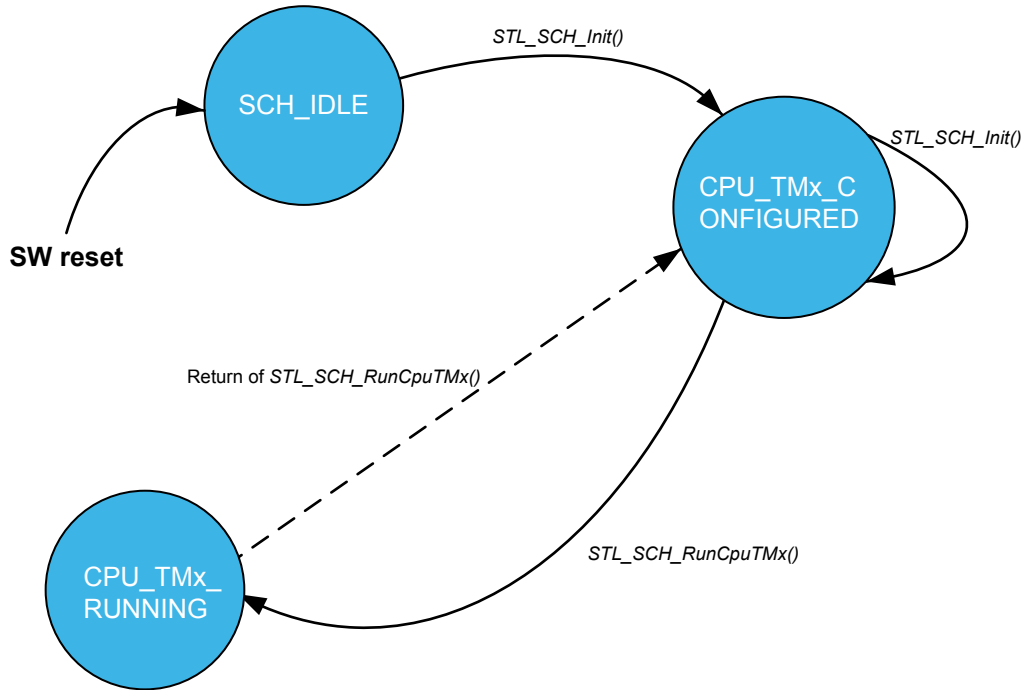
STL_Status_t return value	Comments	Output	Comments
STL_OK	Function successfully executed	No output parameter	No state change
STL_KO	Possible source of defensive programming error: <ul style="list-style-type: none"> • STL internal data corrupted 		

7.3
State machines

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

CPU test APIs

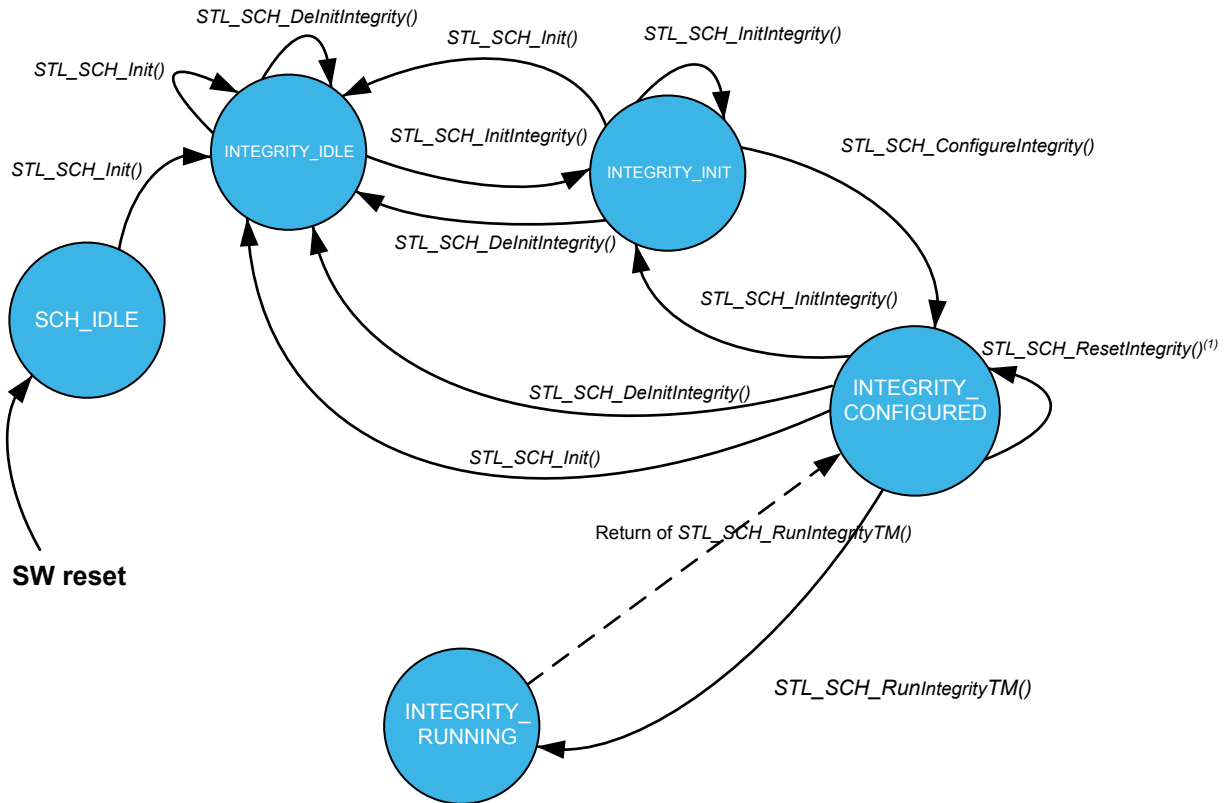
Figure 11. State machine diagram - CPU test APIs



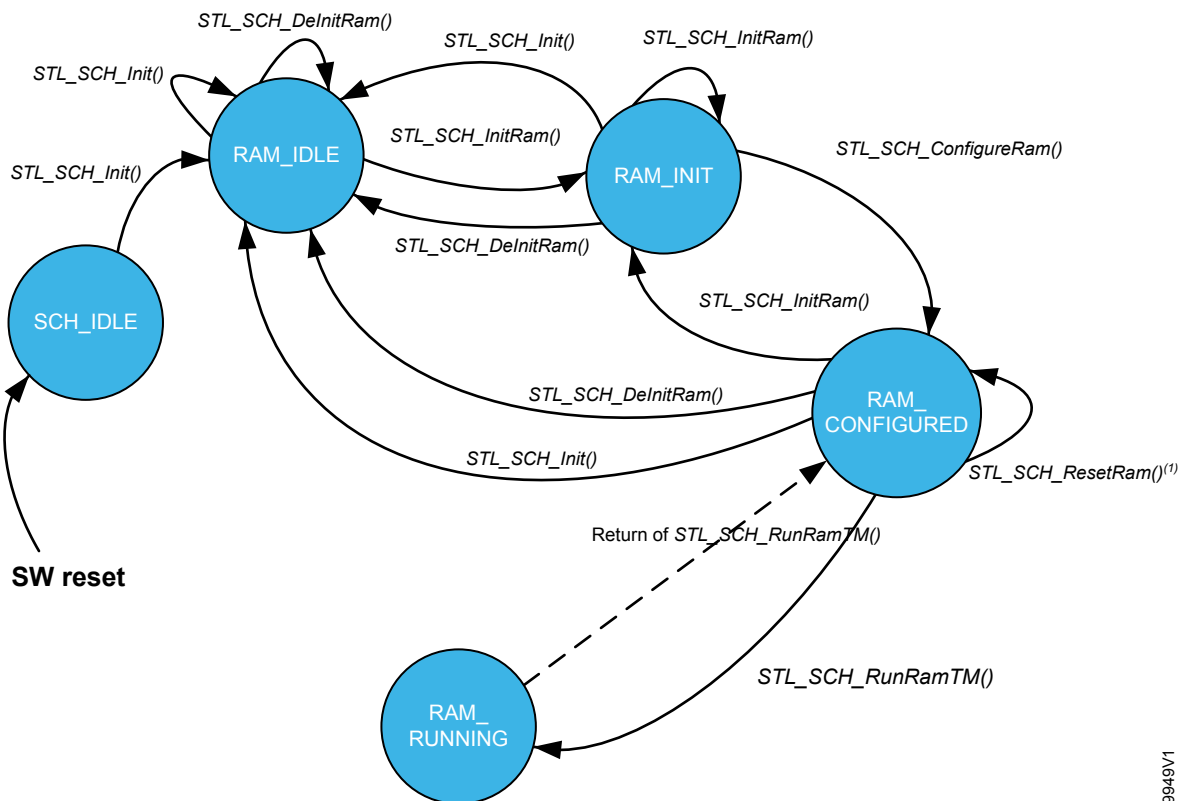
DT69947V1

Program integrity test APIs

Figure 12. State machine diagram - program integrity test APIs



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

RAM test APIs
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.

DT60949V1

7.4 API usage and sequencing

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 accesses directly the memory addresses defined by the application.
- Check the status of function return value (`STL_Status_t`), before checking the test result (`STL_TmStatus_t`). See the example in the delivered applications.

The APIs run independently of each other and therefore can be called in any order.

Only APIs dedicated to the configuration and initialization of the memories tests must be called before any execution of these tests is applied. See Section 7.3: State machines for more details.

The test flow is simplified, all the tests are now executed from C-code. All the modules are common and suitable for both startup and runtime testing. Differentiation between startup and runtime tests can be performed by proper sequencing and configuration of test modules. After application reset, common practice is to perform a full initial sequence including the complete set of tests executed over all the memory areas before the application starts. This sequence is defined in following order:

1. All the CPU tests
2. Complete tests of nonvolatile memory integrity
3. Functional test overall for the available space of volatile memories including the area especially dedicated to the stack

Note: Temporarily suppressing of the memory content backup can be applied to speed up initial testing of huge RAM areas where user does not need to preserve the memory content during this test. For more details see Section 4.3.12: RAM backup buffer. Functional test is not executed over areas containing program code and data when the code is executed from RAM.

4. Specific customer tests

Later, at runtime, the order of the tests can be changed and executed in more relaxed way. The memory regions under tests can be reduced. The test process can even be dynamically modified with prior focus on those areas where the most recently executed safety related code and data are stored. This is especially the case when considering factors like:

- Available application process safety time
- System overall performance
- Concrete status of the application

7.5 User parameters

In addition to parameters set directly inside the *APIs*, there are few parameters to be customized in the `stl_user_param_template.c` file. They are located in the code, with the following comments:

```
/* customisable */
```

Extract from `stl_user_param_template.c`:

```
#if defined (STM32N657xx)
/*Memory configuration*/
#define STL_ROM_TZ_S_START_ADDR (0x34000400UL) /*customizable*/
#define STL_ROM_TZ_S_END_ADDR (0x3405FFFFUL) /*customizable*/
#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 RAM */
/* The RAM backup buffer is placed in "backup_buffer_section". */
/* "backup_buffer_section" section is defined in scatter file */
```

The customizing depends on the user choice.

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

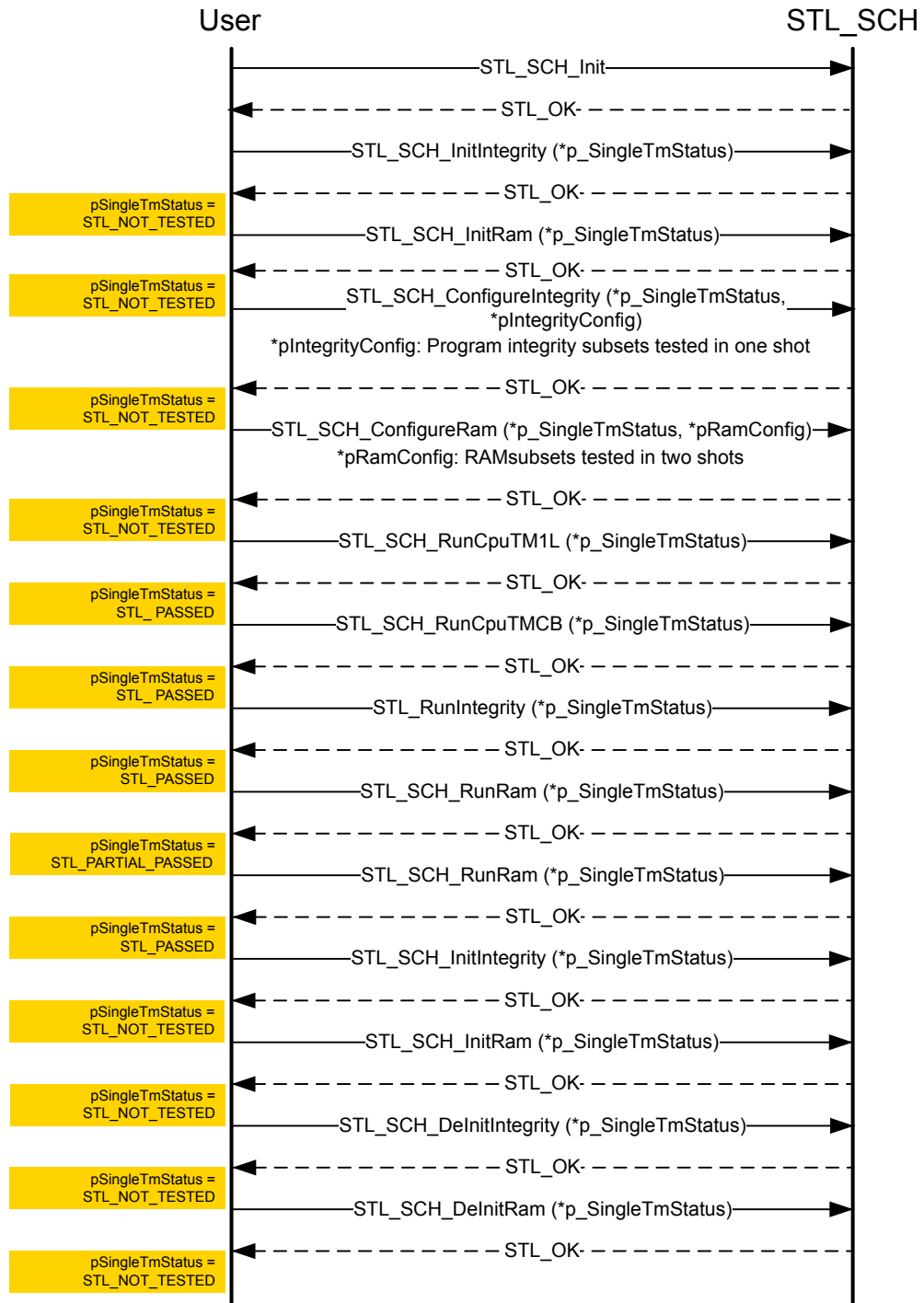
- `stl_user_param_template.c`: use of RAM backup buffer or not
- `stl_util.c`: use of software or hardware *CRC* computation
- `stl_stm32_hw_config.h`: if *CRC* hardware is used, choose the right *CRC* IP configuration according to the STM32 device

Refer to [Section 5.5.2: Steps to build an application from scratch](#) for the flag configuration check.

8 Test examples

Figure 14 shows an example of a possible sequence of STL API calls through the STL scheduler and returned information provided by STL (refer to Figure 1 and Table 2).

Figure 14. Test flow example



Legend:
Test status

DT72480V1

Figure 15 shows a detailed example of program integrity test flow handling:

- Use of two program integrity subsets
- Use of functions
 - STL_SCH_RunIntegrity → only the program integrity test module is executed
 - STL_SCH_ResetIntegrity
- Function return value
- Program integrity test module result value: `pSingleTmStatus` → in this case, it contains the result of the test

Figure 16 shows a detailed example of RAM test flow handling:

- Use of two RAM subsets
- Use of functions:
 - STL_SCH_RunRAM → 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. Program integrity test flow example

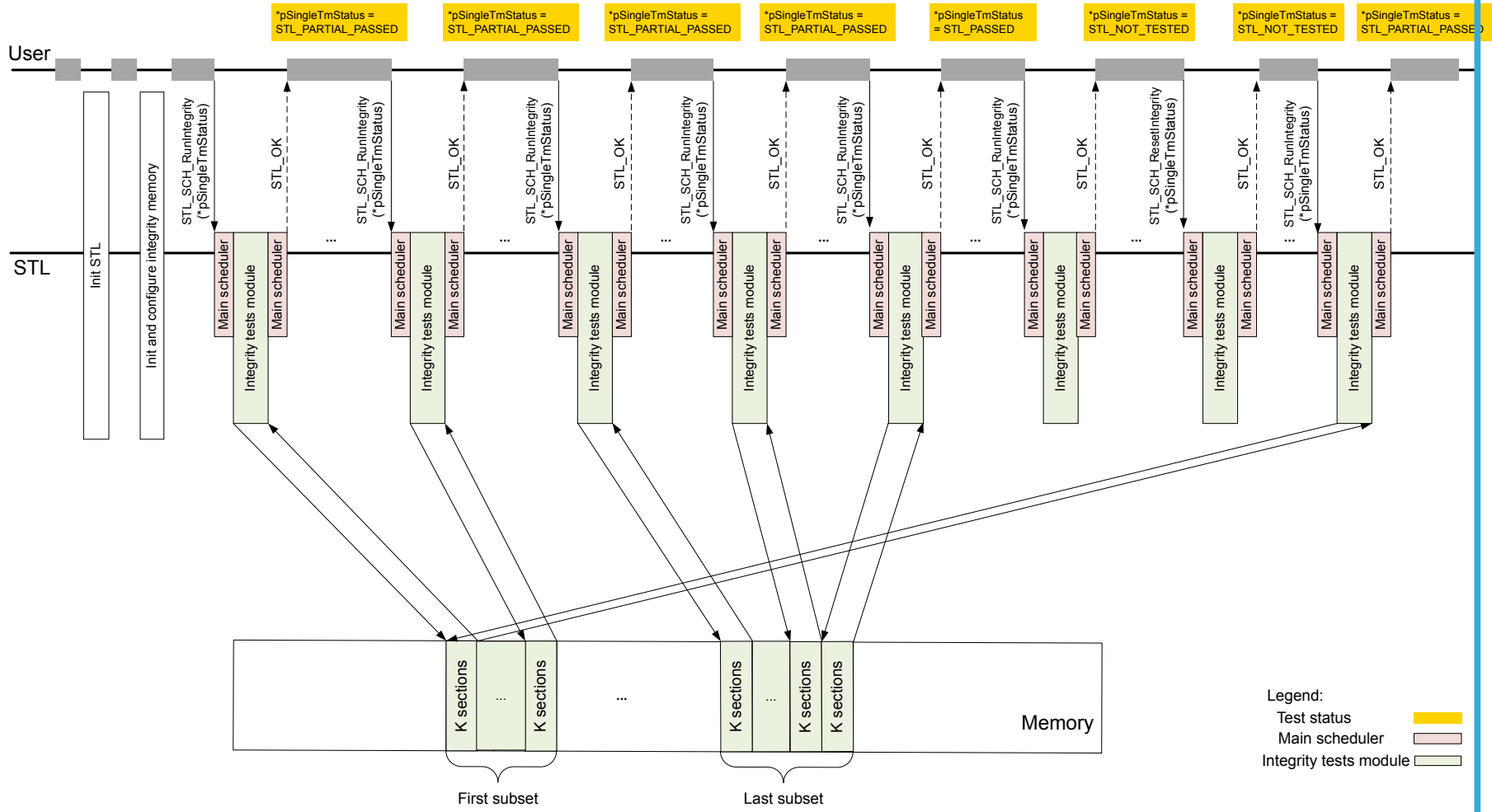
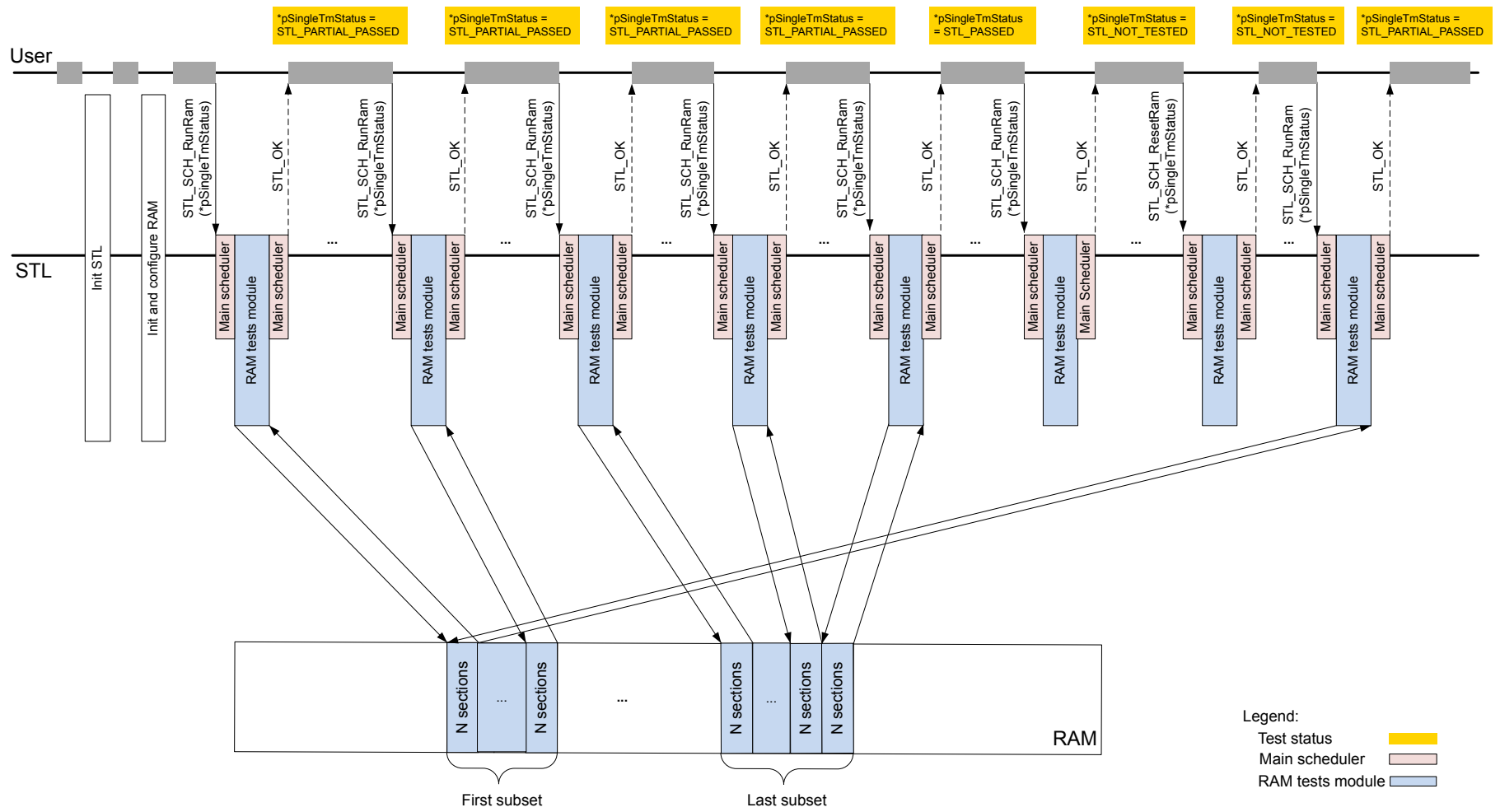


Figure 16. RAM test flow example



9 STL: execution timing details

The data in the following table is obtained with the test set-up described in [Section 4.2: STL performance data](#)

Table 34. Integration tests

Test	Duration (in μ s)		Tested memory
	Hardware CRC	Software CRC ⁽¹⁾	
STL_SCH_InitIntegrity()	0.5	0.5	-
STL_SCH_ConfigureIntegrity()	1	1	-
STL_SCH_RunIntegrityTM()	114	298	17408 bytes tested
STL_SCH_InitRam()	0.7	0.7	-
STL_SCH_ConfigureRam()	1	1	-
STL_SCH_RunRamTM() ⁽²⁾	1860063	1860063	3047384 bytes tested
STL_SCH_RunCpuTM1L()	5	5	-
STL_SCH_RunCpuTM7() ⁽²⁾	2	2	-
STL_SCH_RunCpuTMCB()	1	1	-

1. Test duration increased because of the CRC algorithm (see [Section 4.1.1: Scheduler principle](#)).

2. Default configuration: `STL_ENABLE_IT` not enabled.

10 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.

Note: *The clock frequency measurement is not an integrated part of the STL package. The clock testing module is provided as full source format within STL integration example to demonstrate the capability of implementing additional user defined testing modules which can be included at the STL flow. For more details refer to Section 10.5: Extension capabilities STL library.*

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).

10.1 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).

10.2 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.

10.3 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.

10.4 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.

10.5 Extension capabilities STL library

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 which are already implemented by previous versions of the library:

- Test of registers at *CPU TMs*
- 32-bit CRC calculation compatible with STM32 HW CRC unit at Program integrity *TM*
- March C test respecting physical address order of the RAM *TM*
- Timer triggered by LSI to check system clock frequency of the clock *TM* defined at *STL* integration example

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
- Pre-compiled 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 programming 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.

An example of an additional specific test module implementation is available in the firmware package integration example. A specific test module based on the cross check measurement method of two independent clock sources is delivered as open source format together with the firmware package integration example. This module must be adapted by the end user to take into account specific dependencies on the configuration of the applied clock system.

This module uses the same measurement principle already applied in previous versions of the library. The hardware used for the frequency comparison must initially be configured (Channel 1 of TIM16 triggered by LSI) to invoke clock measurement before the associated *API* is called. This hardware configuration is done at the end of `STL_Init()` procedure in the `main.c` file. The *API* is written to use interface compatibility with the regular *APIs* integrated in the *STL* so the same format is applied in its declaration:

```
STL_Status_t STL_SCH_RunClockTest(STL_TmStatus_t *pSingleTmStatus)
```

The parameter that is passed during this function call acts as a pointer to the clock module measurement status, and the function itself provides a `STL_KO` vs `STL_OK` return status as well as do the regular *STL* modules if defensive programming fails. If the clock measurement hardware is active and the new period value updated by the last measurement cycle (set to 8 consecutive LSI periods) is found at the expected interval (defined by macros `CLK_LimitLow` and `CLK_LimitHigh`), the module measurement status value is changed into `STL_PASSED`. If not it is set to `STL_FAILED` as per the regular *API* modules. This is also the case when artificial failing of the module is invoked.

In a similar way, the user can integrate the following modules. For example, any stack hardening techniques like stack boundary area check or implementing watchdog testing and servicing is no longer included at this new package by default. The source code of these tests is available in older versions of this library see [2]. Refer to [1] for additional information about the commonly recognized safety methods that are not specifically required by the household standard. They may be useful to improve the user application robustness.

11 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 [1].

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 10: 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 <i>STL</i> TM1L, TM7, and TMCB CPU test modules	Combination of functional and pattern tests of the <i>CPU</i> 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 H.2.18.18	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 Periodic run of clock cross-check module. Added at open source format as a user specific test module within the firmware integration example	Clock cross-check measurement between two independent clock sources (system clock and LSI)
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 <i>STL</i> 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 <i>STL</i> 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
5. Internal data path	5.1 Data	X	H.2.19.8.2	Stuck at	-	ECC enable under end-user responsibility ⁽²⁾
	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	Version	Changes
08-Jun-2026	1	Initial release.

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 converter

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

SG Secure gateway

STL Self-test library

TM Test module

Contents

1	General information	2
1.1	Purpose and scope	2
1.2	Reference documents	2
2	STM32Cube overview	3
2.1	What is STM32Cube?	3
2.2	How does this software complement STM32Cube?	3
3	STL overview	4
3.1	Architecture overview	4
3.2	Supported products	5
4	STL description	6
4.1	STL functional description	6
4.1.1	Scheduler principle	6
4.1.2	CPU Arm® core tests	7
4.1.3	Program integrity tests	8
4.1.4	RAM tests	11
4.2	STL performance data	13
4.2.1	STL execution timings	13
4.2.2	STL code and data size	14
4.2.3	STL stack usage	14
4.2.4	STL heap usage	14
4.2.5	STL interrupt masking time	14
4.2.6	Data cache performance impact	15
4.3	STL user constraints	15
4.3.1	Privileged-level	15
4.3.2	RCC resources	15
4.3.3	CRC resources	15
4.3.4	Bit Q of APSR	16
4.3.5	Bits GE of APSR	16
4.3.6	Interrupt management	16
4.3.7	DMA	17
4.3.8	Supported memories	17
4.3.9	RAM TM	17
4.3.10	Program integrity TM	17
4.3.11	Read-only data	17
4.3.12	RAM backup buffer	18

4.3.13	Cortex®-M55 cache resources	18
4.3.14	Processor mode	18
4.3.15	Setting the PSPLIM (process stack pointer limit) CPU register	18
4.4	End-user integration tests	18
4.4.1	Test 1: correct STL execution	18
4.4.2	Test 2: correct STL error-message processing	19
5	Package description	20
5.1	General description	20
5.2	Architecture	20
5.2.1	STM32Cube HAL	20
5.2.2	Board support package (BSP)	21
5.2.3	STL	21
5.2.4	User application example	21
5.2.5	STL integrity	21
5.3	Folder structure	22
5.4	APIs	22
5.4.1	Compliance	22
5.4.2	Dependency	23
5.4.3	Details	23
5.5	Application: compilation process	23
5.5.1	Steps to build a delivered STL example	23
5.5.2	Steps to build an application from scratch	24
6	Hardware and software environment setup	26
6.1	Hardware setup	26
6.2	Software setup	26
6.2.1	Development tool-chains and compilers	26
6.2.2	CRC tool set-up	27
7	STL: User APIs and state machines	28
7.1	User structures	28
7.2	User APIs	29
7.2.1	Common API	29
7.2.2	CPU Arm® core testing APIs	29
7.2.3	Program integrity testing APIs	30
7.2.4	RAM testing APIs	34
7.2.5	Artificial-failing APIs	38
7.3	State machines	39
7.4	API usage and sequencing	42

7.5	User parameters	43
8	Test examples	44
9	STL: execution timing details	48
10	Application-specific tests not included in ST firmware self-test library	49
10.1	Analog signals	49
10.2	Digital I/Os	50
10.3	Interrupts	50
10.4	Communication	51
10.5	Extension capabilities STL library	51
11	Compliance with IEC, UL, and CSA standards	52
	Revision history	54
	Glossary	55
	List of tables	59
	List of figures	60

List of tables

Table 1.	Applicable product	1
Table 2.	STL return information	7
Table 3.	STL execution timings, clock at 600 MHz	14
Table 4.	STL code size and data size (in bytes)	14
Table 5.	STL maximum interrupt masking information	15
Table 6.	STL_SCH_Init input information	29
Table 7.	STL_SCH_Init output information	29
Table 8.	STL_SCH_RunCpuTMx input information	30
Table 9.	STL_SCH_RunCpuTMx output information	30
Table 10.	STL_SCH_InitIntegrity input information	30
Table 11.	STL_SCH_InitIntegrity output information	30
Table 12.	STL_SCH_ConfigureIntegrity input information	31
Table 13.	STL_SCH_ConfigureIntegrity output information	32
Table 14.	STL_SCH_RunIntegrityTM input information	32
Table 15.	STL_SCH_RunIntegrityTM output information	33
Table 16.	STL_SCH_ResetIntegrity input information	33
Table 17.	STL_SCH_ResetIntegrity output information	33
Table 18.	STL_SCH_DeInitIntegrity input information	34
Table 19.	STL_SCH_DeInitIntegrity output information	34
Table 20.	STL_SCH_InitRam input information	34
Table 21.	STL_SCH_InitRam output information	35
Table 22.	STL_SCH_ConfigureRam input information	35
Table 23.	STL_SCH_ConfigureRam output information	36
Table 24.	STL_SCH_RunRamTM input information	36
Table 25.	STL_SCH_RunRamTM output information	37
Table 26.	STL_SCH_ResetRam input information	37
Table 27.	STL_SCH_ResetRam output information	37
Table 28.	STL_SCH_DeInitRam input information	38
Table 29.	STL_SCH_DeInitRam output information	38
Table 30.	STL_SCH_StartArtifFailing input information	38
Table 31.	STL_SCH_StartArtifFailing output information	39
Table 32.	STL_SCH_StopArtifFailing input information	39
Table 33.	STL_SCH_StopArtifFailing output information	39
Table 34.	Integration tests	48
Table 35.	IEC 60335-1 components covered by the X-CUBE-CLASSB library by methods recognized by IEC-60730-1	53
Table 36.	Document revision history	54

List of figures

Figure 1.	STL architecture	4
Figure 2.	Single test control call architecture	7
Figure 3.	Program integrity test: CRC principle	10
Figure 4.	Program integrity test: CRC use cases versus program areas	11
Figure 5.	RAM test: usage	13
Figure 6.	Software architecture overview	20
Figure 7.	Project file structure	22
Figure 8.	IAR™ post-build actions screenshot	25
Figure 9.	CRC tool command line	25
Figure 10.	STM32 Nucleo board example	26
Figure 11.	State machine diagram - CPU test APIs	40
Figure 12.	State machine diagram - program integrity test APIs	41
Figure 13.	State machine diagram - RAM test APIs	42
Figure 14.	Test flow example	44
Figure 15.	Program integrity test flow example	46
Figure 16.	RAM test flow example	47

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.

© 2026 STMicroelectronics – All rights reserved