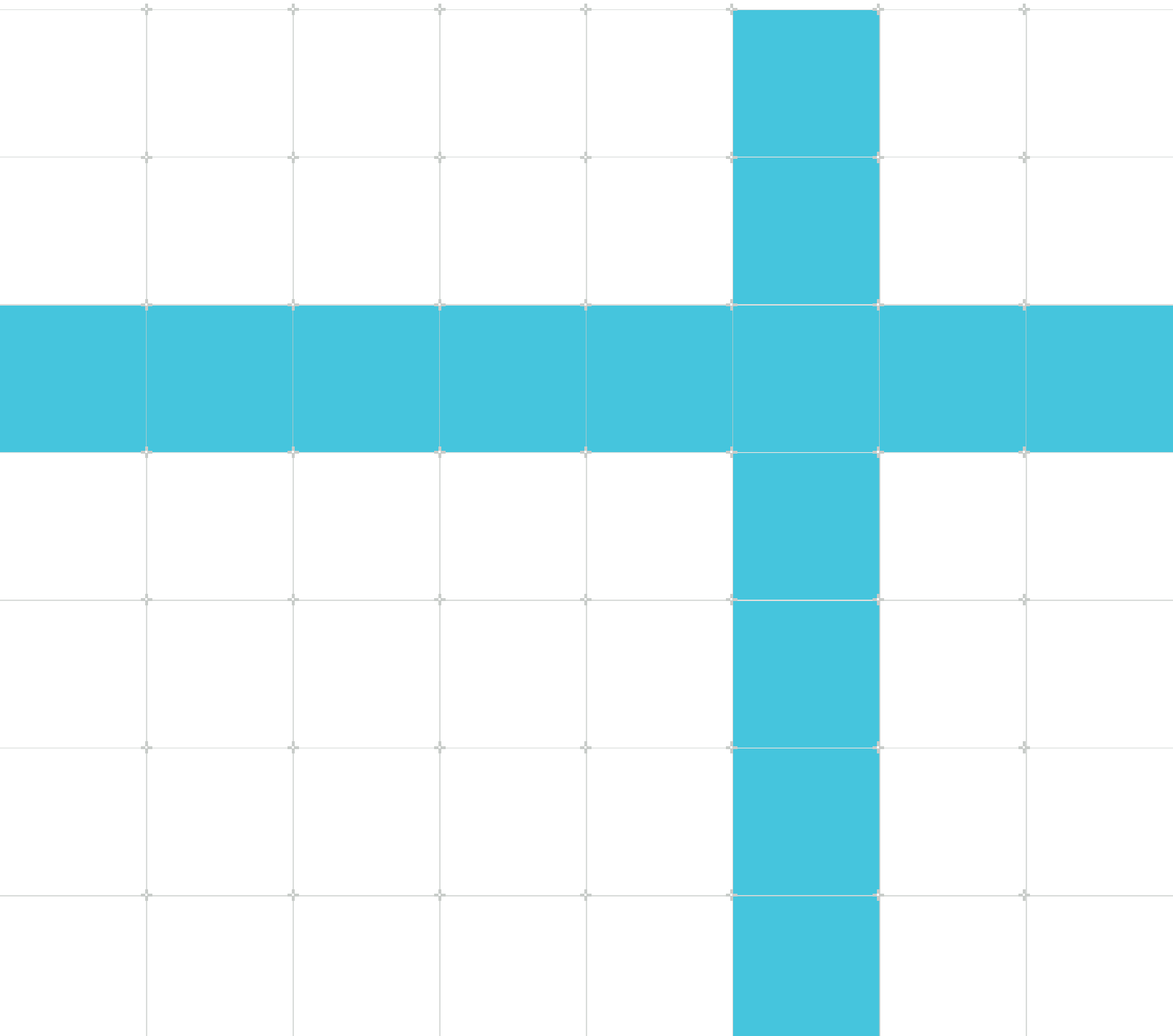
Arm STL Toolchain Interoperability Guide

Version 1

Issue 00
110388_01_00_en



Arm STL Toolchain Interoperability Guide

This document (110388_01_00_en) was issued on 2026-02-20. There might be a later issue at https://developer.arm.com/documentation/110388

The product version is 1.

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

This document is intended for use by an STL licensee. It describes a workflow which can be used to re-build Arm STL or SBIST products in a way that improves interoperability with third-party compiler toolchains. The document uses the term STL to refer to both STL and SBIST interchangeably.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Introduction

Arm® Software Test Library (STL) and Software Built-In Self-Test (SBIST) products are designed to help detect latent faults in safety-critical embedded systems. These components must typically be built using a qualified release of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety, as required by the associated *Safety Manual* for the STL or SBIST product. However, many user applications that integrate STL or SBIST products are built with different compiler toolchains.

To address such use cases, this document describes a workflow that can be used to rebuild STL or SBIST products as a static library using a qualified release of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety, and then link the static library into applications built with a toolchain that is compatible with the Arm ABI (Application Binary Interface). This approach enables interoperability while helping to maintain STL integrity and preserve the assumptions on which STL validation is based.

The document uses the term STL to refer to STL and SBIST interchangeably.

The content of this document is informational only. Any workflow, guidance, or examples described herein are provided without representation or warranty of any kind, whether express or implied, including as to accuracy, completeness, correctness, interoperability, or fitness for a particular purpose. Such information is subject to changing conditions, toolchain behaviors, scope, and available data. This document has been prepared using reasonable efforts based on information available as of the date of issue.

While the workflow described may assist in improving STL compatibility with certain third-party ABI-compatible toolchains, Arm does not represent or warrant that the workflow will ensure interoperability with any specific third-party toolchain or system environment. Toolchain-specific behaviors, assumptions, constraints, or limitations may exist and may not be known at the time of publication. As a result, the workflow may not be sufficient or appropriate in all cases.

The scope of information in this document may exceed that which Arm is required to provide, and such additional information is intended solely to assist the recipient and does not expand or modify any contractual or other obligations of Arm.

Nothing in this document constitutes certification, validation, or approval of any third-party toolchain for safety-related or other regulated use.

You acknowledge and agree that you possess the necessary expertise in system integration, safety, and toolchain validation, and that you shall be solely responsible for all the following:

- Confirming that any third-party ABI-compatible toolchain satisfies the compatibility requirements described in this document.
- Determining the suitability of the workflow for your particular system and use case.

- Verifying that the resulting STL integration behaves as expected within your specific system context.
- Completing all required STL validation and fault-injection testing in accordance with the STL *Safety Manual* and any applicable legal, regulatory, safety, and security requirements.

Notwithstanding any information or support that may be provided by Arm, you remain responsible for the design, validation, and operation of your products, and for implementing appropriate safeguards to minimize risk.

2. STL Toolchain Interoperability Workflow

This chapter describes a workflow that can be used to improve toolchain interoperability for an STL.

The workflow involves using a suitable release of Arm® Compiler for Embedded FuSa or Arm Compiler for Functional Safety to re-build the STL as a static library, and then using an Arm Application Binary Interface (ABI) compatible toolchain to link the static library into the user's application.

Arm STL products must be built using Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety releases. Attempting to build Arm STL source code directly with a different compiler toolchain is not described in this document and may lead to integration issues or unexpected behavior. STL correctness and safety depend on compiler-specific behavior, including code generation, inline assembly handling, and memory layout.

Users must not modify the STL source code. This workflow assumes that only the build system (for example, Makefiles and linker scripts) is adapted to produce a static library and integrate it into an ABI-compatible build environment.


Tip

This workflow can also help to resolve common compatibility issues a user may encounter when using a third-party ABI compatible toolchain to integrate an Arm STL product into the user's application. For example:

- STL source code being incompatible with the third-party toolchain.
- STL build options or Makefiles being incompatible with the third-party toolchain.
- A requirement to use a specific third-party toolchain for the user's application, alongside a requirement to use a specific version of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety to build the STL.

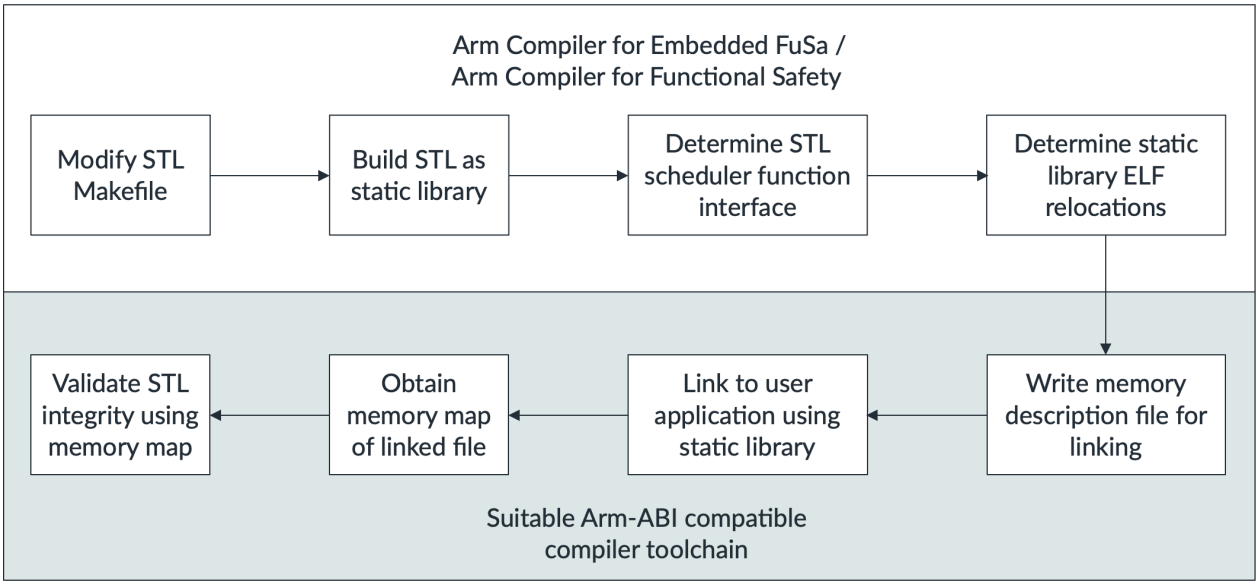
For example, if an STL cannot be built using a specific third-party toolchain, the workflow described in this document can still be used to link the STL with the third-party toolchain.

This document splits the workflow into the following high-level steps:

1. Generating a static library for the STL
2. Determining compatibility of the static library interface
3. Linking the static library
4. Validating the STL memory map
5. Exporting the STL for integration as a static library

For more information about each of these steps, see the corresponding section within this chapter. An overview of the workflow is provided in Figure 1.

Figure 2-1: Overview of STL toolchain interoperability workflow



2.1 Workflow requirements

This section lists the requirements that must be met when using the workflow described in this document.

- You must have a valid license to build and distribute the Arm® STL product you are working with.
- You must have a valid installation of and software tools license for building the STL using the applicable version of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety.
- You must have a valid installation of the applicable version of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety.
- You must not attempt to build the STL source code using a compiler toolchain other than the specific version of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety listed in the *Safety Manual* for the STL product.
- You must know the desired STL configuration you are building for. If you need to support multiple STL configurations, you may need to build a unique STL static library file for each configuration.
- The compiler toolchain used to link the STL static library file with the rest of your application must be compatible with the Arm Application Binary Interface (ABI). For more information about the ABI compatibility requirements relevant to the workflow described in this document, see Step 2: Determining compatibility of the static library interface.
- You must continue to meet the requirements and assumptions of use specified by the *User Guide* and/or *Safety Manual* for the Arm STL product you are using.
- You must independently validate that the contents of the STL have been placed correctly in memory when using the STL as a static library.

Expected user knowledge

This workflow assumes that the reader has familiarity with the following topics that are required to use the workflow:

- The Arm® Application Procedure Call Standard (AAPCS). The AAPCS is described in the following documents:
  - Procedure Call Standard for the Arm Architecture
  - Procedure Call Standard for the Arm 64-bit Architecture (AArch64)
- Linker script or scatter file syntax
- ELF object file structure and relocation types
- Behavior of startup code and memory initialization for the ABI-compatible third-party toolchain
- STL configuration and memory layout constraints

2.2 Step 1: Generating a static library for the STL

This section describes how an STL deliverable from Arm® may be modified to build a static library.

STL deliverables include Makefiles that can be used to build the STL. Typically, a Makefile will build the STL for the purpose of running an Out-of-Box (OOB) test in a fixed configuration by generating a statically linked ELF image instead of building a static library.

In order to generate a static library for an STL, you must modify the Makefile provided by Arm using the following steps:

1. Identify the Makefile you need to modify. You may find the name of this file in the *Safety Manual* or *User Guide* for the STL you are using. For example, for the Cortex®-R52+ STL, this file is called `Makefile`.
2. By reading through the Makefile, determine the Makefile variable or variables that include a list of the object files which must be included in the static library. For example, for the Cortex-R52+ STL, these variables are:
   - `atest_objs_c`
   - `stest_objs_s`
3. Modify the rule that generates a statically linked ELF image to add a command to build a static library instead. Use the librarian tool `armar` included in all releases of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety for this purpose. For example, for the Cortex-R52+ STL, you can add the following command:

   ```
   armar --create libstl.a $(atest_objs_c) $(stest_objs_s)
   ```

   The static library file name in this example is `libstl.a`.
4. After modifying the Makefile, use it with the version of Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety specified in the *Release Notes*, *Safety Manual*, or *User Guide* for the STL to build the static library file.

2.3 Step 2: Determining compatibility of the static library interface

This section describes how to determine the compatibility of the static library interface with the toolchain used to link the STL. In the context of this section, compatibility refers to the compatibility of ELF object files in the static library with the third-party toolchain used to link the static library with a user's application.

The static library interface for an STL is typically limited to a small set of C functions with scalar parameters and return values, passed using registers in accordance with the Arm® Application Procedure Call Standard (AAPCS).

There are no exposed shared data structures, complex types, or C++ exception interfaces. The compatibility checks in this section therefore focus on register use, function prototypes, and relocation compatibility, rather than full Arm ABI semantics.

The interface may include the following items that must be examined:

- C functions to configure, initialize, and start the STL scheduler. For such functions, you must ensure that the toolchain used to link the STL uses the same registers for function parameters and return values as Arm Compiler for Embedded FuSa or Arm Compiler for Functional Safety.
- Header files that must be used with the static library. You must ensure that the source code within these header files is compatible with the toolchain used to link the STL.
- ELF file relocations that a linker must resolve when linking the static library. You must ensure that the toolchain used to link the STL supports all unresolved relocation types in the static library.

The following sub-sections provide further detail about each of the above items.

C functions to configure, initialize, and start the STL scheduler

STL tests are never meant to be run directly by a user. Instead, a user must run a higher-level function or set of functions that schedules STL tests to run appropriately based on the STL configuration. This higher-level function or set of functions is referred to as the STL scheduler.

The STL *User Guide* or *Safety Manual* should include a description and usage instructions for the STL scheduler. Typically, an STL scheduler is used via C functions that can configure, initialize, and start the scheduler. For example, for the Cortex®-R52+ STL, the STL scheduler functions are as follows:

- `int32_t sbist_scheduler_init(uint32_t <paramA>, uint32_t <paramB>);`
- `int32_t sbist_scheduler(int32_t <paramC>, int32_t <paramD>);`

where `<paramA>`, `<paramB>`, `<paramC>`, and `<paramD>` represent different function parameters.

These functions form the C function interface for the static library. You must ensure that you call these functions correctly when linking with the static library.

For each such function, Arm® Compiler for Embedded FuSa and Arm Compiler for Functional Safety will use registers for function parameters and return values according to the Procedure Call Standard for the Arm Architecture or the Procedure Call Standard for the Arm 64-bit Architecture (AArch64). You must ensure that the toolchain you use to link the static library uses the same registers as Arm Compiler for Embedded FuSa and Arm Compiler for Functional Safety. Your toolchain vendor may be able to provide you with information about register allocation.

For example, for the STL scheduler functions from the Cortex-R52+ STL, Arm Compiler for Embedded FuSa and Arm Compiler for Functional Safety use the following registers:

Function	Parameter registers	Return value register
`sbist_scheduler_init()`	`R0` for `<paramA>`, `R1` for `<paramB>`	`R0`
`sbist_scheduler()`	`R0` for `<paramC>`, `R1` for `<paramD>`	`R0`

Header files that must be used with the static library

Each STL scheduler function will have a function prototype defined in a `#include` header file. You must make sure to include this header file or the relevant function prototype from within this header file when compiling code that calls the corresponding STL scheduler function. You must not rely on the default implicit function prototype used by your compiler toolchain, as this can result in unexpected run-time behavior.

For example, the header file `<codename>_sbist_scheduler.h` provides function prototypes for the Cortex®-R52+ STL scheduler functions, where `<codename>` is the processor codename for the Cortex-R52+ processor.

Additionally, the relevant header files may contain useful C preprocessor macros that you can use to simplify the steps required to configure, initialize, and start the STL scheduler.

ELF file relocations that a linker must resolve when linking the static library

The static library may contain one or more unique ELF file relocation types that must be resolved when placing the STL contents in memory. You must ensure that the linker from the toolchain used to link the static library with user code supports each unique ELF file relocation type in the static library.

You can use the `fromelf` utility from Arm® Compiler for Embedded FuSa or Arm Compiler for Functional Safety to obtain a list of unique ELF file relocation types. The `--text -r` options for `fromelf` can be used to print relocation information in a static library file.

For example, for the Cortex®-R52+ STL built as a static library, the output of using the `fromelf` utility with the `--text -r` options may contain instances of relocation information like the following:

```
** Section #4 '<section>' (SHT_REL)
    Size   : 48 bytes (alignment 4)
    Symbol table #6 '.symtab'
    6 relocations applied to section #3 '<section>'

    #      Offset    Relocation Type         Wrt Symbol    Defined in
    ============================================================
```

```
    0  0x000000e8   28 R_ARM_CALL         232 <symbol>  #3 '<section>'
    1  0x00000224   29 R_ARM_JUMP24       241 <symbol>  #3 '<section>'
    2  0x000005dc   43 R_ARM_MOVW_ABS_NC  236 <symbol>  #3 '<section>'
    3  0x000005e0   44 R_ARM_MOVT_ABS     236 <symbol>  #3 '<section>'
    4  0x000006a0   43 R_ARM_MOVW_ABS_NC  237 <symbol>  #3 '<section>'
    5  0x000006a4   44 R_ARM_MOVT_ABS     237 <symbol>  #3 '<section>'
```

The `Relocation Type` column provides information about each relocation type found:

- A number, which is the relocation code. For example, `28`.
- A name, which is the relocation name used by Arm Compiler for Embedded FuSa and Arm Compiler for Functional Safety. For example, `R_ARM_CALL`. Other toolchains may use different relocation names for the same relocation code.

More information about relocation types is available in the following:

- For AArch32 state, in the *Relocation types* section of the *ELF for the Arm Architecture* document.
- For AArch64 state, in the *Relocation* section of the *ELF for the Arm 64-bit Architecture (AArch64)* document.

Tip

The output from `fromelf --text -r` can be processed using Unix text processing utilities such as `grep`, `tr`, `cut`, and `sort` to quickly create a list of unique relocation types. For example:

```
fromelf --text -r /path/to/libstl.a | grep -e "[0-9][0-9]* R_.*" \
| tr -s " " | cut -d " " -f 4,5 | sort -u
```

can generate a list of unique relocation types like the following:

```
10 R_ARM_THM_CALL
2 R_ARM_ABS32
28 R_ARM_CALL
29 R_ARM_JUMP24
30 R_ARM_THM_JUMP24
42 R_ARM_PREL31
43 R_ARM_MOVW_ABS_NC
44 R_ARM_MOVT_ABS
51 R_ARM_THM_JUMP1
```

2.4 Step 3: Linking the static library

This section describes the requirements for linking the STL built as a static library.

The scheduler for Arm® STL products works by sequentially running STL tests from a test table. The order of tests in the test table is hardcoded when the STL is built, and reflects the order in which the STL scheduler runs the STL tests. The memory layout of the application into which the STL is integrated must maintain the same order for the placement of STL tests.

Additionally, each STL requires the user to reserve certain address ranges for exclusive use by the STL. These address ranges may need to be reserved for one or more of the following STL components:

- Test code
- Scheduler code
- Stack and heap space
- Scratchpad memory
- Fault status registers

Therefore, linking the STL built as a static library has the following requirements that a user must follow:

- All STL test code must be placed in a contiguous memory region.
- All STL test code must be placed in memory in the same order as in the test table.
- User code must not overlap with any of the following:
  - STL test code.
  - STL scheduler code.
  - STL scratchpad memory locations.
  - STL stack and heap space.
  - STL fault status registers, for an STL built with memory-mapped fault status registers.

The linker script, scatter file, or any other code and data placement mechanism used by the toolchain used to link the STL as a static library must be written to account for the above requirements.

Some STL configurations assume that specific memory regions are zero-initialized before STL execution begins.

When using a toolchain that does not automatically initialize such regions (for example, via scatter loading or startup code), you must ensure that required memory is manually zero-initialized before invoking the STL scheduler. Failure to do so may result in unexpected run-time behavior.

The order in which STL test code is expected to be placed by the STL scheduler can be determined by reading through the example linker scripts or scatter files provided by Arm as part of the STL deliverable. For example, for the Cortex®-R52+ STL, such files are located in the `link` directory in the STL deliverable package.

Information about the address ranges for STL scratchpad memory locations, STL stack and heap space, and STL fault status registers may be included in one or more of the following resources:

- The *User Guide* or *Safety Manual* for the STL.
- The configuration files for the STL.

- The global definitions C header file for the STL. For example, for the Cortex-R52+ STL, the `<codename>_sbist_global_defs.hs` header file, where `<codename>` is the processor codename for the Cortex-R52+ processor.

Certain STL deliverables include a Python script that is used to generate example linker scripts and/or scatter files to use for the Out-of-Box (OOB) validation tests for the STL. You may be able to modify this Python script yourself to generate a linker script / scatter file snippet for use with your own application instead of manually writing one.

2.5 Step 4: Validating the STL memory map

This section includes recommendations for validating the use of the STL when linked using a static library built according to the workflow described in this document.

In the context of this section, validation refers only to validating the workflow described in this document. You must still separately ensure that you meet all the validation, testing, and fault injection analysis requirements described in the *User Guide* and/or *Safety Manual* for the Arm® STL product you are using.

To help ensure correct operation of the STL, you must validate that the requirements mentioned in Step 3: Linking the static library have been successfully met. This can be done by analyzing the output memory map of an application that has been linked with the STL.

Different compiler toolchains have different output memory map formats. Typically an output memory map provides the following information:

- A list of output ELF sections, with the base address, size, and ELF image attributes for each section.
- A list of input ELF sections included in each output ELF section, with the base address, size, and ELF image attributes for each section.

By comparing this information with the information obtained in Step 3: Linking the static library, you should be able to validate that the following conditions are true:

- All expected STL functions are present in the final ELF binary.
- All STL sections are accounted for in the final ELF binary.
- There is no overlap between STL code and data, and non-STL code and data from your own application.
- All STL code and data is only present in the expected memory regions that were reserved for the STL.
- There is no code or data present in any memory regions that were reserved for use by the STL as scratchpad, stack, or heap memory.

- No STL code or data included in the static library and required by the STL has been inadvertently omitted from the memory map.

For more information about how to obtain an output memory map for the toolchain used to link the STL, refer to the toolchain documentation. The following table lists example command-line options that can be used to generate an output memory map as part of the link step for different toolchains:

Toolchain	Output memory map command-line option
Arm Compiler for Embedded FuSa	`--map`
Arm Compiler for Functional Safety	`--map`
Arm GNU Toolchain	`-Map`

2.6 Step 5: Exporting the STL for integration as a static library

This section provides an overview of what you must include if you need to distribute a package containing a static library build of the STL to another user who integrates the STL into their application.

When distributing a static library build of STL for integration into a separate application, you may find it helpful to organize the required files into a structured export package.

While the exact contents and structure of such an export package depend on the integration requirements, an export package must typically include the following items:

- The static library file.
- The required STL header files for the scheduler interface.
- A list of memory regions that must be reserved for exclusive use by the STL.
- Linker script or scatter file snippets for linking the STL as appropriate.
- Lists of STL function names and STL ELF section names for validation.

Proprietary Notice

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.

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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
01-00	20 February 2026	Non-Confidential	Initial release

Change history

Arm® does not provide a list of changes in each release in the release history of this document.

Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

Tip

A useful tip that might make it easier, better or faster to perform a task.

Remember

A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the following tables provides direct access to the online version of the document.

Arm product resources	Document ID	Confidentiality
Arm Compiler Functional Safety 6.6 documentation index	KA005063	Non-Confidential
Arm Compiler for Embedded FuSa 6.16LTS documentation index	KA005062	Non-Confidential
Arm Compiler for Embedded FuSa 6.22LTS documentation index	KA006002	Non-Confidential
ELF for the Arm 64-bit Architecture (AArch64)	–	Non-Confidential
ELF for the Arm Architecture	–	Non-Confidential