STM32G0 系列 IEC 60730 自检库用户指南

引言

本文档适用于包含 Arm® Cortex®-M0+ 内核的 STM32G0 系列微控制器使用的 X-CUBE-CLASSB 自检库。代码 X-CUBE-CLASSB-G0。

在电子产品应用中安全性是重中之重。面对日益提高的安全等级要求，电子设备制造商在组件设计中运用了大量新技术解决方案。提高产品安全性的技术不断发展进步的同时，也促进了安全标准的升级。

来自世界各地的权威机构所发布的标准中规定了最新的安全建议和要求。这些机构包括：国际电工委员会 (IEC)、美国保险商实验室 (UL) 和加拿大标准协会 (CSA) 等机构。

认证机构主要开展合规性、验证和认证工作。这些机构包括：德国 TUV 和 VDE（主要针对欧洲市场），以及 UL 和 CSA（主要针对美国和加拿大市场）。

与安全要求相关的标准范围非常广泛。这些安全标准涵盖诸多领域，如：分类、方法学、材料、机械、标签、硬件和软件测试。在测试可编程电子组件时需要遵守安全标准中的软件要求。当标准发布新版本时，这些要求通常会发生变化。

另外，在测试微控制器的通用组件（如 *CPU* 或内存）时，常见的安全标准高度相似。

本文档中介绍的库基于 ST 开发和应用的测试模块的子集，以满足严格的 IEC 61508 工业安全标准要求。这些模块经过调整，以满足家电安全的 IEC 60730 标准。因此，这个新库的交付格式有别于之前的版本。该格式源自工业安全库，目前作为一个没有源码但具有清晰外部接口定义的黑盒预编译对象交付。这个解决方案的优势在于它不依赖于特定的编译工具链。它也不依赖于任何其它固件，如 *HAL*、*LL* 或 *CMSIS*层。先前经过验证的旧版本库上的源代码文件由较新的编译器版本重新编译或与最新的固件驱动程序结合使用时，该解决方案可防止出现意外的编译结果。这是一种较为普遍的做法。

表 1. 适用产品

产品编号	订购代码
X-CUBE-CLASSB	X-CUBE-CLASSB-G0

1 概述

1.1 目的和范围

本文档适用于包含 Arm® Cortex®-M0+ 内核的 STM32G0 系列微控制器专用的 X-CUBE-CLASSB 自检库。X-CUBE-CLASSB-G0 扩展包提供不受特定应用限制的软件，符合 UL/CSA/IEC 60730-1 安全标准。UL/CSA/IEC 60730-1 安全标准涉及与家电和类似设备相关的自动电气控制器的安全。

此软件库的主要用途是为了推动：

- 用户软件开发
- 遵守相关要求和认证的应用程序的认证过程。

X-CUBE-CLASSB-G0 扩展包在基于 STM32G0 系列 Cortex®-M0+ 的微控制器上运行。

arm

注意： *Arm 是 Arm 公司（或其子公司）在美国和/或其他地区的注册商标。*

X-CUBE-CLASSB-G0 扩展包（本手册中所述）中提供的软件测试库（自检库 `STL_Lib.a` 文件）的版本为 V4.0.0。

1.2 参考文档

[1] UM2455，针对工业安全应用的 STM32G0 系列安全手册

[2] AN4435，在旧版本自检库的 STM32 应用中通过 UL/CSA/IEC 60730-1/60335-1 B 类认证的指南

2 STM32Cube 概述

2.1 什么是 STM32Cube？

STM32Cube 源自意法半导体，旨在通过减少开发工作量、时间和成本，大幅提高设计人员的工作效率。STM32Cube 涵盖整个 STM32 产品系列。

STM32Cube 包括：

- 一套操作便利的软件开发工具，覆盖从概念到实现的整个项目开发过程，其中包括：
  - 图形化软件配置工具 STM32CubeMX，可通过图形化向导自动生成初始化 C 代码
  - STM32CubeIDE，一种集外设配置、代码生成、代码编译和调试功能于一体的开发工具
  - STM32CubeCLT，一个集代码编译、开发板编程和调试功能于一体的命令行开发工具集
  - STM32CubeProgrammer (STM32CubeProg)，一种编程工具，提供图形用户界面和命令行工具
  - STM32CubeMonitor（STM32CubeMonitor，STM32CubeMonPwr，STM32CubeMonRF，STM32CubeMonUCPD）是功能强大的监控工具，用于实时微调 STM32 应用的行为和性能
- STM32Cube MCU 和 MPU 包，与具体的微控制器和微处理器系列（如用于 STM32G0 系列的 STM32CubeG0）相关的综合嵌入式软件平台，其中包含：
  - STM32Cube 硬件抽象层 (HAL)，增强 STM32 各个产品之间的可移植能力
  - STM32Cube 底层 API，通过硬件提供高度灵活的用户控制，确保最佳性能和内存开销
  - 一套统一的中间件组件，如 ThreadX、FileX/LevelX、NetX Duo、USBX、USB-PD、触摸感应库、网络库、mbed-crypto、TFM、OpenBLRTOS、USB 和图形库
  - 全部嵌入式软件实用工具以及全套外设以及应用示例
- STM32Cube 扩展包，包含的嵌入式软件组件可作为 STM32Cube MCU 和 MPU 包的补充：
  - 中间件扩展和应用层
  - 在特定的意法半导体开发板上运行的实现案例

2.2 此软件如何补充 STM32Cube？

软件扩展包通过中间件组件来扩展 STM32Cube 的功能，以管理特定的基于软件的诊断功能。

该包提供了一个通用的起点，帮助用户构建和完成应用特定的安全解决方案。其中包含：

- *STL*：自检库。它提供了一个二进制文件和一些源代码，用于管理微控制器的通用安全测试的执行。*STL* 是一个独立单元，不受任何 STM32 软件的影响。它包含了 MCU 通用组件的自检测试。
- 用户应用程序：这是一个 *STL* 集成示例，基于 STM32Cube 驱动程序，通过应用特定测试扩展了 *STL*。该部分作为完整源代码提供，可以通过调用终端用户定义的附加应用特定模块来调整或扩展。该示例可用于库测试，包括对所有提供的模块进行人为失效 (artificial failing) 测试。

3 STL 概览

STL 是 ST 发布的一个不受应用限制的软件测试库。其目标是实现与 STM32G0 系列微控制器相关的“B 类”安全标准所要求的安全机制的相关子集。*STL* 是一个独立于 *HAL/LL* 的库，专用于 STM32G0 系列微控制器。*STL* 是一个不依赖于特定编译工具链的库，因此任何标准 C 编译器都能编译它。

STL 是一个自主软件。它根据应用需求执行选定的测试来检测硬件问题，并向应用报告结果。

STL 由库文件（库本身），以及用于用户接口定义和用户参数设置的源代码组成。

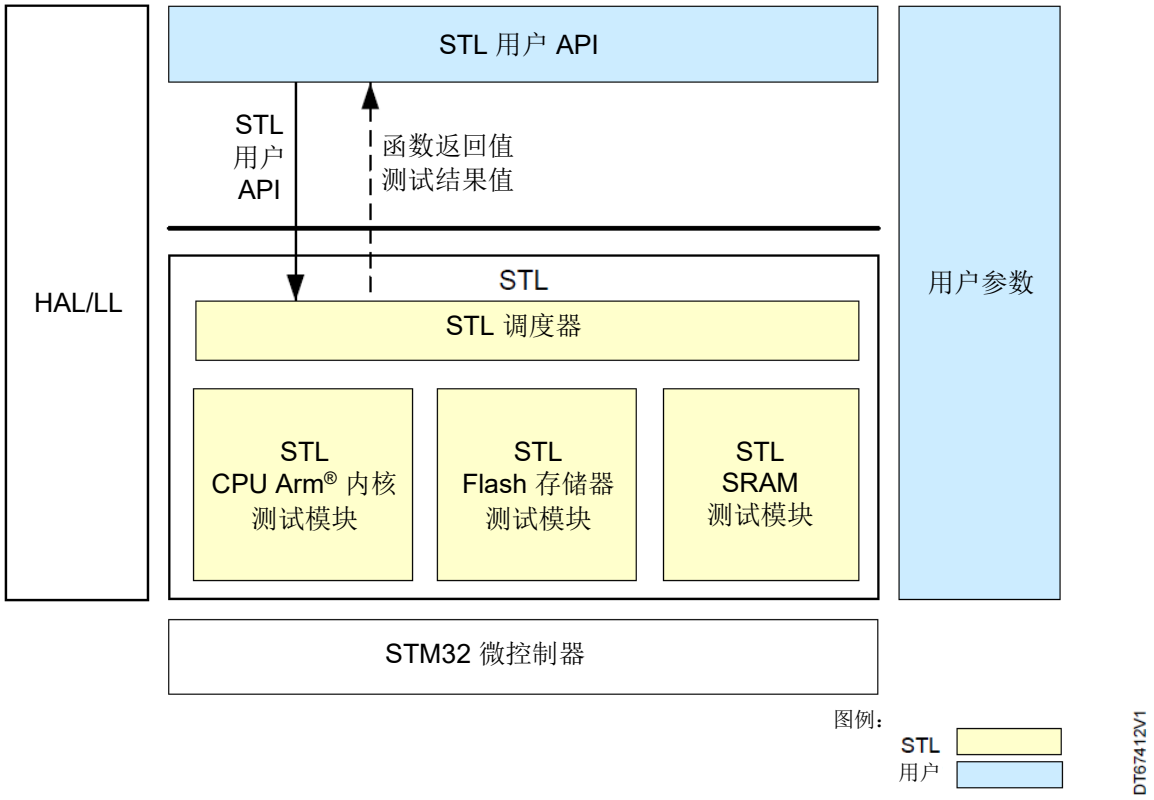
3.1 架构概述

STL 已测试 Arm® Cortex®-M0+ *CPU* 内核、Flash 存储器和 *RAM*。

如下图所示，集成了 *STL* 的终端用户应用程序的系统架构包括：

- 用户应用程序
- 用户参数
- *STL* 调度器：用户应用程序可以通过用户 *API* 直接访问它（不通过 *HAL/LL*）
- *STL* 内部测试模块：由 *STL* 调度器调用（对用户应用程序不可见）。在 *API* 级别返回给用户应用程序的 *STL* 状态信息（参见表 2）为：
- 函数返回值收集内部防御性编程检查的结果。
- 测试模块结果值存储测试结果信息。部分对应于模块的内部状态（参见第 7.3 节状态机）。

图 1. ***STL* 架构**



STL 还允许开发者：

- 使用人为失效功能。开发者可以通过强制 *STL* 返回一个请求的测试结果值来检查应用程序的行为。此功能可通过特定的用户 API 使用。

3.2 支持的产品

STL 在 STM32G0 系列微控制器上运行，这些微控制器具有相同的设计并集成 Cortex®- M0+ 内核和嵌入式存储器。

4 STL 描述

本节描述了 *STL* 功能和性能的基本信息。本节还总结了终端用户必须遵循的限制和强制性操作。

4.1 STL 功能描述

一些测试模块需要暂时屏蔽中断。有关详细信息，请参见第 4.2.5 节 STL 中断 屏蔽时间和第 4.3.4 节中断管理。

4.1.1 调度器原则

调度器是 *API* 模块，用户应用程序需要它来执行 *STL*。

主调度器：

- 使用前必须初始化
- 管理：
  - 应用测试模块的初始化和解除初始化
  - 应用测试模块的配置
  - 应用测试模块的重置。
- 控制应用测试序列的执行（*API* 调用）
- 管理用于用户调试和集成测试的“人为失效”。

通过特定的校验和确保关键内部数据结构的完整性。调度器控制以下测试的执行：

- *CPU* 测试：在执行任何 *CPU* 测试之前，不需要对 *CPU* 测试模块执行特定初始化或配置（参见第 7.2 节用户 API 和图 11）。
- Flash 存储器测试作用于定义了待测试存储器子集的 Flash 配置结构的内容（参见第 7.1 节用户结构）。这些结构必须由终端用户赋值，并在 Flash 存储器测试的配置和执行过程中保持内容不变。在执行任何 Flash 存储器测试之前，必须进行 Flash 测试模块的初始化和配置程序，请参见第 7.2 节用户 API 和图 12。
- RAM 存储器测试作用于定义了待测试存储器子集的 RAM 配置结构的内容（参见第 7.1 节用户结构）。这些结构必须由终端用户赋值，并在 RAM 测试的配置和执行过程中保持内容不变。在执行 RAM 测试之前，必须进行 RAM 测试模块的初始化和配置程序，请参见第 7.2 节用户 API 和图 13。

在轮询模式下，通过调度器 *API*，调用 *STL*。可以在中断上下文中调用 *STL*，但禁止重入。在这种情况下，无法保证 *STL* 的行为。

用户应用程序必须注意从 *STL* 返回的所有信息，这些信息通过一种特定的预定义数据结构提供，用于收集状态信息。详细信息请参见下表。

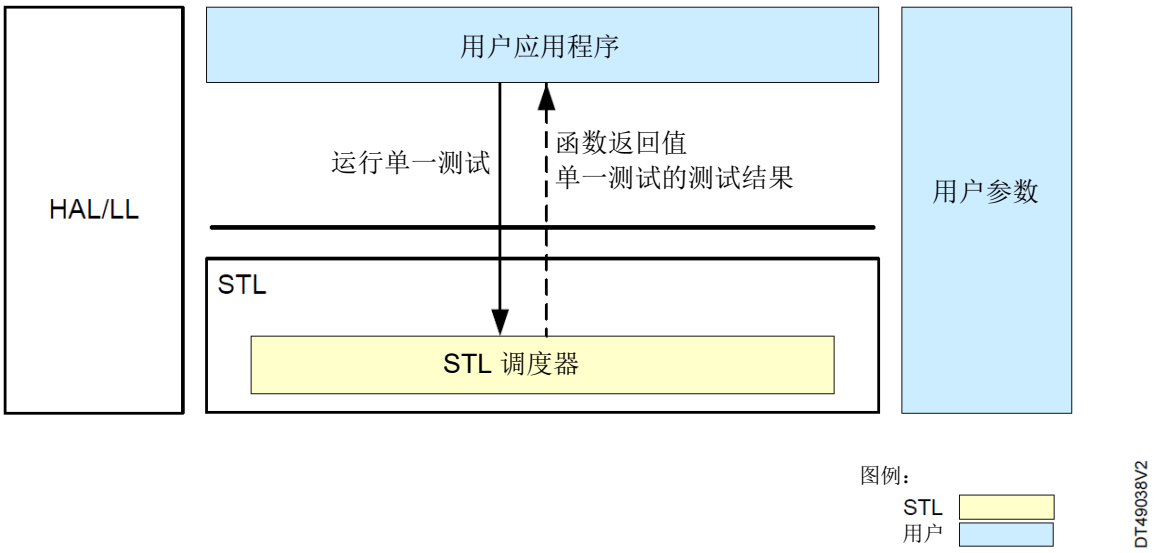
表 2. STL 返回信息

STL 信息	值	说明
函数返回值(1)	STL_OK	成功执行调度器功能
	STL_KO	调度器防御性编程错误（在这种情况下，测试结果无意义）
测试模块结果值(2)	STL_PASSED	通过测试
	STL_PARTIAL_PASSED	仅用于 RAM 和 Flash 存储器测试，当测试通过但尚未到达 RAM/Flash 存储器配置的结束位置时
	STL_FAILED	测试模块检测到的硬件错误
	STL_NOT_TESTED	未执行测试
	STL_ERROR	测试模块防御性编程错误

1. *参见第 7.1 节用户结构中的 `STL_Status_t` 定义。*
2. *参见第 7.1 节用户结构中的 `STL_TmSTatus_t`。*

用户应用程序重复应用下图所示的调用方案，以编程一系列 API 函数调用，从而处理测试模块执行的顺序。

图 2. 单一测试模式架构



调度器与中断

可以随时中断调度器。

4.1.2 CPU Arm® 内核测试

STL 包括下列 *CPU* 测试模块，以及测试能力的通用描述（仅供参考）：

- TM1L：实现通用寄存器的轻量级测试
- TM7：实现两个栈指针的模式和功能测试：*MSP* 和 *PSP*
- TMCB：实现 *APSR 状态寄存器*的测试。

小心： *STL CPU* 测试划分到若干独立的测试模块。并不是说用户程序只需要部分执行 *CPU TM*。而是方便在终端用户应用程序中更好地进行 *CPU* 测试调度，例如时序限制。默认情况下，所有可用的 *TM* 都将执行。

CPU Arm® 内核测试和中断

CPU 测试模块可以随时中断。TM7 仅在最小数据粒度时间内屏蔽中断。

4.1.3 Flash 存储器测试

原理

Flash 测试涉及 STM32G0 系列嵌入式 Flash 存储器。

必须采用以下结构才能提供 Flash 存储器测试的正确配置。

- Block：连续 4 字节 (FLASH_BLOCK_SIZE) 的区域，大小固定为 4 字节，不可更改。
- Section：连续 1024 字节 (FLASH_SECTION_SIZE) 的区域，大小不可更改，与 Flash 存储器的物理扇区无关。Flash 存储器从起始地址开始，被划分多个连续的 Section，比如 Flash 存储器大小为 128KB，那么总共包含 128*1024/1024=128 个 Section。每个 Section 对应一个 *CRC* 校验和，用户需要确保正确计算每个 Section 的 CRC 校验和并烧写到正确的位置，以便在 Flash 存储器完整性测试期间使用。
- Binary（在图 4 中称为“用户程序”）：在编译阶段生成的二进制代码段，通过烧录工具烧写到 Flash 存储器。该 Binary 的结束地址必须 4 字节对齐，起始地址必须和 Section 的起始地址对齐，我们按照 Section 大小把 Binary 划分为多个 Section，如果 Binary 大小不是 Section 大小的整数倍，那么最后一个 Section 就不是一个完整的 Section（请参见下面的 ST *CRC* 工具信息）。
- Subset：用户定义的一个或者连续多个 Section 区域。用户程序可以定义一个或者多个 Subset，Subset 必须在 Binary 范围内指定。Subset 的起始地址必须与 Section 的起始地址一致。而且 Subset 包含的 Section 必须预计算 *CRC* 校验值且存储在 CRC 对应区域。当 Subset 最后一个 Section 为 Binary 的最后一部分时，该 Section 可能是不完整的，也就是说其大小不足 1024 字节。所以用户应用程序必须确保 Subset 的结束位置与 Binary 的结束地址对齐。如果专门测试一组完整的 Section，那么 Subset 的结束地址必须与最后测试的 Section 结束地址对齐。

  Subset 的计算方式如下：

  Subset 大小 = K * FLASH_SECTION_SIZE + L * FLASH_BLOCK_SIZE

  其中：

  - K 是大于 0 的整数。
  - L 为 0 或者小于 (FLASH_SECTION_SIZE / FLASH_BLOCK_SIZE) 的正整数，当 L 不等于 0 的时候，此时 Binary 最后一个 Section 是不完整的。

  用户应用程序定义 Subset。

注意： *可以定义多个 Subset。*

STL 根据以下原则实现 Flash 存储器的测试（基于用户配置结构的实际内容）：

- 对用户程序定义的一个或者多个 Subset 区域，以 section 为单位进行测试。
- 当用户定义的测试区域 (Subset) 包含多个 section 时，测试可以连续进行（一次性完成所有 section 的测试），也可以单步进行（每次测一个 section），具体由用户应用程序定义。
- 测试结果基于计算的 *CRC* 值（在测试执行期间计算）与预期的 *CRC* 值（在 Binary 烧录之前计算）之间的比较。

用户代码执行 Flash 存储器测试的必要步骤是：

- 测试初始化
- 配置一个或多个 Subset
- 执行测试。

所有 Subset 完成测试后，用户必须重置 Flash 存储器测试模块才能再次执行测试。

如果出现 STL_ERROR / STL_FAILED 测试结果，测试模块会卡在失败的存储器 Subset 上。在这种情况下，在重新运行测试之前，需要再次执行去初始化/初始化 Flash 测试模块和配置 Flash 测试模块的操作。

预期的 CRC 预计算

Flash 存储器测试基于内置硬件 *CRC* 单元或软件 *CRC*，可通过宏定义进行配置。默认配置是使用硬件 *CRC*。要使用软件 *CRC*，必须按照第 5.5.2 节从头构建应用程序的步骤第 3 步的定义启用标志 STL_SW_CRC。*CRC* 是一个 32 位的 *CRC*，符合 IEEE 802.3 标准。

预留部分 Flash 存储区作为 *CRC* 专用区域，其大小取决于 Flash 存储器的大小。这个区域采用一种特殊字段格式，其中每个 Flash 存储器区段都有足够的保留空间来存储一个 32 位的 *CRC* 模式。对于所有要测试的区段，用户必须确保计算得出有效的 *CRC* 值，并存储在字段中。相关内容如 图 3 所示。

针对二进制文件的每个连续区段（从二进制文件开始到二进制文件结束）预计算一个预期的 *CRC* 值。这意味着可测试的区段数量取决于二进制文件大小。二进制区域通常与区段大小不一致。在这种情况下，只在覆盖二进制区域的区段部分预计算最后一个不完整区段的 *CRC* 检查值并进行测试。

前提条件：

- 用户程序区域必须从一个区段的起始位置开始
- 用户程序区域的边界必须是 32 位对齐的。
- 根据 Flash 存储器总大小和用户程序的大小，最后一个程序数据和第一个 *CRC* 数据可能都存储在同一个 Flash 存储器区段中（没有任何重叠）。在这种情况下，只能对用户程序数据执行 *CRC* 计算，参见图 4 中的示例 3。

ST CRC 工具信息

ST 提供了一个 *CRC*预计算工具。该工具集成在 STM32CubeProgrammer 内（参见第 6.2.2 节 *CRC* 工具设置），它自动用填充位（0x00 模式）填充二进制文件以实现 32 位对齐。

图 3. **Flash 存储器测试：*CRC* 原理**

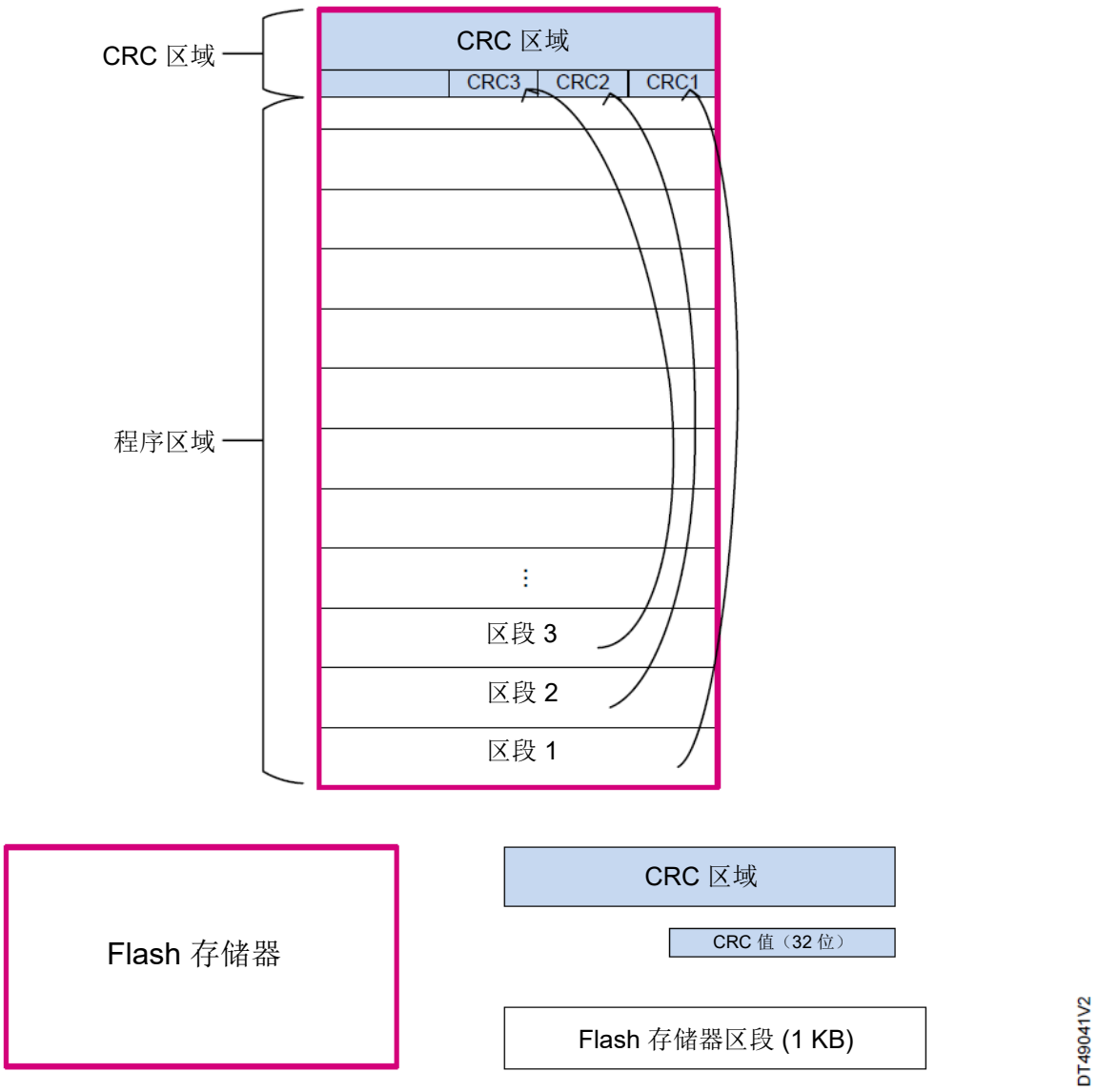
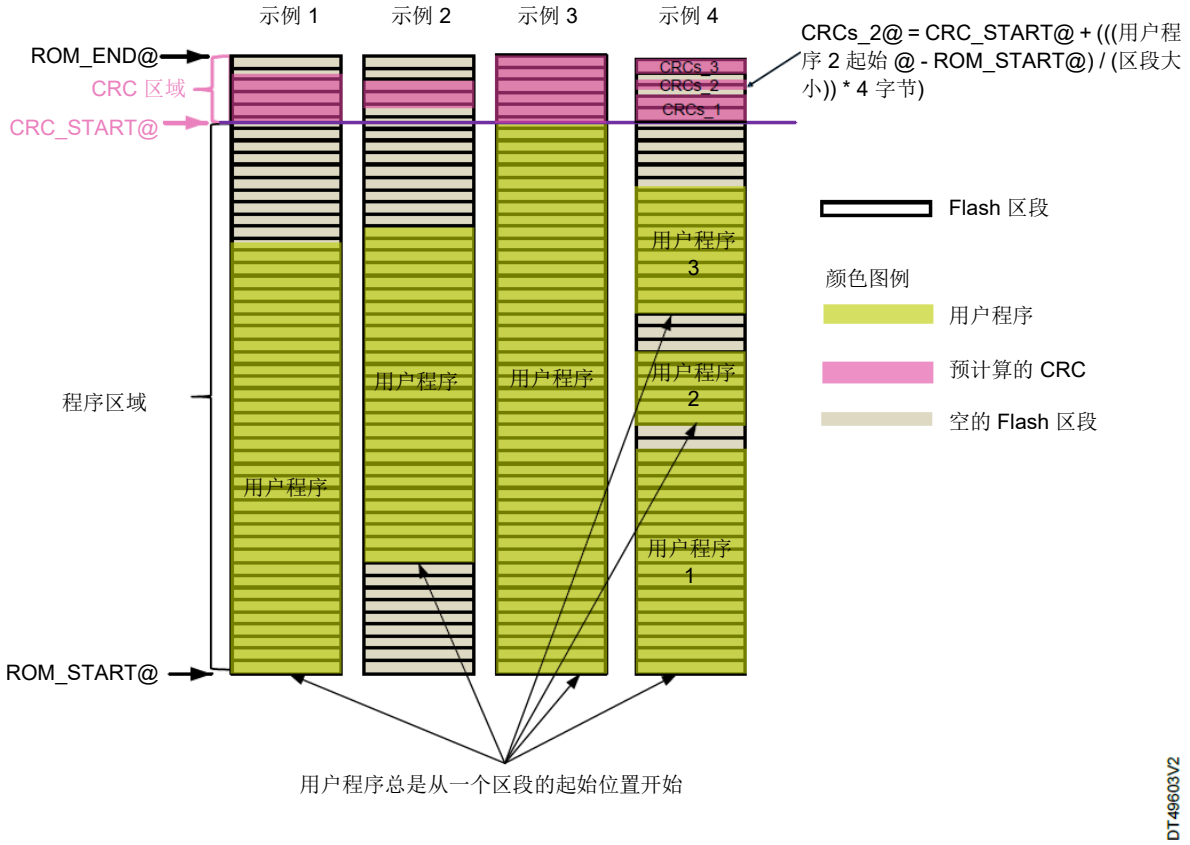


图 4. **Flash 存储器测试：*CRC* 用例与程序区域**



用例描述如图 4 所示:

- 示例 1：用户程序从 ROM_START 地址开始，因此 *CRC* 从 CRC_START 地址存储。
- 示例 2：用户程序从一个区段的起始位置开始，但不是从 ROM_START。存储的 *CRC* 从 *CRC* 区域的正确地址开始。
- 示例 3：用户程序使用完整的程序区域，最后一个程序数据和第一个 *CRC* 数据都存储在同一个 Flash 存储器区段中（没有任何重叠）。
- 示例 4：用户程序在三个独立区域中定义。因此需要三个独立的区域来存储 *CRC* 数据。

CRC 起始地址计算：

- 实际计算：
  CRC_START 地址 = (uint32_t *)(ROM_END - 4 * (ROM_END + 1 - ROM_START)/(FLASH_SECTION_SIZE) + 1)；其中 FLASH_SECTION_SIZE = 1024
- 文字翻译：
  CRC_START = ROM_END - (*CRC* 字节数) * (Flash 存储器区段数量) + 1

Flash 存储器测试与中断

Flash 存储器 *TM* 可随时中断。

4.1.4 RAM 测试

原理

RAM 测试涉及 STM32G0 系列的嵌入式 SRAM 存储器。

必须采用以下结构才能提供 RAM 测试的正确配置。

- Block：一个 16 字节的连续区域 (RAM_BLOCK_SIZE)，由 *STL* 硬编码（与存储器的物理扇区无关）。
- Section：一个连续的 128 字节区域 (RAM_SECTION_SIZE)，由 *STL* 硬编码。
- Subset：连续的区域，其大小是两个块的倍数，起始地址按 32 位对齐。Subset 的大小不一定是区段大小的倍数，因为 Subset 的最后一个部分可能小于一个区段。
- Subset 大小 = N * RAM_SECTION_SIZE + 2 * M * RAM_BLOCK_SIZE，

  其中：
  - N 是一个 ⩾ 0 的整数
  - M 是一个 0 ⩽ M < 4 的整数，当 M > 0 时，最后一个部分 Subset 的大小不与区段大小对齐。

  用户应用程序定义 Subset。

注意： *可以定义多个 Subset。*

STL *根据以下原则实现 RAM 存储器测试（基于用户配置结构的实际内容）：*

- RAM 测试在用户应用程序定义的 RAM 块上进行
- RAM 测试一次性连续完成，或者对用户应用程序定义的区段，以单一原子步骤部分完成
- 测试实现基于 March C- 算法

执行 RAM 测试的必要步骤（对于用户应用程序）是：

- 初始化 RAM 测试
- 配置一个或多个 RAM Subset
- 执行 RAM 测试

一旦所有 Subset 都完成测试，应用程序必须重置 RAM 测试模块以便再次执行测试。

如果出现 STL_ERROR / STL_FAILED 测试结果，测试模块会卡在失败的存储器子集上。在这种情况下，在重新运行测试之前，需要解除初始化、初始化并重新配置 RAM。

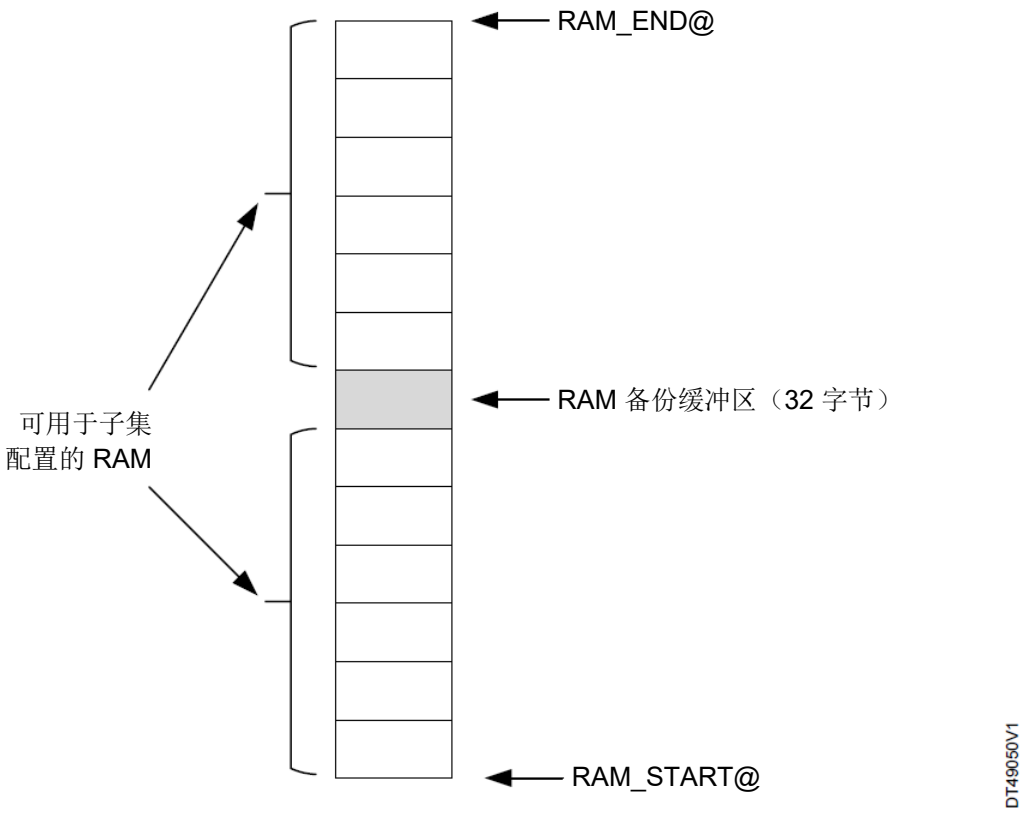
RAM 测试与中断

RAM *TM 可以随时中断，唯一的例外是在执行最小数据粒度块期间，如*第 4.2.5 节 STL 中断屏蔽时间中的定义。在这种情况下，默认情况下 STM32 中断和 Cortex® 异常（具有可配置优先级）会被暂时屏蔽。然而，终端用户可以通过应用专用的 STL_ENABLE_IT 编译开关来停用这种屏蔽（参见第 5.5.2 节从零开始构建应用程序的步骤）。如果激活 STL_ENABLE_IT 编译开关，终端用户需要负责管理 *STL 和应用软件之间的干扰*。这些接口可能因为测试导致 RAM 内容破坏而导致错误的 STL 报告或应用软件故障（参见 March C- 测试原理和存储器备份原理）。

March C- 测试原理和存储器备份原理

RAM 测试基于 March C- 算法，该算法通过特定模式覆写存储器，然后按特定顺序读回。为恢复初始存储器内容，默认启用了备份过程。在这种情况下，用户必须分配一个特定的备份区域，该区域也会被 March C- 测试。如果不需要备份过程，可以选择禁用它。参见第 4.3.7 节 RAM 备份缓冲区了解有关缓冲区控制和分配的详细信息。

图 5. **RAM 测试：用法**



4.2 STL 性能数据

数据是基于下面设置获得的：

- *STL* 库编译细节，请参见“应用：编译过程”。
- 性能测试项目使用 IAR Embedded Workbench® for Arm® (EWARM) toolchain v9.20.1 进行编译
- 编译的软件配置包括：
  - 将 HCLK 时钟设置为 56 MHz
  - 将 Flash 存储器延迟设置为四个等待周期
  - 使能 Flash 预取
  - NUCLEO-G071RB rev B (MB1360 C)

4.2.1 STL 执行时间

表 3 包含了应用最佳默认 *STL 设置时 STL 执行时间的摘要*。每个 *API 的测量数据的详细描述，请参见*第 8 节 STL：执行时间详细信息。

表 3. STL 执行时间，时钟频率为 56 MHz

测试模块	条件		结果 (µs)
CPU	TM1L，TM7，TMCB 默认配置（未启用 STL_ENABLE_IT）：		132
Flash 存储器	默认配置（未启用 STL_SW_CRC）：	测试了 1 KB	65
		测试了 17 KB	860
	启用 STL_SW_CRC：	测试了 1 KB	334
		测试了 17 KB	5289
RAM	默认配置（既未启用 STL_DISABLE_RAM_BCKUP_BUF，也未启用 STL_ENABLE_IT）：	测试了 128 KB	142
		测试了 36 KB	34032

4.2.2 STL 代码和数据大小

下表详细列出了 STL 代码和数据的大小。

表 4. STL 代码大小和数据大小（以字节为单位）

配置	模块	Flash 存储器代码	Flash 存储器 RO 数据	R/W 数据
未启用 STL_SW_CRC	stl_user_param_template.o	-	5	44
	stl_util.o	230(1)	-	8
	stl_lib.a	4696	1459	184
启用 STL_SW_CRC	stl_user_param_template.o	-	5	44
	stl_util.o	114(1)	-	4
	stl_lib.a	4696	1459	184

1. *当使用软件 CRC 计算时，编译的代码/函数会减少（缺少 CRC 硬件初始化），因此与硬件 CRC* 相比，目标代码的大小有所降低。

4.2.3 STL 栈用量

STL 执行可用 *API* 所需的最小栈可用空间为 200 字节。

4.2.4 STL 堆用量

STL 从不使用动态分配，因此堆大小与 *STL* 无关。

4.2.5 STL 中断屏蔽时间

STM32 中断和 Cortex® 异常（具有可配置优先级）在执行 *CPU* TM7 和 RAM 测试期间被 *STL* 多次屏蔽。如下表所示，获得 RAM 测试的最大中断时间（参见表 5）。

表 5. *STL* 最大中断屏蔽信息

测试模块	最大持续时间 (µs)	注释
RAM	12	每次执行 STL_SCH_RunRamTM 功能在测试的部分步骤期间执行一系列中断屏蔽，具体持续时间如下： • 备份缓冲区为 12 µs + • 测试的第一个 RAM 块为 11 µs • 测试的每个中间 RAM 块为 12 µs(1) • 测试的最后一个 RAM 块为 11 µs
CPU TM7	9	屏蔽两次，每次 9 µs

1. *每次执行 RAM 测试涉及的 RAM 块数量（两个 RAM_BLOCK_SIZE 的倍数）取决于用户结构的内容（定义的子集大小与原子步骤比较 - 参见第 4.1.4 节 RAM 测试）*

4.3 STL 用户约束

终端用户需要注意应用程序与 *STL* 之间的干扰。忽视这一点可能导致错误的 *STL* 误报，和/或应用软件执行问题。因此，为了防止任何干扰，应用软件和 *STL* 集成必须遵守本节列出的每个约束。

4.3.1 特权级别

CPU TM7 和 RAM TM 必须以特权级别执行，才能修改某些内核寄存器（例如 PRIMASK 寄存器）。

4.3.2 RCC 资源

在 *STL* 执行期间，RCC 被配置为在执行所有 TM 期间为 *CRC* 提供时钟。这意味着：

- 当 *STL* 返回时，它会恢复用户对 *CRC* 的 RCC 时钟设置（启用或禁用）
- 用户应用程序在 *STL* 执行期间通过保存/恢复 *STL* 设置来配置 RCC 时应小心。

4.3.3 CRC 资源

当使用硬件 *CRC* 时，*STL* 依赖于 STM32 CRC IP。在 *STL* 执行期间使用 *CRC* 资源的两种不同情况：

- 在执行 *STL* 初始化（函数 STL_SCH_Init）时：使用硬件 *CRC*。在此阶段，硬件 *CRC* 的使用不能被应用软件修改，所以在执行 STL_SCH_Init 函数期间 STL_SW_CRC 标志没有影响。
- 在执行其他 STL 函数期间：应用程序可以通过 STL_SW_CRC 标志在硬件 *CRC* 和软件 *CRC* 之间进行选择。默认情况下，使用硬件 *CRC*（STL_SW_CRC 标志被禁用）。

使用硬件 *CRC* 意味着：

- 在调用 *STL* 之前，用户应用程序必须保存完整的硬件 *CRC* 配置。用户配置必须在 *STL* 执行后恢复。
- 在 *STL* 执行期间，配置硬件 *CRC* 并用于 *STL* 需求（用户应用程序在 *STL* 执行期间使用 *CRC* 时必须保存/恢复 *STL* 设置）。

4.3.4 中断管理

升级机制 - Arm® Cortex® 行为提醒

当 *STL* 禁用 STM32 中断和 Cortex® 异常（可配置优先级）时，要注意 Arm® Cortex® 可能会升级到 HardFault。在这种情况下，会调用 HardFault 处理程序而不是故障处理程序。

中断和 CPU TM7

默认情况下，CPU TM7 期间会屏蔽 STM32 中断和 Cortex® 异常（可配置优先级），除非用户应用程序激活 STL_ENABLE_IT（参见第 5.5.2 节从头开始构建应用程序的步骤）。

如果激活了 STL_ENABLE_IT 标志，则无法保证正确的 STL CPU TM7 行为。这可能导致 STL 产生错误的测试误报或无法检测到硬件故障。

中断和 RAM March C- 测试

默认情况下，RAM March C- 测试期间会屏蔽 STM32 中断和 Cortex® 异常（可配置优先级），除非用户应用程序激活 STL_ENABLE_IT（参见第 5.5.2 节从头开始构建应用程序的步骤）。

如果激活了 STL_ENABLE_IT 标志：

- 无法保证正确的 *STL* RAM 测试行为，因为应用程序在中断处理期间可能会重写 STL 测试的 RAM 空间。这可能导致 *STL* 产生错误的 RAM 测试误报。
- 用户应用软件的行为可能会受到损害。可能会从正在被 *STL* March C- 测试修改的 RAM 位置读取或使用错误的数据。

中断和通用寄存器

在 *STL* 执行期间，应用程序必须保存和恢复 STM32 中断和可配置优先级的 Cortex® 异常服务例程中的通用寄存器，以确保正确的 *STL* 行为和防止任何错误误报。

4.3.5 *STL* 如何屏蔽中断

为了屏蔽中断，*STL* 将 PRIMASK 寄存器位设置为 1。将此位设置为 1 将当前执行优先级提升到 0，防止激活所有具有可配置优先级的异常。因此，当当前执行优先级提升到特定值时，所有具有较低或相等优先级的异常都将被屏蔽。

4.3.6 DMA

应用程序必须管理 DMA，以避免在 *STL* March C- 测试期间对 RAM 存储区的意外访问。在这种情况下：

- DMA 写入可能会干扰 *STL* 测试，导致误报
- 因 *STL* 对 DMA 专用 RAM 部分的重写，DMA 读取可能会返回错误数据。

4.3.7 RAM 备份缓冲区

在 RAM 测试期间，默认启用备份过程以保留 RAM 内容。用户必须在编译时为 RAM 备份缓冲区保留一个特定区域，该区域必须在 RAM 子集配置之外分配。测试只定义了一个 RAM 备份缓冲区。每次调用 RAM 运行测试时（在测试用户定义的任何子集之前），RAM 备份缓冲区区域也会被 March C- 算法测试。可以通过激活编译开关 STL_DISABLE_RAM_BCKUP_BUF（参见第 5.5.2 节从头开始构建应用程序的步骤）永久地或通过特定控制序列临时地禁用备份过程（参见下面的说明）。在这种情况下，终端用户有责任确保应用程序软件不使用被 March C- 测试破坏的数据。此选项可用于加快测试那些用户不需要保留存储器内容的 RAM 区域。

注意： *为了临时禁用 RAM 备份缓冲区，用户必须遵循一组序列：*

1. *更改 `STL_pRamTmBckUpBuf` 变量值将其重写为 NULL，同时保留原始值的备份（STL 存储的默认值）。*
2. *然后必须重新启动 RAM 测试。为此，用户可以使用其中一个 API，将 RAM 测试强制转入 RAM_IDLE 或 RAM_INIT 状态（参见第 7.3 节状态机）。*

为了移除备份禁用，用户必须再次执行上述步骤，同时将 `STL_pRamTmBckUpBuf` 的默认值恢复为其原始值并重新初始化 RAM 测试。

4.3.8 存储器映射

由于 RAM 测试模块和 March C 方法的设计，用户必须确保

STL 的“只读”数据位于 Flash 存储器中。这必须通过适当调整相关链接器文件来完成。以下示例适用于 EWARM 和 STM32CubeIDE。

EWARM .icf 文件调整示例

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

STM32CubeIDE.ld 文件调整示例

```
.rodata :
{
……
} >ROM
```

注意： *通常，默认配置将“只读”数据放在 Flash 存储器中。*

4.3.9 处理器模式

为了将活动栈指针设置为进程栈指针，*STL* 必须在线程模式下执行。如果 *STL* 未在线程模式下执行，*CPU* TM7 会返回 STL_ERROR。

4.4 最终用户集成测试

本节描述了在验证阶段终端用户必须执行的强制性测试。这些测试保证了 *STL* 正确集成到应用软件中。

4.4.1 测试 1：正确执行 STL

终端用户必须使用预期的函数返回值和预期的测试模块结果值（参见第 7.2 节用户 API），以检查每个计划的诊断功能是否已正确执行。这涉及到测试模块的执行和它们所有的配置操作。

4.4.2 测试 2：正确处理 STL 错误信息

终端用户必须检查由 *STL* 函数返回值和测试模块结果值产生的任何错误信息是否被正确理解为异常行为，并在其应用软件中得到正确处理。错误信息指的是与预期值不同的值，参见第 7.2 节用户 API）。在验证过程中，针对所使用的每个具体功能必须使用人为失效功能来模拟产生与相关单个软件诊断（*CPU* 测试、RAM 测试、Flash 存储器测试）相关的错误测试模块结果值。

人为失效功能并不是对真实设备上的实际 *CPU* 故障的全面模拟，而只是实现了 *API* 的测试。

5 软件包说明

本节详细介绍了 X-CUBE-CLASSB-G0 扩展包内容及其正确使用方式。

5.1 概述

X-CUBE-CLASSB-G0 是针对 STM32G0 系列微控制器的软件扩展包。

它提供了一个完整的解决方案，帮助最终客户构建安全应用程序：

- 提供一个独立于应用的软件测试库：
  - 部分为目标代码形式：`STL_Lib.a`，即库本身
  - 部分为源文件形式：`stl_user_param_template.c` 和 `stl_util.c`
  - 附带三个头文件：`stl_stm32_hw_config.h`，`stl_user_api.h` 和 `stl_util.h`
- 一个以源代码形式提供的用户应用示例。

X-CUBE-CLASSB-G0 已移植到第 3.2 节支持的产品中列出的产品上。

该软件扩展包中包含应用示例，开发人员可以将其用于试验代码。它以 zip 归档的形式提供，其中包含源代码和库。

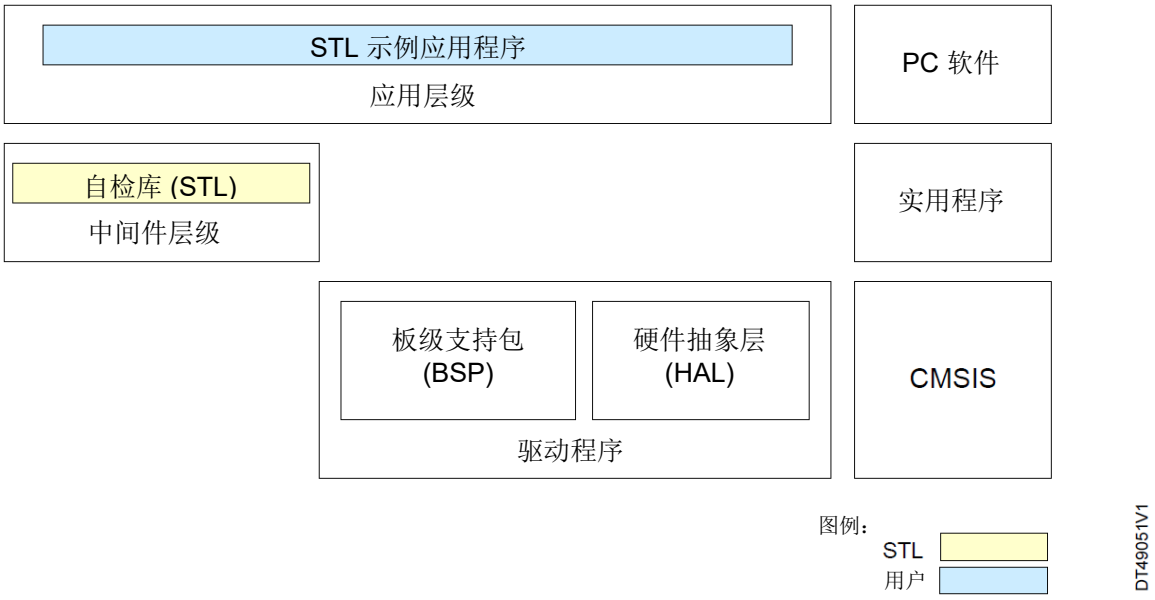
可支持以下集成开发环境：

- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® 微控制器开发套件 (MDK-ARM)
- STM32CubeIDE。

5.2 架构

X-CUBE-CLASSB-G0 扩展包的组件如图 6 中所示。

图 6. 软件架构概述



5.2.1 STM32Cube HAL

HAL 驱动层提供简单通用的 API（应用程序编程接口），以便与上层（应用、库和协议栈）交互。

它包括通用和扩展的 *API*。它以通用架构为基础直接构建，允许在其基础之上构建的层，例如中间件层，实现它们的功能，同时不受给定微控制器的特定硬件配置的限制。

此结构可提高库代码的可复用性，并轻松移植到其他设备。

5.2.2 板级支持包 (BSP)

除了 MCU 之外，软件包需要支持 STM32 开发板上的外设。该软件包含在板级支持包 (BSP) 中。这是一个有限的 *API* 集，为特定板组件（例如 LED 和用户按钮等）提供编程接口。

5.2.3 STL

在中间件层级可用的 *STL* 的一个重要部分是一个黑盒，它管理基于软件的诊断测试。它独立于 *HAL*、*BSP* 和 *CMSIS*，即使 *STL* 的集成示例依赖于某些 *HAL* 驱动程序。

5.2.4 用户应用程序示例

该示例展示了如何将 *STL* 测试模块调用的可能序列集成到应用程序中，验证 *API* 的返回值，并人为模拟它们的失败响应。此外，还包括一个专门用于测试时钟系统的模块（采用符合“*B 类*”标准要求的监视方法）以及完整源代码，以扩展现有的库集。它演示了库如何通过终端用户完全定义的特定测试或模块来实现扩展。

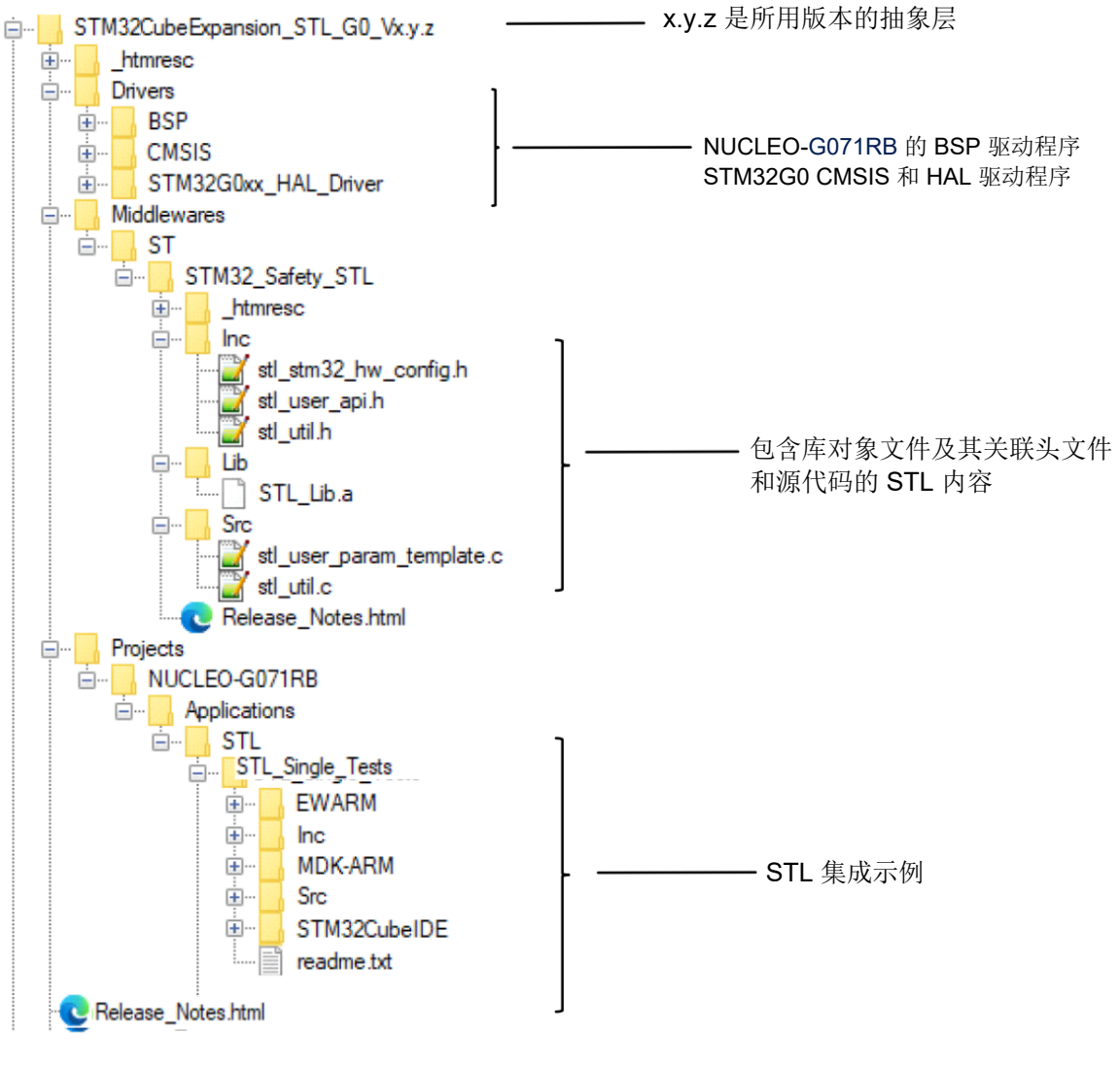
5.2.5 STL 完整性

通过散列 SHA-256 保证 STL 内容的完整性。

5.3 文件夹结构

图 7 中显示了结构的顶层视图。

图 7. 项目文件结构



5.4 API

5.4.1 合规

接口合规性

STL 的库部分不以源代码形式提供。此库已使用 IAR Embedded Workbench® for Arm v9.20.1 编译。编译操作使用 `--aeabi` 和 `--guard_calls` 编译选项完成，以实现 AEABI 合规性，如 EWARM 帮助中“AEABI 合规性”所述。

为了符合 IEC 61508 功能安全标准，经 TUV 认证的工业版本 *STL* 专门由安全认证的 EWARMFS 版本 IAR™ 编译器编译。此 *STL* 的源代码基于库的工业版本，并根据 IEC 60730-1 标准进行调整，并使用最新的 EWARM 编译器重新编译。此库可由任何标准版本的 EWARM 编译器编译，无需专门的安全认证。

安全指南

为了满足 IAR Embedded Workbench® 安全指南（建议 2.1-1、2.2-5、2.4-1a 和 5.4-3）和 Keil® 安全手册 (§4.9.2) 中描述的安全指南合规性，通过 `--strict`、`--remarks`、`--require_prototypes` 和 `--no_unaligned_access` 编译选项实现合规性。

库合规性

STL 的库部分（未以源代码形式提供）符合 C 标准库 ISO C99。

它已使用 IAR™ 选项编译。C 语言版本 = C99。

Arm 编译器 C 工具链厂商/版本独立性

STL 用户 *API* 仅引用“`uint32_t`”和“`enum`”C 类型：

- 根据 C99 标准，“`uint32_t`”C 类型是大小为 32 位的固定类型
- 根据 C99 标准，“`enum`”C 类型的大小由实现定义。它必须能够表示所有枚举成员的值。在 *STL* 接口中，`enum` 类型值是小于等于 (2^{32} - 1) 的无符号整数。用户必须确保 `enum` 类型的值可以存储 32 位的数值。

5.4.2 依赖关系

STL 库调用 `memset` 标准 C 库函数。

此外，IAR™ EWARM 工具链编译器用于编译 *STL* 库。在某些情况下，此编译器可能会调用以下标准 C 库函数：`memcpy`、`memset` 和 `memclr`。这种行为是 IAR™ EWARM 工具链编译器的固有特性。无法禁用或避免此行为。

因此，在关联 *STL* 库时，用户必须确保这些标准 C 库函数已定义。用户可以使用工具链提供的函数或用户自己的函数。

5.4.3 详细信息

有关可用的 *API* 的详细技术信息，参见第 7.2 节用户 API，其中描述了函数和参数。

5.5 应用程序：编译过程

5.5.1 构建交付的 STL 示例的步骤

在交付的 *STL* 示例中，*STL* 在安全二进制文件中运行，而应用程序在非安全二进制文件中运行。确保完成以下步骤：

1. 安装 ST *CRC* 工具（参见第 6.2.2 节 *CRC* 工具设置）或其他 *CRC* 工具，以生成执行 Flash 测试所需的适当结构。
2. 项目选择：选择一个项目示例并打开它。
3. 项目构建：启动编译二进制文件的生成命令，以及调用 *CRC* 工具的后生成命令，以计算和分配 *CRC* 结果。如果出现错误，请检查 *CRC* 工具路径。有关详细信息，参见第 5.5.2 节从头开始构建应用程序的步骤。
4. 加载编译后的二进制文件。
5. 执行。

启动开发板并检查结果：

- LED 定期切换：测试结果符合预期。
- LED 不规则切换：存在错误。

如果任何测试返回失败结果，LED 每 2 秒闪烁一次。如果 *STL* 检测到防御性编程错误，LED 每 4 秒闪烁一次。

然后调用 `FailSafe_Handler` 程序，并传入一个参数以保留失败模块的标识码

注意： *代码定义在 stl_user_api.h 文件中提供，如果发生防御性编程失败，DEF_PROG_OFFSET 将被添加到模块代码中。*

5.5.2 从头开始构建应用程序的步骤

要从头开始构建应用程序，请按照以下步骤操作：

1. 创建一个具有适用目录结构并包含所有适当包的新应用程序项目。使用 STM32CubeMX 工具来自动完成。
2. 如果项目中任何自动包含的 *STL* 选项不受 STM32CubeMX 工具的支持，请从交付的 *STL* 示例中复制 ...Middleware\ST\STM32_Safety_STL 目录中的内容，并粘贴到应用程序项目目录结构中。参见第 5.3 节文件结构。在这种情况下，按照以下步骤手动修改项目设置：
   - 将 SRC 目录下的所有 *STL* 源文件添加到项目中
   - 将 INC 目录指定为要包含在项目中的额外目录
   - 强制链接器将位于 LIB 目录的库对象文件作为附加库包含在内

   注意： *只有在 CubeMX 工具不支持自动包含的选项时，才需要执行这些步骤，否则将完全由工具执行，无需上述人工干预，用户可以略过这些步骤，继续执行第 3 步*
3. 必要时，在项目设置中添加下一个可选的预处理器编译开关：
   - 启用 STL_SW_CRC 选项：用户应用程序在此选择软件 *CRC*。如果未激活，将默认使用硬件 *CRC* 计算。
   - 启用 STL_ENABLE_IT 选项：用户应用程序在 CPU TM7 和 RAM 测试期间在此启用 STM32 中断。如果未激活，这些测试期间中断将被屏蔽。参见第 4.3.4 节中断管理和第 4.1.4 节 RAM 测试。
4. 检查 Flash 存储器容量的配置。

   必须在 *stl_user_param_template.c* 文件中为项目设置正确的 Flash 存储器容量范围。必须更新 STL_ROM_END_ADDR，以确保与关联的链接器分散文件和 *CRC* 工具脚本一致（参见步骤 6.）。
5. 开发用户 *STL* 流控制。根据定义的安全任务要求，这是通过在周期性循环中重复执行 *API* 调用的适当序列实现的。

   必须确保正确填充所有相关用户结构以控制存储器测试，并正确检查 *STL* 返回信息。请参见第 7 节 STL：用户 API 和状态机。
6. 应用 *CRC* 工具构建 *CRC* 计算所需的 *CRC* 区域内容。请参见第 6.2.2 节 *CRC* 工具设置。

   执行 STM32CubeProgrammer 的正确命令行。这可以在编译过程中通过调用 IDE 后生成功能操作自动完成，如图 8 和图 9 所示。
7. 编译、加载并执行二进制文件。

如果 *STL* 检测到硬件故障，人为失效 *API* 可以用于调试经过编程的 *STL* 流的正确行为。

图 8. IAR™ 后生成操作截图

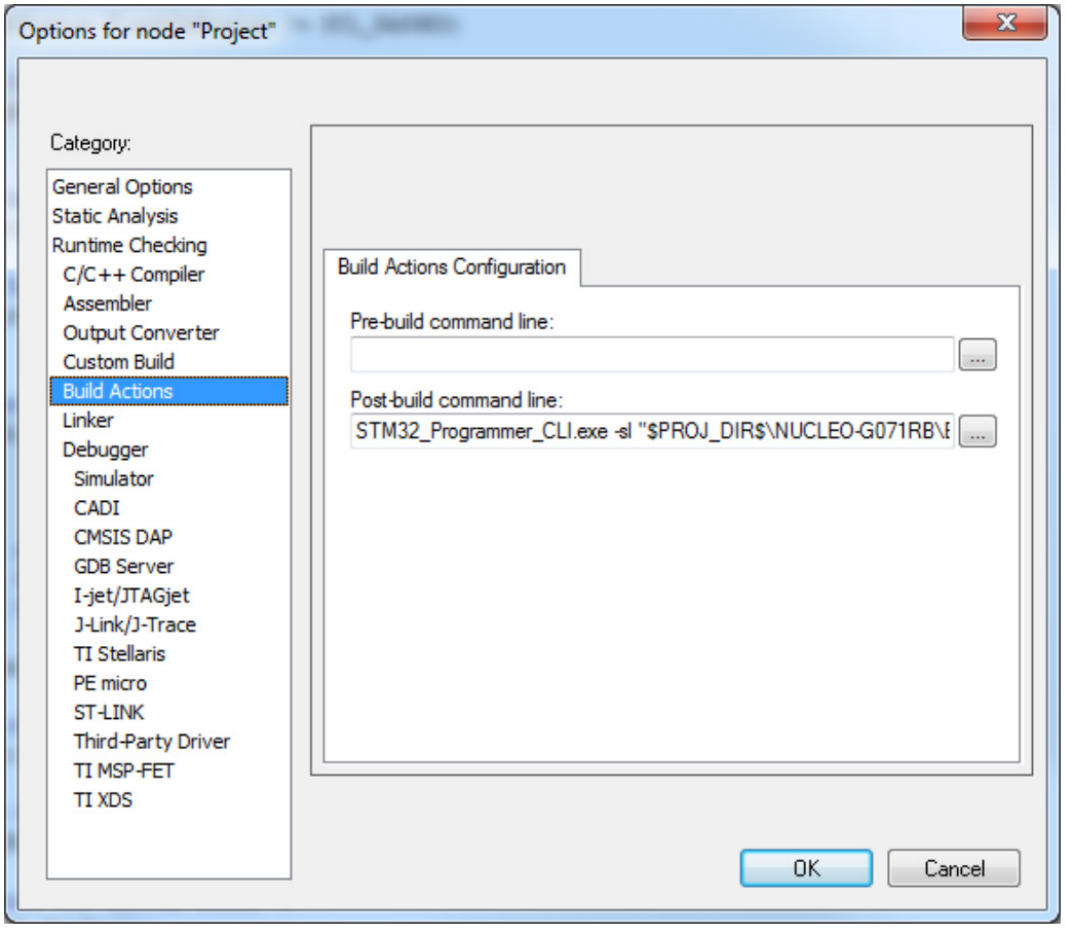
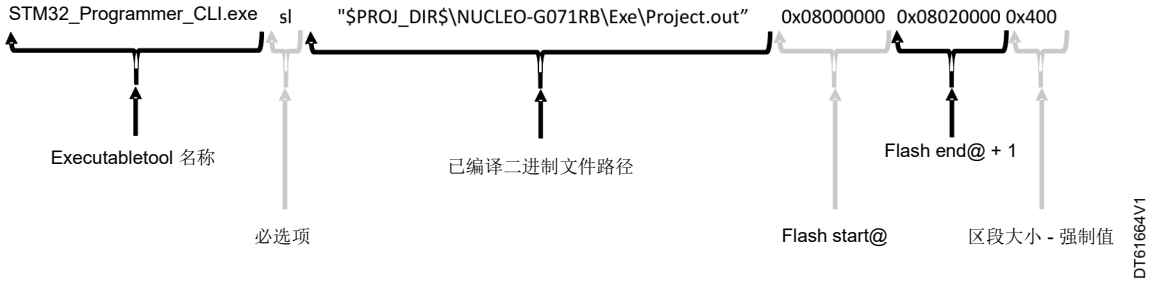


图 9. *CRC* 工具命令行



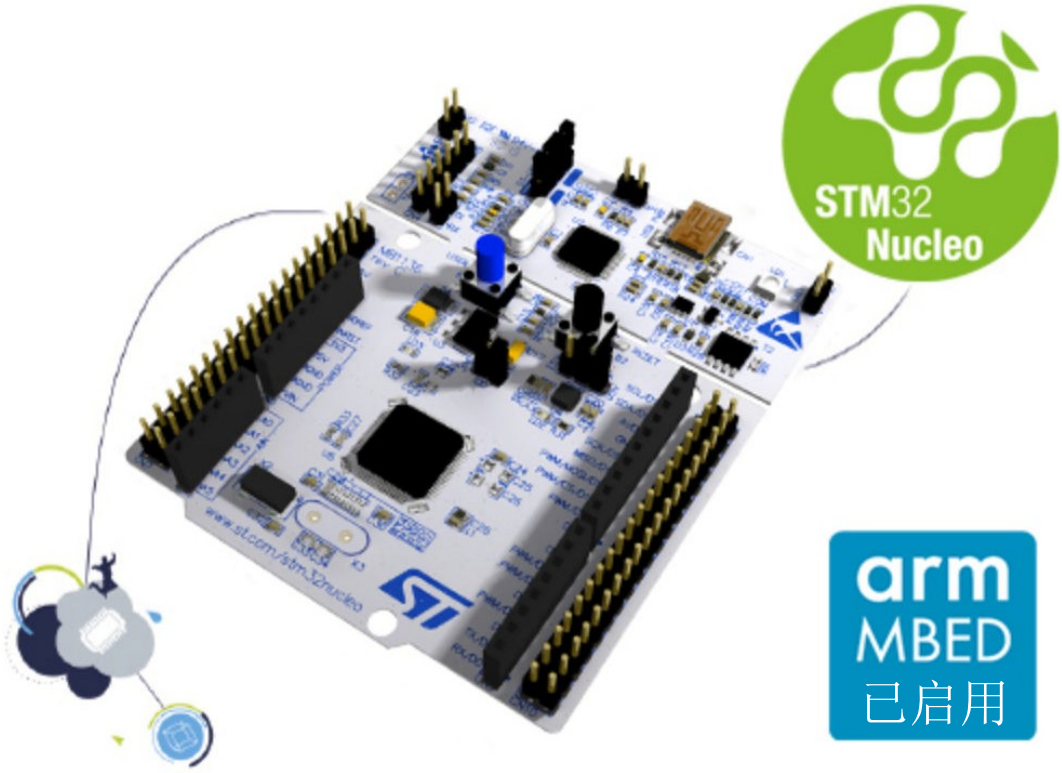
6 硬件和软件环境设置

6.1 硬件设置

STM32 Nucleo 板提供了高性价比解决方案，用户可以灵活使用任何 STM32 微控制器尝试新理念并构建原型。借助 ARDUINO® 连接支持和 ST Morpho 头，能够轻松使用各种专用扩展板来扩展 STM32 Nucleo 开放式开发平台的功能。STM32 Nucleo 板集成了 ST-LINK/V2-1 调试器/编程器。STM32 Nucleo 板附带 STM32 综合软件 HAL 库和各种套装软件示例。

关于 STM32 Nucleo 开发板的详细信息，请访问 http://www.st.com/stm32nucleo 网页。

图 10. **STM32 Nucleo 开发板示例**



需要下列组件：

- NUCLEO-G071RB rev B (MB1360 C) 开发板
- Type A USB 转 Micro-B USB 连接线，用于连接 STM32 Nucleo 和 PC。

6.2 软件设置

本节列出了设置 SDK、运行示例场景以及自定义应用的最低要求，供开发人员参考。

6.2.1 开发工具链和编译器

选择一个 STM32Cube 软件扩展包支持的 *IDE*。

读取所选 *IDE* 供应商提供的系统要求和设置信息。

检查发行包内的项目 Release_Notes.html 文件，并参考 *IDE* 兼容性章节（如果存在）。

6.2.2 *CRC* 工具设置

ST 提供了一个 *CRC* 工具，作为 STM32CubeProgrammer 内部的一个单独功能，用于 Flash 存储器测试。其他 *CRC* 工具也可以使用，前提是它们满足预期 CRC 预计算中的要求。

工具安装步骤：

1. 在 www.st.com 上的专用网页上选择 STM32CubeProgrammer
2. 安装软件包。

最简单的方式是将工具路径添加到环境变量中（需要计算机管理权限）。否则，必须在编译项目的后生成选项中直接添加路径。

7 STL：用户 API 和状态机

7.1 用户结构

结构在 stl_user_api.h 中定义。禁止更改此文件的内容。以下详细介绍的结构是 stl_user_api.h 内容的副本：

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

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

```
typedef enum
{
  STL_CPU_TM1L_IDX = 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 Flash or RAM memory subset */
  uint32_t EndAddr; /* end address of Flash or RAM memory subset */
  STL_CrcTableIdx_t CrcTableIdx; /* Index of the CRC table (used by Flash TM)*/
  struct STL_MemSubset_struct *pNext; /* pointer to the next Flash or RAM memory subset
                                         - to be set to NULL for the last subset */
} STL_MemSubset_t; /* Type used to define Flash
                      or RAM subsets to test */
```

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

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

注意： *请参见 www.st.com 上的 stl_user_api.h，了解最新的结构。*

7.2 用户 API

以下 API 在文件 *stl_user_api.h* 中定义。禁止更改此文件的内容。

小心：

对于用户应用程序定义并用作 *STL API* 参数的指针，用户应用程序必须设置有效的指针，保持指针可用性，并检查指针的完整性。STL *不复制指针内容，而是直接访问应用程序定义的存储器地址。*

整个 *STL* 执行期间均如此。例如，必须保留用于访问保持存储器测试配置的结构内容的指针。即使在调用与这些测试关联的 *API* 时，这些指针不是输入参数列表的一部分，STL_SCH_run_xxx 函数仍会使用这些指针。

有关正确的 *API* 序列调用的更多详细信息，请参见第 7.3 节状态机和第 7.6 节测试示例。

7.2.1 通用 API

以下部分详细介绍了通用 *API*。

7.2.1.1 *STL_SCH_Init*

描述：初始化调度程序。它可以随时用于重新初始化调度程序（它会重置所有测试）。

声明：`STL_Status_t STL_SCH_Init(void)`。

表 6. **STL_SCH_Init 输入信息**

允许的状态	参数
CPU TMx：全部 Flash *TM*：全部 RAM *TM*：全部	-

表 7. **STL_SCH_Init 输出信息**

`STL_Status_t` 返回值		返回状态
值	备注	
STL_OK	功能执行成功	CPU TMx：CPU_TMx_CONFIGURED Flash *TM*：FLASH_IDLE RAM *TM*：RAM_IDLE
STL_KO	防御性编程错误源： • *STL* 内部数据损坏	无状态更改

附加信息：对于 *CPU* 测试模块，没有特定的 *CPU* 初始化功能。

注意： *该功能使用硬件 CRC，如第 4.3.3 节 CRC 资源所述。*

7.2.2 CPU Arm® 内核测试 API

7.2.2.1 *STL_SCH_RunCpuTMx*

描述：运行其中一个 *CPU* 测试模块。

声明：`STL_Status_t STL_SCH_RunCpuTMx(STL_TmStatus_t *pSingleTmStatus)`，其中 `TMx` 可以是 TM1L、TM7 或 TMCB。

表 8. **STL_SCH_RunCpuTMx 输入信息**

允许的状态	参数	
	值	备注
CPU_TMx_CONFIGURED	`*pSingleTmStatus`	参见小心

表 9. STL_SCH_RunCpuTMx 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_PASSED	-	CPU_TMx_CONFIGURED
		STL_FAILED	-	
		STL_ERROR	防御性编程错误源： • *STL* 内部数据损坏 • *CPU* TM7 中软件未以特权级别执行 • *CPU* TM7 中软件未在线程模式下执行	
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.3 Flash 存储器测试 API

7.2.3.1 *STL_SCH_InitFlash*

描述：初始化 Flash 存储器测试。

声明：`STL_Status_t STL_SCH_InitFlash(STL_TmStatus_t *pSingleTmStatus)`

表 10. STL_SCH_InitFlash 输入信息

允许的状态	参数	
	值	备注
FLASH_IDLE FLASH_INIT FLASH_CONFIGURED	*pSingleTmStatus	小心：

表 11. STL_SCH_InitFlash 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	FLASH_INIT
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.3.2 *STL_SCH_ConfigureFlash*

描述：配置 Flash 存储器测试。

声明：`STL_Status_t STL_SCH_ConfigureFlash(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pFlashConfig)`

表 12. **STL_SCH_ConfigureFlash 输入信息**

<table>
<tr><th rowspan="2">允许的状态</th><th colspan="3">参数</th></tr>
<tr><th>值</th><th colspan="2">备注</th></tr>
<tr><td rowspan="11">FLASH_INIT</td><td>*pSingleTmStatus</td><td colspan="2">参见小心</td></tr>
<tr><td rowspan="10">*pFlashConfig</td><td colspan="2">指向 Flash 存储器配置的指针。参见小心。</td></tr>
<tr><td>字段</td><td>备注</td></tr>
<tr><td rowspan="4">*pSubset</td><td>• 指向 Flash 存储器子集的指针。参见小心
• 区段不能与 CRC 区域重叠</td></tr>
<tr><td>字段</td><td>备注</td></tr>
<tr><td>StartAddr</td><td>• 以字节为单位的起始子集地址
• 不能低于 ROM_START 并高于 CRC_START 地址</td></tr>
<tr><td>EndAddr</td><td>• 以字节为单位的结束子集地址
• 不能低于 ROM_START 并高于 CRC_START 地址
• 需要高于 StartAddr</td></tr>
<tr><td>*pNext</td><td>• 指向下一个 Flash 存储器子集的指针。参见 小心
• 对于最后一个子集，必须设置为 NULL</td></tr>
<tr><td>NumSectionsAtomic</td><td colspan="2">• 在原子测试期间要测试的 Flash 存储器区段数
• 设置为 1，作为最小值（每次测试一个区段）
• 如果该值高于所有子集中的区段数，则所有 Flash 存储器子集都将一次性测试</td></tr>
</table>

表 13. **STL_SCH_ConfigureFlash 输出信息**

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	FLASH_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 检测到错误配置 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • pFlashConfig = NULL • STL 内部数据损坏	不相关	不得使用的值	无状态更改

附加信息：如果返回值设置为 STL_KO 或 *pSingleTmStatus 设置为 STL_ERROR，则不应用 Flash 存储器配置。

7.2.3.3 *STL_SCH_RunFlashTM*

描述：运行 Flash 存储器测试。

声明：`STL_Status_t STL_SCH_RunFlashTM(STL_TmStatus_t *pSingleTmStatus)`

表 14. **STL_SCH_RunFlashTM 输入信息**

允许的状态	参数	
	值	备注
FLASH_CONFIGURED	`*pSingleTmStatus`	参见小心

表 15. **STL_SCH_RunFlashTM 输出信息**

STL_Status_t 返回值		`*pSingleTmStatus` 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_PASSED	-	FLASH_CONFIGURED
		STL_PARTIAL_PASSED	-	FLASH_CONFIGURED
		STL_FAILED	-	FLASH_CONFIGURED
		STL_NOT_TESTED	所有子集已经进行测试	FLASH_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 配置损坏 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • `pSingleTmStatus = NULL` • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.3.4 *STL_SCH_ResetFlash*

描述：重置 Flash 存储器测试。

声明：`STL_Status_t STL_SCH_ResetFlash(STL_TmStatus_t *pSingleTmStatus)`

表 16. **STL_SCH_ResetFlash 输入信息**

允许的状态	参数	
	值	备注
FLASH_CONFIGURED	`*pSingleTmStatus`	参见小心

表 17. STL_SCH_ResetFlash 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	配置成功应用	FLASH_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 配置损坏 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

附加信息

- 所有子集完成测试后，用户必须重置测试模块才能再次进行 Flash 存储器测试。
- 如果返回值设置为 STL_KO 或 *pSingleTmStatus 设置为 STL_ERROR，不会重置Flash存储器。

7.2.3.5 *STL_SCH_DeInitFlash*

描述：解除 Flash 存储器测试的初始化 - 仅用于单次测试。

声明：`STL_Status_t STL_SCH_DeInitFlash(STL_TmStatus_t *pSingleTmStatus)`

表 18. STL_SCH_DeInitFlash 输入信息

允许的状态	参数	
	值	备注
FLASH_IDLE FLASH_INIT FLASH_CONFIGURED	*pSingleTmStatus	参见小心

表 19. STL_SCH_DeInitFlash 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	FLASH_IDLE
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • STL 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.4 RAM 测试 API

7.2.4.1 *STL_SCH_InitRam*

描述：初始化 RAM 测试。

声明：`STL_Status_t STL_SCH_InitRam(STL_TmStatus_t *pSingleTmStatus)`。

表 20. STL_SCH_InitRam 输入信息

允许的状态	参数	
	值	备注
RAM_IDLE RAM_INIT RAM_CONFIGURED	*pSingleTmStatus	参见小心

表 21. STL_SCH_InitRam 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	RAM_INIT
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.4.2 *STL_Status_t STL_SCH_ConfigureRam*

说明：描述：配置 RAM 测试。

声明：`STL_Status_t STL_SCH_ConfigureRam(STL_TmStatus_t *pSingleTmStatus, STL_MemConfig_t *pRamConfig)`

表 22. STL_SCH_ConfigureRam 输入信息

<table>
<tr><th rowspan="2">允许的状态</th><th colspan="4">参数</th></tr>
<tr><th>值</th><th colspan="3">备注</th></tr>
<tr><td rowspan="9">RAM_INIT</td><td>*pSingleTmStatus</td><td colspan="3">参见小心</td></tr>
<tr><td rowspan="8">*pRamConfig</td><td colspan="3">此指针包含 RAM 配置。参见小心</td></tr>
<tr><td>字段</td><td colspan="2">备注</td></tr>
<tr><td rowspan="5">*pSubset</td><td colspan="2">• 指向 RAM 子集的指针。参见小心
• 子集不能与 RAM 备份缓冲区重叠</td></tr>
<tr><td>字段</td><td>备注</td></tr>
<tr><td>StartAddr</td><td>• 以字节为单位的起始子集地址
• 起始地址必须是 32 位对齐的
• RAM 子集必须位于 RAM 区域内
• 不能低于 RAM_START 地址且高于 RAM_END 地址</td></tr>
<tr><td>EndAddr</td><td>• 以字节为单位的结束子集地址
• 高于 StartAddr
• 不能低于 RAM_START 地址且高于 RAM_END 地址
• 子集大小 (EndAddr - StartAddr) 需要是 2 * RAM_BLOCK_SIZE，32 字节的倍数
• 子集不能与 RAM 备份缓冲区重叠</td></tr>
<tr><td>*pNext</td><td>• 指向下一个 RAM 子集的指针。参见小心
• 对于最后一个子集，必须设置为 NULL</td></tr>
<tr><td>NumSectionsAtomic</td><td colspan="2">• 在原子测试期间要测试的 RAM 区段数量
• 设置为 1，作为最小值（每次测试一个区段）
• 如果该值高于所有子集中的区段数，则所有 RAM 子集都将一次性测试</td></tr>
</table>

表 23. **STL_SCH_ConfigureRam 输出信息**

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	RAM_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 检测到错误配置 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • pRamConfig = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

附加信息：如果返回值设置为 STL_KO 或 *pSingleTmStatus 设置为 STL_ERROR，不会应用 RAM 配置。

7.2.4.3 *STL_SCH_RunRamTM*

描述：运行RAM 测试。

声明：STL_Status_t STL_SCH_RunRamTM(STL_TmStatus_t *pSingleTmStatus)

表 24. **STL_SCH_RunRamTM 输入信息**

允许的状态	参数	
	值	备注
RAM_CONFIGURED	*pSingleTmStatus	参见小心

表 25. STL_SCH_RunRamTM 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_PASSED	-	RAM_CONFIGURED
		STL_PARTIAL_PASSED	-	RAM_CONFIGURED
		STL_FAILED	-	RAM_CONFIGURED
		STL_NOT_TESTED	所有子集已经进行测试	RAM_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 配置损坏 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.4.4 *STL_Status_t STL_SCH_ResetRam*

描述：重置 RAM 测试。

声明：`STL_Status_t STL_SCH_ResetRam(STL_TmStatus_t *pSingleTmStatus)`

表 26. STL_SCH_ResetRam 输入信息

允许的状态	参数	
	值	备注
RAM_CONFIGURED	*pSingleTmStatus	参见小心

表 27. STL_SCH_ResetRam 输出信息

STL_Status_t 返回值		*pSingleTmStatus 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	配置成功应用	RAM_CONFIGURED
		STL_ERROR	防御性编程错误的可能来源： • 状态不允许 • 配置损坏 • *STL* 内部数据损坏	无状态更改
STL_KO	防御性编程错误的可能来源： • pSingleTmStatus = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

附加信息

- 所有子集完成测试后，用户必须重置测试模块才能再次进行 RAM 测试。
- 如果返回值设置为 STL_KO 或 *pSingleTmStatus 设置为 STL_ERROR，不会应用 RAM 重置。

7.2.4.5 *STL_SCH_DeInitRam*

描述：解除 RAM 测试初始化。

声明：`STL_Status_t STL_SCH_DeInitRam(STL_TmStatus_t *pSingleTmStatus)`

表 28. **STL_SCH_DeInitRam 输入信息**

允许的状态	参数	
	值	备注
RAM_IDLE RAM_INIT RAM_CONFIGURED	`*pSingleTmStatus`	参见小心

表 29. **STL_SCH_DeInitRam 输出信息**

`STL_Status_t` 返回值		`*pSingleTmStatus` 输出		返回状态
值	备注	值	备注	
STL_OK	功能执行成功	STL_NOT_TESTED	-	RAM_IDLE
STL_KO	防御性编程错误的可能来源： • `pSingleTmStatus` = NULL • *STL* 内部数据损坏	不相关	不得使用的值	无状态更改

7.2.5 人为失效 API

7.2.5.1 *STL_SCH_StartArtifFailing*

描述：设置人为失效配置并启动人为失效功能。

声明：`STL_Status_t STL_SCH_StartArtifFailing(const STL_ArtifFailingConfig_t *pArtifFailingConfig)`

表 30. **STL_SCH_StartArtifFailing 输入信息**

允许的状态	参数	
	值	备注
CPU TMx： • CPU_TMx_CONFIGURED Flash *TM*： • FLASH_IDLE • FLASH_INIT • FLASH_CONFIGURED RAM *TM* • RAM_IDLE • RAM_INIT • RAM_CONFIGURED	`*pArtifFailingConfig`	无状态更改

表 31. **STL_SCH_StartArtifFailing 输出信息**

`STL_Status_t` 返回值	备注	输出	备注
STL_OK	功能执行成功	无输出参数	无状态更改
STL_KO	防御性编程错误的可能来源： • `pArtifFailingConfig` = NULL • 个别测试模块未设置配置值 • *STL* 内部数据损坏		

附加信息：以下所有 *API* 调用都会正常执行，除非 *STL_Status_t* 返回值设置为 STL_OK，测试模块状态 (`*pSingleTmStatus`, `*pTmListStatus`) 被强制设置为配置值。

7.2.5.2 *STL_SCH_StopArtifFailing*

描述：停止人为失效功能。

声明：`STL_Status_t STL_SCH_StopArtifFailing(void)`

表 32. STL_SCH_StopArtifFailing 输入信息

允许的状态	参数	
	值	备注
CPU TMx： • CPU_TMx_CONFIGURED Flash *TM*： • FLASH_IDLE • FLASH_INIT • FLASH_CONFIGURED RAM *TM* • RAM_IDLE • RAM_INIT • RAM_CONFIGURED	无输入参数	无状态更改

表 33. STL_SCH_StopArtifFailing 输出信息

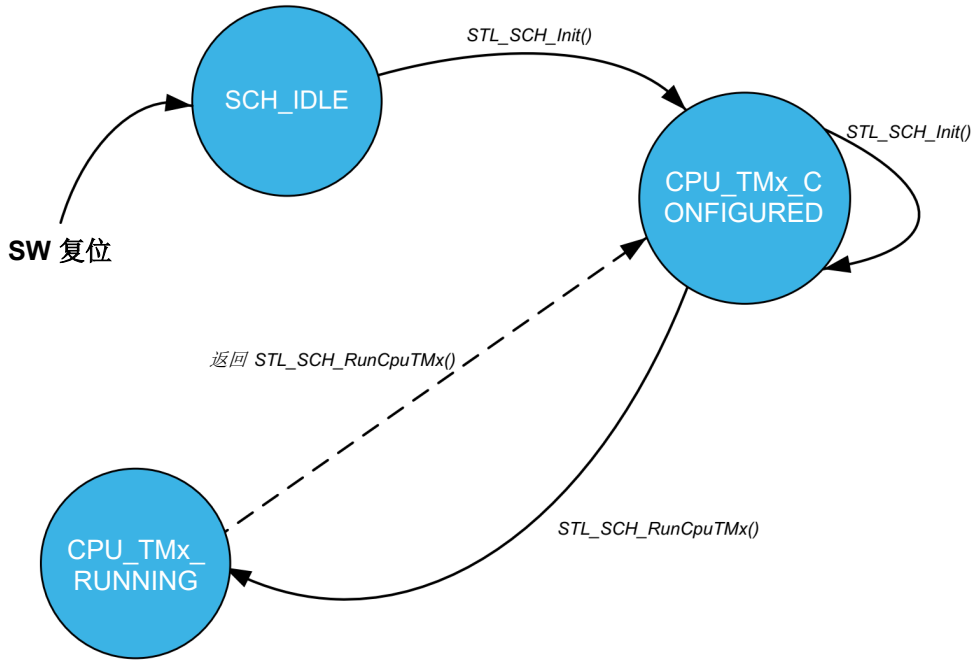
STL_Status_t 返回值	备注	输出	备注
STL_OK	功能执行成功	无输出参数	无状态更改
STL_KO	防御性编程错误的可能来源： • *STL* 内部数据损坏		

7.3 状态机

每个 *CPU* 测试模块都有其专属的与 *CPU* 测试 *API* 相关的状态机图。

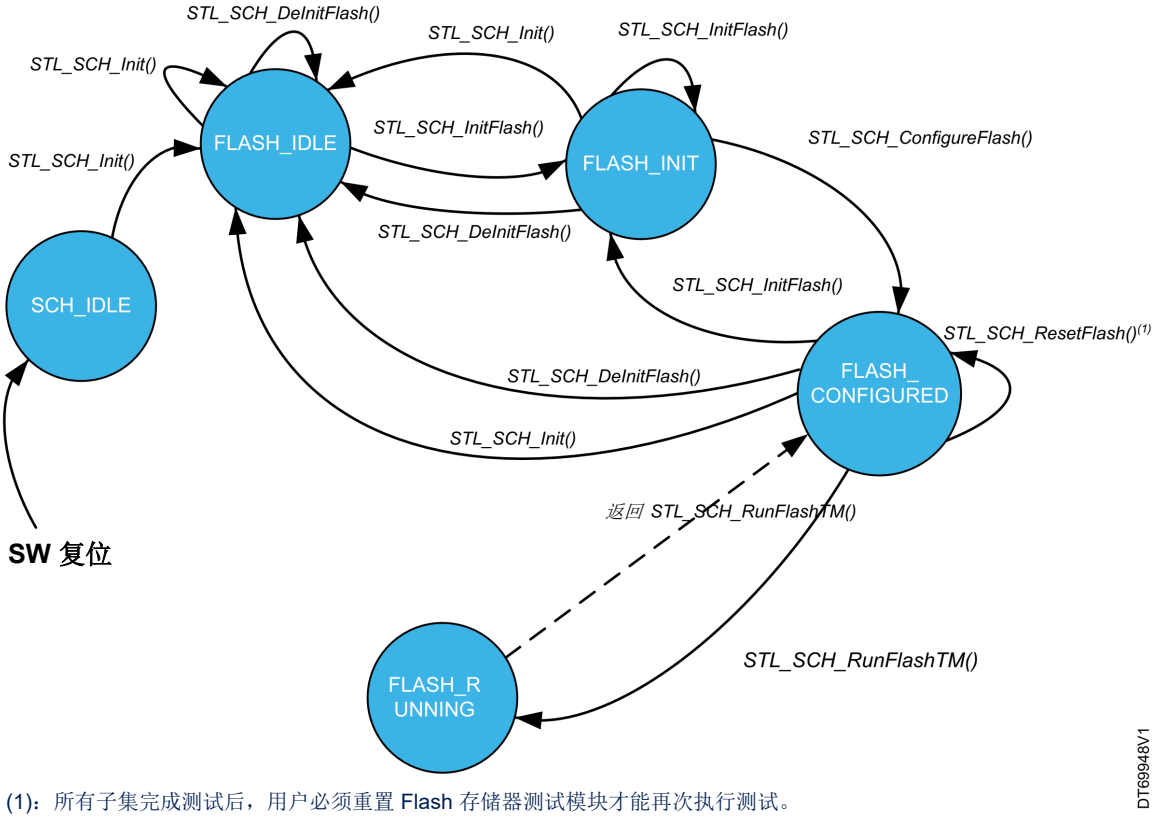
CPU* 测试 *API

图 11. 状态机图 - *CPU* 测试 *API*



Flash 存储器测试 *API*

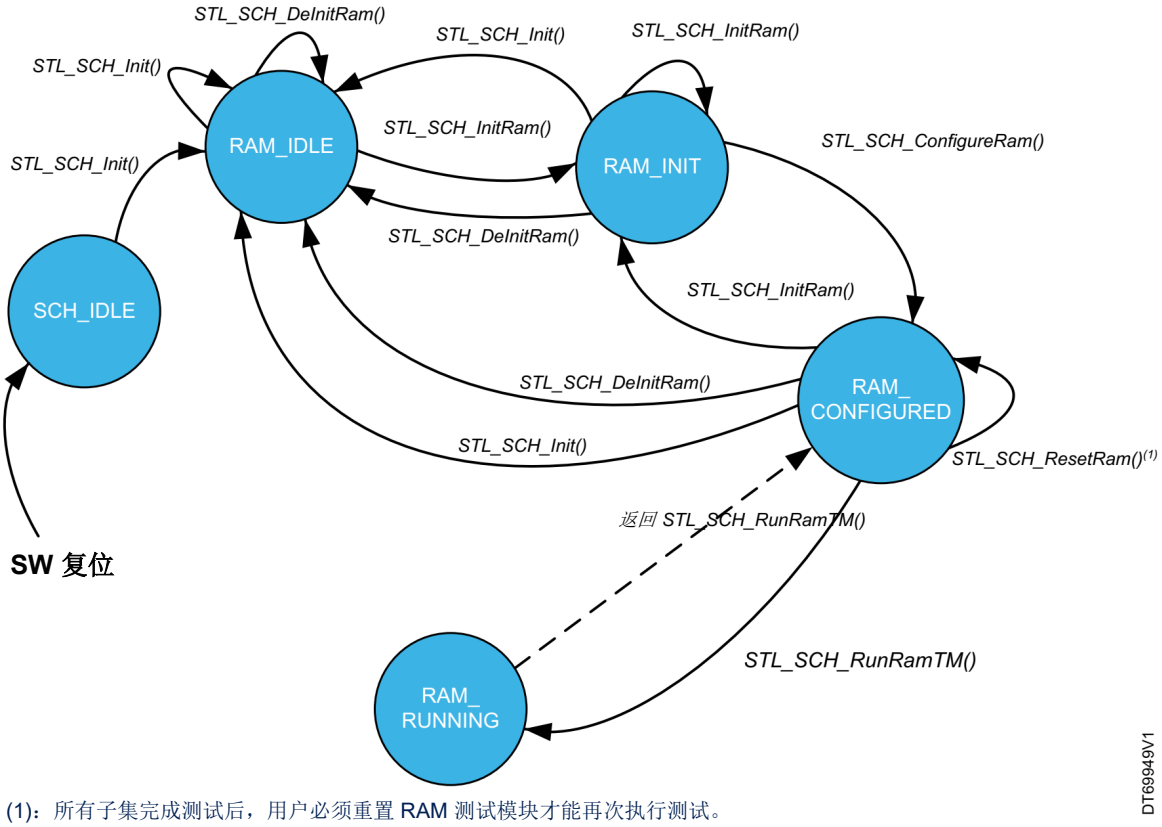
图 12. 状态机图 - Flash 存储器测试 *API*



注 (1)：所有子集完成测试后，用户必须重置 Flash 存储器测试模块才能再次执行测试。

RAM 测试 *API*

图 13. 状态机图 - RAM 测试 *API*



注 (1)：所有子集完成测试后，用户必须重置 RAM 测试模块才能再次执行测试。

7.4 API 使用流程

用户应用程序必须：

- 在测试期间保持作为参数传递的指针的可用性和完整性。*STL* 不复制指针内容，直接访问应用程序定义的存储器地址。
- 在检查测试结果（`STL_TmStatus_t` 或 `STL_TmListStatus_t`）之前，检查函数返回值的状态 (`STL_Status_t`)。参见交付应用程序中的示例。

各个 *API* 彼此独立，因此可以按任何顺序调用。

只有专用于配置和初始化存储器测试的 *API*，必须在执行这些测试之前调用。参见第 7.3 节状态机获取更多详细信息。

测试流程已简化，所有测试现在均从 C 代码执行，启动和运行时测试之间不再有任何区别。在此固件的早期版本中需要进行区分。由于应用程序重置后缺乏具体的初步测试，常规做法是在应用程序启动前执行完整的初始序列，包括在所有存储器区域上执行的完整测试集。该序列按以下顺序定义：

1. 所有 CPU 测试
2. 非易失性存储器的完整测试
3. 包括专门用于栈的区域在内的易失性存储器的完整测试。

注意： *为加快对大型 RAM 区域的初步测试速度，可以暂时禁用存储器内容的备份，在此测试期间用户无需保留存储器内容。有关更多详细信息，请参见第 4.3.7 节 RAM 备份缓冲区。*

4. 特定用户自定义测试

运行期测试，可以更改测试的顺序并以更灵活的方式执行。可减少正在测试的存储器区域。测试过程可以动态修改，优先关注哪些仅存储与安全相关的代码和数据的区域。在特别注意仅存储安全相关代码和数据的区域后尤其需要考虑以下因素：

- 可用的应用程序过程安全时间
- 系统整体性能
- 应用程序的具体状态
- 等等。

7.5 用户参数

除了在 *API* 内部直接设置的参数外，还有一些参数需要在 `stl_user_param_template.c` 文件中进行自定义。它们位于代码中，有以下备注：

```
/* customisable */
```

摘自 `stl_user_param_template.c`：

```
/* Flash configuration */
#define STL_ROM_START (0x08000000U) /* customizable */
#define STL_ROM_END (0x0801FFFFU) /* customizable */
```

自定义取决于 STM32 产品和用户选择。

```
/* 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 */
```

自定义取决于用户选择。

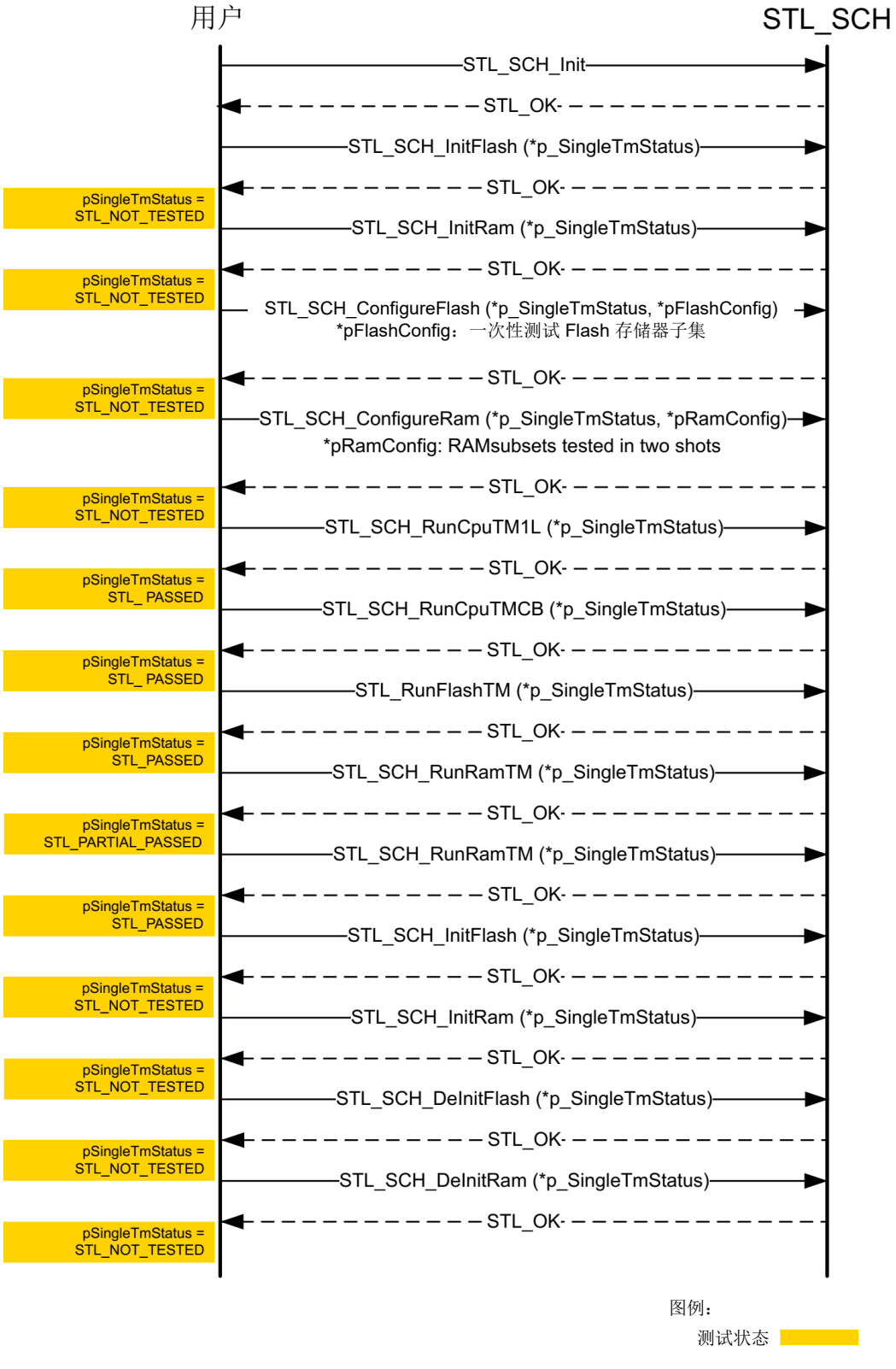
剩余的用户参数由标志定义，可以在以下文件中检查：

- `stl_user_param_template.c`：是否使用 RAM 备份缓冲区
- `stl_util.c`：使用软件或硬件 *CRC* 计算
- `stl_stm32_hw_config.h`：如果使用 *CRC* 硬件，请根据 STM32 器件选择适用的 *CRC* IP 配置

请参见第 5.5.2 节从头开始构建应用程序的步骤进行标志配置检查。

7.6 测试示例

图 14. 单一测试示例



7.7 测试示例的详细信息

7.7.1 Flash 存储器单一测试示例

图 15 显示了 Flash 测试流程处理的一个示例：

- 使用两个 Flash 存储器子集
- 使用函数
  - STL_SCH_RunFlashTM → 仅执行 Flash 存储器测试模块
  - STL_SCH_ResetFlash
- 函数返回值
- Flash 存储器测试模块结果值：`pSingleTmStatus`→在这种情况下，它包含了 Flash 存储器测试的结果

7.7.2 RAM 单一测试示例

图 16 显示了 RAM 测试流程处理的一个示例：

- 使用两个 RAM 子集
- 使用函数：
  - STL_SCH_RunRAMTM → 仅执行 RAM 测试模块
  - STL_SCH_ResetRam
- 函数返回值
- RAM 测试模块结果值：`pSingleTmStatus`→在这种情况下，它包含了 RAM 存储器测试的结果

7.7.3 测试示例图

图 15. Flash 存储器单一测试示例

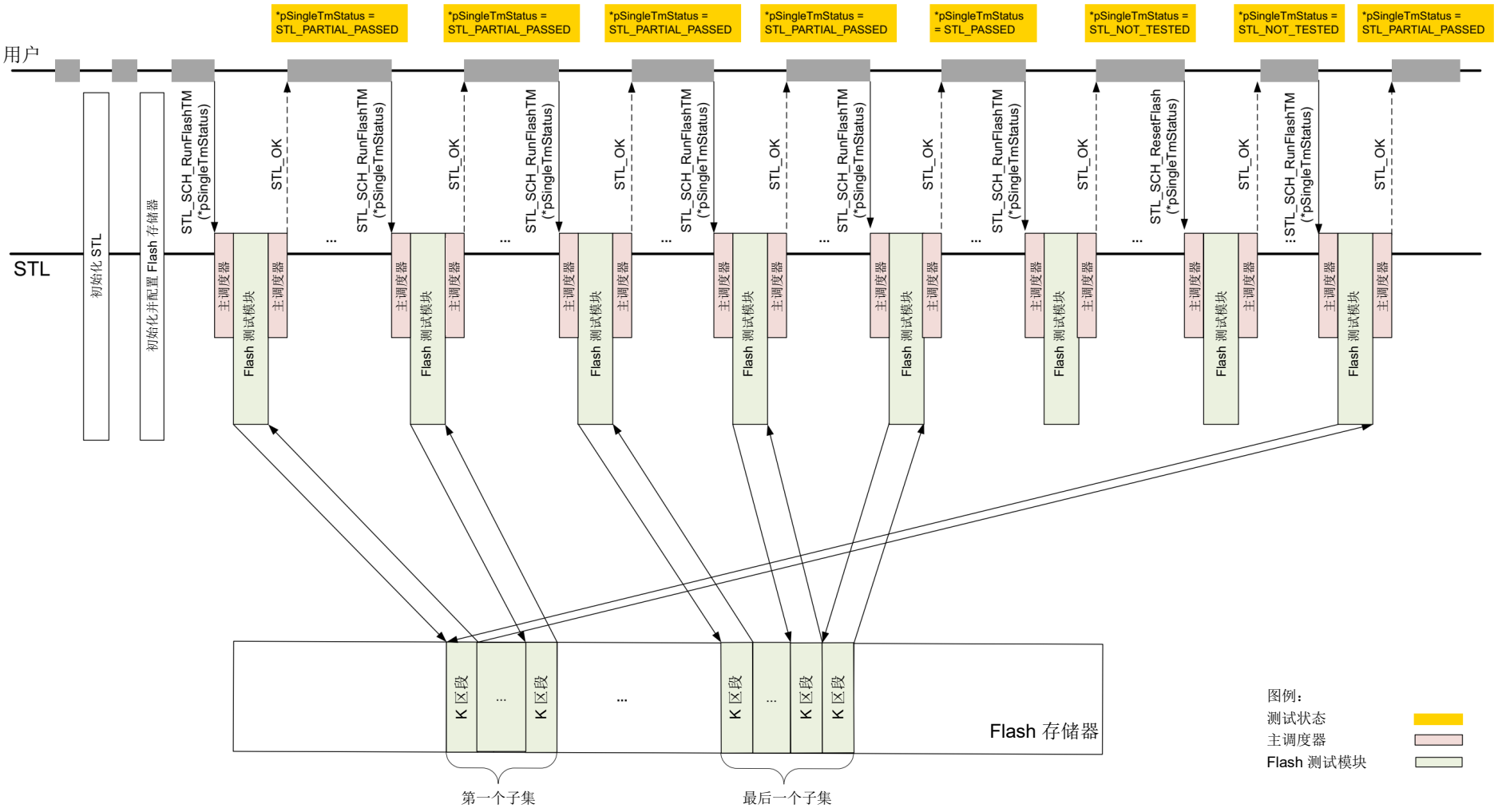
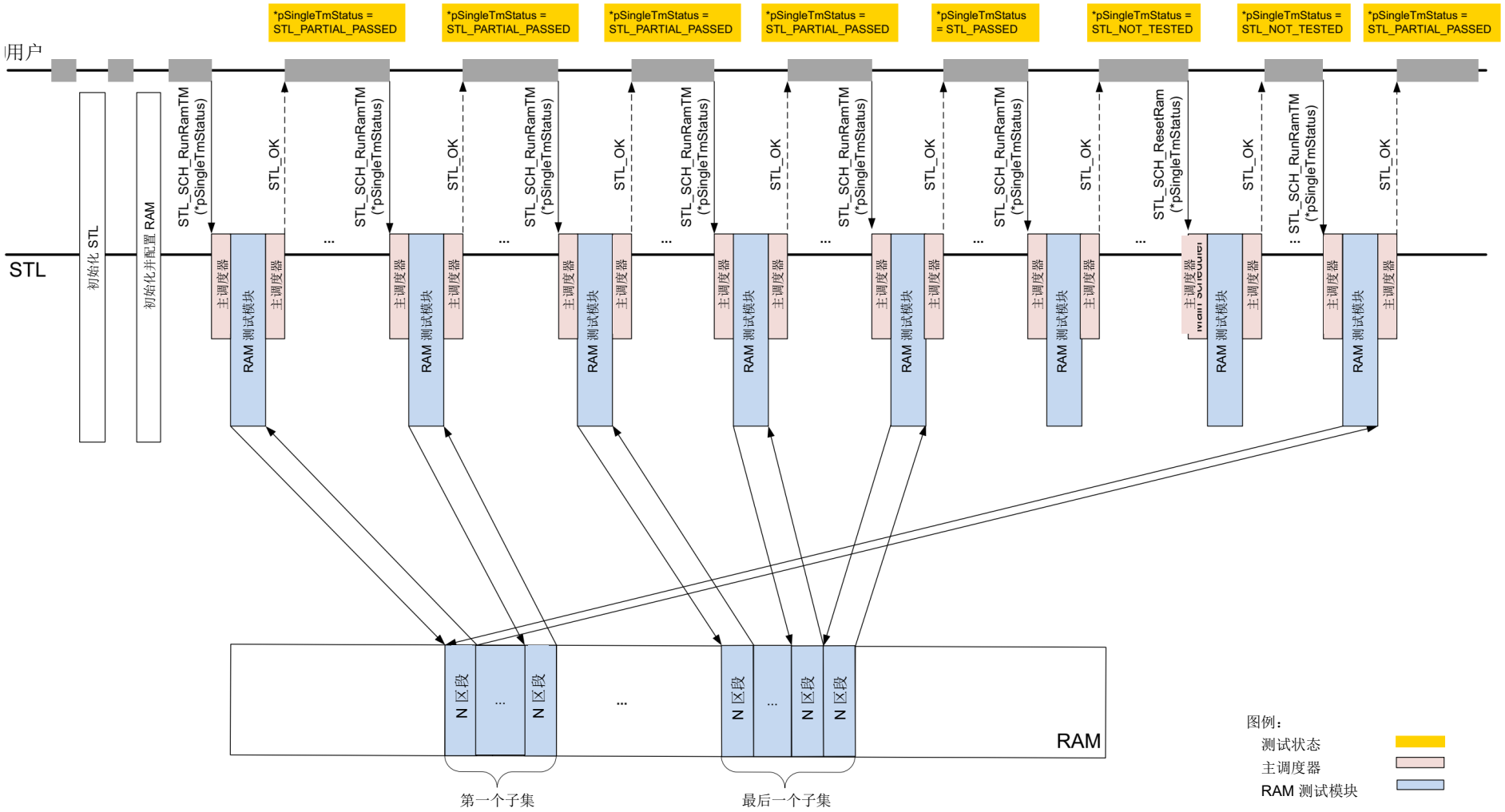


图 16. RAM 单一测试示例



8 STL：执行时间详细信息

表格 34 中的数据是通过以下测试设置获得的：

- *STL* 库编译细节，参见第 5.5 节应用：编译过程。
- 使用 IAR Embedded Workbench® for Arm® (EWARM) toolchain v9.20.1 编译的性能测试项目
- 编译的软件配置包括：
  - 将 HCLK 时钟设置为 56 MHz
  - 将 Flash 存储器延迟设置为四个等待周期
  - *ICache* 已激活
  - NUCLEO-G071RB rev B (MB1360 C)
  - *STL*，以安全二进制方式运行，被非安全二进制方式运行的用户应用程序调用

表 34. 集成测试

测试	持续时间 (µs)		已测试的存储器
	硬件 CRC	软件 CRC	
STL_SCH_InitFlash()	9	9	-
STL_SCH_ConfigureFlash()	13	13	-
STL_SCH_RunFlashTM()	860	5289(1)	测试了 17408 字节
STL_SCH_InitRam()	9	9	-
STL_SCH_ConfigureRam()	13	13	-
STL_SCH_RunRamTM()(2)	34032	33870	测试了 36832 字节
STL_SCH_RunCpuTM1L()	87	87	-
STL_SCH_RunCpuTM7()(2)	34	34	-
STL_SCH_RunCpuTMCB()	12	12	-

1. *测试持续时间因 CRC 算法（参见第 4.1.1 节调度器原则）而增加。*
2. *默认配置：STL_ENABLE_IT 和 STL_DISABLE_RAM_BCKUP_BUF 未启用。*

9 应用特定测试不在 ST 固件自检库范围内

用户必须侧重于 ST 固件库中未包括的涉及特定应用 *MCU* 部分的所有剩余测试：

- 模拟部件测试（ADC/DAC，复用器）
- 数字 I/O 测试
- 外部寻址
- 外部通信
- 定时和中断
- 系统时钟频率测量。

注意： *时钟频率测量不是 STL 软件包的集成部分。时钟测试模块以开源形式在 STL 集成示例中提供，以演示实施附加用户定义测试模块的能力，这些模块可以包含在 STL 流程中。有关更多详细信息，请参见第 9.5 节 STL 库扩展能力。*

这些组件的有效解决方案在很大程度上取决于应用及外设的能力。应用程序必须尽可能准确地遵循从设计初期提出的测试原则。

该方法通常会导致硬件和软件层级的冗余。

硬件方法基于：

- 输入和/或输出的倍增
- 参考点测量
- 模拟或数字输出如 *DAC*、*PWM*、*GPIO* 的回送读取控制
- 配置保护。

软件方法基于：

- 时间上的重复，多次获取，多次检查，不同时间或采用不同方法做出的决策或计算
- 数据冗余（数据复制、奇偶校验、错误修正/检测代码、校验和、协议）
- 真实性检查（有效范围、有效组合、预期变化或趋势）
- 周期性和发生率检查（流程和发生率控制）
- 正确配置的定期检查（如读回配置寄存器）。

9.1 模拟信号

必须检查测量值的一致性，并通过在其他冗余通道上进行的测量进行验证。空闲通道可用于读取参考电压并测试应用程序中使用的模拟复用器。此外，还需要检查内部参考电压。

一些 STM32 微控制器设备具备两个（甚至三个）独立的 *ADC* 块。为确保结果的可靠性，出于安全原因，在同一通道使用两个不同的 *ADC* 块进行多次转换。结果可以通过以下任一方式获取：

- 从一个通道进行多次获取
- 比较冗余通道，随后进行计算平均值操作。

以下是在 STM32 微控制器设备上测试模拟部件功能的一些提示。

ADC 输入引脚断开连接

通过在测试引脚上应用附加的信号源可以测试 ADC 输入引脚断开情况。

- 一些 STM32 微控制器设备在模拟输入上具有内部下拉或上拉电阻激活功能。它们还提供具有 *DAC* 功能的空闲引脚或数字 *GPIO* 输出。其中的任何一个引脚都可以作为 ADC 的已知参考输入。
- 一些 STM32 微控制器设备具有路由接口。该接口可用于引脚之间的内部连接，用于：
  - 测试环回
  - 附加的信号注入
  - 在其他独立通道进行重复测量。

注意： *用户必须阻止任何关键电压注入到模拟引脚。在数字和模拟信号结合时，以及不同的电源等级被应用到模拟和数字部分时 ($V_{DD} > V_{DDA}$)，可能发生这种情况。*

内部参考电压和温度传感器（对于某些设备为 V_{BAT}）

- 这些信号之间的比率可以在允许的范围内进行验证。
- 在 V_{DD} 电压已知的情况下，可以进行附加测试。

ADC 时钟

通过定时器测量的 *ADC* 转换时间可以用来测试独立 *ADC* 时钟功能。

DAC 输出功能

空闲的 *ADC* 通道可以用来检查 *DAC* 输出通道是否正常工作。

连接 *ADC* 输入通道和 *DAC* 输出通道时，可以使用路由接口。

比较器功能

已知电压与 *DAC* 输出或内部参考电压之间的比较结果可用于测试另一个比较器输入上的比较器输出。

通过在测试引脚上激活下拉或上拉功能并将此信号与 *DAC* 电压（作为另一个比较器输入的参考）进行比较，可以测试模拟信号的断开情况。

运算放大器

可以通过强制（或测量）运算放大器 (*OPAMP*) 输入引脚的已知模拟信号，并使用 *ADC* 内部测量输出电压来测试功能。*OPAMP* 的输入信号也可以通过 *ADC*（在另一个通道上）测量。

9.2 数字 I/O

B 类测试同样必须检测数字 I/O 上的所有故障。此检测可以通过真实性检查来实现，并且这种检查可以与其他应用部分结合起来进行。例如，当加热/冷却数字控制开启/关闭时，必须检查来自温度传感器的模拟信号变化。可通过将正确的锁定序列应用于 GPIOx_LCKR 寄存器内的锁定位来锁定选择的端口位。此操作避免了端口配置的意外更改。在这种情况下，仅可在下一个复位序列进行重新配置。此外，位带特征可用于 SRAM 和外围寄存器的原子操纵。

9.3 中断

必须检查事件的发生率和周期性。可采用多种方法；一种方法是，采用一组增量计数器，发生中断事件时特定定时器递增。通过其他独立的时基定期对计数器内的值进行交叉检查。上个周期内发生的事件数量取决于应用要求。

配置锁功能可用于通过由 TIMx_BDTR 寄存器控制的三个层级来保护定时器寄存器设置。未使用的中断向量必须转移至通用错误处理器。如果能简化应用中断方案，对于非安全相关问题最好进行轮询。

9.4 通信

在通信会话期间进行数据交换时，必须通过数据包中包含的冗余信息检查数据的完整性。为此，可使用奇偶校验、同步信号、CRC 校验和、块重复或协议编号。如有必要，稳定的应用软件协议栈（如 TCP/IP）可提供更高级的保护。必须对通信事件的周期性和发生率及协议错误信号进行永久性检查。

用户可在产品专属安全手册中找到更多信息和方法。

9.5 STL 库的扩展能力

与之前版本的 *STL* 库相比，这个框架版本的实施明显更加简便灵活（参见第 1.2 节参考文档），从而更易于扩展。即使采用了新的应用格式，该框架仍保持符合 IEC 60730 标准的同一套自检方法，这些方法已由库的先前版本实现：

- 在 *CPU TM* 上测试寄存器
- Flash *TM* 中，与 STM32 硬件 *CRC* 单元兼容的 32 位 *CRC* 计算
- March C 测试，遵守 RAM *TM* 的物理地址顺序
- 由 LSI 触发的定时器，用于检查在 *STL* 集成示例中定义的时钟 *TM* 的系统时钟频率

新框架版本的主要改进如下：

- 面向模块化
- 支持部分测试
- 基于配置和参数化结构
- 启动测试和运行时测试之间无区别
- 基于 STM32CubeProgrammer 命令行功能提供的格式的 *CRC* 计算支持
- 关键通用模块的预编译和固定对象代码格式
- 通用模块执行不依赖于驱动程序或编译器
- 人为失效控制功能，以验证模块的正确集成，无需额外的仪器代码
- 通过附加的特定应用模块轻松扩展。

固件包集成示例中提供了一个附加特定测试模块实现的示例。一个基于两个独立时钟源交叉检测测量方法的特定测试模块，以开源格式与固件包集成示例一起提供。该模块必须由终端用户调整，以兼顾应用时钟系统配置的特定依赖关系。

此模块使用的测量原则与库的先前版本相同。用于频率比较的硬件必须首先进行配置（由 LSI 触发的 TIM16 的通道 1），以在调用关联的 *API* 之前进行时钟测量。此硬件配置在 `main.c` 文件中的 `STL_Init()` 过程末尾完成。*API* 编写时考虑了与集成在 *STL* 中的常规 *API* 的接口兼容性，因此在声明中应用了相同的格式：

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

在此函数调用期间传递的参数作为指向时钟模块测量状态的指针，如果防御性编程失败，该函数本身提供 STL_KO 与 STL_OK 的返回状态，常规 *STL* 模块也是如此。如果时钟测量硬件处于活动状态，并且在上一测量周期更新的新周期值（设置为连续 8 个 LSI 周期）处于预期区间内（由宏 `CLK_LimitLow` 和 `CLK_LimitHigh` 定义），则模块测量状态值更改为 STL_PASSED。否则，会根据常规 *API* 模块设置为 STL_FAILED。当人为引发模块失效时，情况也是这样。

用户可以以类似的方式集成以下模块。例如，默认情况下，新包不再包括检查栈指针的有效性或实施看门狗测试和服务。这些测试的源代码在该库的旧版本中提供，请参见第 1.2 节参考文档。请参见第 1.2 节参考文档 以获取关于普遍认可的安全方法的额外信息，这些安全方法不是家用标准特别要求的。它们可能有助于提高用户应用程序的鲁棒性。

10 符合 IEC、UL 和 CSA 标准

关键的 IEC 标准为 IEC 60730-1 和 IEC 60335-1，与第四版开始的 UL/CSA 60730-1 和 UL/CSA 60335-1 协调一致。先前的 UL/CSA 还参考了 UL1998 标准。

标准会定期更新。标准涵盖的范围非常广泛；涉及微控制器通用部分软件自检要求的章节非常明确。在大多数情况下，提供的更新完全不会影响标准的这些特定部分。因此，过时的认证对于新版本标准可能依然有效。

所需的相关详细条件定义位于：

- IEC 60335-1 标准的附录 Q 和 R
- IEC 60730-1 标准的附录 H。

IEC 60730-1:2010 H.2.22 定义了三个类别，它们是：

- A 类：与应用安全无关的控制功能。
- B 类：旨在防止受控设备出现不安全状态的控制功能。控制功能故障不会直接导致危险情况。
- C 类：旨在防止特定危险（例如爆炸或可直接导致装置中发生危险的故障）的控制功能。

对于应用安全保护功能的可编程电子组件，IEC 60335-1 标准要求结合软件措施来控制表 R.1 和 R.2 中指定的故障和误差条件：

- 表 R.1 概括了与 B 类要求相当的一般条件
- 表 R.2 概括了与 C 类要求相当的特定条件。

本用户手册主要涉及 B 类软件要求，该要求旨在防止电器其他地方发生故障时造成危害。在这种情况下，故障发生后会在电器上运行自检软件。在执行安全关键例程期间发生的意外软件故障，并不一定会导致危害，因为在此级别要求其他冗余软件程序或硬件保护功能。

在 C 类中，不需要这样的硬件保护，任何安全关键软件的故障都可能导致潜在危险。为了符合这个类别的要求，需要执行更为严格的测试，相对于通常对 STM32 之类的标准工业微控制器执行的测试而言。可接受的解决方案通常会在系统级别实施特定的硬件冗余，如双通道结构。

有关更严格测试方法的更多信息，请参见工业文档 [1]。

IEC 60730-1 定义了一组适用于设计 B 类控制功能的可接受架构：

- 单通道有功能测试。单一 CPU 按需要执行软件控制功能。在软件启动时执行功能测试。它保证所有关键功能正常工作。
- 单通道有定期自检。一个 CPU 执行软件控制功能。嵌入式定期自检检查系统的各种关键功能，而不影响计划中的控制任务的性能。
- 双通道（相同或不同）带比较功能。该软件旨在两个独立的 CPU 上执行控制功能（相同或不同）。在执行任何安全关键任务时，两个 CPU 比较内部信号以便检测故障。

注意： *这种结构也被认为符合 C 类标准。一个普遍原则是，任何符合 C 类的方法自动符合 B 类。*

下表列出了 *STL* 应用的方法及引用的标准概览。*STL* 侧重于在所有应用中重复使用的微控制器的通用组件。其它部分的测试归终端用户负责，因为它们的测试主要与特定应用相关，并且可以在系统设计规划阶段有效完成。请参见第 9 节未包含在 ST 固件自检库中的应用特定测试，了解如何处理这些应用特定测试的更多信息。

表 35. X-CUBE-CLASSB 库涵盖的 IEC 60335-1 组件（遵循 IEC-60730-1 认可的方法）

表 R.1 的组件（IEC 60335-1：附件 R）		B 类	引用 IEC 60730-1：附件 H）	故障/误差	X-CUBE-CLASSB 应用的安全方法	备注
1. CPU	1.1 CPU 寄存器	X	H.2.16.5 H.2.16.6 H.2.19.6	固定型故障	定期运行 *STL* TM1L、TM7 和 TMCB CPU 测试模块	CPU *寄存器的功能模式测试，状态寄存器和栈指针的功能测试*
	1.2 指令解码和执行	N/A				B 类不需要
	1.3 程序计数器	X	H.2.18.10.2 H.2.18.10.4	固定型故障	N/A 终端用户责任	逻辑和时隙程序序列监控，实施看门狗
	1.4 寻址	N/A				B 类不需要
	1.5 数据路径指令解码	N/A				B 类不需要
2. 中断处理和执行		X	H.2.18.10.4 H2.18.18	没有中断或中断过于频繁	结果握手应用于与时钟交叉检查测量模块相关的中断	终端用户负责应用中实现的其他中断
3. 时钟		X	H.2.18.10.1 H.2.18.10.4	频率错误	定期运行时钟交叉检查模块。作为固件集成示例中的用户特定测试模块，以开放源代码格式添加	在两个独立的时钟源（系统时钟和 LSI）之间完成时钟交叉检查测量
4. 存储器	4.1 不可变存储器	X	H.2.19.3.1 H.2.19.3.2 H.2.19.8.2	所有一位故障	定期执行 *STL* FlashTM 测试模块	终端用户负责启用 ECC
	4.2 可变存储器	X	H.2.19.6 H.2.19.8.2	DC 故障	定期执行 *STL* RamTM 测试模块	终端用户负责启用 ECC
	4.3 寻址（适用于可变和不可变存储器）	X	H.2.19.8.2	固定型故障	-	通过执行应用的存储器测试模块间接测试
5. 内部数据路径	5.1 数据	X	H.2.19.8.2	固定型故障	-	终端用户负责启用 ECC
	5.2 寻址	X	H.2.19.8.2	地址错误	-	
6. 外部通信		X	-	-	N/A 终端用户责任	-
7. I/O 外设		X	-	-	N/A 终端用户责任	-
8. 监视设备和比较器		N/A				B 类不需要
9. 定制芯片		X	-	-	N/A	-

版本历史

表 36. 文档版本历史

日期	版本	变更
2023 年 1 月 27 日	1	初始版本。

词汇表

ADC 模数转换器

AEABI Arm® 嵌入式应用程序二进制接口

API 应用编程接口

APSR CPU 状态寄存器

BSP 板级支持包

B 类

面向家用电器安全的中级法规 (UL/CSA/IEC 60730-1/60335-1)

CMSIS 通用微控制器软件接口标准

CPU 中央处理器

CRC 循环冗余校验

DAC 数模转换器

DCache 数据缓存

FPU 浮点单元

GPIO 通用输入/输出

HAL 硬件抽象层

ICache 指令缓存

IDE 集成开发环境

LL 底层

MCU 微控制器单元

MPU 存储器保护单元

MSP 主栈指针

OPAMP 运算放大器

PSP 进程栈指针

PWM 脉冲宽度调制

RAM 随机存取存储器

SDK 软件开发套件

SG 安全网关

STL 自检库

TM 测试模块

目录

表格索引

图片索引

重要通知——请仔细阅读

意法半导体公司及其子公司（“意法半导体”）保留随时对意法半导体产品和/或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。

本文档仅用于获取与意法半导体产品相关的常规信息。因此，您在此同意使用本文档的唯一目的为获取与意法半导体产品相关的常规信息。您进一步知悉并同意，不得在任何法院、仲裁、机构、委员会或其他法庭的任何法律或行政诉讼中，或与任何法律行为、诉因、诉讼、索赔、指控、传唤或任何类型的争议有关的情况下使用本文档。您进一步知悉并同意，本文档不构成任何类型的承认、认可或证据，包括但不限于意法半导体或其任何关联公司的任何责任、过错或职责，或者关于本文档包含的信息的准确性或有效性，或者关于任何声称的产品问题、故障或缺陷。意法半导体不承诺本文档完全准确无误，并明确拒绝对本文档包含的信息的准确性做出所有明示或暗示的保证。因此，您同意，在任何情况下，意法半导体或其关联公司均不对您因参考或使用本文档而引起的任何直接、间接、后果性、示例性、偶然性、惩罚性的或其他损害（包括利润损失）承担责任。

买方在订货之前应获取关于意法半导体产品的最新信息。意法半导体产品的销售依照订单确认时的相关意法半导体销售条款，包括但不限于其中的保修条款。

有鉴于此，请注意，意法半导体产品并非专门设计用于在上述条款和条件中描述的某些特定应用或环境中使用。

买方自行负责对意法半导体产品的选择和使用，意法半导体概不承担与应用协助或买方产品设计相关的任何责任。

提供的信息被视为准确可靠的信息。然而，意法半导体对使用此类信息的后果以及可能由此产生的任何侵犯第三方专利或其他权利的行为不承担任何责任。意法半导体不对任何知识产权进行任何明示或默示的授权或许可。

转售的意法半导体产品如有不同于此处提供的信息的规定，将导致意法半导体针对该产品授予的任何保证失效。

意法半导体和意法半导体徽标是意法半导体的商标。关于意法半导体商标的其他信息，请访问 www.st.com/trademarks。所有其他产品或服务名称是其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。