Integrating Arm STL into Classic AUTOSAR on Cortex-M and Cortex-R

110037

Alex Wilmot, Software Engineer, ETAS
Max Sinclair, Field Application Engineer, ETAS
George Bray, Product Manager, ETAS
Dr Andrew Coombes, Principal Automotive Software Product Manager, Arm
Bernhard Rill, Director Automotive GTM EMEA, Arm

This white paper explores the integration of the Arm Software Test Library (STL) into Classic AUTOSAR (Automotive Open System Architecture) to enhance the diagnostic capabilities and reliability of automotive ECU software.

Overview

The automotive industry is witnessing a period of unprecedented change, and the integration of robust and reliable software components will be essential to ensuring the safety and efficiency of Electronic Control Units (ECUs). Classic AUTOSAR (Automotive Open System Architecture) is a standardized framework for real-time deterministic automotive software, offering a modular and scalable approach to ECU development. Likewise, the Arm Software Test Library (STL) provides a comprehensive suite of diagnostic tests to verify the correct operation and integrity of Arm-based microcontrollers - including those based on Cortex-M and Cortex-R, which are widely used in automotive applications.

This paper explores the integration of Arm STL into Classic AUTOSAR to enhance the diagnostic capabilities and reliability of automotive ECU software. This integration provides a robust solution for fault detection and mitigation, thereby contributing to the overall safety and performance of automotive systems. The structure of the paper is as follows:

1. Introduction to Classic AUTOSAR and Arm STL:
   A brief overview of both technologies.

2. Key Differences Between Armv8-R and Other Cortex-M/Cortex-R Cores:
   How these differences affect the operation of the STL.

3. Run-time Scenarios for STL Integration:
   Considerations for delivering comprehensive coverage with the STL in various scenarios, including Out-of-Reset, Online, and Online Event use cases.

4. Comparison of STL Integration Options in Classic AUTOSAR:
   Evaluating different methods for integrating STL with Classic AUTOSAR.

5. Handling Faults Detected by the STL:
   Strategies for managing faults identified by the STL.

While the use of STL might be necessary to achieve specific ASIL targets for ISO26262, 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 (STL) and ASIL B: Truths, Myths, and Guidance"

Introduction to Classic AUTOSAR

Classic AUTOSAR (Automotive Open System Architecture) is a standardized framework that streamlines the development of automotive software. It provides a modular and scalable architecture, dividing the software into two main layers: Application Software (ASW) and Basic Software (BSW).

Application Software consists of the functional components that perform specific tasks, such as controlling the engine or managing the infotainment system. These components are designed for reusability and can be easily integrated into various vehicle platforms.

Basic Software (BSW) forms the foundation of the AUTOSAR architecture, providing essential services and functions that support the application software. This includes operating system services, communication protocols, and hardware abstraction layers, ensuring that the application software can operate independently of the underlying hardware.

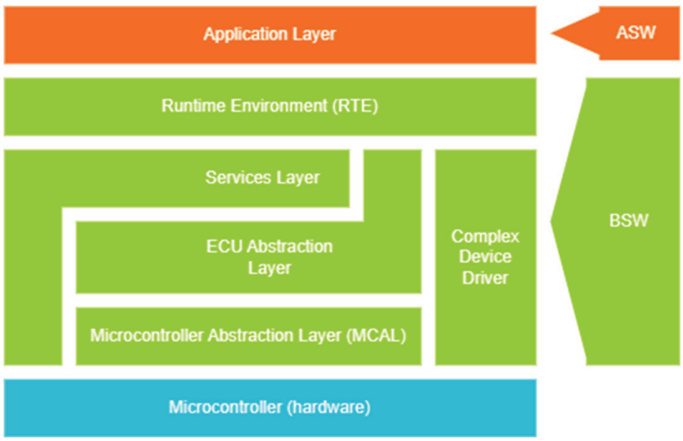

Figure 1: Classic AUTOSAR Stack

Introduction to Arm STL

Arm STLs consist of a set of tests which verify the proper functionality of the individual CPU building blocks. The STLs are one mechanism which enables an application to achieve the ASIL-B rating without dual lock step (DCLS) cores. They provide diagnostic coverage to prevent faults from leading to single-point failures or prevent them becoming latent as the result of multiple-point fault. To ensure that the STL execution can achieve the best-case netlist coverage the STLs need to be executed in the following scenarios:

- Out-of-Reset (OOR) - On startup of the application, the full test suite is executed to ensure the correctness of the underlying hardware before the application is run.
- Online (OL) - A periodic execution of the STLs where all tests are executed at a frequency determined by the application's safety requirements.
- Online Event (OLE) - The same as OL mode, but after each test part, the STLs check for a pending interrupt, allowing control to exit if an interrupt is present.

To ensure full coverage, the STLs require two interactions:

- Initialization - The STLs must be initialized exactly once before Out-Of-Reset scheduling begins and initialized at least once before online mode scheduling begins.
- Scheduling - The STLs require the application to call the scheduler to trigger test execution, using either Online or Online Event triggering.

When executing the STLs (either as OOR, OL or OLE), interrupts are disabled to prevent interference with the tests. The time when the STLs are executing is therefore described as an interrupt blocking window - which prevents interrupts related to the application from being handled. Because OLE allows for interrupts to be handled between test parts, it has a shorter interrupt blocking window than OL.

Running the STL with a Classic AUTOSAR Application

The rest of this paper discusses integrating the STLs into a Classic AUTOSAR Application with Armv8-R series Cortex-R52+ core. ETAS's RTA-OS was selected as the Real-Time Operating System (RTOS) to be used in this practical example.

The strategies discussed below, to integrate the STLs into an AUTOSAR application remain the same for many of Arm's other processor families, thus what has been outlined for the Cortex-R52+ largely applies for other 32bit architectures as well. For example, when looking at the popular Armv7/8-M and Armv7-R architecture families, the main key difference is they don't support hardware virtualization. This means they only have two Exception levels (0 and 1). The STLs must run at the highest Exception Level available (Level 1), which in this case is the same privilege level as the AUTOSAR OS. Executing the STLs at the highest exception level is vital to ensure that you get the highest RTL netlist coverage needed.

On Armv8-R (for example Arm Cortex-R52+)

In the case of Armv8-R (including Cortex-R52 and Cortex-R82AE) the STLs must be executed at EL2. RTA-OS executes 'OS' code at Exception Level 1 (EL1) and untrusted user code at Exception Level 0 (EL0). The OS has no understanding of Exception Level 2, as this is normally reserved for Hypervisor usage.

Hence, to be able to call the STL schedular API after the OS has already been started, the integrator must provide a mechanism to enter EL2. Consider the following chain of events:

1. The processor boots initially in EL2.
2. Startup code initializes the EL2 vector table, providing a handler for the Hypervisor entry handler.
3. Startup code completes. The OS starts at EL1, and may start running applications at EL0.
4. Before calling the STLs from the application, the application stores the current exception level and subsequently executes a HyperVisor Call 'HVC' instruction. This leads to the processor entering the Hypervisor entry handler at EL2.
5. The handler fakes a return from exception the processor remains at EL2.
6. The handler branches to the relevant STL function.

7. Finally, once the STL function has returned, the handler returns from exception, back to the calling code and reinstates the original exception level and the original context.

Example reference assembly code for this mechanism to transfer into EL2, can be found in the appendix.

STL integration test setup

The STLs were tested on a development board with both Cortex-M and Cortex-R cores. They were integrated into an internal RTA-OS testing application which is part of its test suite. Output traces were recorded into a RAM buffer with a synchronized clock, this allowed for accurate recording of the number of CPU cycles taken.

Out of Reset execution

The STLs can be run in a 'one-shot' execution where all configured tests are run consecutively during the boot process, to ensure the correctness of the hardware. Within this mode, the STLs can execute additional tests on the interrupt logic, such as those related to the GIC and GDU. This is possible at this stage in the processor bring-up as interrupts will be disabled.

There are multiple suitable places that this could be done within a classic AUTOSAR application. However, extra steps may be required, depending on the Exception Level of the stage in which they are run. During the startup of the Cortex-R52 cores, the processor will initially be in the Exception Level 2 (EL2) state; this means the processor is already in the correct state to run the STLs without any wrappers.

Careful consideration is required when integrating the STLs at boot-time as real-world applications often have hard timing deadlines on startup and bus communication. This can already be a challenging problem to solve, due to intensive processes on startup such as the initialization of the BSW Non-Volatile Memory module (NvM). Depending on the requirements of the application, the additional time required to execute the STLs out-of-reset may cause startup deadlines to be missed.

Execution Options

ECU State Manager (EcuM)

The EcuM has a notion of an initialization list. This is a list of callouts for driver initialization and other hardware related startup activities to be run after a power-on-reset before the OS is

started. This is the most practical and sensible place to run the STLs on boot as it conforms to the AUTOSAR method of booting, executes the STLs before the OS is started. This means that the EL2 processor state is already present, so no additional exception level switch is required to run the STLs. This cuts down the number of cycles needed to run the STLs at boot, while still conforming to AUTOSAR requirements. The following diagram shows the sequence of events that would occur leading up to the execution of the STLs using this approach

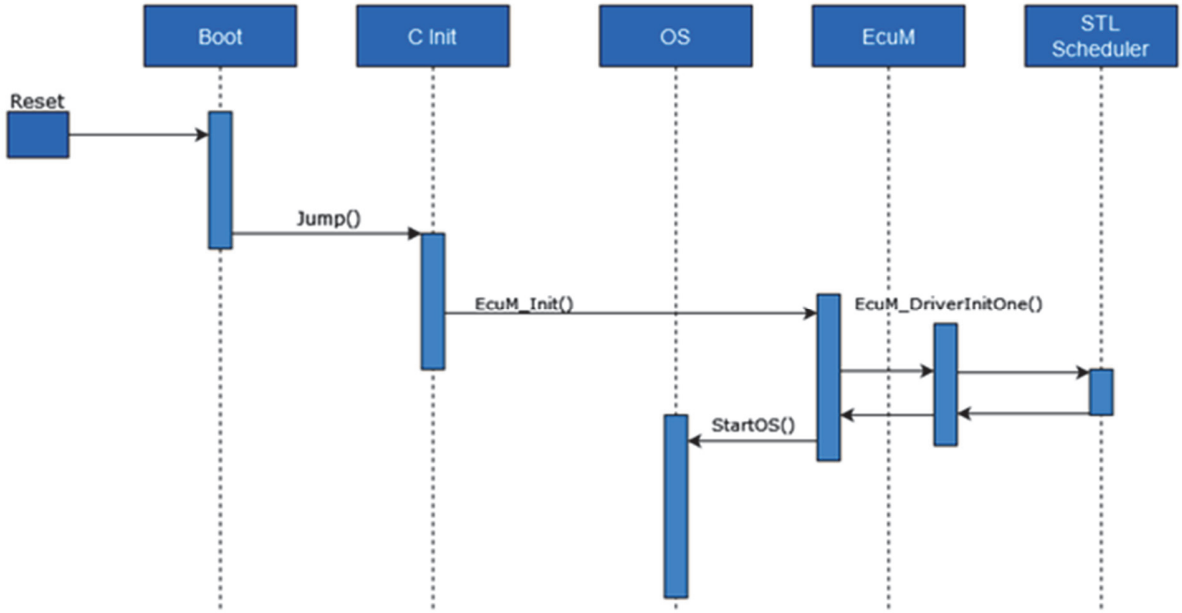

Figure 2: EcuM sequence for OOR execution of STLs

Basic Software State Manager (BswM)

The BswM has the notion of Action Lists. These are a list of functions to execute conditionally, allowing the creation of boot sequences. However, the BswM is only executed within the OS, meaning that the processor exception level will be at EL1. So, additional code will be required to enter EL2 before running any tests, adding more runtime cost. Additionally, these actions lists are further into the boot sequence of the application, meaning MCAL modules and other items may have already been initialized. Requirements may dictate that the STLs must run earlier out of reset to catch hardware faults, so that no hardware initialization is done pointlessly and so that initialization is not at risk of being done using fault present hardware. The following diagram shows the sequence of events that would occur leading up to the execution of the STLs using this approach:

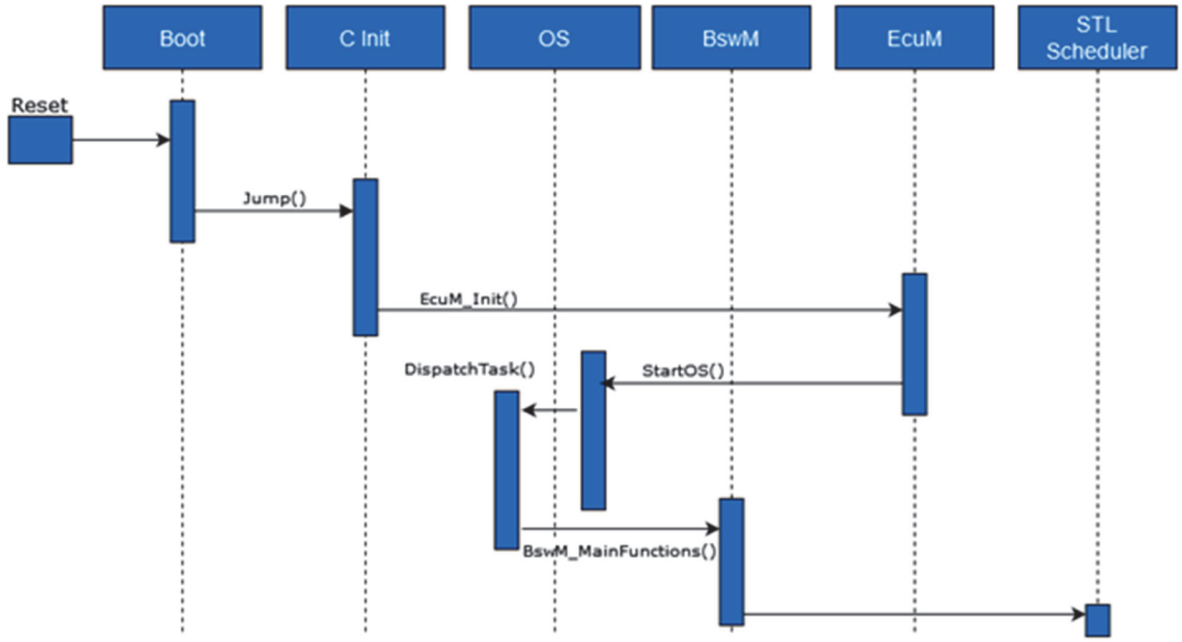

Figure 3: BswM sequence for OOR execution of STLs

Comparison

The two approaches mentioned above only show the phase during startup in which the STLs run in OOR mode. Clearly the methodology does not influence the execution time of the STL OOR test suite itself. The STL OOR test suite was measured to take approximately 420,000 CPU cycles in the ETAS test environment, which would equal an execution time for 1.4ms at a CPU clock speed of 300MHz (this includes the time to initialize the STLs and restore the context after the STL execution).

On average, customer ECUs require that an ECU is online and transmitting/receiving bus traffic within 150-200ms. Executing the STLs would take only 0.7-1% of this startup budget, which suggests that STL overheads will be minimal in most cases.

Online and Online Event Execution

Scheduling Options

For integration into a classic AUTOSAR OS, there are a variety of different scheduling mechanisms that can be employed to run the STLs, all with their associated runtime costs. Depending on the frequency that the STLs need to be periodically executed at, careful considerations need to be made on the overall latency of each option.

Most importantly, during individual tests, there is a blocking window where all interrupts are disabled. Thus, the worst-case reaction time of a system to an event needs careful consideration when integrating STLs into a project.

An advantage of running the STLs in OLE mode is that they can be run periodically, without blocking for the full duration of the number of test parts requested. This means if higher priority events present themselves, the STL code will return, allowing pre-emption.

Periodic Category 1 Interrupt

This is the minimum required for periodically running the STLs and will achieve the lowest number of cycles to run the STLs. It involves a hardware timer that periodically raises an interrupt, for which the user provides an interrupt service routine that switches to EL2 and runs the STL scheduler. However, CAT1s are often frowned upon as they are outside the knowledge of the OS and have no support for AUTOSAR OS features such as timing and memory protection.

The following diagram shows the sequence of events leading to running the STL scheduler using this approach.

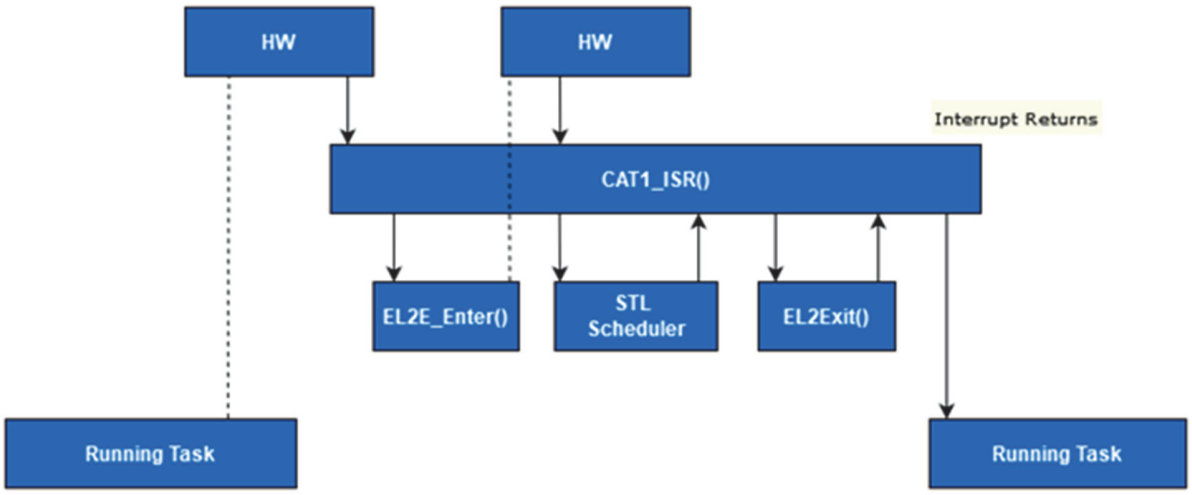

Figure 4: CAT1 sequence for OL execution of STLs

Periodic Category 2 Interrupt

A periodic CAT2 interrupt works on the same basis as above. However, it is known and controlled by the OS, meaning an OS wrapper is used before handing over execution to the user interrupt. This allows support of OS features within the interrupt, and for interaction with OS APIs. Neither of which are allowed in CAT1's. Most importantly, due to the nature of the STLs and their safety aspect, this allows configuration of timing protection to ensure the STLs are run as expected

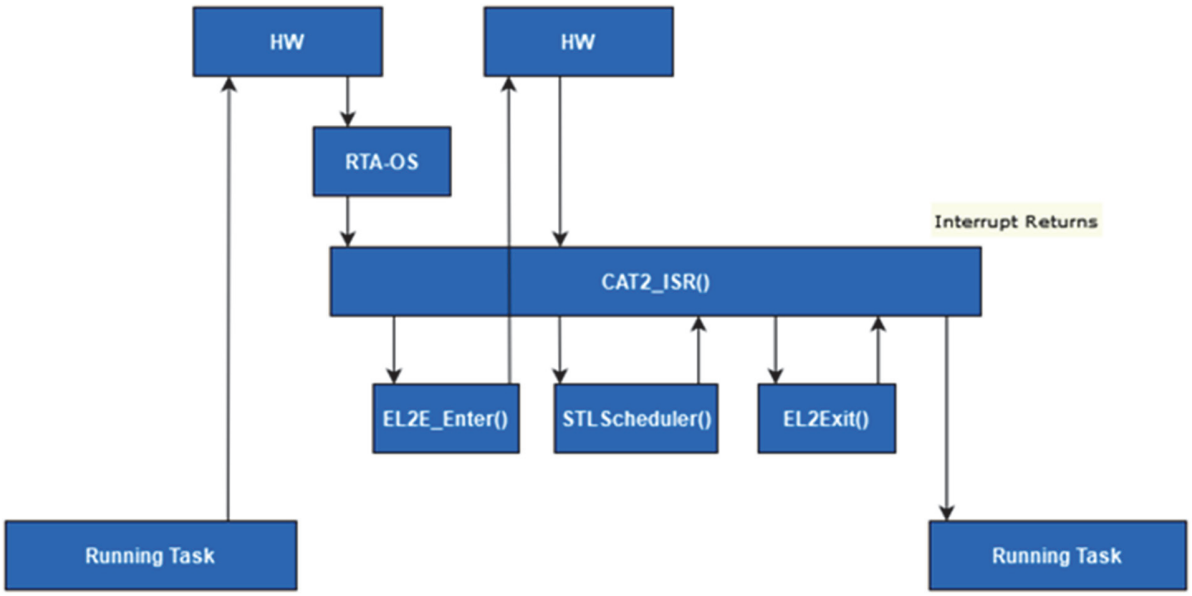

Figure 5: CAT2 sequence for OL execution of STLs

Periodic Task

A periodic task is the most common way that periodic entities are scheduled in an AUTOSAR system. These can be scheduled periodically via a schedule table or via alarms. In both cases, an OS counter is required, which gets incremented leading to the task's activation. For the STLs this will add considerable cycles before execution is handed over to the STL scheduler, resulting in a greater overall overhead for running STLs.

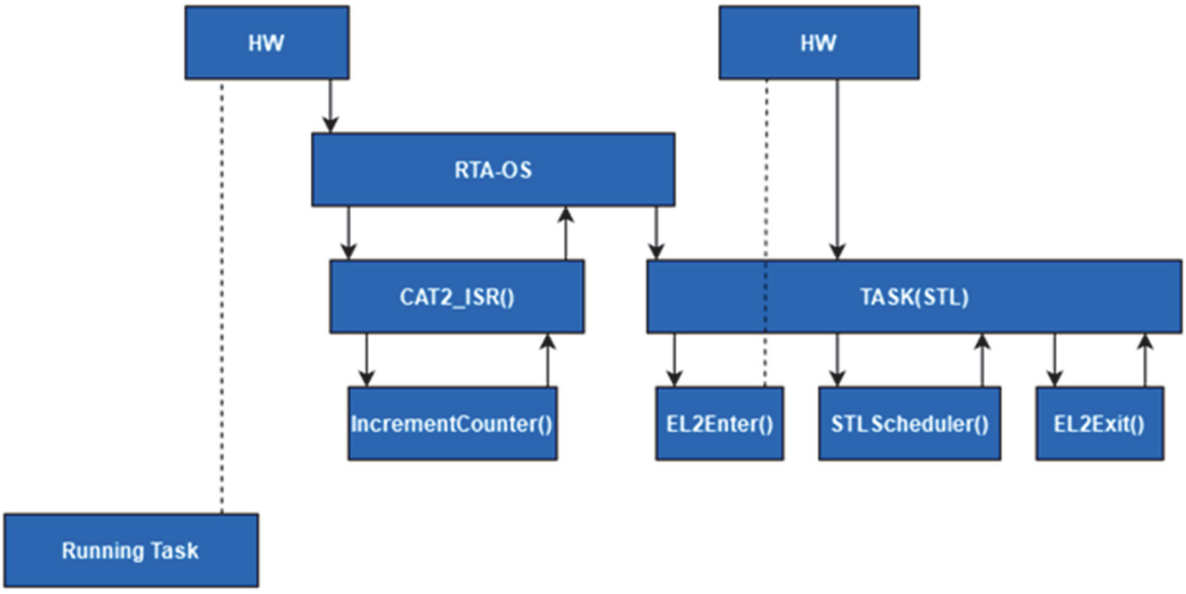

Figure 6: Task sequence for OL execution of STLsComparison

The tables below show the CPU cycles the alternative approaches took to set the context for the STL execution when running in the test environment (approximately the same timings apply for restoring the context).

Method	CPU Cycles
Entering a CAT1 ISR	149
Entering a CAT2 ISR	260
Entering a TASK via Schedule Table	700*

Table 1: Comparison of OL execution methods in classic AUTOSAR application execution

*The time to enter a task via schedule table can differ considerably depending on the number of actions/alarms/events in the configuration. Meaning in a full AUTOSAR stack application, this could be even greater.

The breakdown of the execution of the STLs is shown below

Block	Block description	CPU clock cycles
DPU	Data Processing Unit	181 619
ICU	Instruction Cache Unit	64 454
DCU	Data Cache Unit	24 630
STU	Store Unit	62 252
LSU	Load Store Unit	23 614
TCU	TCM Controller Unit	18 559
BIU	Bus Interface	103 280
GIC	Interrupt Controller	61 292
MMS	Memory System	742 247
GOV	Governor	61 623
L2AXIS	AXI-S Interface	17 491
GDU	GIC Distributor	259 016

Table 2: STL Test Part Execution Cycles

Considering the total number of CPU clock cycles for one complete execution of all tests within the STL, as well as the different scheduling options, we can see the total number of CPU cycles required to schedule and run the STLs. During this measurement, we consider the number of iterations that are passed into the SBIST scheduler API, reflecting how many tests parts to run, in a single invocation of the scheduler. When running the tests in OL mode, this figure is important, as it directly correlates to the interrupt blocking window, meaning in some systems, it may be required to run less test parts, iterations, due to a short worst case response time of the system. However, because of this, the overhead to run a set amount of tests parts will be greater, due to the overhead of scheduling within the operating system.

Method	CPU Cycles			
	10 iterations	20 iterations	50 iterations	100 iterations
CAT1 ISR	1621567	1627077	1627527	1634977
CAT2 ISR	1622677	1625277	1633077	1646077
TASK via Schedule Table	1627077	1634077	1655077	1690077

Table 3: Total number of CPU cycles for the STL execution

The table above shows the calculated CPU cycle overhead for the STLs running with the associated method. This is calculated from the observed methods CPU cycle overhead, plus the stated execution overhead of the STLs, which is found by summing the previous table, Table 2: STL Test Part Execution Cycles.

The table clearly shows that running fewer tests per invocation of the STL scheduler will increase CPU load overall, due to the extra overheads of the scheduling method. However, it may not be practical in real-world applications to run a substantial number of STL tests within one invocation of the STL scheduler, due to the scheduling effect on the rest of the system as well as the continued interrupt blocking window.

This would differ if using OLE mode, as the STLs can return if a pending interrupt is present. This would allow a system to react to an event trigger during the execution of multiple test parts.

Using OLE mode would increase the number of CPU cycles to execute the full test suite. This is because if the STL scheduler is interrupted frequently, more calls to the scheduler will be required to execute the full suite. Essentially this would yield the same behavior as calling the scheduler in OL mode executing just one test part at a time, if interrupted frequently.

A decision of the best approach cannot be drawn from the methodologies described above as the application type and specific customer requirements are unknown. Both influence which approach is favored when executing the STLs in a classic AUTOSAR system.

Integration Notes

It is possible that for a formal integration of the STLs within AUTOSAR, it may be wished to integrate the STLs via a BSW module rather than directly within an OS construct. To do this it would be possible to use either a complex device driver (CDD), or to use the Core Test (CorTst) module.

The AUTOSAR specification of the CorTst module is minimal, specifying only a handful of APIs, such as initialization functions, a main function, and some hooks. This means that integration into a CDD or CorTst would be practically identical; There would be a _Init API for calling the STL initialization mechanism and a periodic _MainFunction API for scheduling of the STLs in OL mode.

In either case, the temporary EL2 switch would be required as a pre- and post-amble to the STL APIs in order to ensure the correct exception level before entry. An example of this can be seen below:

```
void CorTst_Init(const CorTst_ConfigType* cfg)
{
    if(SUCCESS == CallEL2Function( SBIST_START_POS_STESTS_OLE,
          SBIST_LAST_POS_STESTS_OLE, sbist_scheduler_init))
    {
                /* init success*/
    }
    else
    {
            /* init failed; development error*/
    }
}

void CorTst_MainFunction(void)
{
    if(SUCCESS == CallEL2Function( SBIST_START_POS_STESTS_OLE,
          SBIST_LAST_POS_STESTS_OLE, sbist_scheduler))
    {
        /* scheduler executed tests, check results*/
        check_status_registers();
    }
    else
    {
        /* scheduling failed: development error*/
```

```
        }
}
```

Figure 7: Example CDD/CorTst integration of STLs

Since integration via a CDD or the CorTst module is simply a wrapper around the core EL2 switch and STL API invocation, the timing analysis is similar to the scheduling method used to invoke the module (within a few cycles)

Fault Handling

In a classic AUTOSAR system, hardware faults are defined as leading to the operating system 'ProtectionHook'. This is a callback that the OS triggers when a trap or serious fault has occurred. The user can then decide the appropriate response to move forwards.

In the case of a full AUTOSAR application, often the most appropriate response to a serious fault is to trigger a full hardware reset, being careful to shut the system down gracefully. In the case of the STLs, if a fault has occurred on the processor, it would also make sense for the fault to lead to the operating system and cause a total shutdown and reset.

This method allows a central location for error handling within the application, and total visibility over error detection and management.

Integration of the STLs through a module such as CorTst, as mentioned above, could then use feature of that module for error reporting. However, with the asynchronous nature of the CorTst APIs as specified by AUTOSAR, it may not be suitable. This is because the code execution following on from a STL test failure could be using faulty hardware, leading to other processor traps and undefined behavior.

Due to this reason, it would be ideal to use the watchdog to monitor a core for failing STL tests. As when a test fails, the core will execute a 'wait-for-interrupt' (WFI) instruction, putting the core to sleep. In this way, a watchdog surrounding the execution of the STL tests will catch if a test fails or gets stuck, then resetting the core.

Alternatively, it would be possible in a multi-core system, to have a secondary core that checks the status registers of the SBIST controller. This way the secondary core acts as a watchdog, able to reset the core if found to be fault via the status registers.

Summary and Conclusion

This white paper outlines the process of integrating the Arm Software Test Library (STL) with Classic AUTOSAR on Cortex-M and Cortex-R microcontrollers. The integration is designed to be straightforward, ensuring minimal complexity and avoiding significant overheads. By incorporating STL, automotive developers can enhance the diagnostic capabilities of their ECUs, providing a crucial step towards meeting Automotive Safety Integrity Level (ASIL) objectives.

All measurements shown in this white paper are derived from commercially available development boards. If further information is required on the measurements or other topics, contact your representatives at Arm or ETAS.

Glossary

Abbreviation	Explanation
ASIL	Automotive Safety Integrity Level
ASW	Application Software
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software Stack
CDD	Complex Device Driver
ECU	Electronic Control Unit
EL2	Execution level 2 (hypervisor exception level)
OL	Online (method of executing STL)
OLE	Online Event (method of executing STL)
OOR	OOR – Out of reset (method of executing STL)
STL	Software Test Libraries

Appendix

EL1 to EL2 Shim reference code example

```
.global CallEL2Function
.type CallEL2Function, @function
; ------------------------------------------------------------------------
; Function: CallEL2Function
; Arguments: r0 -> arg0
;            r1 -> arg1
;            r2 -> sbist_function to call
; ------------------------------------------------------------------------
CallEL2Function:
     cpsid if
     push { r1-r12, r14 } ; save nv reg's
     mov r4, r13          ; save OS stack
     isb
     dsb
     HVC #0
     pop { r1-r12, r14} ; pop regs
     cpsie if
     BX  lr
; ------------------------------------------------------------------------
.size CallEL2Function, . -CallEL2Function
; ------------------------------------------------------------------------

; ------------------------------------------------------------------------
.global Exit_EL2
.type Exit_EL2, @function
; ------------------------------------------------------------------------
Exit_EL2:
     pop { r4 }
     ERET
; ------------------------------------------------------------------------
.size Exit_EL2, . -Exit_EL2
; ------------------------------------------------------------------------

; ------------------------------------------------------------------------
.global EL2_HypEntry_Handler
.type EL2_HypEntry_Handler, @function
; ------------------------------------------------------------------------
EL2_HypEntry_Handler:
     CPSID if                 ; disable FIQ IRQ
     CPSID a                  ; disable ABORT
     mrs r4, SPSR_hyp
     push { r4 }
     and r4, r4, #0xfffffff0  ; Wipe SPSR_hyp.M[3:0].
     orr r4, r4, #0xA         ; set to Hyp
     msr SPSR_hyp, r4
     mrs r4, ELR_hyp
     push {r4}
     ldr r4, =continue_at_el2 ; set return point
```

```
     msr ELR_hyp, r4
     ERET
continue_at_el2:
     BLX r2                   ; branch to function passed in through r2
     pop {r4}
     msr ELR_hyp, r4
     pop { r4 }
     msr SPSR_hyp, r4
     ERET
; ------------------------------------------------------------------------
.size EL2_HypEntry_Handler, . -EL2_HypEntry_Handler
; ------------------------------------------------------------------------
```