Cortex-M4
Structural Core Self Test Library

User Manual

Rev. 1.5
27 February 2020

Content

1 Introduction

This document contains information and description of the interface to the Structural Core Self Test Library (SCST) for Cortex-M4 platform, defined custom code and data sections, handling of interrupts, and a description of possible results that can be returned by the SCST Library.

1.1 Abbreviations Used in the Document

HW	Hardware
ISR	Interrupt Service Routine
OS	Operating System
RAM	Random Access Memory (Static or Dynamic)
RTM	Released To Market
SCST	Structural Core Self Test
TS	Test Shell

1.2 Compiler Settings and Measurement Conditions

This chapter provides information on execution time and memory consumption of the SCST Library and its individual components. All the measurements were done on the object code produced with the same compiler and same compiler switches.

Measurement of execution time was done on the SCSL Library object code executed on the target device. In the process of measurement execution time, the Cortex-M4 core was forced to run at 112 MHz. Measurements were done for the following case: Instruction and Data Caches both enabled and the caches were invalidated just before every measurement. LPIT (Low Power Interrupt Timer) timer was used to measure an execution time. The timer was forced to run at 112MHz. The timer counter value was read for the first time at the start of execution of the measured part of the SCST Library, and was read for the second time at end of execution of the measured part. Number of elapsed LPIT timer clock cycles was determined as a difference between first read value and the second read value. The obtained difference was divided by 112.000.000 to obtain an execution time of the measured portion of code in seconds. Note, that for devices which do not support HSRUN mode, the Cortex-M4 core was forced to run at 80 MHz.

Measurement of the memory consumption was done based on the information from the .map file generated for a stub-model of application including SCST Library code. When SCST Library code is integrated into the customer application, some minor mismatch in the measured memory consumption can be observed due to different alignment of the custom sections.

One of the following compilers together with compilers options was used to produce an object code which was used for execution time and consumed memory measurements:

Supported compiler:

- GreenHills compiler v.2017.1.4 and v.2017.5.4 Compiler flags:
  ```
  -cpu=cortexm4
  -littleendian
  -G
  -dwarf2
  -thumb_lib
  -DM4_DEVICE_RESERVED_ADDR=0x40080000
  ```

  Assembler flags:
  ```
  -preprocess_assembly_files
  ```

- GCC compiler for ARM v.6.3.1 20170509
  ```
  -mcpu=cortex-m4
  -march=armv7e-m
  -mthumb
  -mlittle-endian
  -Wall
  -gdwarf-2
  -gstrict-dwarf
  -DM4_DEVICE_RESERVED_ADDR=0x40080000
  ```

  Assembler flags:
  ```
  -x assembler-with-cpp
  ```

- IAR Embedded Workbench 8.11.2
  ```
  -cpu=Cortex-M4
  -endian=little
  -fpu=None
  -M4_DEVICE_RESERVED_ADDR=0x40080000
  ```

1.3 Supported Devices

This chapter provides information about supported devices.
Supported devices:

- S32K144
- MWCT101xS
- S32K144W

2 Features

- The SCST library provides tests to achieve the claimed diagnostic coverage (analytically estimated).
- The SCST library can be executed periodically at run time. This way, it contributes to a Single-Point Fault metric. The library preserves execution context of application and device configuration.
- The included tests cover most of the core instructions, as well as the tests targeting specific IP blocks of the core:
    - Core control logic (branch control, exception control);
    - Core data path including:
        - Register file and register multiplexing;
        - ALU, multiplier, divider, load/store, and other execution units;
        - SIMDSAT;
        - Instruction decoder, 16-Bit, 32-Bit;
- Interrupts can be enabled during execution of the most of the tests. SCST library provides its own interrupt vector table and wrappers for interrupt service routines, which in case of unexpected for the library interrupt, forwards it to the corresponding interrupt handler of the OS / user application. SCST library supports nested interrupts without any limitations.

3 Characteristics of the SCST Library

3.1 Coverage

Details regarding the assessment of the coverage are contained in the document M4_S32K144_SCST_Library_FaultCoverage_Estimation.xlsx which is delivered in safety package.

3.2 Memory Consumption

The memory consumption of the SCST Library is specified in Table 1. Note that measurement was done for one randomly selected compiler with compiler options listed in Chapter 1.2.

Table 1. Memory usage by the SCST Library

Stack (Bytes)	RAM Total (Bytes) (excluding stack)	Flash Total (Bytes)
56[1]+104[2]	220	60742

3.3 Execution Time

The SCST Library execution time is specified in Table 2 depending on the test invocation scenario. Execution time of every individual test as well as user ISR invocation latency is specified in Table 9.
Note that measurement was done for one randomly selected compiler with compiler options listed in Chapter 1.2.

Table 2. Execution Time of the SCST Library

Test Invocation Scenario	Accumulated Tests Execution Time µs @ 112 MHz (µs @ 80 MHz)[3]
All atomic tests are invoked through a minimum calls to the test shell: `result = m4_scst_execute_core_tests(0, 43);`	1025,80 (1432,42)[3]

Every atomic test is invoked through a separate call to the test shell:
`result = m4_scst_execute_core_tests(0, 0);`
`result = m4_scst_execute_core_tests(1, 1);`
…
`result = m4_scst_execute_core_tests(43, 43);`
	1067,83 (1504,72)[3]

[1] This is a stack used by the m4_scst_execute_core_tests() function for which assembly commands are generated by compiler as it is written in C.
[2] This is maximum stack value from Table 9.
[3] Measured execution time for devices which do not support HSRUN mode.

4 Integration of the SCST Library to a User Application

4.1 SCST Library-Related Files

The SCST library consists of three parts:

- Set of atomic tests achieving the claimed fault coverage, and
- SCST library-specific interrupt vector table and Interrupt service routines ISR to catch and process expected for the SCST Library interrupts and exceptions, and catch and redirect to application / OS those interrupts and exceptions, occurrence of which was unexpected for the SCST Library;
- Test Shell (TS) library providing run-time interface to the SCST library.

The following subchapters provide an overview of the related to each part files, names of which and relative paths can be added by a user to project.

4.1.1 Core Tests

The SCST Library provides a set of tests which summarily achieve the claimed diagnostic coverage. Table 3 provides an overview of all files that belong to the atomic tests.

Table 3. Core Tests Related Files

File	Description
\SCST\src\asm\core_tests*	Complete set of core tests achieving claimed diagnostic coverage.
\SCST\src\asm\m4_scst_lib.s	Code and data commonly used by two and more core tests.

4.1.2 Interrupt Vector Table and SCST ISRs

The SCST library includes and allocates its own interrupt vector table and provides a complete set of ISRs to catch all interrupts and exceptions. The Table 4 below provides an overview of the related source files.

Table 4. SCST Interrupt Vector Table and ISRs Related Files

File	Description
\SCST\src\asm\m4_scst_exception_wrappers.s	Code of ISR wrappers.
\SCST\src\asm\m4_scst_exception_lib.s	Commonly used functions by more and two exception processing related tests.
\SCST\src\asm\m4_scst_vector_table.s	SCST library specific interrupt vector table.

4.1.3 TS SCST Library

The TS library provides interface through which the core test functions are accessed.

Table 5 lists all files that belong to the TS SCST Library.

Table 5. Content of the TS SCST Library

File	Description
\SCST\src\c\m4_scst_test_shell.c	C-code of the test shell.
\SCST\src\c\m4_scst_data.c	Array of core test descriptors.
\SCST\src\h\m4_scst_data.h	Type definition for core test descriptor.
\SCST\src\h\m4_scst_test_shell.h	Header file of the test shell contains function prototypes and possible return values.
\SCST\src\h\m4_scst_configuration.h	Contains definitions for possible configurations of the SCST Library.
\SCST\src\h\m4_scst_typedefs.h	Type definitions.
\SCST\src\h\m4_scst_compiler.h	Contains compiler abstraction macros.

4.2 Handling of Interrupts

The SCST library includes set of test checking correct functionality of exception logic within the core. The tests intentionally provoke different exceptions and observe whether they are correctly taken, correctly processed, and correct return from a test-specific interrupt handler takes place.

There is another set of tests, which destroy special purpose, configuration, and control registers in the process of their testing. If application / OS specific interrupt is taken, it might be processed incorrectly. Before application / OS ISR gets control, content of destroyed special purpose registers has to be restored.

To support execution of these types of tests, the SCST library includes its own interrupt vector table. When one of such tests is to be executed, the SCST library replaces the application / OS vector tables after which all the interrupts directed to the SCST specific one. The SCST specific ISRs first recognize whether exception condition was triggered by the SCST library and if not, determines matching application ISR, and with the documented in Table 9 latency passes execution control to it.

SCST library fully supports nested interrupts. It correctly restores all the affected registers and passes control to matching application / OS ISR independently on the number of pending interrupts. After the first interrupt request reaches the application / OS ISR, it is safe to assume (in hardware-fault-free case only) that application / OS interrupt vector table was already restored back, so all interrupts would hit application / OS ISRs with no additional delay.

When execution of the test is complete, the SCST library also restores original application / OS vector table.

SCST library exception handlers are located in the custom section `.m4_scst_exception_wrappers`. The SCST library vector table is located in section `.m4_scst_vector_table`.

Application / OS ISRs shall be implemented in their “usual” way. They can rely on the content of related registers and corresponding stack frame. Execution of ISRs shall be terminated identically to the case when they are invoked directly upon taken interrupt.

4.3 Compilation of Assembly Files

Material in this chapter is applicable only in the case when the SCST Library is delivered for evaluation purpose in a form of source code.

Some exception tests requires M4_DEVICE_RESERVED_ADDR preprocessor macro to be defined. This macro is defined directly by the compiler and its value must be in the range of the device reserved address space.

Warning: To avoid potential problems with a cache memory error, e.g. the Processor Code bus (PC) Tag parity error, selected M4_DEVICE_RESERVED_ADDR shall not be linked with any LMEM region. It is also highly recommended to use upper addresses, e.g. addresses in rage 0x40080000 – 0x400FEFFF which are accessed through the Processor System (PS) bus, so there is no risk for the Processor Code bus (PC) Tag parity error.

```
-DM4_DEVICE_RESERVED_ADDR=0x40080000
```

Most of the assembly files contain C directives. Therefore, before assembling, their code has to be preprocessed by the C compiler.
When GreenHills compiler is used, the following directive shall be used:

```
-preprocess_assembly_files
```

When GCC compiler is used, the following directive shall be used:

```
-x assembler-with-cpp
```

4.4 Allocation of Custom Sections in the Linker File

All the code, constants, variables and data structures of the SCST library are placed into the custom sections in the source and assembly code. The Table 6 provides an overview of all custom sections defined with the SCST library. It is a responsibility of a user to allocate these sections correctly in the memory of the device by referencing them in linker command file.

Table 6. Overview of Custom Sections defined within SCST Library

Custom section name	Target memory	Description
`.m4_scst_test_code`	Flash, or RAM when executed out of RAM	This section contains object code of all core self tests that are to be invoked in the privileged mode.
`.m4_scst_test_code_unprivileged`	Flash, or RAM when executed out of RAM	This section contains object code of all core self tests that are to be invoked in the unprivileged mode.
`.m4_scst_test_code1_unprivileged`	Flash, or RAM when executed out of RAM	This region is accessed by the branch test and it contains part of the object code of the branch test. It can be located at any address depending on the available memory and motivation for testing specific target addresses but maximum allowed distance between this section and the `m4_scst_test_code_unprivileged` section cannot be higher than approximately 16Mbytes.
`.m4_scst_test_shell_code`	Flash, or RAM when executed out of RAM	This section contains object code of test shell produced from the C code.
`.m4_scst_test_shell_data`	Initialized RAM data, little endianness	Contains all global variables provided by SCST library.
`.m4_scst_rom_data`	Flash, or RAM when copied to RAM	Contains reference signatures of the tests as well as their start addresses.
`.m4_scst_exception_wrappers`	Flash, or RAM when copied to RAM	Contains SCST library-specific ISR-wrappers.
`.m4_scst_vector_table`	Flash, or RAM when copied to RAM	Contains SCST library-specific vector table.
`.m4_scst_ram_data`	RAM data, little endianness	Contains data structures of all core self tests that are invoked in the privileged mode.
`.m4_scst_ram_data_target0` `.m4_scst_ram_data_target1`	RAM, little endianness	This region is accessed by the load/store tests. Can be located at any address depending on the available memory and motivation for testing specific target addresses.
`.m4_scst_ram_test_code`	RAM code, little endianness	This region is used for code which is executed by the fetch test. Can be located at any address depending on the available memory.

Table 7 provides information on the size of every custom section of the SCST Library depending on the package configuration.
Note that measurements was done for one randomly selected compiler with compiler options listed in Chapter 1.2.

Table 7. SCST Library Custom Section Size

Custom Section Name	Custom Section Size (bytes)
.m4_scst_test_code	10592
.m4_scst_test_code_unprivileged	47792
.m4_scst_test_code1_unprivileged	772
.m4_scst_test_shell_code	278
.m4_scst_test_shell_data	20
.m4_scst_rom_data	352
.m4_scst_exception_wrappers	188
.m4_scst_vector_table	768
.m4_scst_ram_data	72
.m4_scst_ram_data_target0	52
.m4_scst_ram_data_target1	52
.m4_scst_ram_test_code	24

5 Interaction between Application and SCST Library

5.1 Interface to the SCST Library

For using the SCST library, the follwoing header file has to be included into the customer application:

`m4_scst_test_shell.h`

During normal operation, a user application shall interact with the SCST library by calling the `m4_scst_execute_core_tests()` function, provided by the test shell, every time when execution of core self tests is required. The function accepts two arguments for specification of the range of tests to execute. The function executes one test after another – as long as no test fails or interrupted – and generates a 32-bit value as an execution result of the requested tests. Chapter 8.1 provides detailed specification of this function.

The test execution order and invocation time is fully determined by application.

`m4_scst_execute_core_tests()` function shall be invoked in the mode corresponding to the mode of the requested for execution core tests (required execution mode for every core test is apecified in Table 10). SCST library returns control to application in the same mode as it was invoked.

The SCST library restores initial content of all the destroyed dedicated, special purpose, control, and configuration registers before it returns execution control to application.

5.2 Stack Usage by SCST Library

The SCST library uses application stack. First, the `m4_scst_execute_core_tests()` function may use the stack according to the generated assembly code by a C-compiler (since it is written in C). Estimated stack usage is included in the number provided in Table 1.

Stack also is used in addition within each core test individually. Consumed stack size for every core test is specified in Table 9.

5.3 Test Shell Result Processing

The test shell checks an execution result of each individual test even if multiple tests were requested for execution within a single invocation of the `m4_scst_execute_core_tests()` function. A result of each test represents a 32-bit signature value. If the returned by core test value does not match an expected value, the test shell completes its execution and returns the incorrect execution result of the failed test. The number of the failed atomic test is contained in the

m4_scst_last_executed_test_number global variable.

The test shell maintains a combined 32bit signature value, which is updated upon successful completion of each atomic test. The algorithm for calculation of this signature represents an XOR operation over the 32bit signatures of all atomic tests requested for execution. If all requested atomic tests were executed and passed, the m4_scst_execute_core_tests() function returns this combined signature to the user application. A user application, based on the numbers of the executed tests and their expected signatures, shall also calculate the combined signature and compare it to the returned value. If both values are identical, the call to the m4_scst_execute_core_tests() function was successful and all the requested tests passed.

If unexpected for the SCST Library interrupt occurs during its execution, a non-zero value is stored to the global variable m4_scst_test_was_interrupted. In this case, the SCST library returns control after completion of execution of the interrupted test to the application. This is done to allow application requesting an execution of the same test again. The return value from the m4_scst_execute_core_tests() function would be M4_SCST_TEST_WAS_INTERRUPTED. Signature of the interrupted test can be incorrect. Application needs to analyze a content of this global variable first, and only in case it indicates uninterrupted execution of all the requested tests, analyze a returned signature.

For more details, refer to the specification of the m4_scst_execute_core_tests() function. The test shell may also return values notifying about the errors in its invocation or execution. Possible return values are listed in Chapter 8.1.

In some faulty cases, application needs to expect triggering of “illegal opcode” exception by the SCST Library. It is done, for example, in the places where incorrect execution of branch commands is detected.

SCST library provides a redundant test execution result returning path to application in a form of m4_scst_accumulated_signature global variable, that can be accessible also from other core within a multi-code device. Possible content of this variable and meaning of the result is the same as possible return results of the m4_scst_execute_core_tests() function.

5.4 Incorrect Execution Control Flow

The code of the SCST library includes safety measure for recognition incorrect passing of execution control to its code. If this is the case, the safety measure intentionally triggers the “illegal opcode” exception by executing illegal opcode.

5.5 Error Injection into Core Test Execution Result

For the purpose of testing reaction within application on a return of incorrect test execution result, the SCST library provides possibility to inject error into execution of the core test. For this purpose, the library provides the m4_scst_fault_inject_test_index global variable that can be set within application to an index of the core test, in execution result of which an error must be injected. The m4_scst_execute_core_tests() function when gets control, compares an index of the core test it executes next with the index stored in this

global variable. In case of a match, it sets content of other – internal – variable to a non-zero value. The content of this internal variable is XOR-ed to an actual result of executed test.

Error injection affects a result returned by the `m4_scst_execute_core_tests()` function, as well as a result stored in the `m4_scst_accumulated_signature` global variable. Error injection is not applied in case test execution is interrupted.

6 Type Specification

The table below specifies the integer data types used in the self-test code. These data types are defined in the \SCST\src\h\m4_scst_typedefs.h file.

Table 8. Integer Data Types

Data Type	Specification
`m4_scst_int8_t`	signed 8-bit integer
`m4_scst_uint8_t`	unsigned 8-bit integer
`m4_scst_vint8_t`	volatile signed 8-bit integer
`m4_scst_vuint8_t`	volatile unsigned 8-bit integer
`m4_scst_int16_t`	signed 16-bit integer
`m4_scst_uint16_t`	unsigned 16-bit integer
`m4_scst_vint16_t`	volatile signed 16-bit integer
`m4_scst_vuint16_t`	volatile unsigned 16-bit integer
`m4_scst_int32_t`	signed 32-bit integer
`m4_scst_uint32_t`	unsigned 32-bit integer
`m4_scst_vint32_t`	volatile signed 32-bit integer
`m4_scst_vuint32_t`	volatile unsigned 32-bit integer

7 Global Variables

7.1 m4_scst_accumulated_signature

Variable Specification:

```
m4_scst_uint32_t m4_scst_accumulated_signature;
```

Description:

This global variable represents a redundant path of test execution result propagation to the user application comparing to the return value from the `m4_scst_execute_core_tests()` function. In hardware-fault-free case possible values of this variable are exactly the same as possible return values of the `m4_scst_execute_core_tests()` function. This variable can be used by application running on one core for determining core tests execution status running on a different core within the same multi-core device.

The `m4_scst_accumulated_signature` variable is located in the `.m4_scst_test_shell_data` custom section.

7.2 m4_scst_last_executed_test_number

Variable Specification:

```
m4_scst_uint32_t m4_scst_last_executed_test_number;
```

Description:

This global variable contains the number of the atomic test being executed. It is set to a corresponding index of the test before its execution. When test fails or its execution is interrupted with unexpected interrupt, this variable can be used for determining the number of the failed or interrupted test correspondingly. If no test failed and interrupted, and the test shell returns control to the user application, this variable contains the value equal to the value passed to the `end` parameter of the `m4_scst_execute_core_tests()` function.

The `m4_scst_last_executed_test_number` variable is located in the `.m4_scst_test_shell_data` custom section.

7.3 m4_scst_test_was_interrupted

Variable Specification:

```
m4_scst_uint32_t m4_scst_test_was_interrupted;
```

Description:

This global variable is set to a non-zero value in case of interruption of core test execution. If multiple tests were requested for execution, the `m4_scst_last_executed_test_number` variable can be used for determining index of the interrupted test, since the `m4_scst_execute_core_tests()` function does not execute remaining tests in this case.

The `m4_scst_test_was_interrupted` variable is located in the `.m4_scst_test_shell_data` custom section.

8 Function Specification

8.1 m4_scst_execute_core_tests

Call:

```
m4_scst_uint32_t  m4_scst_execute_core_tests(m4_scst_uint32_t  start,
m4_scst_uint32_t end);
```

Arguments:

`start`	in	Index of the first core test to execute. See Table 9 column “Test Index”.
`end`	in	Index of the last atomic test to execute. See Table 9, column “Test Index”.

Description: This function executes all core tests whose numbers are greater or equal to `start` and lower or equal to `end`. The function performs also various checks which may result in error return values listed below. The corresponding constants are defined in the file m4_scst_test_shell.h.

If execution control was passed to this function incorrectly, the function intentionally triggers an exception by executing illegal opcode.

Returns:

Value	Description
`M4_SCST_WRONG_RANGE`	The values retrieved from the `start` and `end` parameters are incorrect.
`M4_SCST_TEST_WAS_INTERRUPTED`	Execution of core test was interrupted.
<signature of the failed test>	The function returns actual (faulty) result of the failed atomic test in a faulty case.
<combined signature of all executed and passed tests>	The function returns combined signature of all executed atomic tests when no failed test detected.

Example:

```
m4_scst_uint32_t result;

/* Request execution of the tests with indices 2, 3, and 4 */
result = m4_scst_execute_core_tests (2, 4);

/* Here comes the code, which analyses:
- Content of global “m4_scst_test_was_interrupted” variable;
- Content of local “result” variable;
- Content of global “m4_scst_last_executed_test_number”
variable (should be 4 in the given example in fault-free
case).
*/
```

9 Core Tests Specification

For each core test, Table 9 provides information on a corresponding index to be passed to the `start` and `end` parameters of the `m4_scst_execute_core_tests()` function, expected test result returned in fault-free case, latency of user ISR invocation once interrupt is triggered during execution of the test, as well as overall test execution time and used stack.
Note that measurement was done for one randomly selected compiler with compiler options listed in Chapter 1.2.

If a user ISR interrupt latency is specified as 0, this means that interrupt request, if occured, directly hits a user application interrupt vector table, so no latency is introduced by the SCST Library ISR wrapper code for the given test.

Table 9. Core Test Specification

Test name	Test Index	Test Reference Signature	User ISR Invocation Latency (µs)	Test Execution Time (µs)	Used Stack Size[1] (bytes)
m4_scst_exception_test_svc	0	0x000012E0	3.6 (5.1)[3]	6.80 (10.17)[3]	72[2]
m4_scst_exception_test_pendsv	1	0x00000E7C	3.6 (5.2)[3]	6.50 (9.92)[3]	72[2]
m4_scst_exception_test_systick	2	0x000006EA	3.6 (5.2)[3]	6.53 (9.92)[3]	72[2]
m4_scst_exception_test_hard_fault1	3	0x00000D89	5.5 (7.9)[3]	16.46 (24.35)[3]	72[2]
m4_scst_exception_test_hard_fault2	4	0x00001A69	4.3 (6.3)[3]	13.7 (19.10)[3]	72[2]
m4_scst_exception_test_usage_fault	5	0x0000205A	3.9 (5.5)[3]	12.12 (17.55)[3]	72[2]
m4_scst_exception_test_mem_fault	6	0x00000C3B	3.8 (5.4)[3]	6.91 (10.15)[3]	72[2]
m4_scst_exception_test_bus_fault	7	0x00000FE3	4.1 (5.8)[3]	8.21 (12.12)[3]	72[2]
m4_scst_exception_test_nmihf	8	0x0000256C	4.0 (5.9)[3]	12.53 (18.20)[3]	96[2]
m4_scst_exception_test_tail_chain	9	0x00002784	5.0 (7.0)[3]	8.17 (11.95)[3]	72[2]
m4_scst_exception_test_masking	10	0x00002B88	4.0 (5.6)[3]	11.96 (17.42)[3]	72[2]
m4_scst_exception_test_handler_thread	11	0x0000302F	7.1 (10.2)[3]	17.0 (24.65)[3]	104[2]

[1] This is a stack used in addition to the portion used within m4_scst_execute_core_tests() function for which assembly commands are generated by compiler as it is written in C.
[2] Includes stack required by the exeptions which are intentionally triggered during tests. In case of tests are called with SP_process configured this values will be distributed between both SP_process, SP_main stacks, where 36 bytes will be stored to the SP_process and the rest to the SP_main.
[3] Measured for devices which do not support HSRUN mode.

Test name	Test Index	Test Reference Signature	User ISR Invocation Latency (µs)	Test Execution Time (µs)	Used Stack Size[1] (bytes)
m4_scst_regbank_test4	12	0x000047DB	3.0 (4.4)[3]	7.41 (10.87)[3]	36[2]
m4_scst_alu_test7	13	0x00001C21	3.9 (5.5)[3]	7.10 (10.52)[3]	72[2]
m4_scst_branch_test3	14	0x7E256E6B	4.1 (5.9)[3]	14.58 (21.30)[3]	72[2]
m4_scst_status_test3	15	0x00003A0E	3.9 (5.6)[3]	7.28 (10.95)[3]	72[2]
m4_scst_regbank_test6	16	0xB91BDE95	1.1 (1.8)[3]	27.10 (38.12)[3]	52
m4_scst_fetch_test	17	0xB14D3A0D	1.0 (1.7)[3]	3.39 (5.00)[3]	52
m4_scst_loadstore_test6	18	0xC19005EE	0.9 (1.6)[3]	53.39 (75.50)[3]	64
m4_scst_loadstore_test1	19	0xC63C044C	0	30.89 (41.25)[3]	76
m4_scst_loadstore_test2	20	0xAE1D1D83	0	43.96 (61.12)[3]	60
m4_scst_loadstore_test3	21	0x5B80787E	0	31.75 (40.85)[3]	60
m4_scst_loadstore_test4	22	0x618271EE	0	34.85 (48.52)[3]	92
m4_scst_loadstore_test5	23	0xF61A1F74	0	106.30 (149.12	60
m4_scst_regbank_test1	24	0xE35DF821	0	56.50 (82.70)[3]	52
m4_scst_regbank_test2	25	0x28D6AF31	0	57.75 (75.27)[3]	52
m4_scst_regbank_test3	26	0xC726A3D2	0	64.12 (90.70)[3]	52
m4_scst_regbank_test5	27	0x00001AC7	0	3.62 (5.37)[3]	52
m4_scst_mac_test1	28	0x0000A828	0	15.12 (22.50)[3]	52
m4_scst_mac_test2	29	0x00011B11	0	14.85 (21.10)[3]	52
m4_scst_alu_test1	30	0x00002C09	0	9.70 (12.70)[3]	52
m4_scst_alu_test2	31	0x0000ADEF	0	35.85 (50.50)[3]	52
m4_scst_alu_test3	32	0x00003F51	0	10.78 (15.82)[3]	52
m4_scst_alu_test4	33	0x00008C65	0	120.41 (168.90) [3]	52
m4_scst_alu_test5	34	0x00006E5A	0	7.57 (10.77) [3]	52

Test name	Test Index	Test Reference Signature	User ISR Invocation Latency (µs)	Test Execution Time (µs)	Used Stack Size[1] (bytes)
m4_scst_alu_test6	35	0x00002022	0	9.33 (13.50)[3]	52
m4_scst_simdsat_test1	36	0x1C464F74	0	30.91 (42.30)[3]	52
m4_scst_simdsat_test2	37	0x78C61FF0	0	20.32 (28.75)[3]	52
m4_scst_simdsat_test3	38	0x24CA596B	0	29.91 (41.37)[3]	52
m4_scst_simdsat_test4	39	0xC4527367	0	23.57 (33.37)[3]	52
m4_scst_branch_test1	40	0x38A82153	0	12.64 (18.60)[3]	60
m4_scst_branch_test2	41	0x53273023	0	16.37 (24.60)[3]	52
m4_scst_status_test1	42	0x17A3FF6A	0	26.67 (37.65)[3]	52
m4_scst_status_test2	43	0x75DB236C	0	8.14 (11.70)[3]	52

Table 10 provides information on the M4 core mode the function `m4_scst_execute_core_tests()` has to be invoked in when a corersponding core test is requested for execution.

Table 10. Core Test Invocation Mode

Test Index	Mode the function `m4_scst_execute_core_tests()` has to be invoked in when execution of corresponding test is requested			Test invocation constraints
	Thread Privileged	Thread Unprivileged	Handler	
0	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
1	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
2	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
3	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
4	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
5	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
6	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
7	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
8	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
9	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
10	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
11	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
12	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
13	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
14	Yes	No	No	This test enables interrupts and fault handlers for testing purposes.
15	Yes	No	No	None
16	Yes	No	No	None
17	Yes	Yes	Yes	None
18	Yes	Yes	Yes	None
19	Yes	Yes	Yes	None
20	Yes	Yes	Yes	None
21	Yes	Yes	Yes	None

Test Index	Mode the function `m4_scst_execute_core_tests()` has to be invoked in when execution of corresponding test is requested			Test invocation constraints
	Thread Privileged	Thread Unprivileged	Handler	
22	Yes	Yes	Yes	None
23	Yes	Yes	Yes	None
24	Yes	Yes	Yes	None
25	Yes	Yes	Yes	None
26	Yes	Yes	Yes	None
27	Yes	Yes	Yes	None
28	Yes	Yes	Yes	None
29	Yes	Yes	Yes	None
30	Yes	Yes	Yes	None
31	Yes	Yes	Yes	None
32	Yes	Yes	Yes	None
33	Yes	Yes	Yes	None
34	Yes	Yes	Yes	None
35	Yes	Yes	Yes	None
36	Yes	Yes	Yes	None
37	Yes	Yes	Yes	None
38	Yes	Yes	Yes	None
39	Yes	Yes	Yes	None
40	Yes	Yes	Yes	None
41	Yes	Yes	Yes	None
42	Yes	Yes	Yes	None
43	Yes	Yes	Yes	None

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support