The latest version of this document is here: www.keil.com/appnotes/docs/apnt_326.asp

Abstract

This application note demonstrates the integration of X-CUBE-STL Software Test Libraries for STM32 devices in an application based on the Arm Functional Safety Run-Time System (FuSa RTS).

In this example, the STM32F413ZH device (on Nucleo-F413ZH board) is used as the target hardware. However, the analysis and principles are mostly universal and can be similarly applied to other STM32 devices.

Prerequisites

To reproduce the example described in this application notes the following components are required:

Components from Arm:

- Arm FuSa RTS for Cortex-M4 core: run-time system for functional safety applications. It includes FuSa RTX RTOS, FuSa CMSIS-Core, FuSa C library, and FuSa Event Recorder.
- Arm Compiler for functional safety: safety-qualified C/C++ compiler for Arm devices. Required for use by Arm FuSa RTS.
- Keil::STM32F4xx_DFP: Device Family Pack (DFP) for STM32F4 devices. Among other items, contains startup and system files used by the application.
- Arm Keil MDK: IDE and debugger used for project development and debug.

Components from ST:

- X-CUBE-STL-F4: Software test library for STM32F4 devices.
- STM32CubeProgrammer: programming utility for STM32 devices. In our example, it is used for ROM CRC calculations.

Example project

The project ZIP file is available for download on www.keil.com/appnotes/docs/apnt_326.asp. Download the ZIP file, unzip it, and double-click the Project.uvprojx file to open it in µVision.

Contents

Introduction

Diagnostic tests are commonly used in safety-related systems to reduce the effect of random hardware failures.

For many of its STM32 series, STMicroelectronics provides a safety-certified Self-Test Library (*X-CUBE-STL*) that implements diagnostic tests covering processor unit, flash and RAM memories.

When using the STL library in a system with the Arm FuSa RTS, the mutual interferences between both components must be analyzed. It needs to be ensured that the assumed safety requirements defined by the FuSa RTS as well as by X-CUBE-STL are satisfied.

This application note provides an example of using X-CUBE-STL in a FuSa RTS application. It is structured as follows:

- "*Introduction*" explains the structure and scope of the application note.
- "*STM32 X-CUBE-STL overview*" introduces X-CUBE-STL, which is part of the STM32 Safety Design Package.
- "*Arm FuSa RTS overview*" gives a brief overview of the Arm Run-Time System for Functional Safety.
- "*Integration of X-CUBE-STL with FuSa RTS*" introduces the approach of executing STL tests in a user SVC handler and analyzes the interferences that FuSa RTS and X-CUBE-STL may have on each other in such case.
- "*Add X-CUBE-STL tests to a FuSa RTS project*" describes the example project setup and goes through the required implementation steps when using X-CUBE-STL in an existing FuSa RTS project.
- "*Analyze X-CUBE-STL integration in MDK*" shows how to analyze the STL operation using MDK debugging capabilities.

Scope of the application note

This application note analyzes potential mutual impacts between the X-CUBE-STL self-test libraries and the Arm FuSa RTS and gives an example for their integration.

An STM32F413ZH device (on Nucleo-F413ZH board) is taken as a referenced platform and the corresponding X-CUBE-STL-F4 and FuSa RTS for Cortex-M4 software are used in the integration example. However, the provided analysis and described principles are mostly universal and can be similarly applied to other STM32 devices.

This application note is not part of the FuSa RTS Safety Package and does not show how to fulfill all conditions of use as specified in the SMT32F4 safety manual [1].

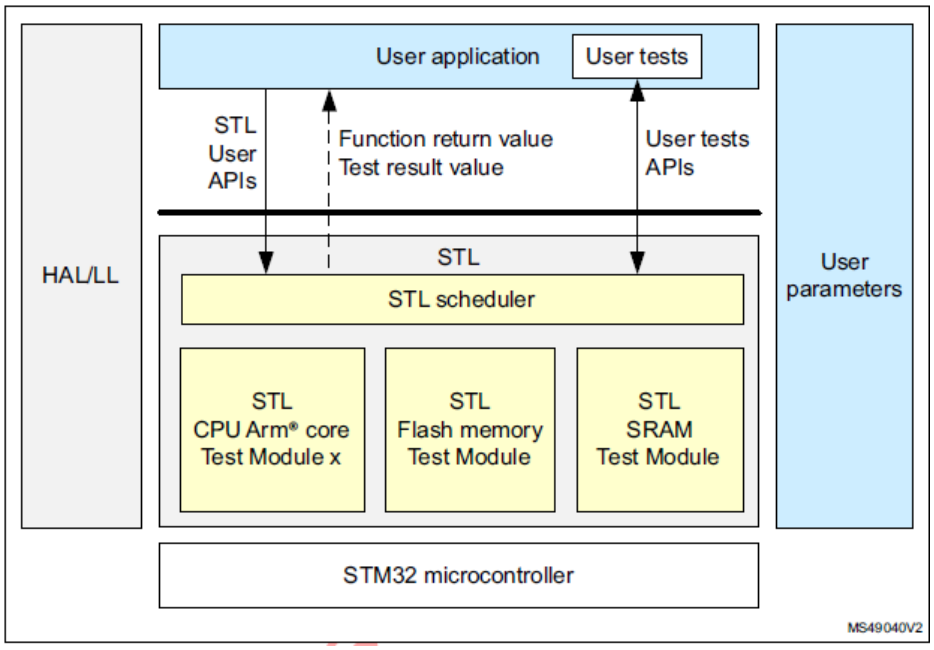
The FuSa RTS Safety Package and the STM32 SIL Functional Safety Design Package are the main references for the user implementing safety-related systems on the specific STM32 device.

STM32 X-CUBE-STL overview

STMicroelectronics provides SIL Functional Safety Design Packages for many of its STM32 device series. It includes a device family safety manual [1], Failure-Modes Effects Analysis (FMEA) [2], Failure Mode and Diagnostics Analysis (FMEDA)[3] and software tests library [6].

Section *"3.7 Conditions of Use"* in the STM32 F4 safety manual [1] lists safety mechanism requirements to be applied to the target device series. Among them are requirements *CPU_SM_0*, *FLASH_SM_0,* and *RAM_SM_0* that highly recommend the use of software tests for detecting permanent faults in CPU, Flash and SRAM memory respectively.

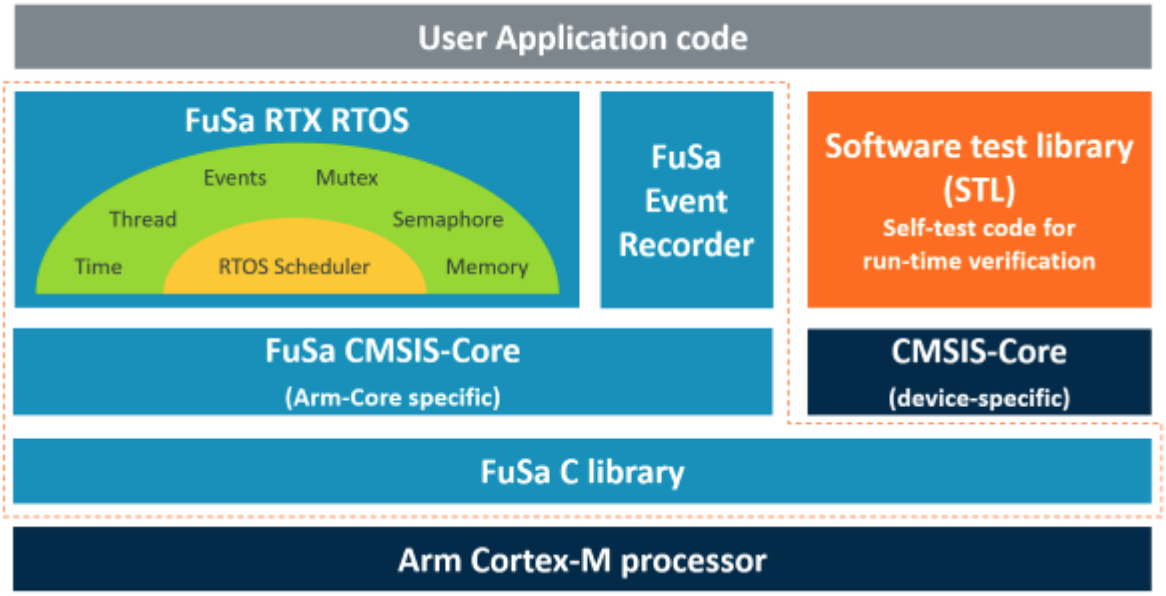
To implement software tests, STMicroelectronics provides *X-CUBE-STL* – a safety-certified software-based diagnostics suite for detecting random hardware failures on STM32 devices. The software test library is application-independent and provides an API for executing selected tests as shown in the figure below.



X-CUBE-STL is supplied with its safety manual [5] that provides requirements for the correct execution of the STL. It references the STL user guide [6] that additionally describes performance, constraints and the API.

Arm FuSa RTS overview

Arm's Run-Time System for Functional Safety (*FuSa RTS*) is a set of safety-certified software components for Cortex-M based devices. It contains the Real-Time Operation System (FuSa RTX RTOS), the processor abstraction layer (FuSa CMSIS-Core), the FuSa Event Recorder and the FuSa C library. The system block diagram below shows FuSa RTS components as blue blocks outlined with a red dotted line.



Arm FuSa RTS doesn't include a software test library (STL). According to the assumed safety requirement *UR_AP_3* from FuSa RTS safety manual [4], the user application is responsible for running a self-test library to detect hardware errors.

Integration of X-CUBE-STL with FuSa RTS

This section analyzes the assumptions of use for both products: X-CUBE-STL and FuSa RTS.

X-CUBE-STL CoUs and FuSa RTS operation

Section "*5.2 Assumptions*" in the X-CUBE-STL-F4 safety manual [5] lists the conditions of use (*CoU_STL*) that the application must fulfill.

Relevant for the FuSa RTS is *CoU_STL_7*. It states that the end user is responsible to guarantee the correct coexistence between the X-CUBE-STL-F4 and its application software according to the guidelines reported in the user guide [6], section "*STL user constraints*". For X-CUBE-STL, any external code is part of the application software and hence this CoU_STL also applies to the FuSa RTS.

Section "*STL user constraints and FuSa RTS operation*" below analyzes how these constraints impact the coexistence between STL and FuSa RTS.

It remains the end user's responsibility to ensure that all CoU_STLs are fulfilled in the system.

STL user constraints and FuSa RTS operation

Section "*4.3 STL user constraints*" in the X-CUBE-STL user guide [6] describes potential interferences between the application and the STL. Below are considerations for those constraints that have a direct impact on the operation of FuSa RTS. If no information is given for a constraint, it means that this constraint has no impact on FuSa RTS.

It stays the user's responsibility to consider all conditions of use and guidelines when adding the STL to the system.

Privilege-level

Subsection "*4.3.1 Privileged-level*" in [6] states that the STL must be executed in privileged mode, as otherwise access to target registers is not possible.

User threads in FuSa RTX RTOS can be configured to run either in privileged or non-privileged mode. However, such configuration is applied for all user threads at once (see *OS_PRIVILEGE_MODE* described in *FuSa RTX RTOS - RTX v5 Implementation – Configure RTX v5 – Thread Configuration* in [4]).

For safety reasons, user threads should execute in non-privileged mode. The portions of the STL code that requires privileged mode can be executed in the SVC exception handler.

According to section "*4.1.1 Scheduler principle*" in the STL user manual [6]: "the STL can be called under interrupt context, but re-entrance is forbidden". The SVC exception is not reentrant, and for STL execution this also guarantees that there are no multiple calls to the STL at the same time from different threads. This application note analyzes and explains this approach (calling the STL functions in SVC context) further.

Interrupt management

Section "*4.3.6 Interrupt management*" in [6] describes multiple interrupt-related interferences that need to be considered when using the STL. Below are considerations for those of them that are relevant for internal FuSa RTS operation.

Interrupt and CPU TM8 / Interrupt and RAM March C-tests

By default, during some CPU TMs and RAM March C-tests mask the STM32 interrupts and Cortex exceptions with configurable priority. Users can disable this by activating *STL_ENABLE_IT* flag, but in such case correct STL RAM test behavior is not warranted.

In this application note, we consider only the case when *ST_ENABLE_IT* flag is deactivated and hence the STM32 interrupts and Cortex exceptions with configurable priority are masked by the STL.

The priorities assigned to system interrupts used in FuSa RTX RTOS (SysTick, PendSV, and SVC) are described in the FuSa RTS safety manual [4], in the book "*FuSa RTX RTOS*", section "*RTX v5 Implementation*", subsections "*Create an RTX5 Project*" and "*Using Interrupts on Cortex-M*". SysTick and PendSV have the lowest priority while SVC has a priority just above the lowest possible. Hence SVC cannot be interrupted by SysTick and PendSV, but will be preempted by an interrupt with a higher priority.

This means that STL tests that are called within the SVC context will postpone the execution of system interrupts used by FuSa RTX RTOS (SysTick and PendSV). The interrupt masking performed within some of the STL tests has no further impact on the FuSa RTS timing behavior.

The application may use additional interrupts. The user shall analyze the impact of interrupt masking by STL on the system behavior.

Interrupt and general-purpose register

The X-CUBE-STL user manual [6] in section 4.3.6 requires that during STL execution, the general-purpose registers must be saved and restored in the STM32 interrupt and Cortex exceptions with configurable priority service routine. As otherwise there is a risk that the STL reports false errors.

As described in the section above, SVC has a higher priority than other system interrupts used by FuSa RTX RTOS (SysTick and PendSV) and will not be interrupted by them. Executing STLs within the SVC context already fulfills this requirement.

The application may use other interrupts and it stays the responsibility of the end user to respect this requirement in the system.

FuSa RTS user requirements and STL operation

Running STL tests in a user SVC handler may postpone the execution of the SysTick interrupt used by FuSa RTX RTOS. The FuSa RTS safety manual [4] has the following user requirement on this:

- ***UR_SYS_1****: The user shall keep ISR short, worst nested ISR time shall be at least shorter than RTX SysTick interrupt period.*

To have the *UR_SYS_1* satisfied, when STL tests are run in the SVC context, the user shall ensure that their execution time is at least shorter than the configured RTX SysTick period.

The X-CUBE-STL user guide provides reference execution times for its API calls. CPU tests are relatively short but Flash and RAM tests may require multiple milliseconds to cover the full memory range available on the device. The STL API allows to run the tests individually and test memory in blocks. Thus, the user can rely on this approach and split memory test execution to fulfill the *UR_SYS_1* requirement.

When *UR_SYS_1* is fulfilled, a SysTick interrupt that occurs during an SVC exception is postponed. The SysTick interrupt handler is executed after the SVC exception and other interrupts with higher priorities complete their execution.

If such a delay of SysTick execution is not desired by the system, the user can schedule the STL test execution at the beginning of a SysTick period and keep it significantly shorter than the SysTick interval. The example in the section "*Create an RTX thread for STL tests*" creates the thread that executes the STL with the highest priority. Using the function *osDelay* function before an STL test call ensures that STL test is executed right after the SysTick interrupt.

Additionally, FuSa RTS has the following timing requirement:

- ***UR_GGR_2:*** *The user shall perform a scheduling analysis to ensure that threads meet their deadline.*

It stays the responsibility of the user to analyze the impact of STL execution on other threads and ensure that the thread deadlines are met.

Add X-CUBE-STL tests to a FuSa RTS project

This chapter describes the steps for adding X-CUBE-STL tests in an existing FuSa RTS project. The implementation follows the approach analyzed above in chapter "*Integration of X-CUBE-STL with FuSa RTS*".

The sections below first describe how the example project is set up and how to add X-CUBE-STL files to it. Then the STL configuration is explained and finally, an example implementation for the integration is shown.

Project setup

The software setup is straightforward and requires that the software listed in the "*Prerequisites*" section is installed on the PC.

We assume that there is already an existing application that uses FuSa RTS in an MDK project. In our example, we use a simple Blinky application that runs on the STMicroelectronics Nucleo-F413ZH. It has 3 user threads: the main thread, the LED toggling thread, and the button reading thread.

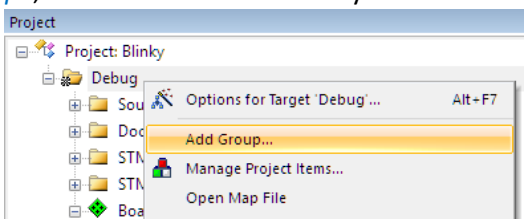
The FuSa RTS and the X-CUBE-STL components are independent of HAL and BSP. To keep the application universal, no HAL and BSP components were used in our example.

Add X-CUBE-STL files to the project

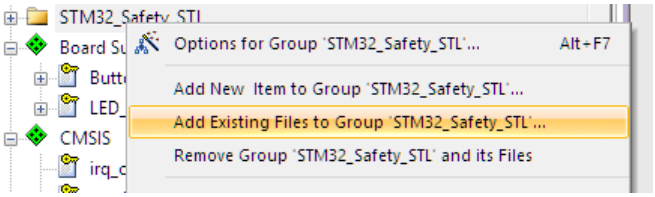
Section "*5.5.2 Steps to build an application from scratch*" in [6] provides the steps required for building a fresh application with X-CUBE-STL.

We use a simple Blinky project described in the section "*Project setup*", that uses FuSa RTS already:

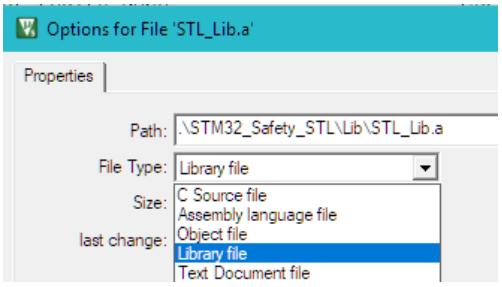
- The X-CUBE-STL libraries, source and header files are available in the X-CUBE-STL installation folder <X-CUBE-STL>\Middlewares\STM32_Safety_STL\. Just copy the STM32_Safety_STL folder to the root of your project.



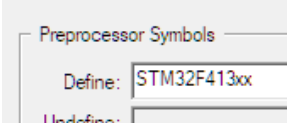
- Next, in the µVision **Project** window, right-click on the Target name (for example Debug) and then **Add Group**. Give it an appropriate name, for example, *STM32_Safety_STL*.
- Right-click the new group and add existing files to it. The following files are needed:
  - *Src\stl_user_param_template.c*
  - *Src\stl_util.c*
  - *Lib\STL_lib.a*

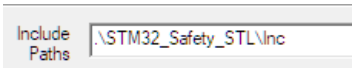


- By default, µVision assumes that files with *.a* extensions are assembler files and processes them accordingly. However, *STL_lib.a* is a library file. To handle it correctly, we need to specify its type manually. In the **Project** window, right-click the *STL_lib.a* file and select its **Options..** In the **File Type** drop-down menu, select **Library file**. Click OK.



- Open the **Options for target..** and go to the **C/C++ (AC6)** tab:
  - In the **Preprocessor Symbols** area, add the device family to the **Define** field. In our example, it is "*STM32F413xx*".
  - Add the path *.\STM32_Safety_STL\Inc* folder to **Include Paths**.



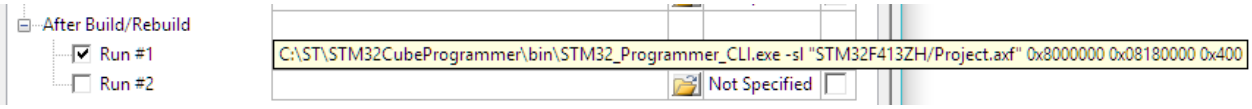


Configurations for the flash memory test

For flash memory tests, X-CUBE-STL relies on pre-calculated CRC values over the firmware image that shall be stored in a dedicated memory area at the end of the ROM. *STM32 Programmer* tool can be used to add a corresponding segment with CRC values into a firmware file.

The ROM address ranges must be consistent in the STM32 Programmer, the scatter-loading file and the STL configuration file *stl_user_param.c*.

- The STM32 Programmer may be called in MDK using the after build setting. Go to **Options for Target...** and select the **User** tab. In the **After Build/Rebuild** section, enable **Run #1** add a call to STM32CubeProgrammer command-line interface:



  X-CUBE-STL user guide [6] provides additional details about flash memory tests and CRC tool setup.

  Note that the STM32 Programmer calculates CRC only over the actually occupied flash area. Hence, an STL flash test will fail if executed outside of this area.

- According to Step 4 from Section "*5.5.2 Steps to build an application from scratch*" in [6], the user shall check the configuration file *stl_user_param.c* and potentially update ROM parameters. In our example for STM32F413xx devices this stays as:

```
#define STL_ROM_START_ADDR (0x08000000UL) /* customizable */
#define STL_ROM_END_ADDR   (0x0817FFFFUL) /* customizable */ /* 1536 Kbytes */
```

- ROM area is specified in the scatter file. Section "*Configurations for RAM memory test*" below provides the complete scatter file used in our example project.
- In our example, the firmware image is just above the 47 KB, so we can test 48 flash sections as shown in section "*Create an RTX thread for STL tests*".

Configurations for RAM memory test

X-CUBE-STL requires a backup buffer in RAM to execute the C-March RAM tests. The linker script shall specify a small RAM section named "*backup_buffer_section*" for this.

To access the scatter file in a µVision project, go to **Options for Target...** dialog, select the **Linker** tab and then click the **Edit...** button next to the **Scatter File** line.

The following linker script is used in the example application:

```
; *************************************************************
; *** Scatter-Loading Description File ***
; *************************************************************

LR_IROM1 0x08000000 0x00180000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00180000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }

  RW_IRAM1 0x20000000 0x00050000{  ; RW data
  *(backup_buffer_section)
  }
  RW_IRAM2 +0  {  ; RW data
  .ANY (+RW +ZI)
  }
  RW_IRAM3 +0 UNINIT {
    EventRecorder.o (+ZI)
  }
  RW_IRAM4 +0 UNINIT {  ; RW data
```

```
  .ANY (STACK)
  }
  RW_IRAM5 +0  {  ; RW data
  .ANY (HEAP)
  }
}
```

Add support for Arm Compiler 6

Section "*2.6 Other compatibilities*" in [5] specifies that the main core of X-CUBE-STL-F4 has no dependencies to the compiler used to build the final application software and that the user is free to select the compiler with no constraints coming from the use of X-CUBE-STL-F4.

FuSa RTS requires the use of the safety-qualified Arm Compiler 6 (v6.6.2). The X-CUBE-STL C files *stl_user_param_template.c* and *stl_util.c* added to the project in the previous step need to be slightly modified for Arm Compiler 6 support:

- In ***stl_user_param_template.c,*** extend the *#elif defined* line for (__CC_ARM) and (__GNUC__) with Arm Compiler 6 defines. This ensures that the buffer *STL_aRamTmBckUpBuf* gets defined and correctly placed into the *backup_buffer_section* in RAM.

  Added code is highlighted in **bold** below. Splitting the code into multiline with "\" is not necessary. It is done here to keep the code correct as full lines don't fit in the document page.

```
…
#elif defined (__CC_ARM) || defined(__GNUC__) || (defined (__ARMCC_VERSION) && \
(__ARMCC_VERSION >= 6010050))
 uint32_t STL_aRamTmBckUpBuf[STL_TM_RAM_BCKP_BUF_SZ]
 __attribute__((section("backup_buffer_section")));
#endif
…
```

- In ***stl_util.c,*** there are three occurrences of *#elif defined (__GNUC__)* that need to be extended with Arm Compiler 6 defines. This ensures that interrupt masking required by some STL tests gets correctly executed with Arm Compiler 6.

```
…
#elif defined (__GNUC__) || (defined (__ARMCC_VERSION) && (__ARMCC_VERSION >= 6010050))
  __asm("cpsid i" : : : "memory");
#endif
…
#elif defined (__GNUC__) || (defined (__ARMCC_VERSION) && (__ARMCC_VERSION >= 6010050))
  __asm("MRS %0, primask" : "=r"(result));
#endif
…
#elif defined (__GNUC__) || (defined (__ARMCC_VERSION) && (__ARMCC_VERSION >= 6010050))
  __asm("MSR primask, %0" : : "r"(MaskReg));
#endif
…
```

Implement STL tests in the application

After the X-CUBE-STL files are added to the project and configured we can start using them in the application. This is done in the following steps that are described in detail in this subsection:

- First, in "*Create STL test execution module*" we create a new C file that implements the STL test functions for CPU, ROM, and RAM testing. Those functions are expected to be called from the SVC handler.
- Then in "*Create user SVC calls*" the STL test functions are wrapped into user SVC calls.
- Finally, in "*Create an RTX thread for STL tests*" we implement a new user RTX thread that uses SVC calls for STL execution.

Create STL test execution module

Correct STL test execution requires multiple API calls to the X-CUBE-STL library for scheduler initialization, test configuration, execution and analysis of the results. To support a modular application structure, we create a new file *stl_run.c* that implements 3 test functions for running CPU, flash and RAM tests respectively. Those functions perform necessary steps required for correct STL test execution. The functions then will be wrapped into SVC calls and used in the application.

Additionally, an error handler function is present `StlErrorHandler()`. Its content is application-specific and is out of scope for this application note.

The test functions are explained in detail below.

- **CPU TM test function**

  To avoid code duplication, a single function `StlCpuTMTest(uint32_t testNumber)` is used for executing *CPU_TM_<testNumber>* STL test depending on the argument value. It is implemented as follows:

```
#include "stdlib.h"
#include "stl_user_api.h"
…
/* Array with pointers to the API functions for corresponding STL CPU TM tests */
static STL_Status_t (*stlCpuTests[STL_CPU_TM_MAX])(STL_TmStatus_t *) = {
  STL_SCH_RunCpuTM1,
  STL_SCH_RunCpuTM1L, // choose one test to run TM1 or TM1L
  STL_SCH_RunCpuTM2,
  STL_SCH_RunCpuTM3,
  STL_SCH_RunCpuTM4,
  STL_SCH_RunCpuTM5,
  STL_SCH_RunCpuTM6,
  STL_SCH_RunCpuTM7,
  STL_SCH_RunCpuTM8,
  STL_SCH_RunCpuTM9,
  STL_SCH_RunCpuTM10,
  STL_SCH_RunCpuTM11,
};

/* Execute STL CPU TM test testNumber */
void StlCpuTMTest(uint32_t testNumber) {
  STL_TmStatus_t StlCpuTmStatus = STL_ERROR;

  /* Verify testNumber value */
  if (testNumber > STL_CPU_TM_MAX) {
    StlErrorHandler();
  }
  /* Init STL */
  else if (STL_SCH_Init() != STL_OK) {
    StlErrorHandler();     /* STL Defence Programming */
  }
  /* Call STL_SCH_RunCpuTMX test */
  else if (((*stlCpuTests[testNumber])(&StlCpuTmStatus)) != STL_OK) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  else if (StlCpuTmStatus != STL_PASSED) {
    stlErrorHandler();     /* STL Tests FAILED  */
  }
}
```

- **Flash test function**

  The function for flash testing is implemented so that corresponding STL tests can be run in the shortest possible blocks - just testing one memory section. This allows us to minimize execution in the SVC context when necessary.

```
/* Execute STL Flash tests for specified memory */
void StlFlashTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections) {
```

```
  /* Flash configurations: */
  STL_MemSubset_t FlashSubset;
  STL_MemConfig_t FlashConfig;

  FlashSubset.StartAddr = startAddr;
  FlashSubset.EndAddr   = endAddr;
  FlashSubset.pNext     = NULL;
  FlashConfig.pSubset   = &FlashSubset;
  FlashConfig.NumSectionsAtomic = sections;

  STL_TmStatus_t StlFlashStatus = STL_ERROR;

  /* Init STL */
  if (STL_SCH_Init() != STL_OK) {
    StlErrorHandler();     /* STL Defence Programming*/
  }
  /* Init Flash TM */
  else if (STL_SCH_InitFlash(&StlFlashStatus) != STL_OK) {
    StlErrorHandler();  /* STL cannot be run */
  }
  /* check Flash TM status is reset to STL_NOT_TESTED */
  else if (StlFlashStatus != STL_NOT_TESTED) {
    StlErrorHandler(); /* STL cannot be run */
  }
  /* configure Flash TM */
  else if (STL_SCH_ConfigureFlash(&StlFlashStatus, &FlashConfig) == STL_KO) {
    StlErrorHandler();     /* STL Defence Programming */
  }
  /* check Flash TM status is reset to STL_NOT_TESTED */
  else if (StlFlashStatus != STL_NOT_TESTED) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  /* Run Flash TM */
  else if (STL_SCH_RunFlashTM(&StlFlashStatus) != STL_OK) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  /* check Flash TM status */
  else if (StlFlashStatus != STL_PASSED) {
    StlErrorHandler(); /* STL Test Failed */
  }
}
```

- **RAM test function**

  The function for RAM testing is implemented so that corresponding STL tests can be run in the shortest possible blocks - just testing one memory section. This allows us to minimize execution in the SVC context when necessary.

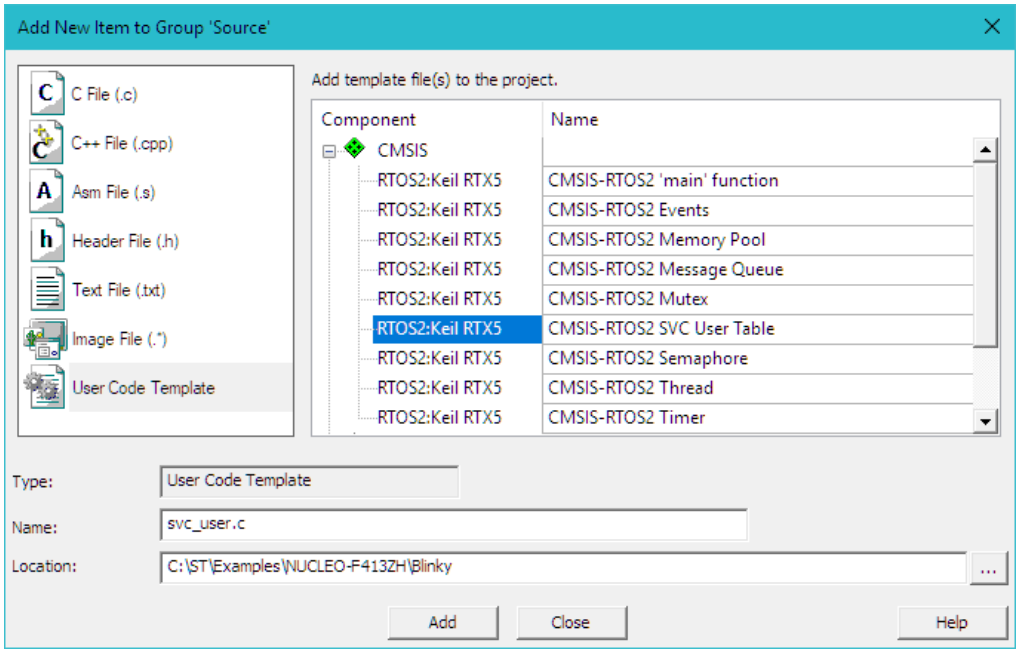
```
/* Execute STL RAM tests for specified memory */
void StlRamTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections) {

  /* RAM configurations*/
  STL_MemSubset_t RamSubset;
  STL_MemConfig_t RamConfig;

  RamSubset.StartAddr = startAddr;
  RamSubset.EndAddr   = endAddr;
  RamSubset.pNext     = NULL;
  RamConfig.pSubset   = &RamSubset;
  RamConfig.NumSectionsAtomic = sections;

  STL_TmStatus_t StlRamStatus = STL_ERROR;

  /* Init STL */
  if (STL_SCH_Init() != STL_OK) {
```

```
    StlErrorHandler();     /* STL Defence Programming*/
  }
  /* Init RAM TM */
  else if (STL_SCH_InitRam(&StlRamStatus) != STL_OK) {
    StlErrorHandler();  /* STL cannot be run */
  }
  /* check RAM TM status is reset to STL_NOT_TESTED */
  else if (StlRamStatus != STL_NOT_TESTED) {
    StlErrorHandler(); /* STL cannot be run */
  }
  /* configure RAM TM */
  else if (STL_SCH_ConfigureRam(&StlRamStatus, &RamConfig) == STL_KO) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  /* check RAM TM status is reset to STL_NOT_TESTED */
  else if (StlRamStatus != STL_NOT_TESTED) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  /* Run Ram TM */
  else if (STL_SCH_RunRamTM(&StlRamStatus) != STL_OK) {
    StlErrorHandler(); /* STL Defence Programming */
  }
  /* check Ram TM status */
  else if (StlRamStatus != STL_PASSED) {
    StlErrorHandler(); /* STL Test Failed */
  }
}
```

Create user SVC calls

Since we want to execute STL tests in SVC context, the test functions implemented above need to be wrapped in corresponding SVC calls. FuSa RTX provides user code templates for that.

In the **Project** window, right-click on the group with the application code and then click **Add New Item to Group…**, select **User Code Template.** Expand **CMSIS** category, select the **CMSIS-RTOS2 SVC User Table** and click **Add**.



A new file called ***svc_user.c*** will be added to the project. This file should specify user functions that the application wants to execute in the SVC context.

In our example, the SVC functions are mapped to the test functions implemented in "*Create STL test execution module*" and the ***svc_user.c*** file has the following code:

```
#include <stdint.h>

extern void StlCpuTMTest(uint32_t testNumber);
extern void svc_StlCpuTMTest(uint32_t testNumber);
__attribute__((always_inline)) void svc_StlCpuTMTest (uint32_t testNumber) {
   register unsigned r0 __asm("r0") = testNumber;
   __asm volatile("SVC #1" : : "r" (r0));
}
extern void StlFlashTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);
extern void svc_StlFlashTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);
__attribute__((always_inline)) void svc_StlFlashTest (uint32_t startAddr, uint32_t endAddr,
uint32_t sections) {    \
   register unsigned r0 __asm("r0") = startAddr;
   register unsigned r1 __asm("r1") = endAddr;
   register unsigned r2 __asm("r2") = sections;
  __asm volatile("SVC #2" : : "r" (r0),"r" (r1),"r" (r2));
}

extern void StlRamTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);
extern void svc_StlRamTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);
__attribute__((always_inline)) void svc_StlRamTest (uint32_t startAddr, uint32_t endAddr,
uint32_t sections) {    \
   register unsigned r0 __asm("r0") = startAddr;
   register unsigned r1 __asm("r1") = endAddr;
   register unsigned r2 __asm("r2") = sections;
  __asm volatile("SVC #3" : : "r" (r0),"r" (r1),"r" (r2));
}

void EnablePrivelegedMode(void);
void EnablePrivelegedMode(void){
__set_CONTROL( __get_CONTROL( ) & ~CONTROL_nPRIV_Msk ) ;
}

extern void svc_EnablePrivelegedMode(void);
__attribute__((always_inline)) void svc_EnablePrivelegedMode (void) {    \
  __asm volatile("SVC #4" : :);
}

void DisablePrivelegedMode(void);
void DisablePrivelegedMode(void){
__set_CONTROL( __get_CONTROL( ) | CONTROL_nPRIV_Msk ) ;
}

extern void svc_DisablePrivelegedMode(void);
__attribute__((always_inline)) void svc_DisablePrivelegedMode (void) {    \
   __asm volatile("SVC #5" : :);
}

#define USER_SVC_COUNT  5     // Number of user SVC functions
extern void * const osRtxUserSVC[1+USER_SVC_COUNT];
void * const osRtxUserSVC[1+USER_SVC_COUNT] = {
  (void *)USER_SVC_COUNT,
  (void *)StlCpuTMTest,
  (void *)StlFlashTest,
  (void *)StlRamTest,
  (void *)EnablePrivelegedMode,
  (void *)DisablePrivelegedMode,
// ...
};
```

Create an RTX thread for STL tests

The exact scheduling of the STL test depends on the application. For simplicity, our application runs a dedicated STL thread that periodically tests CPU, flash and memory.

The STL thread is configured with a priority higher than the other user threads. This shall ensure that the STL thread doesn't get blocked by other threads. X-CUBE-STL user guide provides a stack requirement for STL operation and this is higher than the default 256 Bytes used in our FuSa RTS example. So, the STL thread is configured with 512 Bytes for the stack.

The code below shows how the STL thread is configured and created in the main application thread.

```
#define STL_THREAD_STK_SZ (512U)
static uint64_t stl_thread_stk[STL_THREAD_STK_SZ / 8];
static const osThreadAttr_t stl_thread_attr = {
  .stack_mem  = &stl_thread_stk[0],
  .stack_size = sizeof(stl_thread_stk),
  .priority   = osPriorityHigh
};

static osThreadId_t tid_thrSTL;  /* Thread id of thread: STL */

/*----------------------------------------------------------------------------
 * Application main thread
 *---------------------------------------------------------------------------*/
__NO_RETURN static void app_main (void *argument) {
…
  tid_thrSTL = osThreadNew (thrSTL, NULL, &stl_thread_attr); /* create STL thread */
  if (tid_thrSTL == NULL) { /* add error handling */ }
…
}
```

The implementation of the STL thread itself is rather application specific. In our example, it runs the STL tests every 5 seconds and requires the code shown below. The *svc_xx* functions are directly mapped to corresponding test functions implemented in "*Create STL test execution module*".

```
#include "stl_user_api.h"

extern void svc_StlCpuTMTest(uint32_t testNumber);
extern void svc_StlFlashTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);
extern void svc_StlRamTest(uint32_t startAddr, uint32_t endAddr, uint32_t sections);

/* Array for CPU ARM Core Test Module test list */
static STL_TmEnable_t aCpuTmEnable[STL_CPU_TM_MAX]={
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 1 */
  STL_TEST_DISABLE,/* CPU ARM Core Test Module 1L */ // choose one test to run TM1 or TM1L
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 2 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 3 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 4 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 5 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 6 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 7 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 8 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 9 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 10 */
  STL_TEST_ENABLE, /* CPU ARM Core Test Module 11 */
};

#define FLASH_SECTION_SIZE 1024U /* Fixed by STL */
#define TEST_FLASH_SECTIONS 48  /* Customizable. Amount of sections to be tested in total */
#define TEST_FLASH_SECTION_NB 1 /* Customizable. Sections to be verified in a single run */
#define TEST_ROM_START_ADDR 0x08000000U
#define TEST_ROM_END_ADDR (TEST_ROM_START_ADDR + TEST_FLASH_SECTIONS*FLASH_SECTION_SIZE)

#ifdef STL_DISABLE_RAM_BCKUP_BUF
#define RAM_BACKUP_BUFFER_SIZE 0
#else
```

```
#define RAM_BACKUP_BUFFER_SIZE 32 /* Fixed value as in stl_user_param.c */
#endif
#define RAM_SECTION_SIZE 128U     /* Fixed by STL */
#define TEST_RAM_START_ADDR (0x20000000U + RAM_BACKUP_BUFFER_SIZE)
#define TEST_RAM_END_ADDR 0x20004000U /* customizable */
#define TEST_RAM_SECTION_NB 1 /* Amount of RAM section tested at once*/

/*----------------------------------------------------------------------------
  thrSTL: run STL
 *----------------------------------------------------------------------------*/
__NO_RETURN static void thrSTL(void *argument) {

  (void)argument;
  uint32_t i=0;
  uint32_t currRomStartAddr = 0;
  uint32_t currRomEndAddr   = 0;
  uint32_t RomSections      = TEST_FLASH_SECTION_NB;

  uint32_t currRamStartAddr = 0;
  uint32_t currRamEndAddr   = 0;
  uint32_t RamSections      = TEST_RAM_SECTION_NB;

  for (;;) {
    /** Run CPU tests *******/
    for (i=0; i < STL_CPU_TM_MAX;i++){
      if(aCpuTmEnable[i] == STL_TEST_ENABLE){
        osDelay(1);
        svc_StlCpuTMTest(i);
      }
    }
    /** Run Flash tests *******/
    currRomStartAddr = TEST_ROM_START_ADDR;
    while (currRomStartAddr <= TEST_ROM_END_ADDR-(RomSections*FLASH_SECTION_SIZE)) {
      currRomEndAddr = currRomStartAddr + (RomSections*FLASH_SECTION_SIZE)-1;
      osDelay(1);
      svc_StlFlashTest(currRomStartAddr,currRomEndAddr,RomSections);
      currRomStartAddr = currRomEndAddr+1;
    }
    /** Run RAM tests *******/
    currRamStartAddr = TEST_RAM_START_ADDR;
    while (currRamStartAddr <= TEST_RAM_END_ADDR-(RamSections*RAM_SECTION_SIZE)) {
      currRamEndAddr = currRamStartAddr+(RamSections*RAM_SECTION_SIZE)-1;
      osDelay(1);
      svc_StlRamTest(currRamStartAddr,currRamEndAddr,RamSections);
      currRamStartAddr = currRamEndAddr+1;
    }
    osDelay(5000);
  }
}
```

Execution in thread mode

X-CUBE-STL-F4 used in our example (see "*References and Useful Links*") does not require that any of the STL tests shall be executed in thread mode. But STLs for some other STM32 families (for example STM32F1, STM32F7) require that the CPU TM7 test is executed in privileged thread mode and otherwise an error is returned. This means that it cannot be run in the SVC context that gets executed in handler mode.

The solution could be to enable the privileged mode before executing the test in the STL thread and disable it right after. The implementation for the enable and disable functions is already provided in the ***svc_user.c*** file shown in section "*Create user SVC calls*". The functions can be used as follows:

```
extern void svc_EnablePrivelegedMode(void);
extern void svc_DisablePrivelegedMode(void);
extern void StlCpuTMTest(uint32_t testNumber);

//…
```

```
__NO_RETURN static void thrSTL(void *argument) {
//…
  for (;;) {
    /** Run CPU tests *******/
    for (i=0; i < STL_CPU_TM_MAX;i++){
      if(aCpuTmEnable[i] == STL_TEST_ENABLE){
        osDelay(1);
        if (i == 7){  // Run CPU TM7 in privileged thread mode
          svc_EnablePrivelegedMode();
          StlCpuTMTest(i);
          svc_DisablePrivelegedMode();
        }
        else{
          svc_StlCpuTMTest(i);
        }
      }
    }
  //…
```

Note, that FuSa RTX RTOS kernel does not change the privilege level during a thread switch. Hence if a thread switch occurs when the privileged level is enabled, then also the new thread will be executed in the privileged mode. The user shall analyze potential safety impact of such behavior on its application.

Analyze X-CUBE-STL integration in MDK

Arm Keil MDK has advanced debug and trace capabilities that allow us to analyze the X-CUBE-STL integration in our example. The System Analyzer window graphically displays the exceptions and RTOS events synchronized over time. µVision User's Guide [7] provides more details on the System Analyzer.

In our example, we use a high-speed ULINK*pro* adapter for the debug connection to the MCU. ULINK*pro* is connected to the NUCLEO-F413ZH board via an SWD adapter.

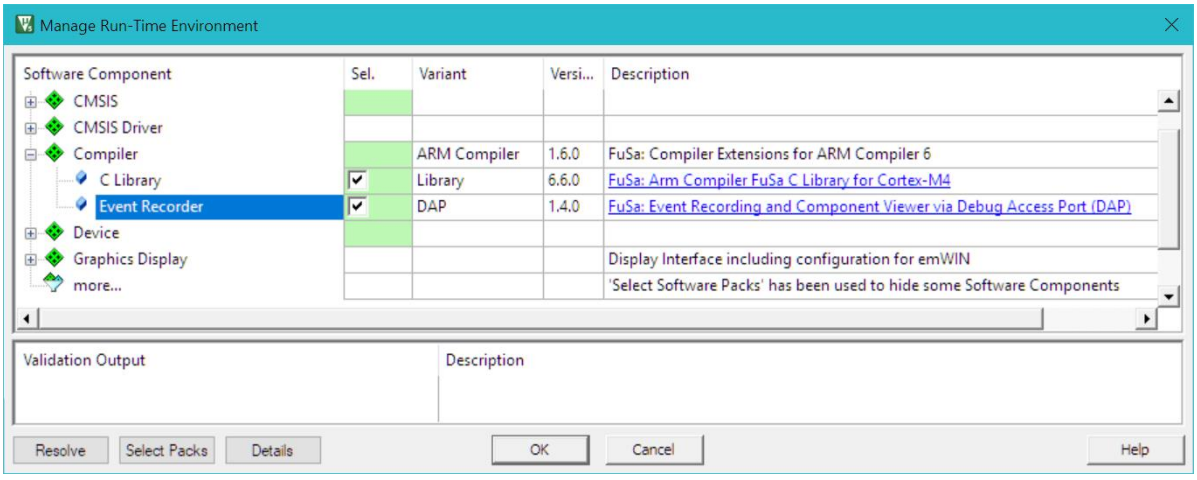
Project configuration

Configure SWO Trace

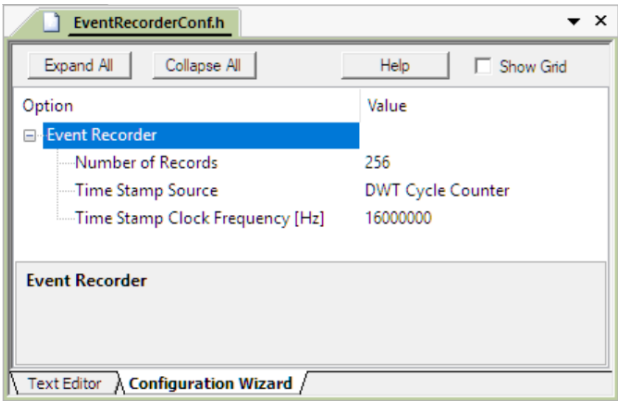
The SWO trace needs to be enabled and configured for the target debug adapter. Refer to the ULINK*pro* User's Guide [8] for details on how to do this.

Configure FuSa Event Recorder

The FuSa Event Recorder component shall be present in the application to enable RTOS-aware debugging. To add it to the project open **Manage Run-Time Environment** window and then in the **Compiler** group enable **Event Recorder** software component.



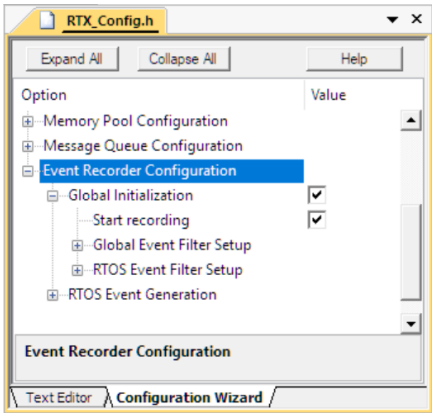
Then the FuSa Event Recorder files will be added to the project under the **Compiler** group. In the *EventRecorderConf.h* file we need to specify the number of records for the Event Recorder buffer in RAM, time stamp source and clock frequency. Configuration Wizard view allows GUI-like editing of the file.



Event Recorder buffer needs to be placed into a non-initialized RAM area. Corresponding entry can be seen in the scatter file described in the section "*Configurations for RAM memory test*".

Configure FuSa RTX RTOS Events

In the FuSa RTX configuration file *RTX_Config.h* we need to enable Event Recorder initialization and recording start.

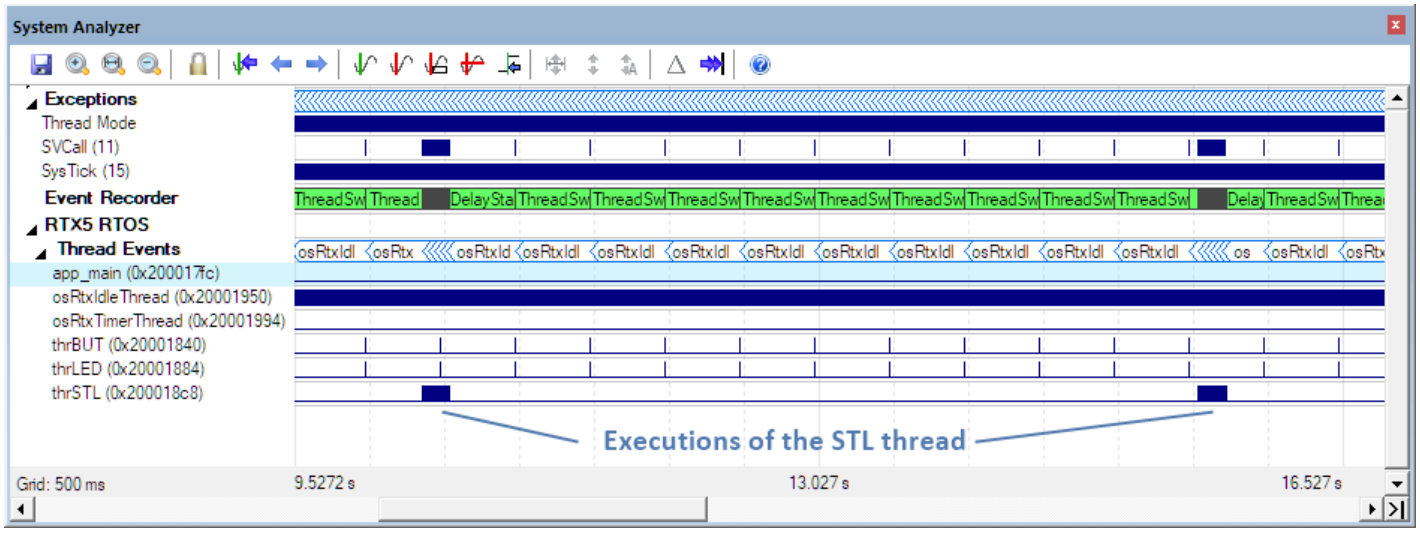


We keep all event categories enabled in the **RTOS Event Generation** group. But to reduce the number of events placed in the limited Event Recorder buffer we need to configure **RTOS Event Filter Setup** for our debug purpose. We enable Error, Operation and Detailed Operation event types for **Kernel**, **Thread** and **Generic Wait** categories. For other categories, only Error events are enabled.

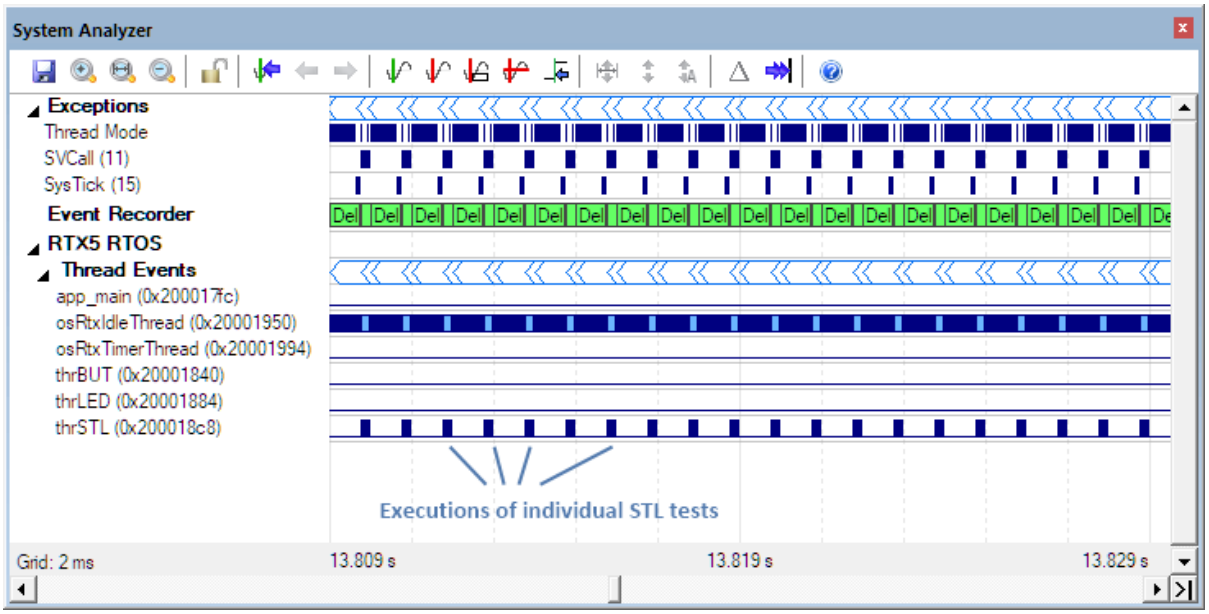
Observed System Behavior

When the debug session is started and the application is running we can open System Analyzer window and see the exceptions and RTOS threads appearing in the window over time.

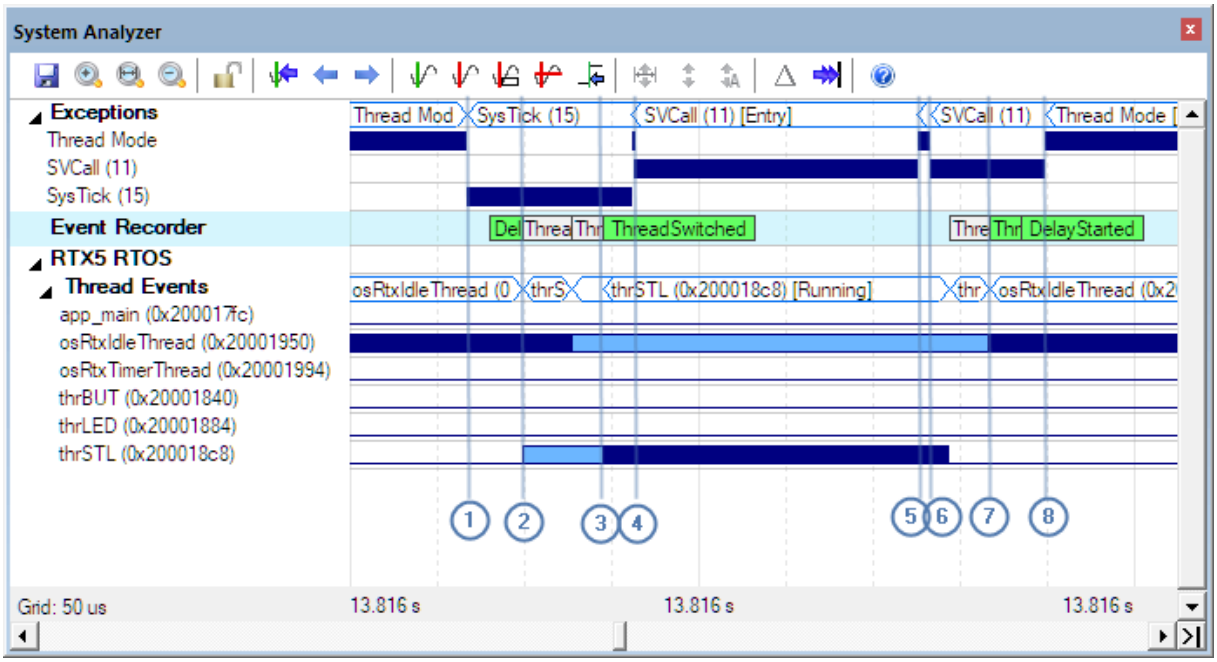
From the default view, as shown in the figure below, we can observe large blocks of activity in the STL thread every 5 seconds. This is our periodic STL test executions.



Zooming in to one of such areas we can see that this is not a continuous execution, but many short runs of the STL thread. This is also how it should be per our implementation. We execute STL tests individually in the shortest possible blocks and have a 1 SysTick delay in between.



Zooming further into a single continuous STL thread run, we can analyze the STL test execution in more detail, as shown in the figure below.



Here are the explanations for the highlighted events:

1) osRtxIdleThread is in Running state. STL thread is in Blocked state because it has called `osDelay(1)` function before and is still waiting for the delay to end. At this point, SysTick execution is started and shortly after the delay is completed.
2) Then the STL thread is changed from Blocked to Ready state.

3) Since STL thread has a higher priority it preempts the osRtxIdleThread and is now changed to Running state.
4) The *svc_xx* function is called within the STL thread and its execution in the SVC context is started.
5) The *svc_xx* function is completed, and the processor exits from the SVC exception. The STL thread executes some code in the thread mode (the `for()` loops)
6) *osDelay* function is called in the STL thread. It is executed in the SVC context and changes the STL thread state from Running to Blocked.
7) osRtxIdleThread state is changed to Running state.
8) osRtxIdleThread is executed in thread mode.

Observed operation is as expected according to the implementation. The STL tests are kept short and executed one by one in the SVC exceptions with delays in between. Execution of each STL test is started right after a SysTick interrupt to reduce the risk of postponing the next SysTick.

Summary

In this application note, we have analyzed the possibility of using X-CUBE-STL-F4 in a FuSa RTS based application. It shows that individual test execution using the SVC exception can be a valid approach for the integration of X-CUBE-STL. An example implementation for such integration is provided for the STM32F413ZH device.

References and Useful Links

[1] UM1840 STM32F4 Series safety manual, Rev. 6 -June 2019
[2] AN5141 Results of FMEA on STM32F4 Series microcontrollers, Rev. 3, June 2019
[3] AN5140 FMEDA snapshots for STM32F4 Series microcontrollers, Rev. 4, June 2019
[4] FuSa RTS v1.0.1 Safety manual
[5] UM2490 STM32F4 Series self-test library safety manual, Rev. 4, April 2019
[6] UM2494 STM32F4 Series self-test library user guide, Rev. 2, February 2019
[7] µVision User's Guide
[8] ULINKpro User's Guide
[9] Arm Safety Compiler
[10] Arm FuSa RTS