Integrating the Arm Software Test Library into the Linux software stack

110142_0100_00

Ben Dooks, Senior Software Engineer (Codethink)
Dickon Hood, Senior Software Engineer (Codethink)
Roan Richmond, Software Engineer (Codethink)
Julien Jayat, Staff Applications Engineer (Arm)
Bernhard Rill, Director Automotive Partnerships EMEA (Arm)

This white paper, jointly written by Arm and Codethink, demonstrates the integration of the Arm Software Test Library (STL) into the Linux software stack.

1 Introduction

In the realm of embedded systems, ensuring the reliability and safety of hardware components is paramount. As systems become more complex and integrated, the potential for hardware failures increases, posing significant risks to system integrity and functionality.

While ensuring the integrity of execution is crucial in safety-critical embedded systems, integrating powerful testing tools into existing systems presents significant challenges. These challenges are further compounded by the need for elevated privileges to access certain system registers, complicating the testing process and hindering the seamless operation of embedded systems.

Arm's Software Test Libraries (STLs) are designed to address these challenges by providing comprehensive diagnostic coverage of hardware failure modes in Arm CPUs. These libraries play a crucial role in verifying CPU functionality and detecting permanent hardware failures, ensuring that systems continue to operate safely throughout their lifecycle.

This whitepaper, developed in collaboration with Codethink, explains how to integrate Arm STLs into Trusted Firmware for the Cortex-A Profile (TF-A). The primary focus is on the mechanisms for triggering STL tests in both Out-of-Reset (OOR) and Online (OnL) modes, which aid in maintaining system reliability and achieving maximum diagnostic coverage.

The STL offers two classes of tests:

1. Out-of-Reset (OOR): This mode is called during power-up. The software routine tests the logic in the design before any software is run on the system. Boot time diagnostics can only be run once per power cycle because they can interfere with a running system.

2. Online (OnL): This mode is non-destructive when executed during normal system operation and can be scheduled periodically.

Each test has a minimum Exception Level (EL) at which it may run, ranging from EL0 (least-privileged) to EL3 (most-privileged). In A-class CPUs, user processes typically run in EL0, the operating system kernel in EL1, an optional hypervisor in EL2, and an optional system monitor in EL3. TF-A provides a reference EL3 system monitor, facilitating the execution of these tests.

By examining implementation details across various Cortex-A CPUs, this document aims to provide a robust framework for developers and engineers.

Key problems addressed in this whitepaper include:

1. Diagnostic coverage: Ensuring comprehensive testing of CPU components to detect and diagnose hardware failures effectively. Arm's STLs provide diagnostic capabilities for permanent faults, making them ideal for applications with lower safety integrity requirements, such as ISO 26262 ASIL B and IEC 61508 SIL 2.

2. Integration complexity: Dealing with the complexities of integrating STLs into existing firmware and software stacks, particularly at different exception levels (EL0 to EL3). Arm's STLs can be integrated into applications using a standard API and scheduler, supporting both bare-metal implementations and operating systems.

3. Licensing and compatibility: Addressing the challenges posed by licensing constraints and ensuring compatibility between proprietary STLs and open-source software environments like Linux. The integration of STLs into Linux environments, as demonstrated on platforms like the Raspberry Pi 4, showcases the versatility and robustness of the approach.

4. Performance overhead: Managing the performance overhead associated with running diagnostic tests, particularly in resource-constrained environments. The runtime duration of all tests is less than 500 microseconds at the lowest CPU frequency and less than 275 microseconds at the maximum frequency on the Raspberry Pi 4.

Arm offers STLs on various Cortex-A CPUs, and while the overall architecture remains the same, implementation details can vary for different CPUs, impacting integration into the software stack.

For example:

- For the Cortex-A53, Cortex-A72, and Cortex-A55 CPUs, the STL includes an MMU test that can only run Out-of-Reset (OOR) before the MMU is configured by the boot software.
- For the Cortex-A78AE and Cortex-A65AE CPUs, there is no MMU test Out-of-Reset (OOR), so the test can only run later in the boot software.
- For the Cortex-A720AE and upcoming STL, to increase diagnostic coverage, the STL mandates the use of an additional SBIST (Software Built-In Self-Test) Controller, which can only be accessed from Secure Memory. Consequently, it is not possible to run the STL at EL0 non-secure (Linux user space), EL1 non-secure (Linux kernel), or EL2 non-secure (hypervisor) with the A720AE STL. Integration at EL3 is possible.

Because all the STL tests can run at EL3, providing the most diagnostic coverage, this whitepaper focuses on the integration of the STL at EL3. By leveraging the capabilities of EL3, the integration framework not only maximizes diagnostic coverage but also addresses the complexities of interfacing with secure monitors and managing exception levels.

The collaborative efforts of Arm and Codethink provide a valuable blueprint for future developments in hardware diagnostics and reliability.

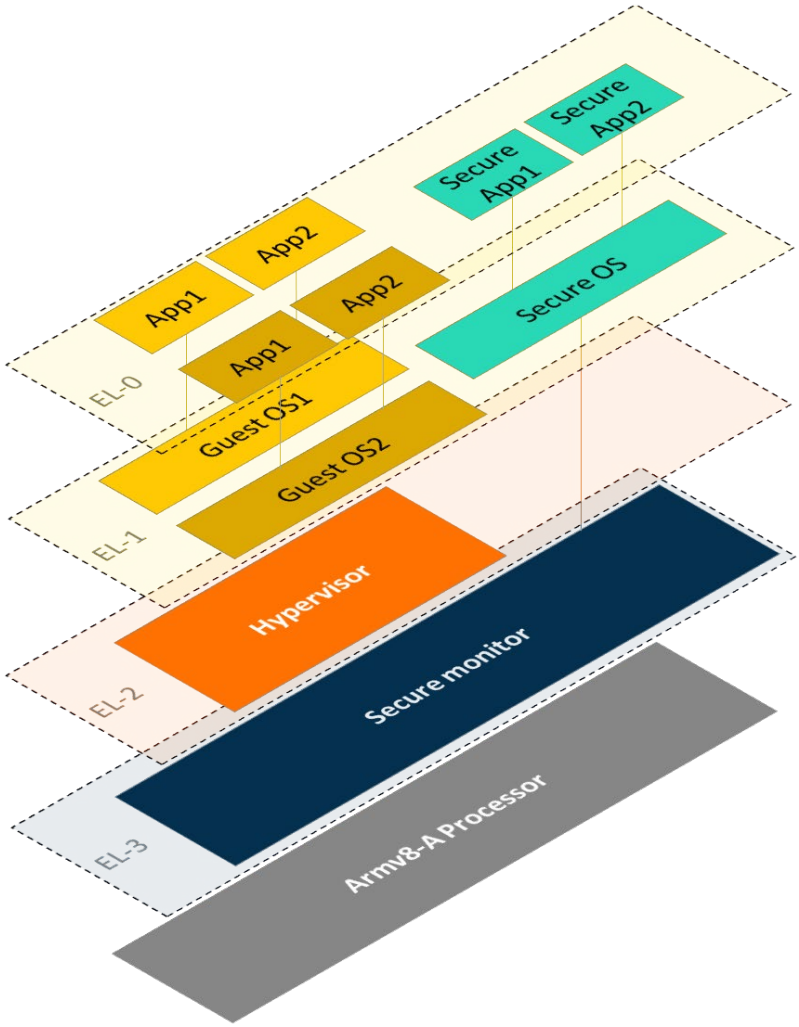

Figure 1: Armv8/9-A exception levels

Some STL tests require elevated privileges to read and write system registers that are not accessible in lower-privileged ELs. For example, only basic Arithmetic Logic Unit (ALU) functions are testable in EL0, whereas EL3 has full access to all the registers across the core. Processes in EL0 cannot directly call EL3, requiring a mechanism to call the STL in EL3 using a kernel. Because some customers have restrictions on running software at EL3, this document briefly explores the integration possibilities at other exception levels. However, because all tests can run at EL3, providing the most comprehensive diagnostic coverage, and because integration at EL3 is feasible with all current and upcoming STLs, this document primarily focuses on EL3 integration.

While the use of STL might be necessary to achieve specific ASIL targets for ISO 26262, this white paper does not specifically address the topic of argumentation required to achieve functional safety certification. For an overview on the topic of safety certification, refer to the joint Exida/Arm paper "State of the Art Software Test Libraries (STLs) and ASIL B: Truths, Myths, and Guidance".

Similarly, this Codethink/Arm STL paper does not cover cybersecurity-related aspects of STL integration.

2 STL execution on application processors

Arm commissioned Codethink to integrate Arm-developed STLs in application processors executing a Linux software ecosystem. The solution was a proof of concept demonstrating STL execution with the Linux kernel, including execution in out-of-reset (OOR) and online (OnL) modes. The results were then made available to user space applications post-boot-up.

Key challenges identified included dealing with the complexity of changing exception levels (EL) and interfacing with the TF-A Secure Monitor to access EL3. Potential licensing issues between the Linux kernel and the Arm STL were also considered. The hardware under test was a Raspberry Pi 4 which has four Cortex-A72 cores in a single cluster.

This section reviews the proof of concept in depth, describing the results and the assumptions developed throughout the project.

2.1 Getting started

The goal for the project is that the solution should be generic to most modern Linux kernel versions. The solution adds a minimal kernel driver, which provides an interface for user space to make calls to the Secure Monitor running in privileged EL3.

The solution uses a stable Linux kernel API to implement the calling chain. Further details of the APIs are described in EL3 integration in TF-A. This is because the Linux kernel internal APIs are subject to change and are not stable. By choosing an API in the Linux kernel that has not changed in an extended period, the integration should also apply to many recent kernels and future versions.

2.2 EL0: User-based execution

At this exception level, the solution provides a simple daemon which periodically executes the STL. For this, the team did not directly interact with Linux kernel code; therefore, licensing was not an issue. Instead, the STL code is directly linked with the user space daemon.
The main challenge for this part of the project was guaranteeing that the STL executes each test without interruption. This is achieved by creating one thread per core, where each thread changes the scheduling priority to the maximum before starting the execution.

2.3 EL1: Linux kernel execution using SMC (Secure Monitor Call)

The Linux kernel is licensed under GPLv2 (General Public License), meaning any code linked directly must be GPL-compatible. Adding software into EL1, such as a kernel driver to run the STL directly, requires all the code linked with it to be open-source, which is impossible because the STL are proprietary as they expose CPU-related details.

Instead, thesolution calls the Secure Monitor ‘TF-A' to run the STL tests. This means these are not run in EL1 but via it. The advantage of this architecture approach is that all the tests can be executed in EL3 ensuring the highest netlist coverage.

2.4 EL3: Secure Monitor

The Secure Monitor is part of Arm's open-source Trusted Firmware for Cortex-A (TF-A). The code to run the STL can be integrated directly into the Secure Monitor, which runs at EL3. The firmware starts the Secure Monitor during the boot process and then becomes available to interact with using a Secure Monitor Call (SMC).

Because the SMC instruction is available only for software executing at EL1 or higher, a user space application first uses a Supervisor call (SVC) instruction to request the execution of the STL to an EL1 software. In Linux this can be achieved through a system call. A kernel module running at EL1 handles an IOCTL (input/output control) and then forwards the STL request to EL3 using SMC.

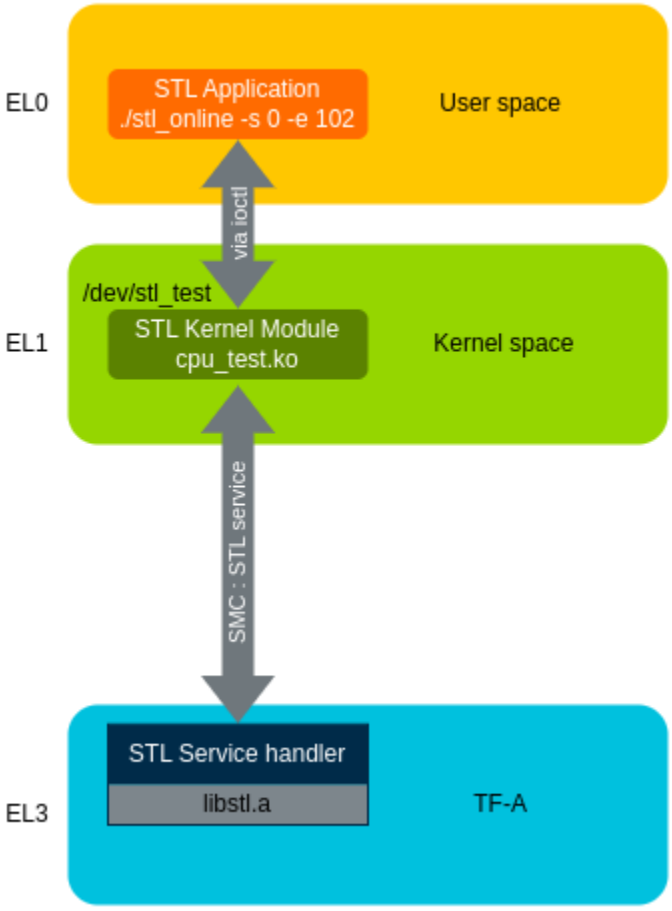

Figure 2: STL SW stack interaction between the exception levels

Adding a new SMC allows the user to trigger the Secure Monitor to run the STL. To do this, a glue file was added to TF-A to advertize the SMC number and register the handler. The purpose of this SMC is twofold: The first is to run the STL in online mode, and the second is to access the results of the STL out-of-reset tests.

The Secure Monitor runs all the tests offered by the STL in OL mode, including those which test EL0-2. These are run at EL3 for the entire duration, even if they only require EL1 or EL2 permissions. If a test fails, the STL stops executing the following tests and returns the values stored in the various fault registers using the same SMC. This design was chosen as it provides a constant return value size and allows the caller to handle the failure immediately.

In multi-core systems, it is crucial for each core to run the STL to detect its permanent hardware faults. To ensure that the STL runs on a specific core, the application must bind itself to that core. This is done by setting the CPU affinity of the application using the `sched_setaffinity` function from the Linux CPU scheduling infrastructure. By configuring the affinity, the application guarantees that both SVC and SMC will be executed on the same core ensuring that the STL runs on the designated core.

2.5 Out of reset integration observations

OOR tests are performed from the early BL31 stage of the TF-A bootloader. The tests must run at this stage to strike a careful balance between ensuring enough environment has been set up to facilitate them and not running the tests too late that they disrupt some part of memory that a later stage relies on. This is especially true for the STL Memory Management Unit (MMU) because parts of these tests include reading and writing pages of memory.

Naturally there is an interest to execute the STLs during a cold boot process. Due to some of the circumstances outlined below, it is not possible to run the OOR tests when cores are hot plugged. In this case, one logical solution is to run the Online tests instead, which can be executed in this scenario.

The STL Out of Reset (OOR) tests have certain requirements on when they can be run during the boot process. These requirements are mainly because of the thorough tests requiring access to the Memory Management Unit (MMU) to test physical memory. This resulted in some special considerations in the boot process of TF-A.

A general boot process without integrating the STLs involves the secondary cores being put into a wait state. The primary core then continues and runs various early-stage bootloaders to initialize early core functions. Eventually, TF-A hands control to the Linux kernel to begin its boot process, and which brings the secondary cores out of their wait state.

With the inclusion of the STLs, the proposed solution includes the primary core running the OOR tests early in its boot process before the bootloader sets up the MMU (in early BL31). This allows the MMU to be tested without affecting any physical memory already set up.

The primary core then sequentially pulls the secondary cores out of reset, and they run the OOR tests before re-entering the same wait state as before. It is important to note that the

secondary cores need to run the OOR tests sequentially because one of the MMU tests has a dependency of writing a pattern to a physical address and rereading it using a virtual address.

Once all the OOR tests have completed, the results are saved and can be accessed using an SMC after properly starting Linux. Another alternative is to deal with test failures when they happen inside the TF-A boot process. However, the STL test failure mitigation choice depends on the overall application requirements.

2.6 Managing and mitigating test failures

The STL implementation from Arm exits on the first test failure and stores various fault information in the relevant fault registers. In online mode, the implementation returns all the fault register values to EL0 on failure, allowing the calling process to handle the issue.

When the tests run in OOR mode as part of the boot sequence, the fault register values are stored if there is a failure, and the boot process continues. Implementing fault management at the point of failure can be trivial, for example, signaling the error to external hardware or restarting the system.

It is advisable to implement an addition to the SMC, allowing an EL0 process to check the outcome of the OOR tests and get the results for each fault control register. This makes it possible for non-critical failures to be resolved after the boot-up is complete.

2.7 Raspberry Pi 4 STL integration

The Raspberry Pi 4 Model B features a Broadcom BCM2711 system-on-chip (SoC), which includes a quad-core Cortex-A72 (Armv8.0-A) 64-bit CPU. This easily procurable board is well-supported by the open-source community, making it an excellent choice for a wide range of projects. This reference platform and the methods described for integrating STLs can be mapped to any selected target platform that partners use, provided STLs have been developed for those platforms. For further details, please contact your Arm partner manager.

The ARM STL is specifically designed to test the CPU interface of the GICv3. Because the Raspberry Pi 4 implements the GICv2 interface, which uses memory-mapped registers, the GIC tests cannot run on the Raspberry Pi 4. Consequently, the GIC tests are excluded from the list of tests when building the STL for the Raspberry Pi 4.

2.7.1 EL0 application integration

At this exception level, the calling of the STLs can be triggered using a simple daemon which periodically executes the STL. With this approach there is no direct interaction with Linux kernel code; therefore, licensing is not an issue. Instead, the STL static library is linked with the user-space daemon, and the daemon executes the EL0 OnL tests and reports the results using syslog.

The STL must execute tests without interruption. This is achieved by creating one thread per core, with each thread changing the scheduling priority to the maximum before starting execution.

All the cores are tested by binding each task to a different CPU using CPU affinity.

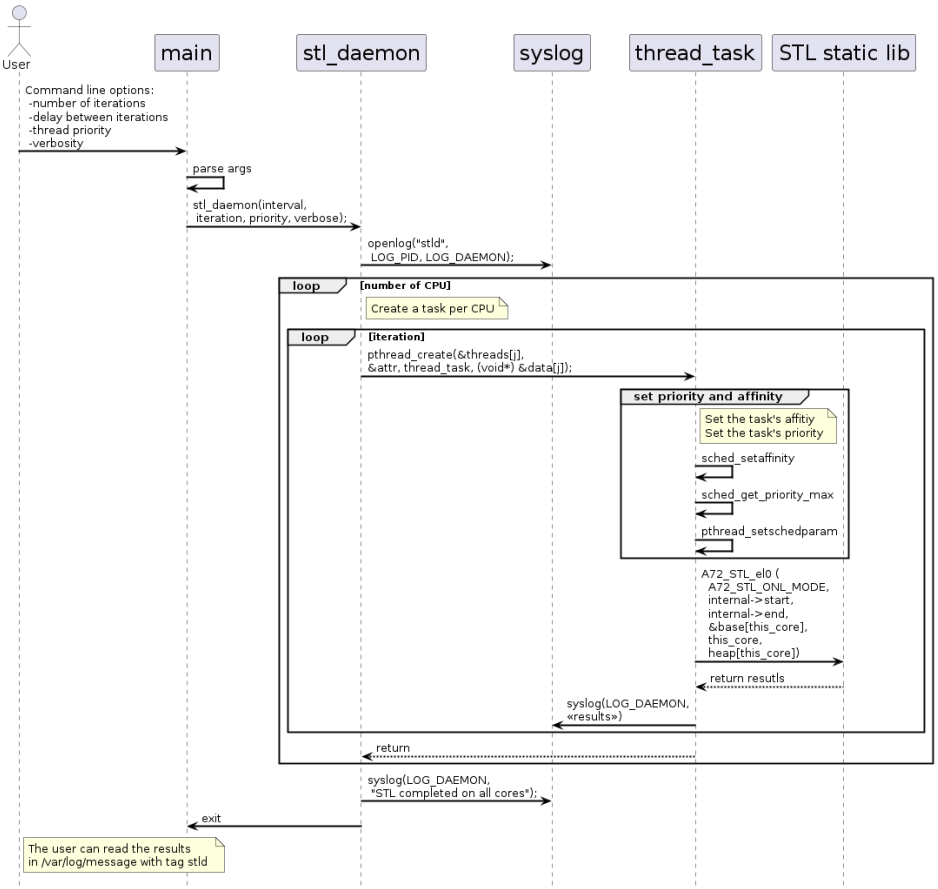

Figure 3: Sequence diagram of the EL0NS daemon

It is feasible to create as many threads as the system has cores to run tests in parallel. When the tests are executed, the results are logged. The above example shows how to integrate the STL

library and log the results. In a production system the integrator would need to implement in addition a safety mechanism in case of hardware failure detection.

The user can review the result of the test using the following command:

```
# stl_daemon -l 10 -i $((1*1000*1000*1000)) -p 80
# cat /var/log/messages | grep stld | tail -n 11
Jan  1 00:11:21 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:22 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:23 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:24 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:25 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:26 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:27 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:28 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:29 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:30 buildroot daemon.emerg stld[700]: STL completed on all cores
Jan  1 00:11:30 buildroot daemon.emerg stld[700]: Finished and Exiting now...
# 
```

Figure 4: STL test results overview

2.7.2 EL3 integration in TF-A

To address EL3 integration it is advisable to create an out-of-tree Linux kernel. This small module uses two stable kernel APIs:

- the file_operations API, via the misc device handler
- the Arm Secure Monitor Call Calling Convention (SMCCC) API

The module registers as a device driver for a `/dev/stl_test` pseudo-device and accepts an `ioctl()` call to trigger any given range of tests.

Using this approach, the STL can be executed using a simple command-line application, which uses the CPU process affinity to bind the process to a specific core ID, calls the ioctl() function, which calls the SMCCC function, and executes the requested STL tests on the appropriate core.

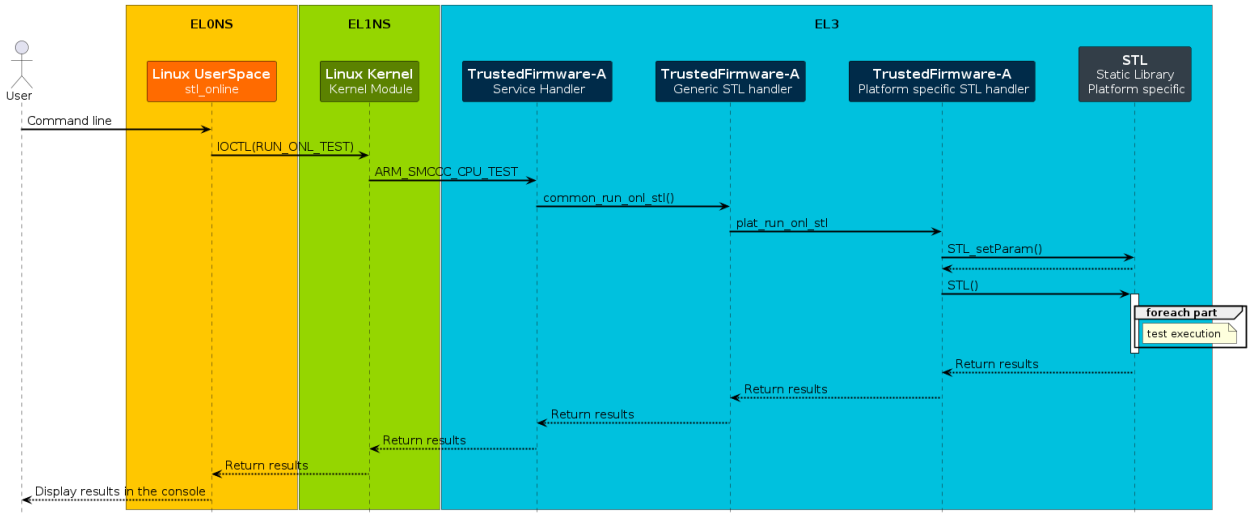

Figure 5: Simplified sequence diagram for EL3 online tests when called from kernel user space

In the Raspberry Pi 4, the boot process for TF-A results in the secondary cores being put into a wait state, with only the primary core continuing to run most of the bootloader. To take advantage of this, the solution includes the primary core running the OOR STL when it reaches BL31. Then, after it had completes the tests, it configures the entry point of the secondary cores so that when they are pulled out of the wait state, they immediately start running the STL OOR tests themselves. This order was used because it is the most straightforward architectural approach.

2.7.3 How to add the static library to the TF-A

Because the STL source code is delivered as a standalone software artefact and access is restricted to Arm licensees, it has been chosen to link the STL as an external static library. The STL source code is not included in the TF-A code, only the header file of the library is included. The STL code must be configured to run the specific platform, with all user parameters properly set. This includes setting the correct value in `stl/tests/scheduler/inc/a72_stl_global_defs.h` and in the case of the Raspberry PI 4 removing the GIC tests from the EL3 list `stl/tests/cfg/list/a72_stl_el3.lst`.

Once compiled, it is possible to link the static library in the TF-A by adding the library path to the linker library variable. This is done in `bl31/bl31.mk`:

```
ifeq ($(ENABLE_STL),1)
LDLIBS += ${STL_LIB}
endif
```

The addition of the library is conditional, so the variable `ENABLE_STL` should also be enabled.

To compile the TF-A including those variables, add the following the `make` command line:

```
ENABLE_STL=1 STL_LIB=$(CONFIG_DIR)/a72_RPI4_stl.a
```

Using this definition, the platform-specific STL implementation can be added in `plat/rpi/rpi4/platform.mk`:

```
ifeq ($(ENABLE_STL), 1)
BL31_SOURCES += plat/rpi/rpi4/rpi4_stl.c
endif
```

2.7.4 Build integration

The approach used for the build integration on RPI4 is a generic STL service handler utilizing an operations structure to call platform-specific STL functions. During the setup phase, the platform-specific handler registers a function pointer table for use by the generic driver.

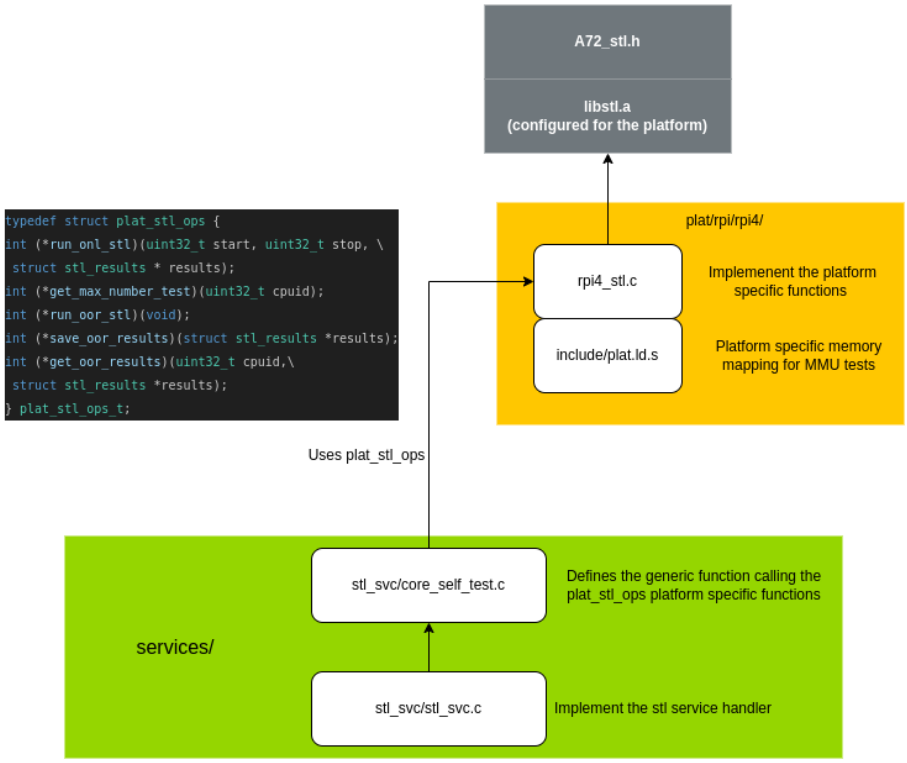

Figure 6: Architecture for TF-A integration

This integration method offers several benefits:

1. It isolates platform-specific code within the platform folder. For instance, different platforms may have different storage locations for OOR test results.

2. It permits the use of platform-specific linker files to define symbols that the static library needs when built with MMU tests.

3. It enables the potential utilization of multiple STL implementations. Although not yet deployed, this architecture could support big.LITTLE cores, for example Cortex-A72 and Cortex-A53.

2.7.5 Special consideration about MMU tests (STL A53, A72 and A55)

When linking the STL library compiled with the MMU tests, the library expects to find some specific symbols. These can be provided using the platform-specific linker file. The library

expects symbols to be generated at link time using scatter files from `armlink`. The symbols have a specific denomination.

It is possible to provide those symbols when compiling the TF-A with GCC by using the platform-specific linker file `/plat/rpi/rpi4/include/plat.ld.S`.

For the RPI4, use the following:

```
MEMORY {
     STLMEM1 (rwx): ORIGIN = 0x80000000, LENGTH = 4
     STLMEM2 (rwx): ORIGIN = 0xBFFFFFFC, LENGTH = 4
}

SECTIONS
{
     .stl_mem_1   (NOLOAD)   : {
         KEEP(*(.stlmem1))
     } > STLMEM1

     .stl_mem_2   (NOLOAD)   : {
         KEEP(*(.stlmem2))
     } > STLMEM2
}
```

The variables with the name of the symbols are defined as in the platform-specific STL handler:

```
uint32_t Image$$TEST_UTLB_REG1$$ZI$$Base __section(".stlmem1");
uint32_t Image$$TEST_UTLB_REG2$$ZI$$Base __section(".stlmem2");
```

The symbols are created and located at defined addresses that should be configured accordingly in the STL library's source, `stl/tests/scheduler/inc/a72_stl_global_defs.h`.

```
#define A72_STL_PA_1         (0x0000000080000000)
#define A72_STL_PA_2         (0x00000000FFFFF000)
```

The STL can test the MMU using those physical memory addresses.

The MMU tests also require memory to create an MMU translation table. The memory does not need to be placed in an exact location. The MMU table is filled at runtime and located at the address of the symbols of type:

```
Image$$TTB0_L1$$ZI$$Base
```

The symbols can be provided as follows:

```
#define make_pgarea(__name, __size, __align) \
  extern uint8_t Image$$##__name##$$ZI$$Base[];  \
  uint8_t Image$$##__name##$$ZI$$Base[__size]
__attribute__((aligned(__align)))

make_pgarea(TTB0_L1, 0x1000, 0x1000);
make_pgarea(TT_S1L1_EL1, 0x1000, 0x1000);
make_pgarea(TT_S2L1_EL1, 0x1000, 0x1000);
make_pgarea(TT_S1L1_EL2, 0x1000, 0x1000);
make_pgarea(TT_S1L2_EL2, 0x1000, 0x1000);
make_pgarea(TT_L0_EL3, 0x1000, 0x1000);
make_pgarea(TT_L1_T1_EL3, 0x1000, 0x1000);
make_pgarea(TT_L1_T2_EL3, 0x1000, 0x1000);
make_pgarea(TT_L1_T3_EL3, 0x1000, 0x1000);
make_pgarea(TT_L2_T1_EL3, 0x1000, 0x1000);
make_pgarea(TT_L2_T2_EL3, 0x1000, 0x1000);
make_pgarea(TT_L2_T3_EL3, 0x1000, 0x1000);
make_pgarea(TT_L2_T4_EL3, 0x1000, 0x1000);
make_pgarea(TT_L3_T1_EL3, 0x1000, 0x1000);
make_pgarea(TT_L3_T2_EL3, 0x1000, 0x1000);
make_pgarea(TT_L3_T3_EL3, 0x1000, 0x1000);
make_pgarea(TT_L3_T4_EL3, 0x1000, 0x1000);
make_pgarea(TT_L3_T5_EL3, 0x1000, 0x1000);
```

The unmodified linker file from the TF-A then has to allocate memory to those tables into the `.bss` section of `BL31.elf`. Consequently, it might be necessary to increase the memory size reserved for the TF-A.

2.7.6 Performance overhead

It is possible to instrument the TF-A to measure the execution time and cycle count taken by the STL execution. These measurements are for reference only but provide an accurate order of magnitude.

The runtime duration (in the TF-A handler) of all the tests is less than 500 microseconds at the lowest RPI4 CPU frequency (600 MHz) and less than 275 microseconds at the maximum frequency (1.5 GHz). Because the tests can run individually, it is possible to reduce the test execution duration at the cost of more overhead from the triggering mechanism. A single test can be triggered and executed in less than 15 microseconds.

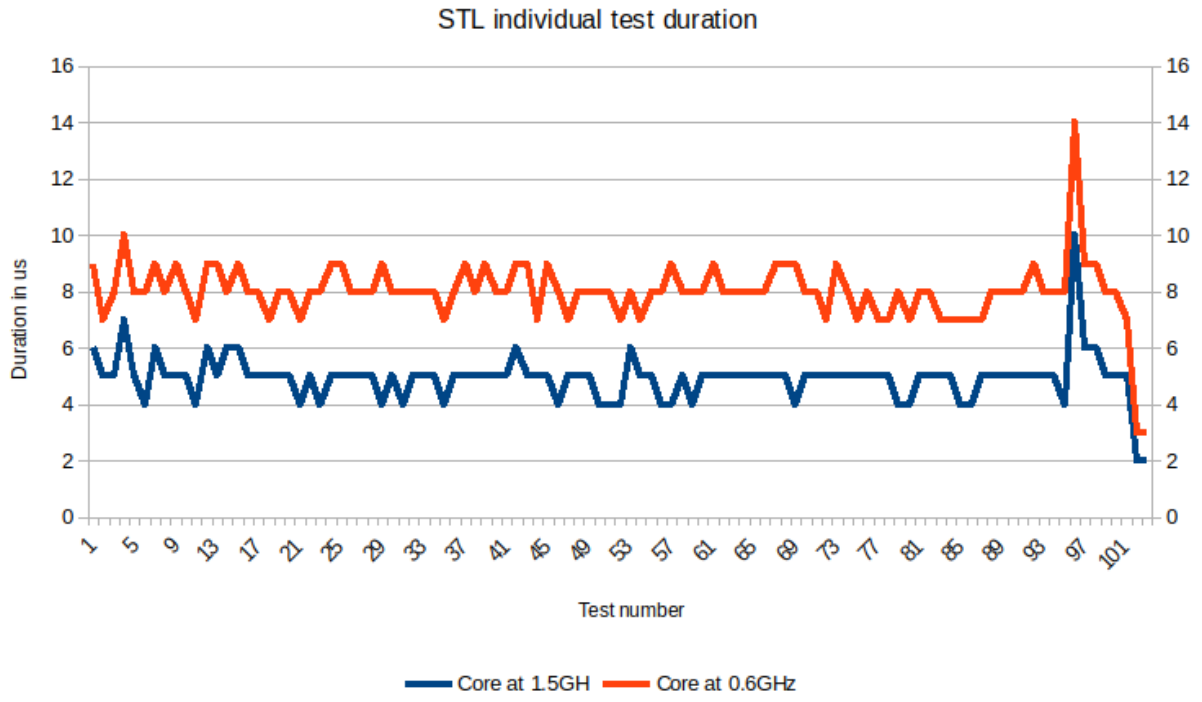

Figure 7: STL execution duration graph

Note that the STL executes tests with IRQs disabled; thus, there is an interrupt-blocking window. This has implications for the application execution, which needs to service dependencies like external devices in a timely manner. Where IRQ service latency is a problem, consider executing tests sequentially rather than in a blockso that any interrupts can be serviced between calls to the STL. This increases the overall runtime overhead of the STL but

should mitigate the IRQ servicing delay to levels acceptable for all but the most demanding of applications.

2.7.7 Memory footprint

Adding the STL library to TF-A increases the build size according to the library's size and runtime memory needs. This is manageable for the Raspberry Pi 4 because TF-A is in DDR memory, allowing for an easy increase in BL31 size. However, not all platforms can do this; some require TF-A to run from On-Chip RAM (OCRAM).

The size of the TF-A BL31 can be compared with and without the STL integration. No modification to the TF-A linker script has been made to optimize the STL's size or placement.

Section	With STL	Without STL	Difference
.text	0x1e000	0xb000	0x13000
.rodata	0x5000	0x2000	0x3000
.data	0x73	0x73	0x0
.stacks	0x4000	0x4000	0x0
.bss	0xe7a0	0xf20	0xd8a0
.xlat_table	0x4000	0x4000	0x0
.coherent_ram	0x1000	0x1000	0x0

Table 1: Size of the code sections in the TF-A (Raspberry Pi 4)

Independently of any memory footprint optimization option, this data shows that adding STL to the TF-A has a significant impact on the memory size of the TF-A.

2.8 Other consideration: number of tests

Software applications are independently built at EL0, EL1 at EL3. The STL static library is also built independently. The number of tests supported by the library depends on the exception level and the platform's support of the features.

The application must determine the number of tests supported by the library at their exception level to validate request inputs.

No API directly supplies this maximum number of tests; thus it must be provided in one of the following two ways:

- Statically, by adding `#define` in a platform-specific header.
- Dynamically, by probing the library. Use `SetParam()` to iteratively increase the test count until an error occurs. Although this approach takes more time, it simplifies integration, because linking to a modified library does not necessitate adapting the header file. However, probing can only return the number of tests available for the current exception level.

Each approach has its advantages. The library should remain unchanged in functional safety scenarios, with a predefined number of tests. However, eliminating the need to update a header file from another project during software integration eases build dependencies.

3 Summary

This document describes in detail the integration of the A72 STL into a Raspberry Pi4 in a Linux environment. The Codethink team implemented an EL3 TF-A-based approach that allows end users to use STLs within a Linux-based software architecture. Based on this approach it has been demonstrated how proprietary register-level software tests can be integrated into an open-source operating system.

Integrating STLs within a Linux application context revealed several issues to be aware of. The results outlined in this paper provide viable solutions the these issues. Licensing concerns, mainly due to the incompatibility between Linux's GLPv2 and the proprietary STL, were a known but recurring obstacle which were carefully managed throughout the development process.

Note that the approach to integrating the STLs into a Linux environment described in this document is not the only method and other integration techniques can be followed.

If you have any questions about the content of this white paper, please reach out to your Codethink or Arm partner manager.

Arm’s commitment to providing robust diagnostic tools, combined with Codethink’s expertise in software integration, has resulted in a comprehensive solution that enhances system reliability and safety. This collaboration underscores the importance of industry partnerships in driving innovation and ensuring the highest standards of functional safety.

Together with Codethink, Arm is paving the way for more reliable and effective embedded systems.

4 Abbreviations

Abbreviation	Description
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
CPU	Central Processing Unit
DDR	Double Data Rate Memory
ECU	Electronic Control Unit
GPL	General Public License
IRQ	Interrupt Request
MMU	Memory Management Unit
OCRAM	On Chip Random Access Memory
OnL	Online Mode
OOR	Out-Of-Reset
SMC	Secure Monitor Call
STL	Software Test Libraries
SVC	Supervisor Call
TF-A	Trusted Firmware for Cortex-A

Table 2: List of abbreviations used in this document

5 Further reading

- Arm STL product information overview

  https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries

- Arm Community blog: Wind River and Arm collaboration accelerates journey to functional safety compliance in centralized vehicle controllers

  https://community.arm.com/arm-community-blogs/b/automotive-blog/posts/wind-river-functional-safety-collaboration

- Arm Community blog: How to integrate Arm STL for a safer Automotive experience

  https://community.arm.com/arm-community-blogs/b/automotive-blog/posts/arm-stl-automotive-experience