Arm® Cortex®-M33 Processor STL

Revision: r0p0

User Guide

Confidential

101170_0000_01_en

Arm® Cortex®-M33 Processor STL

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0000-00	15 June 2018	Confidential	First beta release for r0p0
0000-01	17 August 2020	Confidential	First early access release for r0p0

Confidential Proprietary Notice

This document is CONFIDENTIAL and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorised by Arm to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

You acknowledge and agree that you possess the necessary expertise in functional safety and that you shall be solely responsible for compliance with all legal, regulatory and safety-related requirements concerning your products, notwithstanding any information provided herein.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: (i) for the purposes of determining whether implementations infringe any third party patents;(ii) for developing technology or products which avoid any of Arm's intellectual property; or (iii) as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or (iv) for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Contents
Arm® Cortex®-M33 Processor STL User Guide

Chapter 4

STL Out-of-Box tests

Chapter 5

Fault simulation

Appendix A

Revisions

Preface

This preface introduces the *Arm® Cortex®-M33 Processor STL User Guide*.

It contains the following:

- *About this book* on page 7.
- *Feedback* on page 10.

About this book

This book is for the Cortex®-M33 processor *Software Test Library* (STL). It explains how to integrate and implement the STL software into your system.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This guide is for experienced engineers who are responsible for software integration and verification of the Arm Software Test Library.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an overview of the Cortex-M33 processor, *Software Test Library* (STL), and the STL package deliverables.

Chapter 2 STL software integration
This chapter describes the software integration for STL implementation.

Chapter 3 Faults
This chapter introduces the concepts of faults and fault control.

Chapter 4 STL Out-of-Box tests
This chapter describes how to check if the *Software Test Library* (STL) deliverables have been set up correctly, test the STL functionality, and to fault grade the processor.

Chapter 5 Fault simulation
This chapter describes the activities required to run a fault simulation.

Appendix A Revisions
This appendix describes the technical changes between released issues of this book.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

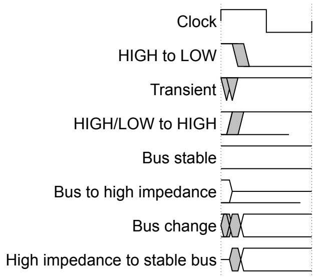


Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name, n denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm publications

- *Arm®v8-M Architecture Reference Manual* (DDI 0553)
- *Arm® Cortex®-M33 Processor Technical Reference Manual* (100230)

The following confidential documents are only available to licensees:

- *Arm® Cortex®-M33 Processor Integration and Implementation Manual* (100323)
- *Arm® Cortex®-M33 Processor STL Safety Manual* (101327)
- *Arm® Cortex®-M33 Processor STL Development Interface Report* (101328)
- *Arm® Cortex®-M33 Processor STL Release Note* (PJDOC-466751330-17543)
- *Arm®Cortex®-M33 Processor STL Failure Modes, Effects and Diagnostic Analysis Report* (PJDOC-466751330-16128)
- *Arm® Cortex®-M33 Processor STL Dependent Failure Analysis Report* (PJDOC-466751330-16127)

Other publications

- *Z01X Simulator Safety Verification User Guide*
- *Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC 61508:2010.
- *Road vehicles - Functional safety*, ISO 26262:2018

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Cortex-M33 Processor STL User Guide*.
- The number 101170_0000_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter provides an overview of the Cortex-M33 processor, *Software Test Library* (STL), and the STL package deliverables.

It contains the following sections:

1.1 About the processor

The Cortex-M33 processor is a low gate count, highly energy efficient processor that is intended for microcontroller and deeply embedded applications. The processor is based on the Armv8-M architecture and is primarily for use in environments where security is an important consideration.

The interfaces that the processor supports include:

- *Code AHB* (C-AHB) interface
- *System AHB* (S-AHB) interface
- *External PPB* (EPPB) APB interface
- *Debug AHB* (D-AHB) interface

The processor has optional:

- Arm TrustZone® technology, using the Armv8-M Security Extension supporting Secure and Non-secure states
- *Memory Protection Units* (MPUs), which you can configure to protect regions of memory
- Floating-point arithmetic functionality with support for single precision arithmetic
- Support for ETM and MTB trace

The processor is highly configurable and is intended for a wide range of high-performance, deeply embedded applications that require fast interrupt response features.

The following figure shows the processor in a typical system.

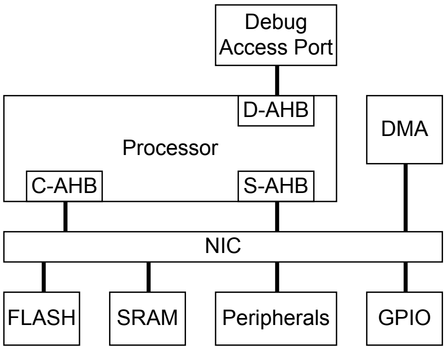


Figure 1-1 Example processor system

1.2 About STL

The *Software Test Library* (STL) provides diagnostic testing for the Cortex-M33 processor, r1p0.

The tests can be run either when the processor powers up or periodically, to detect:

- Faults that cause single points of failure.
- Latent faults caused by multiple points of failure.

The STL is not a replacement for scan-based manufacturing tests, but can be used for additional in-field testing at either boot time or runtime.

The STL deliverable contains:

- A scheduler that is written in C that determines the sequence of testing.
- A set of subroutines that tests individual functional blocks. Each subroutine is written using GNU Assembly syntax.

The STL is designed to detect faults in the functional logic, excluding memories. Any logic that is not used in a functional environment, such as debug and trace, is excluded. The fault detection is realized entirely by software which performs self-tests using a sequence of instructions. The STL tests can detect permanent stuck-at faults in logic. However, they are not suited to detect transient errors.

1.3 STL deliverables

The STL package contains the following deliverables:

- Software that includes the source code of a library of test routines written in C and assembly code. It includes:
  - — A scheduler written in C that enables you to specify the number of parts to be executed.
  - — A set of subroutines that tests individual functional blocks. Each subroutine is written using GNU Assembly syntax.
- Documentation that includes:
  - — This user guide, the *Arm® Cortex®-M33 Processor STL User Guide*, which describes the software integration of the STL, and the *Out-of-Box* (OoB) tests.
  - — The *Arm® Cortex®-M33 Processor STL Safety Manual*, which provides a summary of results after fault grading the Cortex-M33 processor, and the *Assumptions of Use* (AoU) to be considered when using the STL.
  - — The *Arm® Cortex®-M33 Processor STL Development Interface Report*, which describes the activities conducted by Arm related to the development activities of the Cortex-M33 processor STL deliverable in the context of the ISO 26262:2018 framework.

Note

A complete list of the deliverables is provided in the *Arm® Cortex®-M33 Processor STL Release Note*.

1.4 STL tool resources

Various tools are used to develop the *Software Test Library* (STL).

The following table shows the tools that are used.

Table 1-1 Tool resources

Purpose	Vendor	Tool	<simulator>[a] tool specifier
HDL simulator	Synopsys®	VCS®	`vcs`
		Z01X™	`zoix`
	Mentor Graphics®	Questa® SIM	`mti`
Software development tools	-	Arm Compiler 6.6.2	-
	-	Python 2.7	-

Note

See the *Arm® Cortex®-M33 Processor STL Release Note* for the specific tool versions that are required.

[a] `<simulator>` is used as part of a command to run your chosen simulator.

1.5 Hardware methods

The STL can be configured to execute using one of the two following status reporting methods:

- Method M0. The STL execution status is reported at a configurable memory location.
- Method M1. An SBIST controller (SBISTC), provided as part of the Cortex-M33 processor package, is integrated into the system. The STL execution status is reported in the memory-mapped registers which are part of the SBISTC.

Note

A detailed description of the SBISTC specification and integration can be found in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

The SBIST parameter in the processor package needs to be set to 1 to enable Method M1.

Chapter 2
STL software integration

This chapter describes the software integration for STL implementation.

It contains the following sections:

2.1 STL Assumptions of Use

Arm recommends that the system integrator and system developer take the relevant steps to ensure that the *Software Test Library* (STL) software meets the requirements of their application. This can be done by ensuring that the design is in line with the Assumptions of Use.

Refer to *STL Assumptions of Use* in the *Arm® Cortex®-M33 Processor STL Safety Manual* for more information.

2.2 STL terminology

This section explains the *Software Test Library* (STL) terminology.

A typical STL file has the format `stl_<block>_<optional_testname>_p<part>_n<iter>.S`, and the following table describes terms used.

Table 2-1 Terminology

Term	Description
`<block>`	The block that is being tested. This is one of the following: • Core • *Floating-Point Unit* (FPU) • *Memory-Protection Unit* (MPU) • *Nested Vector Interrupt Controller* (NVIC)
`<optional_testname>`	A short description of the test.
`<part>`	The routine that is testing a part of the block. The `<part>` field must be an integer in the range 1-255, for example 004. In this document, a part is also referred to as a module. See *2.3.2 STL architectural overview* on page 2-22 for more information.
`<iter>`	The number of iterations that must be completed to test the part.

2.3 STL software

The main components of the *Software Test Library* (STL) are:

- A range of STL parts that test individual functions of the processor. These are subroutines which test one of the following blocks:
  - core
  - FPU
  - MPU
  - NVIC
- An STL scheduler, which controls how many STL parts are run, when they are run, and reports the diagnostic results.

Note

The STL software is implemented with the Armv8-M instruction set and a C-code interface.

2.3.1 STL file descriptions

This section describes the main files of the *Software Test Library* (STL) software.

The numbers of STL parts available for testing are:

- 9 parts for testing the core
- 22 parts for testing the *Floating-Point Unit* (FPU)
- 9 parts for testing the *Memory-Protection Unit* (MPU)
- 21 parts for testing the *Nested Vector Interrupt Controller* (NVIC)

The following table describes the content and function of the main STL files.

Table 2-2 STL file descriptions

File name	Description
`stl_common_macros.S`	Contains common macros used by STL tests or common functions
`stl_common_mpu_macros.S`	Contains common macros used by MPU STL tests
`stl_common_nvic_macros.S`	Contains common macros used by NVIC STL tests
`stl_core_defs.h`	Contains common macros used by CORE STL tests
`stl_defs.h`	Contains common macros used by STL tests
`stl_fpu_defs.h`	Contains common macros used by FPU STL tests
`stl_global_defs.h`	Contains common global macros used by STL
`stl_global_macros.h`	Contains common global macros used by scatter files
`stl_mpu_defs.h`	Contains common macros used by MPU STL tests
`stl_nvic_defs.h`	Contains common macros used by NVIC STL tests
`stl_common_core_func.S`	Contains common functions used by CORE STL tests
`stl_common_data.S`	Contains data sections defining heap/scratch
`stl_common_data_ns.S`	Contains Non-secure data sections defining heap and global variables
`stl_common_fpu_func.S`	Contains common functions used by FPU STL tests
`stl_common_func.S`	Contains common functions used by STL tests
`stl_common_mpu_func.S`	Contains common functions used by MPU STL tests

Table 2-2 STL file descriptions (continued)

File name	Description
`stl_common_nvic_func.S`	Contains common functions used by NVIC STL tests
`stl_common_nvic_func_intr.S`	Contains common functions used by NVIC STL interrupt based tests
`stl_core_handler_funcs.S`	Contains exception handlers used by CORE STL tests
`stl_fpu_handler_funcs.S`	Contains exception handlers used by FPU STL tests
`stl_global_funcs.S`	Contains user modifiable functions used by STL tests
`stl_mpu_data.S`	Contains data sections used by MPU STL tests
`stl_mpu_data_ns.S`	Contains Non-secure data sections used by MPU STL tests
`stl_mpu_handler_funcs.S`	Contains exception handlers used by MPU STL tests
`stl_nvic_data.S`	Contains data sections used by NVIC STL tests
`stl_nvic_funcs_nsc.S`	Contains Non-secure callable functions used by NVIC STL tests
`stl_nvic_handler_funcs.S`	Contains exception handlers used by NVIC STL tests
`stl_nvic_handler_funcs_ns.S`	Contains Non-secure exception handlers used by NVIC STL tests
`stl_system_data.S`	Contains data-sections defining main and process stacks used by boot/initialization code
`stl_system_data_ns.S`	Contains Non-secure data-sections defining Non-secure main and process stacks used by boot/initialization code
`stl_system_init.S`	Initialization functions used by one of the following: • stl_core.S • stl_mpu.S • stl_nvic.S • stl_fpu.S • stl_test_suite
`stl_system_init_ns.S`	Non-secure initialization functions used by one of the following: • stl_core.S • stl_mpu.S • stl_nvic.S • stl_fpu.S • stl_test_suite
`stl_vectors.S`	Secure vector-table used by STL tests
`stl_vectors_ns.S`	Non-secure vector table used by STL tests
`stl_core.S`	Contains basic framework needed for running all CORE STL tests
`stl_core_save.S`	Contains save-restore functions used by CORE STL tests
`stl_core_restore.S`	
`stl_core_*_p*_n*.S`	Contains CORE STL tests
`stl_fpu.S`	Contains basic framework needed for running all FPU STL tests
`stl_fpu_save.S`	Contains save-restore functions used by FPU STL tests
`stl_fpu_restore.S`	
`stl_fpu_*_p*_n*.S`	Contains FPU STL tests
`stl_mpu.S`	Contains basic framework needed for running all MPU STL tests

Table 2-2 STL file descriptions (continued)

File name	Description
`stl_mpu_save.S`	Contains save-restore functions used by MPU STL tests
`stl_mpu_restore.S`	
`stl_mpu_p*_n*.S`	Contains MPU STL tests
`stl_nvic.S`	Contains basic framework needed for running all NVIC STL tests
`stl_nvic_save.S`	Contains save-restore functions used by NVIC STL tests
`stl_mpu_restore.S`	
`stl_nvic_*_p*_n*.S`	Contains NVIC STL tests
`stl_nvic_*_ns.S`	Contains Non-secure code or data sections used by NVIC STL tests
`stl_scheduler.c`	Contains STL-scheduler API
`stl_scheduler.h`	Contains macros, type-defines and prototypes used by STL-scheduler
`stl_scheduler_asm.S`	Contains assembly functions used by STL-scheduler
`stl_scheduler_asm.h`	Contains macros used by assembly functions in STL-scheduler
`stl_scheduler_def.h`	Contains save-restore macros, type-defines and prototypes used by STL-scheduler
`stl_scheduler_dfa.c`	Contains DFA functionality used by STL-scheduler
`stl_scheduler_dfa.h`	Contains function-prototypes of DFA-functionality
`stl_scheduler_init.c`	Contains STL-scheduler initialization API
`stl_scheduler_m33.c`	Contains save-restore functionality used by STL-scheduler
`stl_scheduler_m33.h`	Contains macros, type-defines and prototypes used by save-restore functionality in STL-scheduler
`stl_scheduler_tests_table.c`	Contains save-restore functionality used by STL-scheduler
`stl_scheduler_tests_table.h`	Contains macros and function-prototypes used by save-restore functionality in STL-scheduler
`stl_test_suite.c`	Contains an example to showing STL-usage
`stl_test_suite.h`	Contains macros needed by the example setup
`stl_test_suite_asm.S`	Contains assembly functions needed by the example setup
`stl_test_suite_asm.h`	Contains macros used by the assembly functions in the example setup
`stl_core.scf`	Contains scatter files for stl_core.S
`stl_fpu.scf`	Contains scatter files for stl_fpu.S
`stl_mpu.scf`	Contains scatter files for stl_mpu.S
`stl_nvic.scf`	Contains scatter files for stl_nvic.S
`stl_test_suite.scf`	Contains scatter files for stl_test_suite (example setup)

2.3.2 STL architectural overview

The following diagram provides a high-level overview of the Cortex-M33 *Software Test Library* (STL) architecture.

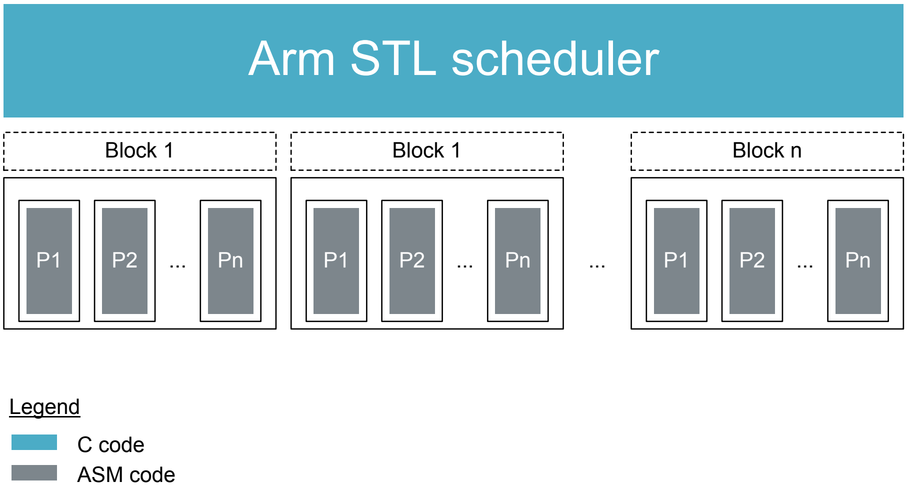


Figure 2-1 Overview of STL architecture

The STL has been written in C code and Assembly (ASM) code. There are two functional levels to the code:

Arm STL Scheduler

The first functional level, written in C, is the Arm STL Scheduler. The scheduler controls when the diagnostic tests are run and reports the test results. Functions at this level have the prefix `stl_scheduler_`.

Block1-Blockn

Blocks are logical groups into which the functions are grouped to test individual hardware units in the Cortex-M33 processor. Each block contains a number of functions, p1, p2, ... pn, which are called parts, as described in *2.3.3 STL part descriptions* on page 2-23. The parts are written in ASM and they perform the fault diagnosis of the internal hardware units. The Cortex-M33 STL has the following blocks:

- Core, consisting of the *Prefetch Unit* (PFU) and the *Data Processing Unit* (DPU)
- *Floating-Point Unit* (FPU)
- *Memory-Protection Unit* (MPU)
- *Nested Vector Interrupt Controller* (NVIC)

Each Cortex-M33 STL test has the format `stl_<block>_<optional>_p<parts>_n<iter>.S`.

For example, `stl_core_dpu_p005_n001.S`, where:

- `core` is the hardware unit this block is testing
- `dpu` is the sub-unit under test in the block
- `p005` is the number of the part
- `n001` represents the number of iterations

`M33_STL` allows you to select and sequentially run one or more `stl_<block>` functions in an order specified by a list, and collect the diagnostic results. See *2.3.4 STL APIs* on page 2-31 for more information.

2.3.3 STL part descriptions

A *Software Test Library* (STL) part is a routine that tests a part of a block, which can be the Core, *Floating-Point Unit* (FPU), *Memory-Protection Unit* (MPU), or *Nested Vector Interrupt Controller* (NVIC).

The following table describes the STL parts that test the Cortex-M33 processor.

Table 2-3 STL part descriptions

Test ID	Part name	Function or operation tested
`CORE_P001`	`stl_core_ris_p001_n001`	This test detects faults in the core by creating a sequence of random instructions.
`CORE_P002`	`stl_core_ris_p002_n001`	This test detects faults in the core by creating a sequence of random instructions.
`CORE_P003`	`stl_core_dataproc_p003_n001`	This test executes various data processing instructions to detect stuck-at faults in the DPU. The instructions tested are: • AND • MUL • ADD • EXOR • SBC • TST • RSB • CMP • CMN • ORR • BIC • UDIV • SDIV
`CORE_P004`	`stl_core_decoder_p004_n001`	This test executes various data processing instructions to detect stuck-at faults in the DPU. The instructions tested are: • LDA • LDAH • LDAB • STL • STLB • LDAEX • LDAEXB • LDAEXH • STLEX • STLEXB • STLEXH • LDR (register) • STR (register) • LDRSB (register) • LDRSH (register) • LDRB (register) • LDRH (register) • STRB (register) • STRH (register) • LDRB (literal) • LDRH (literal) • ADR • ADR.w • ADD (sp plus immediate) • LDR (sp plus immediate) • STR (sp plus immediate) • LDRSB (register)

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
CORE_P005	stl_core_dpu_p005_n001	This test detects stuck-at faults in the following registers: • SPLIM • SP • LR This test also detects stuck-at faults in the DPU with the following instructions: • RBIT • SBFX • USAT • SSAT • LSRS
CORE_P007	stl_core_ris_p007_n001	This test detects faults in the core by creating a sequence of random instructions.
CORE_P008	stl_core_ris_p008_n001	This test detects faults in the core by creating a sequence of random instructions.
CORE_P009	stl_core_dataproc_p009_n001	This test executes various instructions to detect stuck-at faults in the DPU. The instructions tested are: • QADD8 • QSUB8 • QADD16 • QSUB16 • QSAX • QASX • QADD • QDADD • QDSUB • SSAT16 • SMLABB • SMLABT • SMLATB • SMLATT • SMLAD • SMLADX • SMLAWB • SMLAWT • SMLSD • SMLSDX • SMMLA • SMMLAR • SMMLS • SMMLSR • SMLALBB • SMLALBT • SMLALTB • SMLALTT • SMLALD • SMLALDX • SMLSLD

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
`CORE_P009` (continued)	`stl_core_dataproc_p009_n001`	• SMLSLDX • SMMUL • SMMULR • SMUAD • SMUADX • SMULBB • SMULBT • SMULTB • SMULTT • SMULWB • SMULWT • SMUSD • SMUSDX • CBZ • CBNZ
`CORE_P010`	`stl_core_ris_p010_n001`	This test detects faults in the core by creating a sequence of random instructions. The instructions tested are: • ADC • ADCS • ASRS • LDRSB • LSLS • MVNS • ORRS • QSUB16 • RBIT • RORS • RSB • SBCS • SEV • SHADD8 • SMLAL • SMLATB • SMLATT • SMLAWB • SMLAWT • SMLSD • SMMLAR • SMMLSR • SMUAD • SMULBB • SMULBT • SSAX • SXTAB • SXTAH • SXTB • SXTH • UADD16 • UHSUB8 • UMAAL

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
CORE_P010 (continued)	stl_core_ris_p010_n001	- UQADD8 - UQSUB8 - UXTAH - UXTH
MPU_P001	stl_mpu_p001_n002.S	This test scans for faults in the following registers: - Secure MPU_CTRL, MPU_RNR, MPU_MAIR0/1 registers - Non-Secure MPU_CTRL, MPU_RNR, MPU_MAIR0/1 registers - SAU_CTRL, SAU_RNR registers This part iterates twice. The first iteration completes a scan of the secure MPU registers. The second iteration scans the Non-secure MPU and SAU registers.
MPU_P002	stl_mpu_p002_n032.S	This test scans for faults in the following registers: - Secure MPU_RBAR, MPU_RLAR registers - Non-Secure MPU_RBAR, MPU_RLAR registers This part iterates32 times. The first 16 iterations complete a scan of the secure MPU_RBAR and MPU_RLAR registers. The second 16 iterations scan the Non-secure MPU_RBAR and MPU_RLAR registers.
MPU_P003	stl_mpu_p003_n008.S	This test scans for faults in the SAU_RBAR and SAU_RLAR registers.
MPU_P004	stl_mpu_p004_n032.S	This test detects faults on the D-side of the MPU comparator by scanning for faults along the data path interface between the MPU and Core. This part iterates 32 times. The first 16 iterations completes a scan of the secure MPU RBLAR region for `sbist_scratch_1`. The second 16 iterations scans the secure MPU RBLAR regions for `sbist_scratch_2`.
MPU_P005	stl_mpu_p005_n032.S	This test detects faults on the D-side of the MPUNS comparator by scanning for faults along the data path interface between the MPU and Core. This part iterates 32 times. The first 16 iterations completes a scan of the Non-secure MPU RBLAR region for `sbist_scratch_1`. The second 16 iterations scans the Non-secure MPU RBLAR regions for `sbist_scratch_2`.
MPU_P006	stl_mpu_p006_n016.S	This test detects faults on the D-side of the SAU comparator by scanning for faults along the data path interface between the MPU (SAU) and Core. This part iterates 16 times. The first 8 iterations completes a scan of the secure SAU RBLAR region for `sbist_scratch_1`. The second 8 iterations scans the secure SAU RBLAR regions for `sbist_scratch_2`.

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
MPU_P007	stl_mpu_p007_n256.S	This test detects faults on the D-side and I-side exception handling of the MPU bank by scanning for faults on the secure MPU D/I-side comparators for MemManage exceptions. This part iterates 256 times. Each specified scratch region is used through 16 iterations to scan the D-side MemManage (MMDACC) and I-side MemManage (MMIACC) fault interfaces. D-side faults are induced using appropriate read/write access permissions, while I-side faults are induced based on execute-never (XN) regions. Furthermore, privileges are tested by setting up appropriate permissions.
MPU_P008	stl_mpu_p008_n256.S	This test detects faults on the D-side and I-side exception handling of the NS-MPU bank by scanning for faults on the Non-secure MPU D/I-side comparators for MemManage exceptions. This part iterates 256 times. Each specified scratch region is used through 16 iterations to scan the Non-secure D-side MemManage (MMDACC) and I-side MemManage (MMIACC) fault interfaces. D-side faults are induced using appropriate read/write access permissions, while, I-side faults are induced based on execute-never (XN) regions. Furthermore, privileges are tested by setting up appropriate permissions.
MPU_P009	stl_mpu_p009_n128.S	This test detects faults on the D-side and I-side exception handling of the SAU bank by scanning for faults on the Non-secure SAU D/I-side comparators for Secure exceptions. This part iterates 128 times. Each specified scratch region is used through 8 iterations to scan the D-side and I-side Secure exceptions.
FPU_P001	stl_fpu_reg_p001_n001.S	This test detects faults in FPU register file.
FPU_P002	stl_fpu_reg_p002_n001.S	This test detects faults in Secure and Non-secure FPU System registers.
FPU_P003	stl_fpu_dataproc_p003_n001.S	This test detects fault in FPU datapath for various instructions.
FPU_P004	stl_fpu_dataproc_p004_n001.S	This test executes VMOV and VABS to test the FPU datapath, and also UNDEF instructions in FPU space.
FPU_P005	stl_fpu_context_p005_n001.S	This test checks faults in context stacking and unstacking logic in FPU.
FPU_P006	stl_fpu_div2nd_p006_n001.S	This test checks faults in division logic in FPU.
FPU_P007	stl_fpu_divsqrt_p007_n001.S	This test checks faults in square-root logic in FPU.
FPU_P008	stl_fpu_maxmin_p008_n001.S	This test executes VMAXNM and VMINNM instructions to check faults in FPU datapath.
FPU_P009	stl_fpu_vadd_p009_n001.S	This test executes variants of VADD instruction in various modes to check faults in FPU datapath.
FPU_P010	stl_fpu_vcvt_p010_n001.S	This test executes variants of VCVT instruction in various modes to check faults in FPU datapath.

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
FPU_P011	stl_fpu_vfma_p011_n001.S	This test executes VFMA instruction in various modes to check faults in FPU datapath.
FPU_P012	stl_fpu_vfms_p012_n001.S	This test executes VFMS instruction in various modes to check faults in FPU datapath.
FPU_P013	stl_fpu_vfnma_p013_n001.S	This test executes VFNMA instruction in various modes to check faults in FPU datapath.
FPU_P014	stl_fpu_vmls_p014_n001.S	This test executes VMLS instruction in various modes to check faults in FPU datapath.
FPU_P015	stl_fpu_vnmla_p015_n001.S	This test executes VNMLA instruction in various modes to check faults in FPU datapath.
FPU_P016	stl_fpu_vsqrt_p016_n001.S	This test executes VSQRT instruction in various modes to check faults in FPU datapath.
FPU_P017	stl_fpu_vnmls_p017_n001.S	This test executes VNMLS instruction in various modes to check faults in FPU datapath.
FPU_P018	stl_fpu_round_p018_n001.S	This test executes VRINT instruction in various modes to check faults in FPU datapath.
FPU_P019	stl_fpu_clz32b_p019_n001.S	This test executes various Floating Point instructions to detect faults in CLZ logic in FPU datapath.
FPU_P020	stl_fpu_clz32a_p020_n001.S	This test executes various Floating Point instructions to detect faults in CLZ logic in FPU datapath.
FPU_P021	stl_fpu_vcmp_p021_n001.S	This test executes variants of VCMP instruction to detect faults in FPU comparison logic.
FPU_P022	stl_fpu_lza_p022_n001.S	This test executes various Floating Point instructions to detect faults in leading zero alignment logic in FPU datapath.
NVIC_P001	stl_nvic_regro_p001_n001.S	This test detects errors on following read-only registers: • Secure registers: CTR, CPUID, PFR0, ISAR4, MMFR2, MMFR0, PIDR3, PIDR5, PIDR6, PIDR7 • Non-secure registers: PIDR3, PIDR5, PIDR6, PIDR7
NVIC_P002	stl_nvic_regro_p002_n001.S	This test detects errors on Non-secure read-only register ICTR.
NVIC_P003	stl_nvic_fpuro_p003_n001.S	This test detects errors on following read-only registers: • Secure registers: MVFR2, MVFR0, MVFR1 • Non-secure registers: MVFR2, MVFR0, MVFR1
NVIC_P004	stl_nvic_statusregro_p004_n001.S	This test detects errors on following read-only registers: • Secure registers: CFSR, HFSR, DFSR, SFSR • Non-secure registers: CFSR, HFSR, DFSR
NVIC_P005	stl_nvic_dbglvlro_p005_n001.S	This test detects errors on following secure read-only registers: • CIDR0, CIDR1, CIDR2, CIDR3 • PIDR0, PIDR1, PIDR2, PIDR4 • DFR0, DEVARCH

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
NVIC_P006	stl_nvic_secextro_p006_n001.S	This test detects errors on following read-only registers: • Secure registers: PFR1 • Non-secure registers: CTR, MMFR2, MMFR0, CPUID, PFR0, PFR1, ISAR4, SYSCALIB, PFR1
NVIC_P007	stl_nvic_secextdbglvlro_p007_n001.S	This test detects errors on following Non-secure read-only registers: • DFR0, DEVARCH • CIDR3, CIDR2, CIDR1, CIDR0 • PIDR0, PIDR1, PIDR2, PIDR4
NVIC_P008	stl_nvic_regrw_p008_n001.S	This test detects errors on following secure read/write registers: • VTOR • SYST_RVR • SHPR1 • SHPR2 • SHPR3 • MMFAR • BFAR • SFAR • FPDSCR • FPCAR • SCR • CCR • AIRCR • BASEPRI • PRIMASK • FAULTMASK
NVIC_P009	stl_nvic_regrw_p009_n001.S	This test detects errors on following read/write registers: • Secure registers: ACTLR, FPCCR, NSACR, CPACR, CPPWR, DHCSR • Non-secure registers: ACTLR, DHCSR, FPCAR, CPPWR, FPCCR, FPDSCR, CPACR, DCRDR
NVIC_P010	stl_nvic_regrw_p010_n001.S	This test detects errors on following read/write registers: • All IPR • All ITNS
NVIC_P011	stl_nvic_secextrw_p011_n001.S	This test detects errors on following Non-secure read/write registers: • VTOR • SYST_RVR • SHPR1 • SHPR2 • SHPR3 • SCR • CCR • AIRCR • MMFAR • BASEPRI • PRIMASK • FAULTMASK

Table 2-3 STL part descriptions (continued)

Test ID	Part name	Function or operation tested
`NVIC_P012`	`stl_nvic_dbglvlrw_p012_n001.S`	This test detects errors on following secure read/write registers: • DCRDR • DAUTHSTATUS
`NVIC_P014`	`stl_nvic_internalexcep_p014_n001.S`	This test detects errors on secure and Non-secure: • SVC • Usage Fault • Hard Fault • PendSV
`NVIC_P015`	`stl_nvic_systick_p015_n001.S`	This test detects errors on secure and Non-secure SYSTICK registers.
`NVIC_P016`	`stl_nvic_wic_p016_n001.S`	This test detects faults in the *Wake-up Interrupt Controller* (WIC).
`NVIC_P017`	`stl_nvic_core_decoder_p017_n001.S`	This test checks *Non-Maskable Interrupt* (NMI) interrupting load, store multiples.
`NVIC_P018`	`stl_nvic_exceptree_p018_n001.S`	This test detects errors on secure exception tree.
`NVIC_P019`	`stl_nvic_exceptree_ns_p019_n001.S`	This test detects errors in exception based active and pending tree.
`NVIC_P020`	`stl_nvic_actvtree_p020_n001.S`	This test detects errors on active and pending trees.
`NVIC_P021`	`stl_nvic_wfecheck_p021_n001.S`	This test detects errors on wfe and sevonpend based logic.
`NVIC_P022`	`stl_nvic_fsm_p022_n001.S`	This test generates interrupts in descending order of interrupt number.

2.3.4 STL APIs

The *Application Program Interfaces* (APIs) used by the Cortex-M33 *Software Test Library* (STL) are described in the following sections.

Set-Parameters

The Set-Parameters API, `M33_STL_Init`, is used to configure the STL-Parts to run.

Scope System-Interface

Parameters Start (`part_start_num`)

First STL-Part to run.

Last-Part (`part_max_num`)

Last STL-Part to potentially run

Return Value `0x0`

STL parameter setup failed

`0x1`

STL parameter setup successful

Format

```
init M33_STL_Init(int part_start_num, int part_max_num) ;
```

Run STL

The STL can be called using the `M33_STL` API.

Scope System-Interface

Parameters `num_subtests`

Number of STL parts (including iterations) to schedule

`tmode`

Mode, which can be one of the following:

- `0x1`: Out of reset mode
- `0x2`: Online time triggered mode
- `0x3`: Online event driven mode
- `0x4`: Force failure (debug) triggered mode

Return Value `0x1`

Potentially dangerous fault detected

`0x0`

No dangerous faults detected

Format

```
int M33_STL(int num_subtests, int tmode);
```

2.3.5 STL scheduler

The scheduler can be configured to specify the number of *Software Test Library* (STL) parts to execute each time it is called.

The scheduler can be called in two modes:

Out-of-Reset mode

During powerup, the software routine is used to diagnose the logic in the design before the operating system can schedule an application. This is referred to as *Out-of-Reset* (OoR) test mode.

Online mode

The STL routine is scheduled as an application, but with full privileges (in privilege mode). If this method is adopted, the routine can be scheduled periodically. This is referred to as *Online* (OL) test mode. There are two options available for OL:

- Time-triggered online (OLT) mode, see *Time-triggered OL scheduling* on page 2-35.
- Event-driven online (OLE) mode, see *Event-triggered OL scheduling* on page 2-36.

Note

All STL parts can be scheduled in the *Out-of-Reset* (OoR) mode. All STL parts except for NVIC can be scheduled in the *time-triggered online* (OLT) and *event-driven online* (OLE) modes. For details of the scheduler modes, see *2.3.5 STL scheduler* on page 2-32.

The following figure shows the Cortex-M33 STL time-triggered OL execution process.

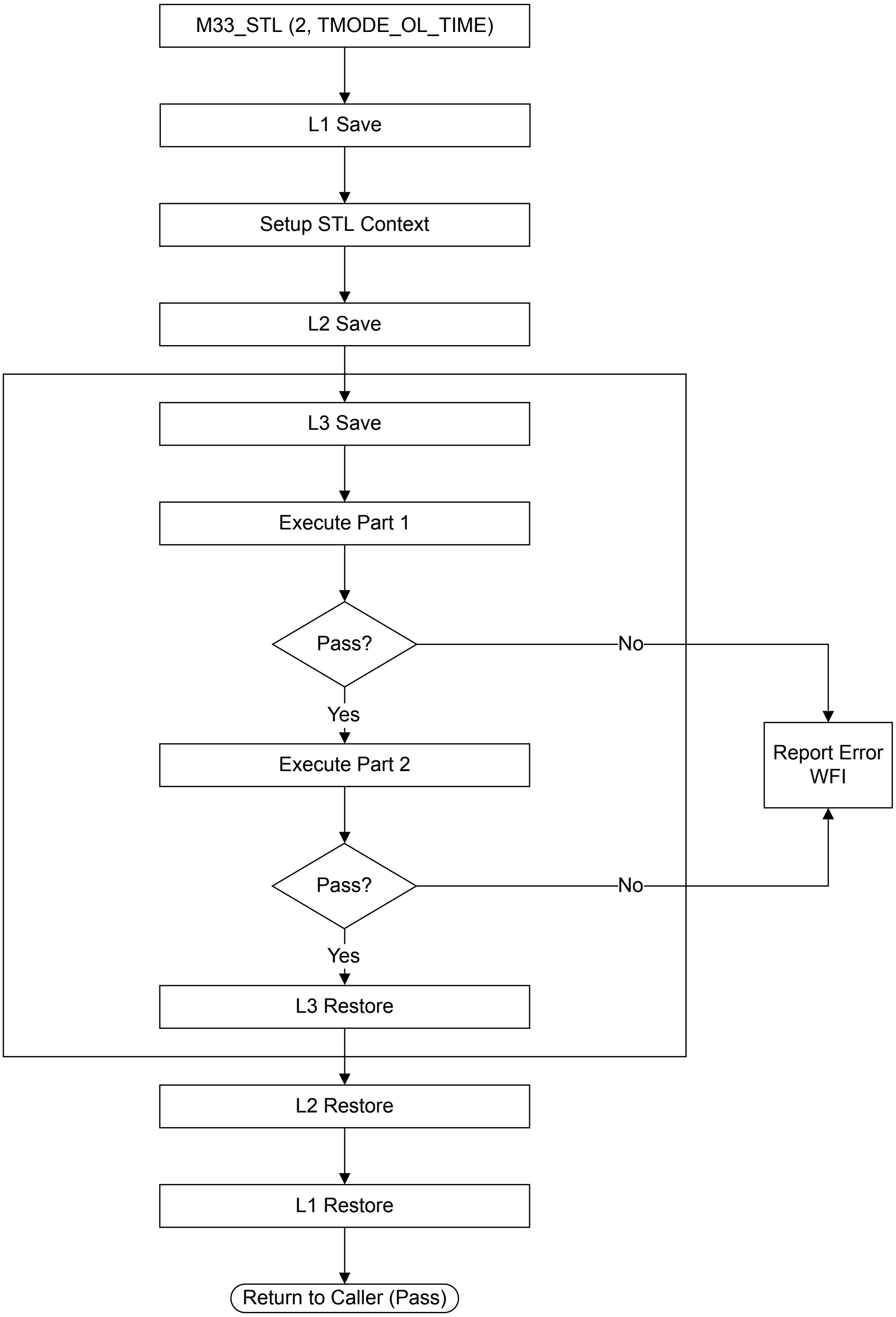


Figure 2-2 OL test scheduling process

Note

- In *Example time-triggered OL scheduling process* on page 2-37:
  - — Level 1 save and restore is the global save and restore. An STL global restore occurs after the test routine has completed.
  - — Level 2 save and restore is the unit-specific save and restore and STL C save and restore.
  - — Level 3 save and restore is intrinsic and occurs within each element that is run.
  - — The Check element predefined process checks if each element has passed. If an element fails, then the test fails and the processor enters *Wait For Interrupt* (WFI) state.

For more information on the save and restore methodology, see *Save and restore methodology* on page 2-38.

- The OL tests are listed in `stl_scheduler_parts.lst`. For more details, see *2.3.6 STL release directory structure* on page 2-39.

The following figure shows the scheduling process.

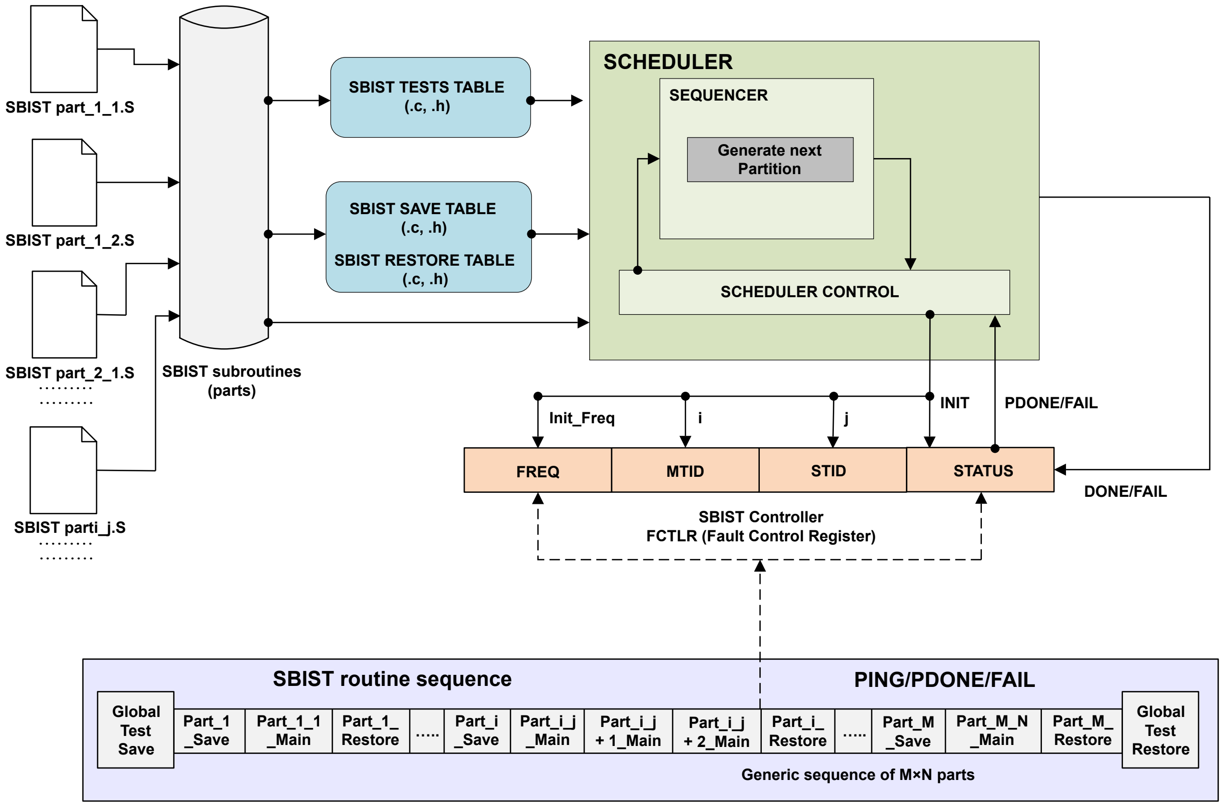


Figure 2-3 STL scheduling process

The STL scheduling process generically follows these steps:

1. Based on the two scheduling methods, OoR and OL, there is a single test list that is a repository for all the parts to test your chosen blocks. After the parts are determined, a Python script is run that generates the STL test table.
2. An STL test suite (`stl_test_suite`) is provided as a verification program that allows you to verify that all the parts are running correctly. The test suite demonstrates the usage of the STL scheduler. The following steps are performed when the test suite is run.
   a. Initialize the STL scheduler.
   b. Call the STL scheduler. If you call the STL scheduler using time-triggered scheduling, then disable all interrupts. For more information on time-triggered and event-triggered scheduling, see *Time-triggered OL scheduling* on page 2-35 and *Event-triggered OL scheduling* on page 2-36.
   c. Run the OoR tests using the STL scheduler. All the OoR tests must be run before the bootloader starts the caller, or the caller sets up the interrupt controller.

      Example code

```
// Initialize test table
// This needs to be called only once
setup_table();
// OOR mode
valid = M33_STL_Init(START_POS_STESTS_OOR, LAST_POS_STESTS_OOR + 1);
if (valid)
```

```
{
result = M33_STL(NUM_STESTS_OOR, TMODE_OOR);
}
```

d. Run the OL tests. The caller can schedule the OL tests periodically.

Example code

```
// OLT mode
__disable_irq();
valid = M33_STL_Init(START_POS_STESTS_OLT, LAST_POS_STESTS_OLT + 1);
if (valid)
{
result = M33_STL(NUM_STESTS_OLT, TMODE_OL_TIME);
}
__enable_irq();


// OLE mode
__disable_irq();
valid = M33_STL_Init(START_POS_STESTS_OLE, LAST_POS_STESTS_OLE + 1);
if (valid)
{
result = M33_STL(NUM_STESTS_OLE, TMODE_OL_EVENT);
}
__enable_irq();
```

Time-triggered OL scheduling

In time-triggered OL scheduling, the caller calls the STL periodically whenever it finds an interrupt-free time window.

The STL must not be prevented from executing. This means that if the caller cannot find an interrupt-free time window, it must disable interrupts before calling the STL.

The STL scheduler calls the specified parts sequentially and returns a status indicating success or failure to the caller.

The total time taken by the STL test is given by the following equation:

$$N = S + \sum_{i=0}^{(K-1)} [O(i) + S(i) + P(i) + R(i)] + R$$

- N is the total time it takes to run the test.
- S is the time it takes to save the processor state on the heap.
- R is the time taken by the test to restore the processor state from the heap.
- P(i) is the time taken by an individual part that is optimized to target different functions of the processor.
- S(i) and R(i) indicate the local save and restore.
- K is the number of elements that the STL implements, which depends on the configuration of the processor and the tests selected in the list.
- O(i) is the overhead to schedule the next part. The overhead is generally negligible.

The following figure shows time-triggered STL test scheduling.

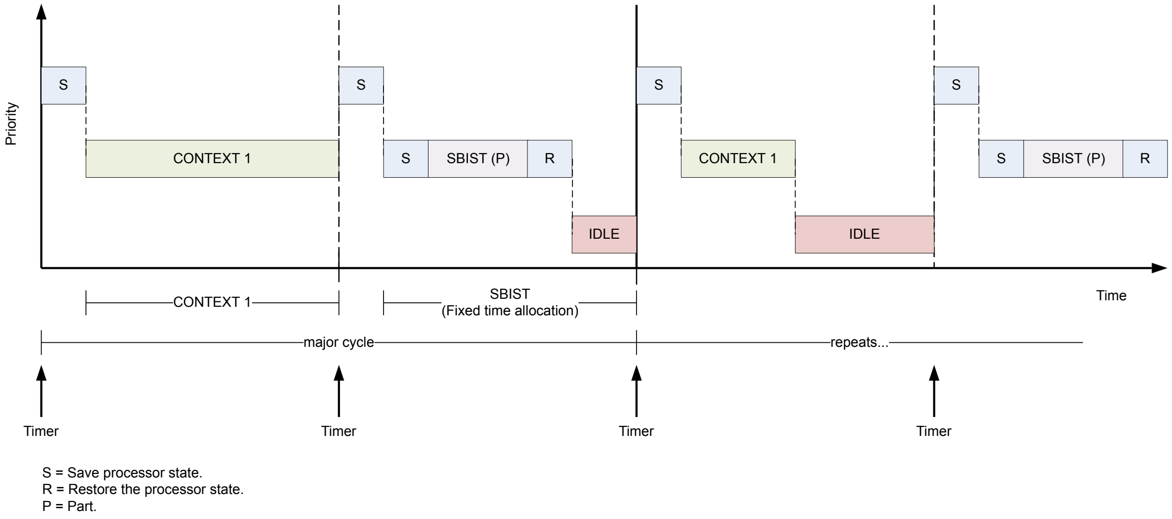


Figure 2-4 Time-triggered system

Note

In a *Figure 2-4 Time-triggered system* on page 2-36, the priority of every job and task is the same.

Example time-triggered OL scheduling process

The following example shows when the scheduler is called to run two parts in time-triggered OL mode:

1. Global (L1) Save.
2. Sets up the Context for STL.
3. Unit (L2) Save.
4. Element or Part (L3) Save.
5. Executes the first part and checks the result. If the first part passes, execution continues. If the result is an error, the processor enters WFI state. The WFI state is the default safe state, but user code can be invoked instead of the WFI state. Returning back to the caller from the safe-state cannot be guaranteed as this state is a result of a potentially dangerous fault being detected.
6. Executes the second part and checks the result. If the second part passes, execution continues. If the result is an error, the processor enters WFI state. The WFI state is the default safe state, but user code can be invoked instead of the WFI state. Returning back to the caller from the safe-state cannot be guaranteed as this state is a result of a potentially dangerous fault being detected.
7. L3 Restore.
8. L2 Restore.
9. L1 Restore.
10. Returns to caller.

Event-triggered OL scheduling

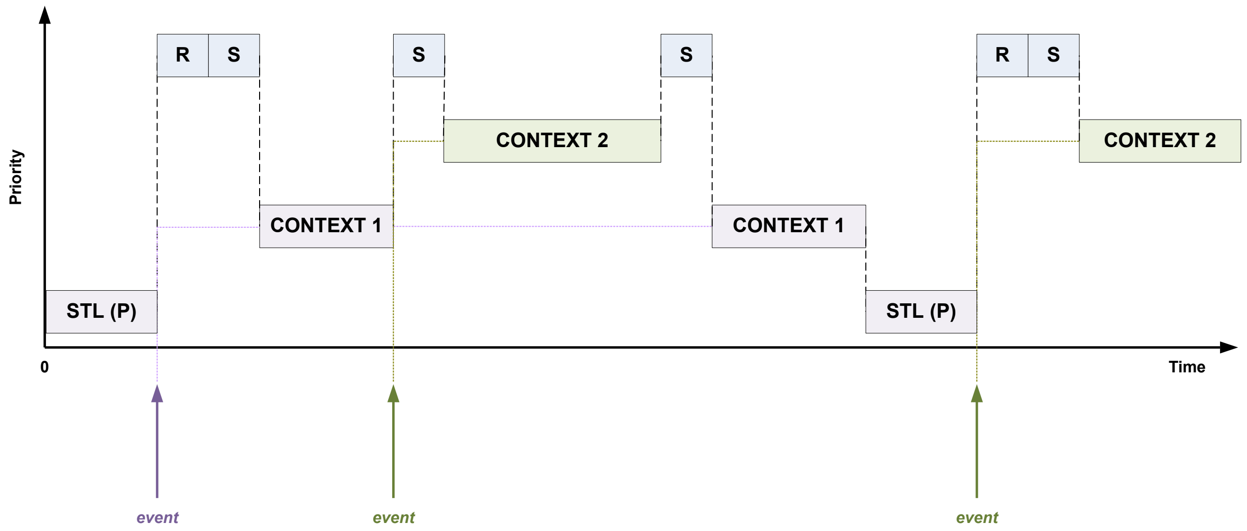
In event-triggered OL scheduling, the STL routine executes during periods when higher priority applications are not scheduled to execute.

This avoids having to schedule the STL routine within a fixed period based on the *Worst Case Execution Time* (WCET) allocated to all critical routines. Therefore, event-triggered scheduling utilizes the spare capacity within an allocated period.

The STL scheduler must not be preempted during execution. Therefore, if the caller cannot find an interrupt-free period, it must disable interrupts before calling the STL scheduler, which can increase interrupt latency.

The STL event-triggered OL (online) mode reduces interrupt latency. In this mode, after each part is run, the STL scheduler checks if an interrupt is pending. If an interrupt is pending, the STL scheduler returns to the caller to allow it to handle the interrupt, without executing any parts that are waiting to execute. The STL scheduler executes the remaining parts the next time it is called by the caller.

The following figure shows event-triggered STL test scheduling.



S = Save processor state.
R = Restore the processor state.
P = Part.

Figure 2-5 Event-triggered system

There are no IDLE periods in event-triggered scheduling because the STL test runs in the background when there are no critical tasks scheduled.

The STL routine can be preempted after a determined, worst-case period. At the point where the routine is preempted, it is possible to return to the routine without restarting. This implies that the STL routine can be implemented in a `while (1)` loop that constantly tests logic in the design, but can be preempted to allow critical routines to execute before returning to complete execution of the STL routine. The only exception is when the processor interrupt handling logic needs to be tested.

Example time-triggered OL scheduling process

This section shows an example time-triggered OL scheduling process.

The following example shows when the scheduler is called to run two parts in time-triggered OL mode:

1. Global (L1) Save
2. Sets up the Context for STL
3. Unit (L2) Save
4. Element or Part (L3) Save
5. Executes the first part and checks the result. If the first part passes, execution continues. If the result is an error, the processor enters WFI state. The WFI state is the default safe state and can be overridden.
6. Executes the second part and checks the result. If the second part passes, execution continues. If the result is an error, the processor enters WFI state. The WFI state is the default safe state, but user code can be invoked instead of the WFI state. Returning back to the caller from the safe-state cannot be guaranteed as this state is a result of a potentially dangerous fault being detected.
7. L3 Restore
8. L2 Restore

9. L1 Restore
10. Returns to caller

Structure of Parts list

This section describes the structure of the list used to specify the relevant *Software Test Library* (STL) Parts and their compatibility in different modes.

The format used to specify the list is as follows:

<STL-Part-Name> ((OoR-Compatibility (<Y/N>), OLE-Compatibility (<Y/N>), OLT-Compatibility (<Y/N>)).

The STL deliverables include a script, called `gen_table`, which processes the list file and generates the relevant source file(s). The STL-Parts list can be modified by the user, allowing addition or removal of STL-Parts and changing the OoR/OnL mode compatibility. The ordering in this file is preserved at run-time when the STL is called.

The following is an example of the STL-Parts list:

`stl_mpu_p001_n002`	`(Y, Y, Y)`	`# MPU-Part 001 runs in the OLE, OLT, and OoR modes (2 iterations)`
`stl_mpu_p002_n032`	`(Y, N, N)`	`# MPU-Part 002 runs in the OoR mode (32 iterations)`
…	…	…
`stl_mpu_p008_n256`	`(N, N, N)`	`# MPU-Part 008 does not run in OLE, OLT, or OoR mode (256 iterations)`
`stl_mpu_p009_n126`	`(N, Y, Y)`	`# MPU-Part 009 runs in the OLE and OLT modes (128 iterations)`

The subsequent source files and the contents generated from this list are used by the scheduler during the course of STL execution.

In Cortex-M33, a Python script (`list/gen_table.py`) is used to process the STL-Parts list and a source file (`m33_stl_scheduler_list.c`) is generated. Changes to the list should be applied to this source file. A header (`m33_stl_parts.h`) is also generated from the STL-Parts list and contains a macro defining the number of valid parts in the list.

Usage guidelines for `gen_table.py` can be found using the following command:

```
python gen_table.py -help
```

Save and restore methodology

Cortex-M33 STL uses three levels of save and restore.

Level 1 save and restore

This is the global save and restore.

Global save preserves System registers that the STL might corrupt, irrespective of the parts that are run.

Global restore restores the set of System registers that are saved by global save.

Level 2 save and restore

This is the test-specific save and restore.

The test-specific global save is done to:
- Save and set up the relevant registers
- Program the test-specific MPU regions

The test-specific restore is done to restore the configuration registers.

Level 3 save and restore

This is the part-specific save and restore.

Part-level save saves System registers corrupted by a particular part.

Part-level restore restores the set of System registers that are saved by part-level save.

2.3.6 STL release directory structure

The following figure shows the directory structure for the Cortex-M33 *Software Test Library* (STL).

```
stl
|-- documentation
|-- faultsim
|   |-- bsub.sh
|   |-- cortexm33_net.sff
|   |-- sbist_fmsh_cmd.lst
|   `-- zoix.vc
|-- sim
|   |-- Makefile
|   `-- Makefile.asm
`-- tests
    |-- stl_core
    |-- stl_fpu
    |-- stl_link
    |-- stl_list
    |-- stl_mpu
    |-- stl_nvic
    |-- stl_scheduler
    |-- stl_shared
    `-- stl_test_suite
```

Note

`Makefile`, `Makefile.asm`, and linker files are provided to verify the execution of the STL software, they are not necessary for software integration.

Details of the directory structure

The following table describes the structure of the directory.

Table 2-4 Directory structure

Directory/File	Purpose
`stl/documentation`	Top level directory to hold STL documents, such as this User Guide
`stl/sim`	Top level directory for the `makefiles` to compile the STL for logical fault simulations
`stl/tests/stl_<block>`	Top level directory for a logical group of STL, such as core, fpu, mpu, and nvic
`stl/tests/stl_shared`	Top level directory for shared files
`stl/tests/stl_scheduler`	Top level directory for scheduler files
`stl/tests/stl_link`	Top level directory for linker scripts
`stl/tests/stl_list`	Top level directory for lists containing references STL parts
`stl/tests/stl_test_suite`	Top level directory for example suites. There can be more than one suite, so that single core configurations or dual core configurations can be run.

2.3.7 Makefiles

The makefiles available in the `stl/sim` directory can be used to compile the source code and generate executable binaries, which can be used with the execution testbench delivered with the Cortex-M33 processor.

2.3.8 Linker scripts

The linker script, also known as the scatter file, is an example memory map of a specific system described by the execution testbench.

The linker script tells the linker how to place the vector table, application code, data, stacks, and heap at suitable addresses in the memory map.

It is generated to support the `armcc` compiler flow.

2.3.9 STL configuration

The global configuration controls are a set of static parameters which must be initialized before building the *Software Test Library* (STL).

These parameters specify the system information needed by the STL to function correctly. This information is used to make the following inferences:

- SBIST Method and base address used for status-reporting
- Code, Data, and Scratch space information available in the system
- Disabled IRQ information relevant for the system
- Miscellaneous system configuration information that cannot be derived at runtime

stl_global_defs.h

The `stl_global_defs.h` file is customer-modifiable to set the hardware method and memory locations for STL implementation.

Before you build the `stl_test_suite`, you must update the `stl_global_defs.h` file to match your system integration, then copy the updated file to `tests/stl_shared/include/gnuasm/stl_global_defs.h`.

The following table shows the variables that can be modified in the `stl_global_defs.h` file.

Table 2-5 Modifiable variables in stl_global_defs.h

Control name	Default value	Description	Safety implications
`sbist_method`	`0x1`	The Arm SBIST controller provides a method to control the execution of the STL on the Cortex-M33 processor. It also has built-in checks to detect deadlock faults and support a means to drive processor inputs and check processor outputs. The STL can execute without an Arm SBIST controller: `0x0` = No Arm SBIST controller `0x1` = Arm SBIST controller	You must consider `sbist_method` = `0x1` to get the maximum possible coverage from the *Software Test Library* (STL).
`cpu_sbist_base`	• `0x20000000` when `sbist_method` = `0x0` • `0xE0044000` when `sbist_method` = `0x1`	The memory address where the Arm SBIST controller is integrated into the system. If `sbist_method` == `0x0`, then the address must be mapped to a memory instead of the hardware controller so that the STL can read and write to memory mapped registers regardless of the method of integration.	You must specify a valid address for this region for compilation and functioning of STL.

Table 2-5 Modifiable variables in stl_global_defs.h (continued)

Control name	Default value	Description	Safety implications
code_region	0x00000000	The base address of the Secure ROM where the Secure code lives. This is applicable to hardware configurations where SECEXT == 1. If SECEXT == 0 this code can be placed in Non-secure ROM.	Specifying a valid address for this region is necessary for compilation and functioning of STL.
code_region_size	0xCFFE0	The maximum size of the Secure code offset from code_region	-
nsc_code_region	0x000D0000	The base address of the Non-secure Callable ROM where the Non-secure Callable code lives. This is applicable to hardware configurations where SECEXT == 1. If SECEXT == 0, this code can be in Non-secure ROM.	This region is necessary for STL compilation. If this cannot be provided, then you must remove from the list the parts that depend on SECEXT enabled. This region can be placed anywhere.
nsc_code_region_size	0xFFE0	The maximum size of the Non-secure Callable code offset from the nsc_code_region	-
ns_code_region	0x000E0000	The base address of the Non-secure ROM where the Non-secure code lives	This region is necessary for STL compilation. If this cannot be provided, then you must remove from the list the parts that depend on SECEXT enabled. This region can be placed anywhere.
ns_code_region_size	0x1FFE0	This is the maximum size of the Non-secure code offset from ns_code_region	-
data_region	0x20000080	The base address of the Secure RAM where the Secure data lives This is applicable to hardware configurations where SECEXT == 1. If SECEXT == 0, this data can be placed in Non-secure RAM.	This region is necessary.
data_region_size	0x10000	The maximum size of the Secure data offset from data_region	-
ns_data_region	0x20100000	The base address of the Non-secure RAM where the Non-secure data lives	This region is necessary.
ns_data_region_size	0x10000	The maximum size of the Non-secure data offset from ns_data_region	-

Table 2-5 Modifiable variables in stl_global_defs.h (continued)

Control name	Default value	Description	Safety implications
`scratch_space_1`	`0x7FFEFFF0`	A 128 byte region in Non-secure RAM. For maximum diagnostic coverage, this address must satisfy the following condition. The bit ranges [31:5] for the address indicated by scratch_space_1+32Bytes must be the complement of bit ranges [31:5] of the address indicated by scratch_space_2+32Bytes: (scratch_space_1 + 32B)[31:5] = ~(scratch_space_2 + 32B)[31:5].	This region is necessary for compilation.
`scratch_space_2`	`0x8000FFD0`	A 128 byte region in Non-secure RAM. For maximum diagnostic coverage, this address must satisfy the following condition. The bit ranges [31:5] for the address indicated by scratch_space_1+32Bytes must be the complement of bit ranges [31:5] of the address indicated by scratch_space_2+32Bytes: (scratch_space_2 + 32B)[31:5] = ~(scratch_space_1 + 32B)[31:5].	This region is necessary for compilation.

`scratch_space_3 - scratch_space_16`	`0x87FFFFF0,`
	`0x83FFFFF0,`
	`0x81FFFFF0,`
	`0x80FFFFF0,`
	`0x807FFFF0,`
	`0x803FFFF0,`
	`0x801FFFF0,`
	`0x800FFFF0,`
	`0x8007FFF0,`
	`0x8003FFF0,`
	`0x8001FFF0,`
	`0x80000FF0,`
	`0x800007F0,`
	`0x800003F0,`
		These are 96 byte regions in the Non-secure RAM.	These regions are optional. Any number of scratch regions can be instantiated. If defined, none of these scratch regions should overlap with any other scratch region locations.

`irqdis0-irqdis15`	`0x00000000`	Software interrupt disable mask. Setting a bit prevents the corresponding interrupt from being tested by NVIC STL.	

Table 2-5 Modifiable variables in stl_global_defs.h (continued)

Control name	Default value	Description	Safety implications
`systick_mask`	`0x00FFFFF8`	STLs write `0x00FFFFFF` into SYST RVR, and check if the value if loaded correctly into SYST CVR. The value read back from SYST CVR is indeterministic because of any delays that the system might introduce. For example, delay in instruction fetch. To account for any such indeterministic behavior, lower significant bits of the CVR are masked. You can determine the number of bits to be masked according to any potential delays. If the degree of indeterminism is high, more bits need to be masked. Consequently, the fault coverage numbers would be lower. The 0s in LSB of the mask imply that the corresponding bits of SYSTICK timer would not be tested. The fields of this mask are as follows: • [31:24] - {8{1'b0}} • [23:N] - {(24-N){1'b1}} • [N-1:0] - {N{1'b0}} where N is the number of bits to be masked. Example: If N = 5, then • [31:24]: `0b00000000` • [23:N]: `0b1111111111111111111` • [N-1:0]: `0b00000` Total mask: `0x00FFFFE0`	More bits masked implies lesser fault coverage. The following test cases are affected by this parameter: `stl_nvic_systick_p015_n001`
`ldm_stm_interrupt_latency`	3	**Note** This part (`stl_nvic_core_decoder_p017_n001`) is heavily reliant on the precise timing of interrupts and fetches. Any element that can cause delays along this path must be analyzed with the below considerations. Failure to do so could result in the STL part malfunctioning. This variable is used in test `stl_nvic_core_decoder_p017_n001`, which aims at catching those faults affecting the logic relative to the NMI and LDM/STM. For more details, see *Load-Multiple/Store-Multiple Interrupt Latency* on page 2-44.	Incorrectly setting this parameter could either decrease the number of detected faults or fail the test `stl_nvic_core_decoder_p017_n001`. Extra care MUST be taken to set this parameter correctly for your system.

Load-Multiple/Store-Multiple Interrupt Latency

The `ldm_stm_interrupt_latency` value is loaded into the SBISTC register FPSPR0.Count to tune the latency of the NMI signal generated to interrupt *Load-Multiple* (LDM) and *Store-Multiple* (STM) instructions of part `stl_nvic_core_decoder_p017_n001`.

This is shown in the following figure:

```
        bl              stl_core_generate_interrupt
         ldm             r0!,  {r1-r7}

...

         bl              stl_core_generate_interrupt
         ldm             r0!,  {r1-r12}

...
         bl              stl_core_generate_interrupt
         stm             r0!,  {r1-r11}
...

         bl              stl_core_generate_interrupt
         ldm             r0!,  {r1-r7}
```

On the Arm example system, with this value set to 1, the LDM and STM targeted instructions are interrupted after the third register is serviced. With a value of 4, the LDM and STM targeted instructions are interrupted after the sixth register is serviced. For the Arm example system, 4 is the maximum recommended value and 1 is the minimum. This value should be adjusted to allow some variance in your system latency.

This value is strictly related to the time it takes to interrupt the targeted LDM/STM instructions. For example, on the Arm example system, from the time the FPSPR0.Count register value of `0x33bc1` (`ldm_stm_interrupt_latency` = 3) is issued on the APB bus (signal PWDATAat Cursor 1 on the waveform below) to the time the LDM/STM instruction is interrupted (signal `run_irq` at Cursor 2) at register r05 being serviced, 11 cycles of signal tbench.CLK will have clocked.

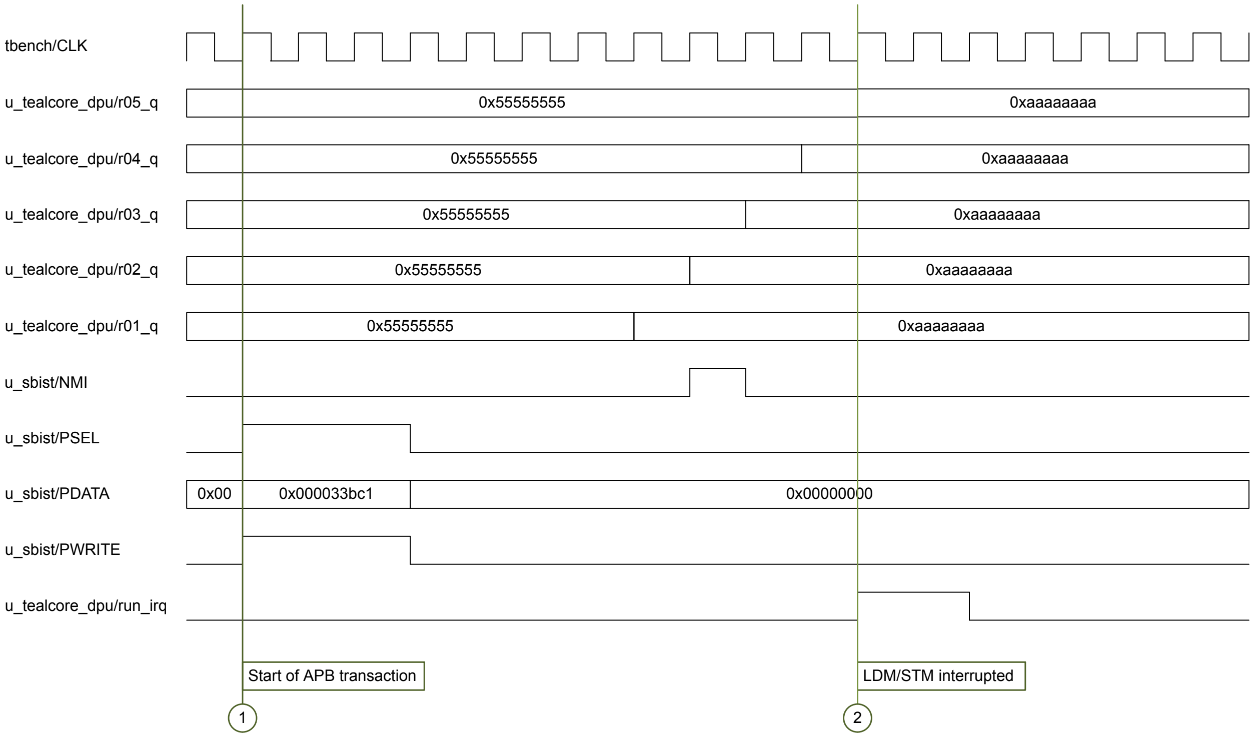


Figure 2-6 Cortex-M33 STL waveforms demonstrating Interrupt Latency

With reference to the following diagram, if the timing path is varied with respect to the reference system, the value of `ldm_stm_interrupt_latency` will need adjusting accordingly. More precisely, `ldm_stm_interrupt_latency` must be decremented once for each clock cycle delay added to this or any corresponding path, and incremented for any reduction in clock cycle delay.

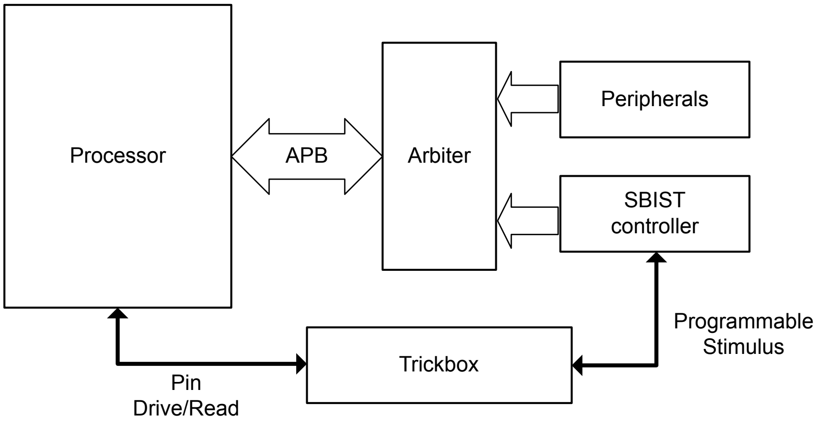


Figure 2-7 SBIST configuration for Cortex-M33

Note

For this test to work correctly, ACTLR.DISMCYCINT must be set to 0.

2.3.10 Status reporting and programmable stimulus

The *Software Test Library* (STL) writes information about the current execution to a configurable RW memory location.

Each processor in a multiprocessor configuration would report the status, in the same format, at uniquely specified memory locations.

The STL reports the execution status in the following format, where `SBIST_BASE` is configured using the `cpu_sbist_base` parameter in the global configuration controls.

Note

Register memory locations in this section are actual System registers only when the CPU contains an SBISTC (M1-method). Without the SBISTC, these are only indicative of memory locations.

Table 2-6 STL status reporting register memory locations

Offset	Register memory location	Type	Function
`SBIST_BASE` + `0x00`	FCTLR	RW	Fault Control Register
`SBIST_BASE` + `0x04`	FPIR	RW	Fault Partition Identifier Register
`SBIST_BASE` + `0x08`	FFMIR	RW	Fault Failure Mode Identification Register
`SBIST_BASE` + `0x0C`	FPSTR	RO	Fault Programmable Stimulus Type Register

Table 2-6 STL status reporting register memory locations (continued)

Offset	Register memory location	Type	Function
`SBIST_BASE` + `0x10`	FPSPR0	RW	Fault Programmable Stimulus Payload Register 0
`SBIST_BASE` + `0x14`	FPSPR1	RW	Fault Programmable Stimulus Payload Register 1

——— **Note** ———

The `SBIST_BASE` parameter must be mapped to a specified RW memory location.

For details of the SBISTC registers, refer to the *SBIST Controller* appendix in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

Programmable check register

Programmable checking is a method to sample a triggered event.

A trigger is an event generated in the system. An event can be generated because of a state upgrade, a pulse that a primary output on the Cortex-M33 processor drives, or an error signal that goes HIGH because a fault is detected.

The following table shows the programmable check register WICSENSEREG.

Table 2-7 Programmable check register

Offset	Name	Type	Description
cpu_sbist_base + `0x24`	WICSENSEREG	RW	32-bit vector indicating the WICSENSE output. For details on sampling this output, see below.

——— **Note** ———

The offset value is dependent upon the number of Stimulus Payload registers implemented. The value is based on trickbox register WICSENSEREG. If additional FPSPRs are implemented, the addresses of the check registers will need to be adjusted accordingly.

To detect faults, the Cortex-M33 processor output interface is sampled. The sampled data signal called WICSENSE is read by trickbox register WICSENSEREG and then compared with the expected value.

The following figure shows the WICSENSE read path.

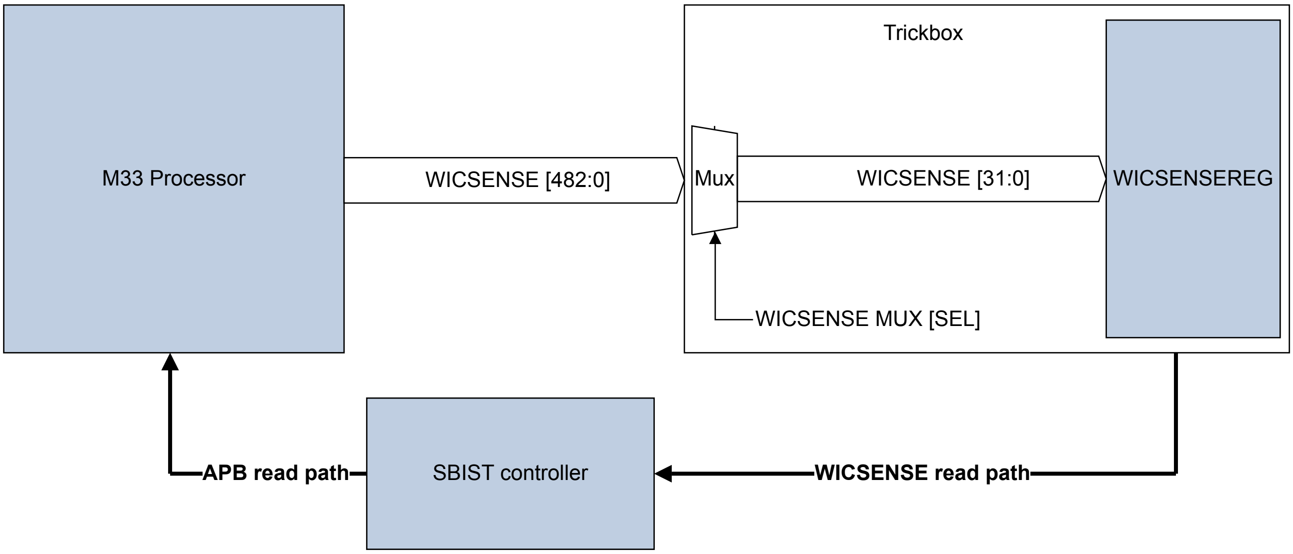


Figure 2-8 WICSENSE read path

Programmable stimulus registers

To exercise the input signals of the processor that are generally not in the control of the processor, the STL architecture provides a method to drive these inputs in a programmable and controlled way. The programming and control is achieved through the programmable stimulus memory-mapped registers.

An SBIST Controller implementation specifies a list of programmable stimulus payload registers and its specification. The specification includes the register description, which processor signals to drive, and any additional information to control the values driven on the input. All programmable stimulus features are discoverable (that is, an STL routine must not enable the stimulus if it is not implemented). The discoverability is controlled through a type register which specifies if a particular stimulus feature is implemented in the system.

Note

SBISTC-Cortex-M33 might implement a physical register, but the stimulus could be non-implemented depending on the complexity of the system.

For details of the Fault Programmable Stimulus Payload registers, refer to the *SBIST Controller* appendix in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

2.3.11 Customizable functions

You can modify the following functions:

stl_preamble

Location

`tests/stl_shared/asm/gnuasm/stl_global_funcs.S`

Arguments

None

Return value

None

Registers used

R0, R1, R11, and LR

Description

This function initializes the watchdog timer.

stl_postamble

Location

`tests/stl_shared/asm/gnuasm/stl_global_funcs.S`

Arguments

None

Return value

None

Registers used

R0, R1, and LR

Description

This function checks if the part has passed or failed, based on the value in register R0. If the part passed, this function calls `stl_part_pass`, otherwise it calls `stl_fail`. If the part fails, this function enters the safe state by creating a WFI loop. This contains the fault to prevent UNPREDICTABLE behavior. However, you can choose to override this behavior depending on your requirements.

2.3.12 Processor configuration

The *Software Test Library* (STL) depends the following components of the Cortex-M33 r1p0 RTL package:

- RTL configuration
- SBIST-Controller and Trickbox
- SBIST specific HW-features

RTL configuration

The STL needs to be configured as per the RTL implementation. Failure to do so can potentially cause runtime issues when an incorrectly configured version of the STL is used on an incompatible RTL implementation. The following table shows the RTL and the equivalent legitimate parameters along with the STL files where these configurations can be set.

Note

The SBIST parameter needs to be set to 1 for the full STL functionality.

Table 2-8 RTL configuration details for STL

HW package	Values	RTL file	SW Package	STL files
FPU	0, 1	`TEAL_CONFIG.v`	`FPU_PRESENT = (FPU == 1 && CFGFPU == 1) ? 1: 0`	`Makefile` & `Makefile.asm`
CFGFPU	0, 1	`TEALMCU.v`		
DSP	0, 1	`TEAL_CONFIG.v`	`DSP_PRESENT = DSP`	`Makefile` & `Makefile.asm`
SECEXT	0, 1	`TEAL_CONFIG.v`	`SECEXT_PRESENT = SECEXT`	`Makefile` & `Makefile.asm`
CPIF	0, 1	`TEAL_CONFIG.v`	`CP_PRESENT = CPIF`	`Makefile` & `Makefile.asm`

Table 2-8 RTL configuration details for STL (continued)

HW package	Values	RTL file	SW Package	STL files

MPU_NS	0, 4, 8, 12, 16	`TEAL_CONFIG.v`	`MPUNS_PRESENT = (SECEXT == 1 && MPU_NS != 0)? 1 : 0`
			`MPUS_PRESENT = (SECEXT == 1 && MPU_S != 0)? 1 : (SECEXT == 0 && MPU_NS != 0)? 1 : 0`
				`Makefile` & `Makefile.asm`

MPU_S	0, 4, 8, 12, 16	`TEAL_CONFIG.v`		`Makefile` & `Makefile.asm`
SAU	0, 4, 8	`TEAL_CONFIG.v`	`SAU_PRESENT = (SAU != 0)? 1 : 0`	`Makefile` & `Makefile.asm`
NUMIRQ	1-480	`TEAL_CONFIG.v`	`NUMIRQ`	`Makefile` & `Makefile.asm`
WIC	0, 1	`TEAL_CONFIG.v`	`WIC_PRESENT = WIC`	`Makefile` & `Makefile.asm`
SBIST	1	`TEAL_CONFIG.v`	-	-
CFGBIGEND	0, 1	`TEALMCU.v`	`BIG_END = CFGBIGEND`	`Makefile` & `Makefile.asm`

SBIST-Controller and Trickbox:

The SBIST-Controller and the Trickbox are components used by STL routines to aid fault detection.

- The SBIST-Controller contains a series of memory mapped registers to report status and monitor deadlocks during STL execution.
- The trickbox contains a set of programmable stimulus features used by the STL for enhanced stimulus.

The HW integration details for the SBIST-Controller can be found in *SBIST Controller* appendix of the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

The STL can be configured to either use the SBIST-Controller and Trickbox or not rely on either. This can be done by building the appropriate STL implementation for the targeted platform. The following table shows the RTL implementation specifications mapped to the SW configurations.

Table 2-9

SBIST method	Compatible PSI	Equivalent SW-defines and macros	STL file
M0	0 (specified in `tealsbistpsi_defs.v`)	`sbist_method`	`stl_global_defs.h`
		`SBIST_M0`	`Makefile` & `Makefile.asm`
M1	0, 3 (specified in `tealsbistpsi_defs.v`)	`sbist_method`	`stl_global_defs.h`
		`SBIST_M1`	`Makefile` & `Makefile.asm`

Note

The STL is capable of reading the *Programmable Stimulus* (PSI) parameter at run time and does not need build-time configurations to infer the same.

2.4 STL code size and execution time

The system integrator and system developer should consider the memory resources and execution times required to integrate the Cortex-M33 processor *Software Test Library* (STL) in terms of its intended use in their system.

The following tables detail the memory resources used by the code files of the STL and the execution times:

- *Table 2-10 CORE memory resources and execution times* on page 2-50
- *Table 2-11 MPU memory resources and execution times* on page 2-51
- *Table 2-12 FPU memory resources and execution times* on page 2-51
- *Table 2-13 NVIC memory resources and execution times* on page 2-52
- *Table 2-14 Scheduler memory resources and execution times* on page 2-53
- *Table 2-15 Shared memory resources and execution times* on page 2-53
- *Table 2-16 Total STL memory resources* on page 2-54

Maximum stack usage is 512 bytes.

Note

No dynamic memory allocation is implemented.

Table 2-10 CORE memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_core_ris_p001_n001`	2846	0	0	1	1765	-
`stl_core_ris_p002_n001`	1648	0	0	1	5948	-
`stl_core_dataproc_p003_n001`	876	0	0	1	2841	-
`stl_core_decoder_p004_n001`	2960	0	0	1	1526	-
`stl_core_dpu_p005_n001`	776	0	0	1	335	-
`stl_core_ris_p007_n001`	2880	0	0	1	1779	-
`stl_core_ris_p008_n001`	2950	0	0	1	1819	-
`stl_core_dataproc_p009_n001`	628	0	0	1	250	-
`stl_core_ris_p010_n001`	1464	0	0	1	847	-
`stl_core_save`	102	0	0	N/A	N/A	-
`stl_core_restore`	96	0	0	N/A	N/A	-
Total CORE	17226	0	0	N/A	N/A	-

Table 2-11 MPU memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_mpu_p001_n002`	420	0	0	2	1237	Longest iteration takes 662 clock cycles
`stl_mpu_p002_n032`	228	0	0	32	9360	Longest iteration takes 525 clock cycles
`stl_mpu_p003_n008`	182	0	0	8	2292	Longest iteration takes 511 clock cycles
`stl_mpu_p004_n032`	404	0	0	32	39136	Longest iteration takes 2391 clock cycles
`stl_mpu_p005_n032`	400	0	0	32	37952	Longest iteration takes 2309 clock cycles
`stl_mpu_p006_n016`	362	0	0	16	17664	Longest iteration takes 2146 clock cycles
`stl_mpu_p007_n256`	980	0	0	256	279488	Longest iteration takes 2886 clock cycles
`stl_mpu_p008_n256`	952	0	0	256	286656	Longest iteration takes 2832 clock cycles
`stl_mpu_p009_n128`	740	0	0	128	115136	Longest iteration takes 1743 clock cycles
`stl_mpu_save`	46	0	0	N/A	N/A	-
`stl_mpu_restore`	50	0	0	N/A	N/A	-
Total MPU	4764	0	0	N/A	N/A	-

Table 2-12 FPU memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_fpu_reg_p001_n001`	276	0	0	1	788	-
`stl_fpu_reg_p002_n001`	532	0	0	1	293	-
`stl_fpu_dataproc_p003_n001`	372	0	0	1	689	-
`stl_fpu_dataproc_p004_n001`	1204	0	0	1	3629	-
`stl_fpu_context_p005_n001`	268	0	0	1	1652	-
`stl_fpu_div2nd_p006_n001`	496	0	0	1	844	-
`stl_fpu_divsqrt_p007_n001`	176	0	0	1	337	-
`stl_fpu_maxmin_p008_n001`	384	0	0	1	418	-
`stl_fpu_vadd_p009_n001`	280	0	0	1	3022	-
`stl_fpu_vcvt_p010_n001`	644	0	0	1	539	-
`stl_fpu_vfma_p011_n001`	2048	0	0	1	2906	-

Table 2-12 FPU memory resources and execution times (continued)

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_fpu_vfms_p012_n001`	724	0	0	1	4436	-
`stl_fpu_vfnma_p013_n001`	1040	0	0	1	1389	-
`stl_fpu_vmls_p014_n001`	636	0	0	1	3145	-
`stl_fpu_vnmla_p015_n001`	644	0	0	1	828	-
`stl_fpu_vsqrt_p016_n001`	212	0	0	1	3960	-
`stl_fpu_vnmls_p017_n001`	512	0	0	1	4518	-
`stl_fpu_round_p018_n001`	1596	0	0	1	2518	-
`stl_fpu_clz32b_p019_n001`	920	0	0	1	1403	-
`stl_fpu_clz32a_p020_n001`	452	0	0	1	435	-
`stl_fpu_vcmp_p021_n001`	588	0	0	1	622	-
`stl_fpu_lza_p022_n001`	576	0	0	1	775	-
`stl_fpu_save`	90	0	0	N/A	N/A	-
`stl_fpu_restore`	44	0	0	N/A	N/A	-
Total FPU	14714	0	0	N/A	N/A	-

Table 2-13 NVIC memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_nvic_regro_p001_n001`	248	0	0	1	110	-
`stl_nvic_regro_p002_n001`	68	0	0	1	42	-
`stl_nvic_fpuro_p003_n001`	244	0	0	1	79	-
`stl_nvic_statusregro_p004_n001`	168	0	0	1	90	-
`stl_nvic_dbglvlro_p005_n001`	356	0	0	1	102	-
`stl_nvic_secextro_p006_n001`	304	0	0	1	88	-
`stl_nvic_secextdbglvlro_p007_n001`	368	0	0	1	106	-
`stl_nvic_regrw_p008_n001`	784	0	0	1	704	-
`stl_nvic_regrw_p009_n001`	1184	0	0	1	727	-
`stl_nvic_regrw_p010_n001`	268	0	0	1	10611	-
`stl_nvic_secextrw_p011_n001`	756	0	0	1	694	-
`stl_nvic_dbglvlrw_p012_n001`	176	0	0	1	130	-
`stl_nvic_internalexcep_p014_n001`	472	0	0	1	1595	-
`stl_nvic_systick_p015_n001`	400	0	0	1	1397	-

Table 2-13 NVIC memory resources and execution times (continued)

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_nvic_wic_p016_n001`	1508	480	0	1	10536	-
`stl_nvic_core_decoder_p017_n001`	592	0	0	1	734	-
`stl_nvic_exceptree_p018_n001`	544	0	0	1	6202	-
`stl_nvic_exceptree_ns_p019_n001`	464	0	0	1	6097	-
`stl_nvic_actvtree_p020_n001`	404	0	0	1	35418	-
`stl_nvic_wfecheck_p021_n001`	404	0	0	1	16753	-
`stl_nvic_fsm_p022_n001`	404	0	0	1	19274	-
`stl_nvic_save`	228	0	0	N/A	N/A	-
`stl_nvic_restore`	240	0	0	N/A	N/A	-
Total NVIC	10584	480	0	N/A	N/A	-

Table 2-14 Scheduler memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_scheduler_asm`	188	0	0	N/A	N/A	-
`stl_scheduler`	1394	0	0	N/A	N/A	-
`stl_scheduler_init`	102	0	16	N/A	N/A	-
`stl_scheduler_m33`	4134	0	520	N/A	N/A	-
`stl_scheduler_dfa`	90	0	0	N/A	N/A	-
`stl_scheduler`	1324	0	16	N/A	N/A	-
`stl_test_suite_init`	2428	0	0	N/A	N/A	-
`stl_scheduler_tests_table`	0	0	886	N/A	N/A	-
Total Scheduler	9660	0	1438	N/A	N/A	-

Table 2-15 Shared memory resources and execution times

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_vectors`	532	0	0	N/A	N/A	-
`stl_core_handler_funcs`	16	0	0	N/A	N/A	-
`stl_fpu_handlers_funcs`	632	0	0	N/A	N/A	-
`stl_mpu_handler_funcs`	684	0	0	N/A	N/A	-

Table 2-15 Shared memory resources and execution times (continued)

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
`stl_nvic_handler_funcs`	248	0	0	N/A	N/A	-
`stl_common_core_func`	148	0	0	N/A	N/A	-
`stl_common_fpu_func`	438	0	0	N/A	N/A	-
`stl_common_mpu_func`	2204	0	128	N/A	N/A	-
`stl_common_nvic_func`	840	0	0	N/A	N/A	-
`stl_common_nvic_func_intr`	1856	0	0	N/A	N/A	-
`stl_common_func`	564	0	0	N/A	N/A	-
`stl_global_funcs`	56	0	0	N/A	N/A	-
Total Shared	8216	0	128	N/A	N/A	-

Table 2-16 Total STL memory resources

Object	ROM		RAM	Iterations	Runtime (clock cycles)	Notes
	Code + RO Data (bytes)	Data (bytes)	RW data + ZI Data (bytes)			
Total STL	65164	480	1566	N/A	N/A	-

Note

These code-size and execution time measurements are performed under the following conditions:

- HW configurations as specified by the *Arm® Cortex®-M33 Processor STL Safety Manual*.
- Pin configurations as specified in the reference execution testbench in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.
- Memory latency shown in the reference execution testbench.
- Software configurations as specified in *2.3.9 STL configuration* on page 2-40.

2.5 STL setup

The *Software Test Library* (STL) software must save and restore configurations when necessary, and requires full access to registers.

Follow the integration guidelines in *2.5.1 Cortex®-M33 STL integration guidelines* on page 2-55.

2.5.1 Cortex®-M33 STL integration guidelines

Use the following guidelines to integrate the *Software Test Library* (STL) software.

Environment for the `M33_STL` function call

1. Include STL header files as shown in the `stl_test_suite`, such as namely `stl_scheduler.h`.
2. Create a variable to hold the result of the test, `uint32_t test_status`, then assign the variable to the `M33_STL` function. This is shown in the following *Integration example* on page 2-55.
3. Call `M33_STL_Init(..)`, and `M33_STL(..)`, as shown in the following *Integration example* on page 2-55.

Integration example

The following code is an example integration of the `M33_STL` function. It is similar, but not identical, to the code that is in the file `stl_test_suite.c` that Arm provides in the STL software package.

```
#include "stl_test_suite.h"
int main (void) {
    int test_status = 0;
    int oor_test_part_start = (START_POS_STESTS_OOR != -1)? START_POS_STESTS_OOR : 0;
    int oor_part_max        = LAST_POS_STESTS_OOR + 1;
    int oor_part_elem_num   = NUM_STESTS_OOR;
    int olt_test_part_start = (START_POS_STESTS_OLT != -1)? START_POS_STESTS_OLT : 0;
    int olt_part_elem_num   = NUM_PARTS_GRP_OLT;
    int olt_part_max        = LAST_POS_STESTS_OLT + 1;
    int ole_test_part_start = (START_POS_STESTS_OLE != -1)? START_POS_STESTS_OLE : 0;
    int ole_part_max        = LAST_POS_STESTS_OLE + 1;


    setup_table(); // One time call to ensure test table RW-data structures are initialized


    // Call STL in OoR mode
    // It is assumed that external IRQs are not enabled
    if (M33_STL_Init(oor_test_part_start, oor_part_max)) {
        test_mode = TMODE_OOR;  // TMODE_OOR = 0x1
        test_status = M33_STL(oor_part_elem_num, test_mode);
    }
    // Call STL in OLT mode
    // It is assumed that no external IRQs will interrupt the STL
    if (M33_STL_Init(olt_test_part_start, olt_part_max)) {
        test_mode = TMODE_OL_TIME;  // TMODE_OL_TIME = 0x2
        test_status = M33_STL(olt_part_elem_num, test_mode);
    }
    // Call STL in OLE mode
    // External IRQs can be enabled, but need to be masked by setting primask. The STL
figures out the presence of a pending IRQ and exits as soon as context restore is complete.
    if (M33_STL_Init(ole_test_part_start, ole_part_max)) {
        test_mode = TMODE_OL_EVENT; // TMODE_OL_EVENT = 0x3
        set_primask();
        test_status = M33_STL(ole_part_elem_num, test_mode);
    }
}
```

In this example, `main` is the main example test case `main.c`, which shows how the scheduler is called and also shows that all parts supplied in the STL software deliverable are complete and have run.

To detect faults, after the `M33_STL` function returns, check the result output value as shown in this example; `test_status = 0` reflects no dangerous faults detected, while `test_status = 1` indicates the presence of potentially dangerous functional failures.

Note

The STL is programmed to trap any potentially dangerous condition within its execution as safe context restore cannot be guaranteed. This results in `test-status` not being returned.

2.5.2 STL code integration

To integrate the STL code.

1. Before calling the *Software Test Library* (STL) software, you must configure the stack for all the STL parts.
2. If you use the verification example system that is described in *4.2 Setting up the Cortex®-M33 processor STL OoB test environment* on page 4-69, you must specify the memory map of an image to the linker file, for example, through the scatter file, `stl_test_suite.scf`. The scatter file describes to the linker file how to place the vector table, application code, data, stacks, and heap at suitable addresses in the memory map. The scatter file is used in the Arm compiler flow.
3. Integrate the STL code as shown in the following example.

```
#! armclang --target=armv8-arm-none-eabi -E -x c
#include "../../stl_shared/include/gnuasm/stl_global_defs.h"
#include "../../stl_shared/include/gnuasm/stl_global_macros.h"


LOAD_CODE code_region code_region_size
{
    TEST_VECTORS +0
    {
        stl_vectors.o(.init, +First)
        stl_core_handler_funcs.o(+RO)
        stl_mpu_handler_funcs.o(+RO)
        stl_fpu_handler_funcs.o(+RO)
        stl_nvic_handler_funcs.o(+RO)
    }
    TEST_COMMON_PARTS +0
    {
        stl_global_funcs.o(+RO)
        stl_common_func.o(+RO)
        stl_common_mpu_func.o(+RO)
        stl_common_fpu_func.o(+RO)
        stl_common_nvic_func.o(+RO)
        stl_common_nvic_func_intr.o(+RO)
        stl_common_core_func.o(+RO)
    }
    TEST_STL_TEST_SUITE +0
    {
        stl_test_suite*.o(+RO)
    }
    TEST_SCHEDULER +0
    {
        stl_scheduler*.o(+RO)
        stl_test_suite_init.o(+RO)
    }
    TEST_CORE_PARTS +0
    {
        stl_core_*_p*_n*.o(+RO)
        stl_core_save.o(+RO)
        stl_core_restore.o(+RO)
    }
    TEST_MMS_PARTS +0
    {
        stl_mpu_p*_n*.o(+RO)
        stl_mpu_save.o(+RO)
        stl_mpu_restore.o(+RO)
    }
    TEST_FPU_PARTS +0
    {
        stl_fpu_*_p*_n*.o(+RO)
        stl_fpu_save.o(+RO)
        stl_fpu_restore.o(+RO)
    }
    TEST_NVIC_PARTS +0
    {
        stl_nvic_*_p*_n*.o(+RO)
        stl_nvic_save.o(+RO)
        stl_nvic_restore.o(+RO)
    }
    TEST_MISC +0
    {
          *.o (+RO)
    }
}
LOAD_CODE_NSC nsc_code_region nsc_code_region_size
{
    TEST_NVIC_PARTS_NSC +0
    {
        stl_nvic_funcs_nsc.o(+RO)
    }
}
LOAD_CODE_NS ns_code_region ns_code_region_size
{
    TEST_VECTORS_NS +0
    {
        stl_vectors_ns.o(.init_ns, +RO)
        stl_nvic_handler_funcs_ns.o(+RO)
    }
    TEST_MMS_PARTS_NS +0
    {
        stl_mpu_funcs_ns.o(+RO)
    }
    TEST_NVIC_PARTS_NS +0
    {
        stl_nvic_nsregion_ns.o(+RO)
        stl_nvic_wic_ns.o(+RO)
    }
}


LOAD_DATA data_region data_region_size
{
    ARM_LIB_HEAP +0
    {
       stl_common_data.o(.data.heap)
    }
    TEST_NVIC_DATA +0
    {
       stl_nvic_data.o(+RW)
    }
    TEST_MPU_DATA +0
    {
        stl_mpu_data.o(.data)
    }
    TEST_MISC_DATA +0
    {
       *.o(+RW)
       *.o(+ZI)
    }
}
;; System Data - This is a dependency on the system
;; If this is being integrated into a system-level environment, do not include this load
region
LOAD_SYSTEM_DATA system_space system_space_size
{
    ARM_LIB_STACK +0
    {
       stl_system_data.o(.data.msp)
       stl_system_data.o(.data.psp)
    }
}
LOAD_DATA_NS ns_data_region ns_data_region_size
{
    TEST_COMMON_DATA_NS +0
    {
       stl_common_data_ns.o(.data, +RW)
    }
    TEST_MPU_DATA_NS +0
    {
        stl_mpu_data_ns.o(.data)
    }
}
;; System Data (NS) - This is a dependency on the system
;; If this is being integrated into a system-level environment, do not include this load
region
LOAD_SYSTEM_DATA_NS +0
{
    ARM_LIB_STACK_NS +0
    {
        stl_system_data_ns.o(.data.msp_ns)
        stl_system_data_ns.o(.data.psp_ns)
    }
}



#ifdef scratch_space_1
CREATE_MPU_REGS(1)
#endif
#ifdef scratch_space_2
CREATE_MPU_REGS(2)
#endif
#ifdef scratch_space_3
CREATE_MPU_REGS(3)
#endif
#ifdef scratch_space_4
CREATE_MPU_REGS(4)
#endif
#ifdef scratch_space_5
CREATE_MPU_REGS(5)
#endif
#ifdef scratch_space_6
CREATE_MPU_REGS(6)
#endif
#ifdef scratch_space_7
CREATE_MPU_REGS(7)
#endif
#ifdef scratch_space_8
CREATE_MPU_REGS(8)
#endif
#ifdef scratch_space_9
CREATE_MPU_REGS(9)
#endif
#ifdef scratch_space_10
CREATE_MPU_REGS(10)
#endif
#ifdef scratch_space_11
CREATE_MPU_REGS(11)
#endif
#ifdef scratch_space_12
CREATE_MPU_REGS(12)
#endif
#ifdef scratch_space_13
CREATE_MPU_REGS(13)
#endif
#ifdef scratch_space_14
CREATE_MPU_REGS(14)
#endif
#ifdef scratch_space_15
CREATE_MPU_REGS(15)
#endif
#ifdef scratch_space_16
CREATE_MPU_REGS(16)
#endif
```

2.5.3 Initialization

Before calling the STL scheduler, you must call the initialize function `M33_STL_Init()`.

The initialize function:
- Initializes the part number pointers.
- Initializes the fault registers.
- Checks the maximum part number.

Valid values of part number pointers are typically different for each test mode. Therefore, when you change the test mode (OOR, OLT, OLE), or the part number pointers, you must call the initialize function before the STL scheduler function. The part number pointers that the initialization function uses must be valid for the test mode in which the STL scheduler is called. The part number pointers define the range, from start to end, of parts that are run by the STL scheduler.

The following example code uses a part list in which the first 50 parts only support OOR mode. The next 50 parts only support OLT mode, and the last 50 parts only support OLE mode. Each part runs for one iteration only.

Example code

```
// OOR mode
valid = M33_STL_Init(0, 49);
if (valid == 1)
{
// This will run parts 0-49 once
M33_STL(50, OOR);
}

// OLT mode
valid = M33_STL_Init(50, 99);
if (valid == 1)
{
// This will run parts 50-99 twice
M33_STL(100, OLT);
}

// OLE mode
valid = M33_STL_Init(100, 149);
if (valid == 1)
{
// This will run parts 100-149 thrice
M33_STL(150, OLE);
}
```

The following example code shows an invalid initialization for the part list shown in the example code.

```
valid = M33_STL_Init(0, 149);
if (valid == 1)
{
M33_STL(50, OOR);
}
```

The following table describes the parameters that are passed to the initialization function.

Table 2-17 Scheduler initialization parameters

Parameter	Type	Description
`part_start_num`	Input	Indicates the number of the first part in the sequence. This value must be valid for the test mode in which you call the STL scheduler. `part_start_num` must be less than or equal to `START_POS_STESTS_<MODE>`. `START_POS_STESTS_<MODE>` is defined in `sbist_scheduler_test_table_config.h`.
`part_max_num`	Input	Indicates the last part number that you want to run. This value must be valid for the test mode in which you call STL scheduler. `part_max_num` must be less than or equal to `LAST_POS_STESTS_<MODE>` + 1. `LAST_POS_STESTS_<MODE>` is defined in `sbist_scheduler_test_table_config.h`.
`valid_init`	Output	Indicates if initialization is valid: 0 = Initialization failed. 1 = Initialization passed. **Note** Before checking initialization validity, the values of `part_start_num` and `part_max_num` are checked to ensure that they are not NULL. If initialization fails, you must not call the STL scheduler.

Chapter 3
Faults

This chapter introduces the concepts of faults and fault control.

The chapter also describes the fault list generated for the Cortex-M33 processor, and methods to categorize and determine the status of a fault which forms a part of the final fault report. The chapter also helps you to understand the fault grading report that is generated by the *Software Test Library* (STL) running on the Cortex-M33 processor.

It contains the following sections:

3.1 STL fault model

A fault model is expected to emulate the behavior of a real-life defect. Therefore, a part that is defective because of manufacturing or aging can be represented in a simulation environment.

Logical fault model examples

There are several types of physical faults that map to one or more logical faults. These logical faults can be modeled.

Stuck-at faults

Stuck-at faults are inputs or output of a gate or net that are stuck-at 0 or stuck-at 1. A stuck-at fault model ties the value of the input or the output of a gate LOW or HIGH for the complete duration of the simulation.

Transition delay faults

Transition delay faults are caused by signal transitions at gate inputs or outputs, which are slow to rise or slow to fall. The critical path might become slower because of transition delay faults.

Bridging faults

Bridge or bridging faults occur when two normally separate logic signals become connected together.

Note

The only logical fault model supported by the *Software Test Library* (STL) software running on the Cortex-M33 processor is the stuck-at fault model.

3.1.1 Stuck-at faults

The stuck-at fault model that is used assumes that any physical effect can only lead to two types of logical faults, namely stuck-at 0 and stuck-at 1 faults.

This section provides examples of stuck-at 1 faults. The *Software Test Library* (STL) aims to activate and propagate the fault to an observation point.

To simulate a net fault, a single stuck-at 1 fault can be inserted on the connection between two gates as shown in the following figure.

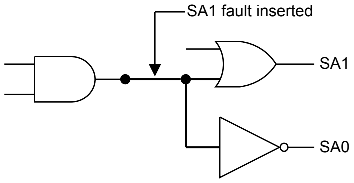


Figure 3-1 Example of a stuck-at 1 fault on a net

To simulate an input fault, a single stuck-at 1 fault can be inserted on the input of a gate as shown in the following figure.

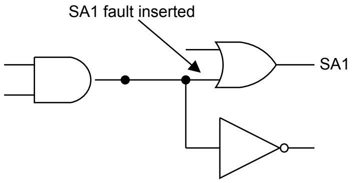


Figure 3-2 Example of a stuck-at 1 fault on an input of a gate

To simulate an output fault, a single stuck-at 1 fault can be inserted on the output of a preceding gate as shown in the following figure.

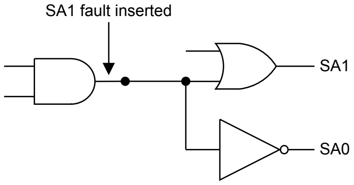


Figure 3-3 Example of a stuck-at 1 fault on the output of a gate

A fault can also be inserted on the inputs and output of buffers or on the inputs of registers. To provide information that is useful for failure mode analysis, a fault list is separately created by inserting stuck-at faults on the output of registers.

For the STL package deliverable, the only fault model simulated is a single stuck-at fault on the inputs or the output of a gate.

A fault on a net can provide optimistic results as it simulates a stuck-at value on the inputs of all gates the net connects to, assuming only net faults are selected to generate a fault list. Therefore, simulating a stuck-at fault on the inputs of a gate only covers the input and not the net that is connected to the input. The output of a gate must be simulated to model the behavior of the following inputs that are stuck-at 1 or stuck-at 0. This simulation of gates on the inputs or the output is called a node or port fault simulation. Performing net fault simulation does not provide the same coverage as port faults. However, performing port fault simulations can also cover net faults.

There are two types of fault list:

- A list containing ports of cells. For example, gates, registers, and buffers.
- A list containing outputs of registers for failure mode analysis.

Note

This STL package deliverable does not support a multiple stuck-at fault model.

3.2 Fault status

Arm relies on the default fault status defined by the simulator tool, for more information about the fault statuses see the *Z01X Simulator Safety Verification User Guide*.

3.3 Fault list

A single stuck-at fault model is used to generate a fault universe on logic. The fault universe contains all possible single stuck-at faults that can occur in the Cortex-M33 processor. The following figure shows the fault universe.

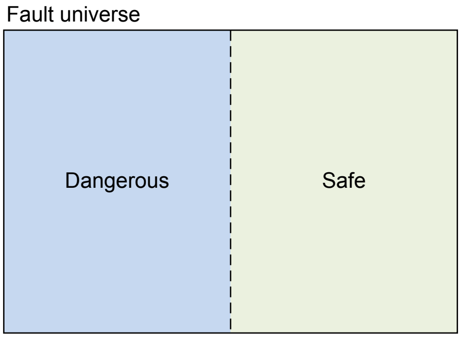


Figure 3-4 Fault universe

The *Software Test Library* (STL) only covers the dangerous fault subset shown in the figure. Dangerous faults are those that can lead to incorrect execution of software. Therefore, the fault universe is segregated into safe faults and dangerous faults. By default, any fault that is not categorized as safe is a dangerous fault.

You can specify the fault model as stuck-at 0 or stuck-at 1, or both, on ports of a cell using a standard fault simulator to generate the fault universe.

For more information on categories of safe faults and safe fault analysis, see the *Arm® Cortex®-M33 Processor STL Safety Manual*.

3.4 Categories of fault simulation

In a fault simulation, there are two categories of the *Design Under Test* (DUT).

The two categories of DUT are:

Good machine simulation
A simulation which covers the actual behavior of the circuit or DUT

Faulty machine simulation
A simulation which covers the behavior of the circuit or DUT with the stuck-at fault injected

The good and faulty machine simulations can run separately or concurrently on the same machine. The results of the two simulations are compared to determine if a fault is detected or not.

In the processor testbench setup, a logic simulation is first performed to determine the correctness of the test, that is, the test must complete successfully without flagging a fault.

A fault simulation determines if the *Software Test Library* (STL) part can detect the fault if the strobed location holds an unexpected value.

3.5 Fault grading

The Cortex-M33 processor is fault graded using the EDA tool Synopsys Z01X.

3.5.1 Collapsed faults

To optimize the fault grading execution time, a fault simulator performs fault collapsing and only simulates prime faults. A fault that represents one or more faults is called a prime fault. A fault that has the same observable effect as its prime fault is called a collapsed fault. A fault simulator typically performs the step of fault collapsing to only simulate prime faults to improve the efficiency of simulations.

Examples of collapsed faults include:

- A stuck-at fault on the output of a gate that has a fan-out to only one other gate
- A stuck-at fault on the input of a buffer that can be collapsed with the output
- An output of a module that connects through the hierarchy to an input of a gate in another module

3.5.2 Reporting

Determining the number of faults that a test detects is one of the parts of the fault grading process called reporting.

For more information, refer to *About STL fault grading* in the *Arm® Cortex®-M33 Processor STL Safety Manual*.

Chapter 4
STL Out-of-Box tests

This chapter describes how to check if the *Software Test Library* (STL) deliverables have been set up correctly, test the STL functionality, and to fault grade the processor.

It contains the following sections:

4.1 About the STL OoB tests

After you have installed the product bundle, Arm recommends that you run the *Out-of-Box* (OoB) tests.

The OoB test procedures use the execution testbench. The execution testbench is a platform, which enables you to run the supplied OoB tests, and which indicates whether all the parts intended have downloaded and run successfully.

The testbench is not intended to be used for full verification, but it can demonstrate the functionality of the *Software Test Library* (STL) software.

The testbench is part of the *Integration Kit* (IK) that is in the IP deliverable supplied by Arm with the Cortex-M33 (r1p0) processor. If the STL software is supplied by Arm without the processor, you must use the IK that is already installed on your system.

Arm recommends validating the RTL supplied, before you start RTL configuration, to check that you have unpacked the testbench correctly. To do this, see the chapter *Checking your RTL* in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

4.2 Setting up the Cortex®-M33 processor STL OoB test environment

The *Software Test Library* (STL) OoB test environment requires the Cortex-M33 processor and the resources listed in *1.4 STL tool resources* on page 1-15.

Before you can run the STL OoB tests, the Cortex-M33 example system must be built correctly, and simulation tests run successfully, as described in the *Arm® Cortex®-M33 Processor Integration and Implementation Manual*.

4.3 STL OoB tests

Arm supplies *Software Test Library* (STL) *Out-of-Box* (OoB) tests that you can run with the Cortex-M33 processor. These tests also show that the STL deliverables are set up correctly.

The STL deliverables include test source code and Makefiles to compile the tests. See *2.3.7 Makefiles* on page 2-39 for more information.

The tests are organized in the `stl/tests` directory as follows:

- A directory called `stl_link`, which contains the linker script that the Makefile uses. For more information, see *2.3.8 Linker scripts* on page 2-39.
- A directory called `stl_scheduler`. For more information on the scheduler, see *2.3.5 STL scheduler* on page 2-32.
- A directory called `stl_<block>`, which contains files of format `stl_<block>_<optional>_p<parts>_n<iter>.S`, where `<block>` is either `core`, `fpu`, `mpu`, or `nvic`.
- A directory called `stl_test_suite`, which contains the code file `stl_test_suite.c`. This file is the main example test case. Run this test to show how the scheduler is called and to ensure that all parts supplied in the STL software deliverable are complete and have run.
- A directory called `stl_shared`, which contains common test source files that are used for the tests.

For more information on all the directories, see *2.3.6 STL release directory structure* on page 2-39.

4.4 Flow for validating STL

This section describes how to build, and run tests on, the simulation environment for your STL functionality.

The following figure shows the build and validate flow for a test.

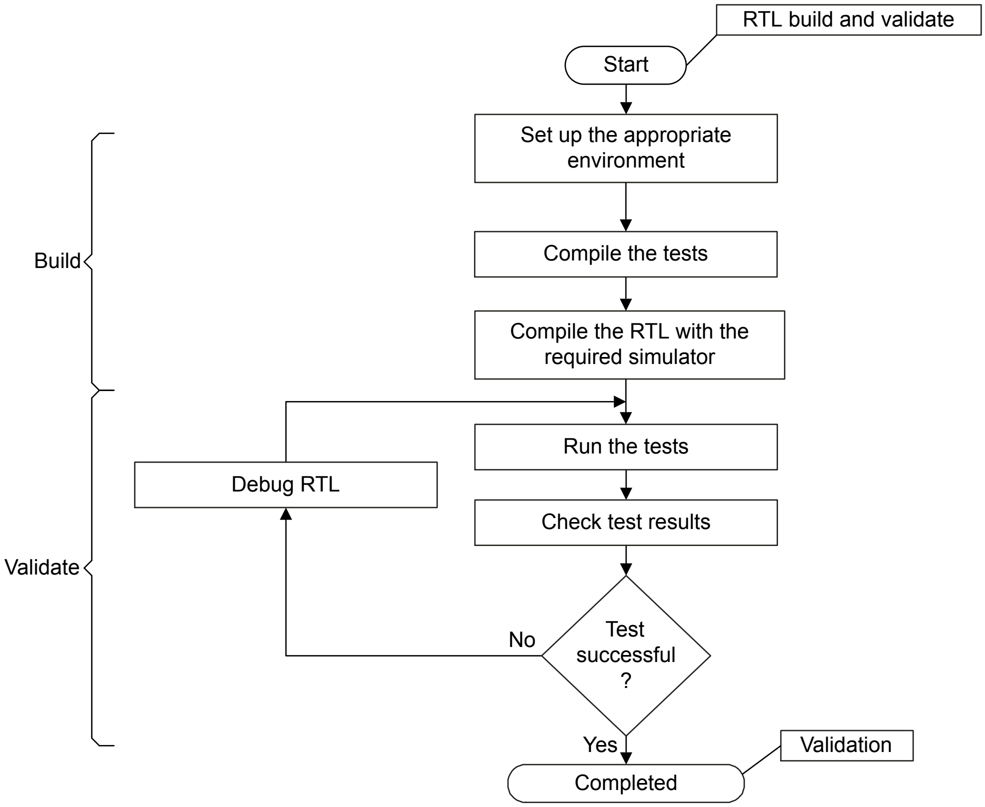


Figure 4-1 Build and validate flow

4.4.1 Prerequisites for compiling and running the example system

STL parts are run using the Integration Kit.

The testbench is part of the Integration Kit that is in the IP deliverable supplied by Arm with the Cortex-M33 processor. If the STL software is supplied by Arm without the processor, you must use the example system that is already installed on your system.

Before using this environment, you must ensure that the following tools are located in the execution path:

- The Arm compiler tools
- The simulation tools

4.5 Running the Cortex®-M33 processor STL OoB tests

To run the *Software Test Library* (STL) software OoB tests:

1. Ensure that all the necessary tools are loaded and available.
2. Create a testing directory:

   ```
   mkdir testing
   cd testing
   ```
3. Extract the release bundle into the testing directory as described in the release notes.
4. Copy the CPU hardware under the testing directory:

   ```
   cp -r ~/<path>/AT623-* && cp -r /<path>/AT-624-*
   ```
5. Copy the STL package to the testing directory:

   ```
   cp -r ~/<path>/MT098-* cp -r /<path>/AT-624-*
   ```
6. Extract the contents of these packages and set the environment variable `M33_HOME` to point to the extracted hardware package. This should be an absolute path.
7. Compile STL:

   ```
   cd stl/sim
   <bsub> make SBIST_METHOD=<M1/M0> <HW-ConfigOptions>
   ```

   Where, `<HWConfigOptions>` must reflect the hardware (RTL) configuration that is rendered. The options are:

   FPU_PRESENT=<0,1> 1 is the default

   DSP_PRESENT=<0,1> 1 is the default

   SECEXT_PRESENT=<0,1> 1 is the default

   CP_PRESENT=<0,1> 1 is the default

   MPUNS_PRESENT=<0,1> 1 is the default

   MPU_PRESENT=<0,1> 1 is the default

   SAU_PRESENT=<0,1> 1 is the default

   WIC_PRESENT=<0,1> 1 is the default

   NUMIRQ=<1,480> 32 is the default

   BIG_END=<0,1> 0 is the default.
8. Copy the generated binaries and map files in `stl/tests` to `$(M33_HOME)/logical/testbench/execution_tb/tests`
9. Run the logic simulation:

   ```
   cd $M33_HOME/logical/testbench/execution_tb
   <bsub> make run TESTNAME=stl_test_suite SBIST_METHOD=<M1/M0> SIM=<mti/vcs/ius> GUI=<no/
   yes>
   ```

   ——— **Note** ———

   You should see the following message if the command is successful:

   ```
   ** TEST PASSED **
   ```

   ————————

——— **Note** ———

Return to the original state of the software package by removing all the object files, binaries, and related test files:

```
cd testing/stl/sim
make clean
```

The STL object files must recompiled after returning to the original state before switching between M0 and M1 methods.

Chapter 5
Fault simulation

This chapter describes the activities required to run a fault simulation.

It contains the following sections:

5.1 Fault simulation files

The files relating to fault simulation are found in the `stl/faultsim` directory.

faultsim directory files

The following table describes the files in the `faultsim` directory.

Table 5-1 Files in the faultsim directory

File	Description
`zoix.vc`	VC file containing the necessary defines for Z01X
`sbist_fmsh_cmd.lst`	SFF script which specifies the commands for the Fault Manager. For more information on SFF, refer to *Synopsys Z01X User Guide*.
`bsub.sh`	Script used by `sbist_fmsh_cmd.lst` to launch child jobs for fault simulation
`cortexm33_net.sff`	SFF file which contains the faults location within the design. For more information on SFF, refer to *Synopsys Z01X User Guide*.
`setup.sh`	Script to set up the fault sim

5.2 Fault simulation environment

For fault simulation, the existing Cortex-M33 execution testbench is updated to enable fault detection and grading when running the *Software Test Library* (STL).

These features are enabled by using the `vc` files provided along with the execution testbench and the STL package. The `vc` files include the sbist.vc/ sbistc.vc file, delivered along with the processor package, and the zoix.vc file, delivered in the STL software package.

The fault simulation uses the SBIST-Controller as a watchdog and the FCTLR.STATUS field as strobes for detecting faults. This setup can be found in `logical/tealsbistc/verilog/teal_sbistc_wrapper_ctl.v`.

The following figure is an example Cortex-M33 testbench structure.

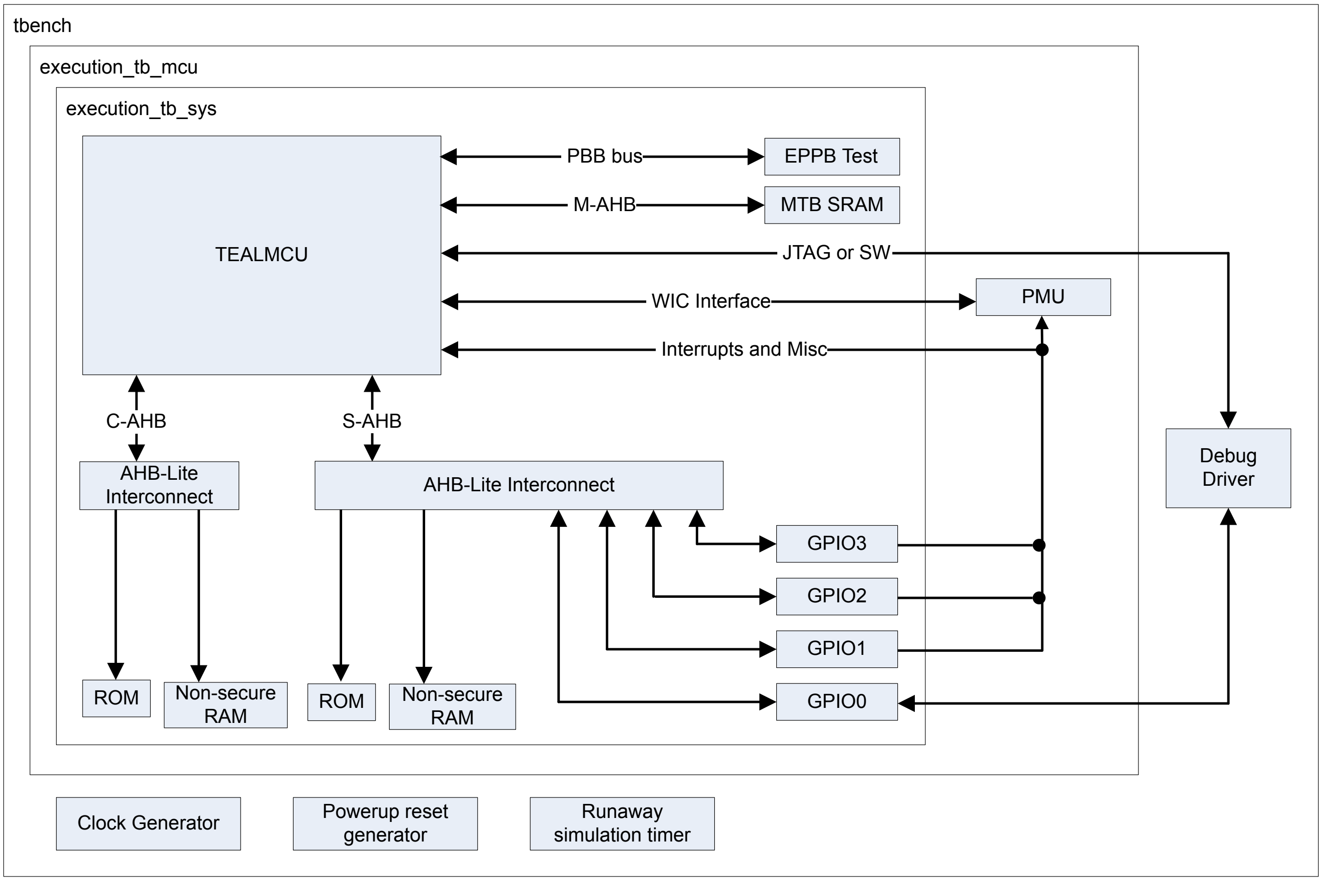


Figure 5-1 Example Cortex-M33 testbench

5.3 Performing fault simulation

This section describes the steps that are required for performing a Z01X fault simulation on the Cortex-M33 processor.

Note

- Ensure that you have performed the steps mentioned in *Chapter 4 STL Out-of-Box tests* on page 4-67 before performing fault simulation.
- Use Z01X v2017.12-SP2 for fault simulation.

Setup

1. Ensure that a netlist directory exists in the execution testbench:

   ```
   mkdir $M33_HOME/logical/testbench/execution_tb/netlist
   ```

2. For fault simulation, a top-level netlist must be generated for the Cortex-M33 processor. This netlist must be placed at `$M33_HOME/logical/testbench/execution_tb/netlist`, along with the necessary verilog library files for compilation.
3. Build and test the individual unit-tests `stl_core`, `stl_fpu`, `stl_mpu`, `stl_nvic` using the following commands:

   ```
   cd stl/sim/
   <bsub> make -f Makefile.asm TEST=<stl_core/stl_fpu/stl_mpu/stl_nvic> SBIST_METHOD=<M0/M1>
   cd $M33_HOME/logical/testbench/execution_tb/
   <bsub> make run TESTNAME=<stl_core/stl_fpu/stl_mpu/stl_nvic> SBIST_METHOD=<M0/M1>
   SIM=<MTI/VCS/IUS> GUI=<NO/YES>
   ```

4. Ensure that you are able to run logical simulations on the netlist with the supported compilers in the integration kit:

   ```
   cd $M33_HOME/logical/testbench/execution_tb/
   ```

   ```
   <bsub> make run TESTNAME=<stl_core/stl_fpu/stl_mpu/stl_nvic> SBIST_METHOD=<M1/M0>
   NETLIST=yes
   ```

5. Copy the fault-simulation essentials from `stl/faultsim` as shown below:

   ```
   cp stl/faultsim/zoix.vc $(M33_HOME)/logical/testbench/execution_tb/verilog
   ```

   ```
   cp stl/faultsim/bsub.sh $(M33_HOME)/logical/testbench/execution_tb
   ```

   ```
   cp stl/faultsim/cortexm33_net.sff $(M33_HOME)/logical/testbench/execution_tb
   ```

   ```
   cp stl/faultsim/sbist_fmsh_cmd.lst $(M33_HOME)/logical/testbench/execution_tb
   ```

6. Compile the netlist with Z01X and run logical simulations:

   ```
   cd $(M33_HOME)/logical/testbench/execution_tb
   ```

   If, in step 4, SBIST_METHOD=M0:

   ```
   <bsub> zoix -f verilog/sbist.vc -f verilog/zoix.vc -f verilog/execution_tb.vc -f verilog/
   netlist.vc +fault+var +suppress+cell +nolibcell +notimingchecks +nospecify verilog/
   tbench.v
   <bsub> ./zoix.sim +rom_image=tests/<stl_core/stl_fpu/stl_mpu/stl_nvic>.bin -maxt
   20000000000
   ```

   If, in step 4, SBIST_METHOD=M1:

   ```
   <bsub> zoix -f verilog/sbistc.vc -f verilog/zoix.vc -f verilog/execution_tb.vc -f verilog/
   netlist.vc +fault+var +suppress+cell +nolibcell +notimingchecks +nospecify verilog/
   tbench.v
   <bsub> ./zoix.sim +rom_image=tests/<stl_core/stl_fpu/stl_mpu/stl_nvic>.bin -maxt
   20000000000
   ```

This compiles the netlist and runs a logical simulation with Z01X. Logical simulation results with Z01X can be checked at:

```
$(M33_HOME)/logical/testbench/execution_tb/zoix_rt.log
```

Successful logical simulation is necessary for accurate fault simulation results.

Fault Simulation Parameters

An example standard fault format file for Fault Generation has been included, `cortexm33_net.sff`. This points to the top module to recursively inject faults, in this example it is called `u_tealcore_dpu_udfdec`. You can update this to some other of your choice where you want to inject faults.

Note

The full hierarchy needs to be specified.

An example fault manager command list is provided, `sbist_fmsh_cmd.lst` and is used for setting simulation parameters before launch. It also invokes the `fdef` script to generate fault lists that occur in the design. This can be modified and extended as required for other features available in Z01X.

Running the fault simulation

The example fault simulation environment uses LSF queue natively in Z01X. Comment the following lines in the `sbist_fmsh_cmd.lst` if you do not want it to use LSF:

```
set(var=[resources], parts=[1])
set(var=[queue], name=[lsf], parts=[1], submit=[./bsub.sh], options=[], tools=[faultsim])
```

This allocates 1 job for fault simulation which will be invoked with the included `bsub.sh` example script. You can update this number based on what resources you have available. For more information on the submit feature, refer to the Z01X User Manual. The default is set to 1.

Additionally, update `bsub.sh` to add an appropriate value for the `-P` option. For more information on the bsub command, refer to the LSF documentation. No default value is set for this parameter.

To run the simulation:

```
<bsub> fmsh -load sbist_fmsh_cmd.lst
```

Fault simulation results

You should see the following message if the example fault simulation was successful:

```
#                      Total Run Times
#
#                              Queued             CPU                Wall
# Fault simulation            05:17:59          00:24:53          05:45:54
#
#
#                            Time Allocation Per Tool
#
#                                   Queued                Running
# Fault coverage              00:00:00     0.00%     00:00:02     0.01%
# Testability                 04:04:15    71.18%     00:00:16     0.08%
# Fault creation              00:00:00     0.00%     00:00:15     0.07%
# Fault results recording     00:00:00     0.00%     00:00:00     0.00%
# Fault simulation            01:13:44    21.49%     00:24:38     7.18%
#----------------------------------------------------------------------
# Total                       05:17:59    92.66%     00:25:11     7.34%
CPU Time: 22.0s ; Elapsed Time: 20787.8s ; Data Size: 68.4MB
```

When all the jobs are complete, results for the fault simulation will appear at:

```
$M33_HOME/logical/testbench/execution_tb
```

The default coverage results files will be:

```
cm33_fault_coverage.hierarchical_rpt
cm33_fault_coverage.rpt
```

All logs from Z01X simulations are preserved for debug and reference.

Appendix A
Revisions

This appendix describes the technical changes between released issues of this book.

It contains the following section:

A.1 Revisions

This section describes the technical changes between released issues of this book.

Table A-1 Issue 0000-00

Change	Location
First beta release for r0p0	-

Table A-2 Differences between issue 0000-00 and issue 0000-01

Change	Location
First early access release for r0p0	-
Editorial corrections and changes	All
Revised the presentation and content of the introductor chapter, improving the quality of the content and its flow	*Chapter 1 Introduction* on page 1-11
STL terminology section updated and corrected	*2.2 STL terminology* on page 2-19
STL software section updated and corrected, including: • Removal of superseded subsections and content from the STL software section • Revised introduction to STL software • Updated file descriptions • Revised image and content revisions in the architectural overview • Updated part descriptions • Updated STL APIs • Scheduler subsection restructured and content updated • Directory structure updated • Makefiles subsection updated • STL configuration subsection updated • Status reporting and programmable stimulus update to align with the content of the *Arm® Cortex®-M33 Processor Integration and Implementation Manual* • Customizable functions subsection updated	• *2.3 STL software* on page 2-20 • *2.3.1 STL file descriptions* on page 2-20 • *2.3.2 STL architectural overview* on page 2-22 • *2.3.3 STL part descriptions* on page 2-23 • *2.3.4 STL APIs* on page 2-31 • *2.3.5 STL scheduler* on page 2-32 • *2.3.6 STL release directory structure* on page 2-39 • *2.3.7 Makefiles* on page 2-39 • *2.3.9 STL configuration* on page 2-40 • *2.3.10 Status reporting and programmable stimulus* on page 2-45 • *2.3.11 Customizable functions* on page 2-47
STL code size and execution time raised in the hierarchy of Chapter 2, and content updated	*2.4 STL code size and execution time* on page 2-50
STL Setup section updated and corrected	*2.5 STL setup* on page 2-55
STL Out-of-Box chapter restructured, updated and corrected, including: • Revised structure • Removal of superseded subsections and content • Updated content	*Chapter 4 STL Out-of-Box tests* on page 4-67
Fault simulation chapter updated and corrected	*Chapter 5 Fault simulation* on page 5-74
Added a new section about Load-Multiple/Store-Multiple Interrupt Latency	*Load-Multiple/Store-Multiple Interrupt Latency* on page 2-44