User Manual

for the S32K118 Cortex M0+ SCST Library

Document Number: S32K118M0SCSTRTM1.0.1
Rev. 1.4, 02/2020

Contents

Section number **Title** **Page**

Chapter 7
Global Variables

Chapter 8
Function Specification

Chapter 9
Core Test Specification

Chapter 1
Introduction

This document contains information and description of the interface to the Structural Core Self Test Library (SCST) for Cortex-M0+ 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 Acronyms and Definitions

Table 1-1. Acronyms and Definitions

Term	Definition
ALU	Arithmetic Logic Unit
ASM	Assembler
HW	Hardware
ISR	Interrupt Service Routine
LPIT	Low Power Interrupt Timer
MCU	Micro Controller Unit
N/A	Not Applicable
OS	Operating System
RAM	Random Access Memory (Static or Dynamic)
ROM	Read Only Memory
SCST	Structural Core Self Test
SPF	Single-Point Fault
SW	Software
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.

Measurements of execution time and user ISR invocation latency were done on the SCST Library object code executed on the target device. In the process of measurements of execution time and ISR latency, the Cortex-M0+ core was forced to run at 48MHz. LPIT timer was used for these measurements. The timer was forced to run at 48MHz. 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 48 000 000 to obtain an execution time of the measured portion of code in seconds.

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.

NOTE

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.

Chapter 2
Features

This chapter summarizes main features of the M0+ SCST Library software product.

2.1 Software Design Summary

- 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 SPF 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, load/store, and other execution units;
    - Instruction decoder, 16-Bit, 32-Bit;
- Interrupts can be enabled during execution of all tests. SCST library provides its own interrupt vector table and wrappers for exception testing purposes. During exception testing the SCST Library disables all external interrupts. The interrupt processing latency introduced by the SCST library is specified in the Table 9-1. Exceptions which occur during testing and are detected as unexpected for the SCST library are forwarded to the corresponding exception handler of the OS / user application.

Chapter 3
Characteristics of the SCST Library

3.1 Coverage

Details regarding the assessment of the coverage are contained in the document *M0_S32K118_SCST_Library_FaultCoverage_Estimation.xlsx* which is part of the safety package.

3.2 Memory Consumption

The memory consumption of the SCST Library is specified in the Table 3-1 table. Note that measurement was done for one randomly selected compiler with compiler options listed in Compiler Settings and Measurement Conditions.

Table 3-1. Memory usage by the SCST Library

Stack (Bytes)	RAM Total (Bytes)	Flash Total (Bytes)
88	152	5576

3.3 Execution Time

The SCST Library execution time is specified in Table 3-2 depending on the test invocation scenario.

Execution time of every individual test as well as user ISR invocation latency is specified in Table 9-1.

Table 3-2. Memory usage by the SCST Library

Test Invocation Scenario	Accumulated Tests Execution Time (us @ 48 MHz)
All atomic tests are invoked through a minimum calls to the test shell: `result=m0_scst_execute_core_tests(0, 18);`	306,41

Every atomic test is invoked through a separate call to the test shell:
 	
`result=m0_scst_execute_core_tests(0, 0);`
`result=m0_scst_execute_core_tests(1, 1);`
`...`
`result=m0_scst_execute_core_tests(18, 18);`
	343,60

Chapter 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 exceptions, and catch and redirect to application / OS those 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 4-1 provides an overview of all files that belong to the atomic tests.

Table 4-1. Core Tests Related Files

File	Description
\SCST\src\asm\core_tests.s*	Complete set of core tests achieving claimed diagnostic coverage.
\SCST\src\asm\m0_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. Table 4-2 below provides an overview of the related source files.

Table 4-2. SCST Interrupt Vector Table and ISRs Related Files

File	Description
\SCST\src\asm\m0_scst_exception_wrappers.s	Code of ISR wrappers.
\SCST\src\asm\m0_scst_exception_lib.s	Commonly used functions by more and two exception processing related tests.
\SCST\src\asm\m0_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 4-3 lists all files that belong to the TS SCST Library.

Table 4-3. Content of the TS SCST Library

File	Description
\SCST\src\c\m0_scst_test_shell.c	C-code of the test shell.
\SCST\src\c\m0_scst_data.c	Array of core test descriptors.
\SCST\src\h\m0_scst_data.h	Type definition for core test descriptor.
\SCST\src\h\m0_scst_test_shell.h	Header file of the test shell contains function prototypes and possible return values.
\SCST\src\h\m0_scst_configuration.h	Contains definitions for possible configurations of the SCST Library.
\SCST\src\h\m0_scst_typedefs.h	Types definitions.
\SCST\src\h\m0_scst_compiler.h	Contains compiler abstraction macros.

4.2 Handling of Interrupts

The SCST library includes set of tests checking correct functionality of exception logic within the core. These tests use SCST library-specific interrupt vector table and intentionally provoke different exceptions and observe whether they are correctly taken, correctly processed, and a correct return from the test-specific exception handler takes place. When test execution is complete, the SCST library also restores original application / OS vector table.

There is another set of tests, which destroy special purpose, configuration, and control registers in the process of their testing.

To support execution of these types of tests, the SCST library disables the user application interrupt processing before destroying the content of dedicated registers. To allow application interrupt processing, each test contains several “windows” in which the dedicated registers, including those enabling the interrupts, are temporally restored. The interrupt processing latency introduced by the SCST library is specified in the Table 9-1.

SCST library exception handlers are located in the custom section `.m0_scst_test_code`. The SCST library vector table is located in section `.m0_scst_vector_table`.

4.3 Compilation of Assembly Files

Most of the assembly files contain C directives. Therefore, before assembling, their code has to be preprocessed by the C compiler.

When IAR compiler is used, preprocessing is done automatically, no compiler option is needed.

When GreenHills compiler is used, the following option shall be used:

- *-preprocess_assembly_files*

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

- *-x assembler-with-cpp*

When WindRiver Diab compiler is used, the following options shall be used:

- *-Xpreprocess-assembly*
- *-Xalign-power2*

NOTE

If not used, the *-Xalign-power2* option is the default for WindRiver Diab compiler.

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 4-4 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 4-4. Overview of Custom Sections defined within SCST Library

Custom section name	Target memory	Description
.m0_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.
.m0_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.
.m0_scst_vector_table	Flash, or RAM when copied to RAM	Contains SCST library-specific vector table.
.m0_scst_rom_data	Flash, or RAM when copied to RAM	Contains reference signatures of the tests as well as their start addresses.
.m0_scst_ram_data	RAM data, little endianness	Contains all global variables and data structures of the SCST Library.
.m0_scst_test_shell_data	Initialized RAM data, little endianness	Contains all global variables provided by SCST library.
.m0_scst_ram_data_target0	RAM data, 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.
.m0_scst_ram_data_target1	RAM data, 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.
.m0_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 4-5 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 Compiler Settings and Measurement Conditions.

Table 4-5. SCST Library Custom Section Size

Custom section name	Custom section size (Bytes)
.m0_scst_test_code	5088
.m0_scst_test_shell_code	144

Table continues on the next page...

Table 4-5. SCST Library Custom Section Size (continued)

Custom section name	Custom section size (Bytes)
.m0_scst_vector_table	192
.m0_scst_rom_data	152
.m0_scst_ram_data	20
.m0_scst_test_shell_data	16
.m0_scst_ram_data_target0	52
.m0_scst_ram_data_target1	52
.m0_scst_ram_test_code	12

Chapter 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:

- *m0_scst_test_shell.h*

During normal operation, a user application shall interact with the SCST library by calling the `m0_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 – and generates a 32-bit value as an execution result of the requested tests. Chapter Function Specification provides detailed specification of this function.

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

`m0_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 9-2. 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 `m0_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 3-1.

Stack also is used in addition within each core test individually. Consumed stack size for every core test is specified in Table 9-1.

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 `m0_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 `m0_scst_last_executed_test_number` global variable.

The test shell maintains a combined 32-bit 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 32-bit signatures of all atomic tests requested for execution. If all requested atomic tests were executed and passed, the `m0_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 `m0_scst_execute_core_tests()` function was successful and all the requested tests passed.

SCST library provides a redundant test execution result returning path to application in a form of `m0_scst_accumulated_signature` global variable, that can be accessible also from other core within a multi-core device. Possible content of this variable and meaning of the result is the same as possible return results of the `m0_scst_execute_core_tests()` function.

5.4 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 `m0_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 `m0_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 `m0_scst_execute_core_tests()` function, as well as a result stored in the `m0_scst_accumulated_signature` global variable.

Chapter 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\m0_scst_typedefs.h* file.

Table 6-1. Integer SCST Data Types

Data Type	Specification
m0_scst_int8_t	signed 8-bit integer
m0_scst_uint8_t	unsigned 8-bit integer
m0_scst_vint8_t	volatile signed 8-bit integer
m0_scst_vuint8_t	volatile unsigned 8-bit integer
m0_scst_int16_t	signed 16-bit integer
m0_scst_uint16_t	unsigned 16-bit integer
m0_scst_vint16_t	volatile signed 16-bit integer
m0_scst_vuint16_t	volatile unsigned 16-bit integer
m0_scst_int32_t	signed 32-bit integer
m0_scst_uint32_t	unsigned 32-bit integer
m0_scst_vint32_t	volatile signed 32-bit integer
m0_scst_vuint32_t	volatile unsigned 32-bit integer

Chapter 7
Global Variables

This section describes all global variables provided by the M0+ SCST Library to a user application.

7.1 m0_scst_accumulated_signature

Variable Specification:

```
m0_scst_uint32_t m0_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 `m0_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 `m0_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 `m0_scst_accumulated_signature` variable is located in the `.m0_scst_test_shell_data` custom section.

7.2 m0_scst_last_executed_test_number

Variable Specification:

```
m0_scst_uint32_t m0_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, this variable can be used for determining the number of the failed test. If no test failed, 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 `m0_scst_execute_core_tests()` function.

The `m0_scst_last_executed_test_number` variable is located in the `.m0_scst_test_shell_data` custom section.

7.3 m0_scst_fault_inject_test_index

Variable Specification:

```
m0_scst_uint32_t m0_scst_fault_inject_test_index;
```

Description:

This global variable can be set by a user application to specify an index of the core test into whose result an error will be injected, see Error Injection into Core Test Execution Result chapter.

The `m0_scst_fault_inject_test_index` variable is located in the `.m0_scst_test_shell_data` custom section.

Chapter 8
Function Specification

This section contains description of the API provided by the Cortex M0+ SCST Library to a user application.

8.1 m0_scst_execute_core_tests

Function Call:

```
m0_scst_uint32_t m0_scst_execute_core_tests(m0_scst_uint32_t start, m0_scst_uint32_t end);
```

Arguments:

Name	Direction	Description
`start`	in	Index of the first atomic test to execute.
`end`	in	Index of the last atomic test to execute.

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 *m0_scst_test_shell.h*.

The `m0_scst_execute_core_tests()` function is located in the `.m0_scst_test_shell_code` custom section.

Returns:

Value	Description
`M0_SCST_WRONG_RANGE`	The values retrieved in case of the `start` or `end` parameters are incorrect.
<signature of the failed test>	The function returns actual (faulty) result of the failed atomic test in a faulty case.

Table continues on the next page...

Value	Description
<combined signature of all executed and passed tests>	The function returns combined signature of all executed atomic tests when no failed test detected.

Example:

```
m0_scst_uint32_t result;

/* Request execution of the tests with indices 2, 3, and 4 */
result = m0_scst_execute_core_tests (2, 4);

/* Here comes the code, which analyses:
 - Content of local “result” variable;
 - Content of global “m0_scst_last_executed_test_number” variable (should be 4 in the given
example in fault-free case).
 */
```

Chapter 9
Core Test Specification

For each core test, Table 9-1 provides information on a corresponding index to be passed to the `start` and `end` parameters of the `m0_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 Compiler Settings and Measurement Conditions chapter.

Table 9-1. Core Test Specification

Test Name	Test Index	Test Reference Signature	User ISR Invocation Latency (us)	Test Execution Time (us)	Used Stack Size (Bytes)
m0_scst_exception_test_svc	0	0x05D86B22	7,44	11,91	88
m0_scst_exception_test_pendsv	1	0x71DDFB5B	6,13	10,64	56
m0_scst_exception_test_systick	2	0x1A6AE17F	6,62	10,54	56
m0_scst_exception_hardfault_test1	3	0x428DE81A	4,96	22,40	56
m0_scst_exception_hardfault_test2	4	0xE27BBB8D	5,85	17,37	68
m0_scst_exception_nmi_test	5	0xC2262509	9,79	13,85	88
m0_scst_exception_priority_test	6	0x1971A0EA	7,79	18,20	56
m0_scst_loadstore_test1	7	0x6B8D4B1D	0	8,18	32
m0_scst_loadstore_test2	8	0xF3668954	2,58	10,43	44
m0_scst_loadstore_test3	9	0xD56B91A4	0	10,33	68
m0_scst_alu_test1	10	0x8F6CFDFC	2,48	11,10	32
m0_scst_alu_test2	11	0x48808A28	0	7,41	20
m0_scst_alu_test3	12	0x8E4542E0	0	88,97	20
m0_scst_alu_test4	13	0x5058B6F2	0	7,8	20
m0_scst_alu_test5	14	0xBFFAE07E	0	28,91	20
m0_scst_branch_test1	15	0x444E09B5	0	13,68	24
m0_scst_branch_test2	16	0xDB1C1FEE	2,48	11,00	36
m0_scst_regfile_test1	17	0x4674E8F8	0	22,91	52
m0_scst_regfile_test2	18	0x7EE47D44	2,90	18,95	40

NOTE

If a user ISR interrupt latency is specified as 0, this means that interrupts are not disabled within the atomic test and interrupt request, if occurred, directly hits a user application interrupt vector table, so no latency is introduced by the SCST Library code for the given atomic test.

Table 9-2. Core Test Invocation Mode

Test Index	Thread Mode (Privileged/ Unprivileged)	Handler Mode	Test invocation constraints
0	Yes/No	No	None
1	Yes/No	No	None
2	Yes/No	No	None
3	Yes/No	No	None
4	Yes/No	No	None
5	Yes/No	No	None
6	Yes/No	No	None
7	Yes/Yes	Yes	None
8	Yes/No	Yes	None
9	Yes/Yes	Yes	None
10	Yes/No	Yes	None
11	Yes/Yes	Yes	None
12	Yes/Yes	Yes	None
13	Yes/Yes	Yes	None
14	Yes/Yes	Yes	None
15	Yes/Yes	Yes	None
16	Yes/No	Yes	None
17	Yes/Yes	Yes	None
18	Yes/No	Yes	None

NOTE

It is recommended to run tests in the Thread Privileged mode, which is supported by all atomic tests. Running tests in other invocation modes is limited as it is summarized in Table 9-2.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number S32K118M0SCSTRTM1.0.1
Revision 1.4, 02/2020