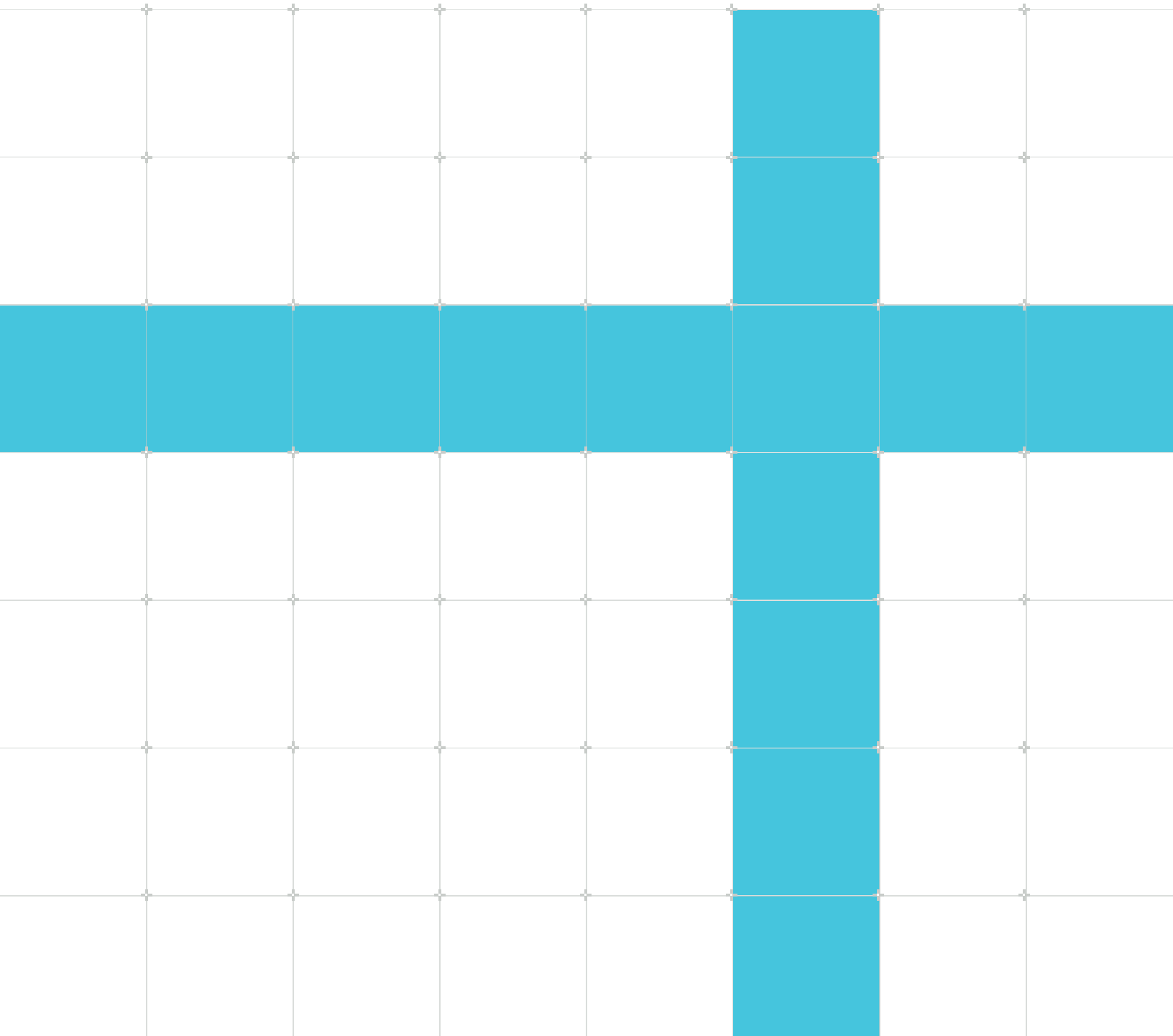
Using the CoreSight ELA-500 Embedded Logic Analyzer with Arm DS-5

Version 1.0

Issue 01
102623_0100_01_en



Using the CoreSight ELA-500 Embedded Logic Analyzer with Arm DS-5

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2020	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

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.

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.

Contents

1. Introduction

The Arm CoreSight ELA-500 Embedded Logic Analyzer provides low level signal visibility into Arm IP and third party IP. When used with a processor, it provides visibility of load, stores, speculative fetches, cache activity, and transaction life cycle, none of which are available through instruction tracing.

CoreSight ELA-500 enables swift hardware assisted debug of otherwise hard-to-trace issues, including data corruption and dead/live locks. As well as accelerating debug cycles during complex IP bring up, it provides extra assistance for post deployment debug.

CoreSight ELA-500 offers on-chip visibility of both Arm and proprietary IP blocks. Trigger conditions can be programmed over standard debug interfaces either directly by an on-chip processor or an external debugger.

The Problem

One of the most common deadlock scenarios can be caused when a processor initiates memory transactions to a location in the system in which no bus slave exists or the bus slave has limitations such as not being able to handle burst transactions. This type of incomplete transaction can ultimately lead to the processor locking-up (deadlock).

In a perfect world, systems should be designed in such a way that all the entire physical memory map is fully populated. This means that all memory transactions, to all addresses, will correctly respond with either a valid transaction result or a bus fault. However, for certain designs this may not always be the case. The aggressive speculation and prefetching performed by Arm processors mean that these memory map “holes” are more likely to be exposed by incorrect software, even if these memory “holes” are not explicitly referenced by software.

Software can prevent this by correctly configuring the MMU translation tables to accurately describe the physical memory map. Software should configure any memory map “holes” as being Invalid. Configuring the MMU this way will prevent the processor from making any physical bus transactions to that location, and ultimately preventing this type of deadlock scenario.

Debugging these types of deadlock scenarios pose an issue when debugging using traditional methods, such as external debug, and instruction / data trace. A processor core which has locked-up due to an incomplete transaction, might not be able to enter halt mode debug. Effectively, the external debugger is unable to break the processor and inspect its internal state. Trace capture might still be available, but it will not provide any record of the speculative or prefetched transactions that could be responsible for the deadlock.

The Solution

The CoreSight ELA-500 can be used in this scenario to trace the external bus transactions made by the processor (both explicitly and speculatively). This tutorial intends to show the use case scripting capabilities of DS-5, and demonstrate the example CoreSight ELA-500 use case script shipped with Arm DS-5 Development Studio.

About the CoreSight ELA-500

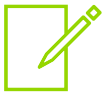
The ELA-500 can be implemented with up to 12 Signal Groups, each containing 64, 128, or 256 signals. The connections between the signals in the signal groups is dependent on the system and the IP that it is connected to. The specific signal interfaces is documented in the relevant documentation (low-level signal description documents like this are typically not publicly available, and are made available only to licensees of the Arm IP). Arm IP connected to an ELA is supplied with a JSON file which documents and annotates the signal group connections for that particular IP, in a machine-readable format. The JSON file can be interpreted by DS-5 to allow seamless debugging of a piece of IP using DS-5 and the ELA.

Signals typically consist of debug signals (status or output), and qualifiers (trigger). Qualifier signals might be required to determine that the debug signal is valid. Debug signals are valid when the qualifier signal(s) are asserted.

The System

For the purposes of this tutorial, the Cortex-A72 + ELA-500 system utilizes the LAK-500A. The LAK-500A is an Integration Kit for the ELA-500, and the Cortex-A72, it is an add-on to the ELA-500. The LAK-500A exposes a number of pre-defined debug observation ports to the Cortex-A72 (Signal Groups), and provides the corresponding JSON signal mapping file.

As part of the LAK-500A, one of the debug observation ports to the Cortex-A72 exposes the physical read address signal bus `ARADDR` and an address valid signal, `ARVALID`.



These signal names have been obfuscated for this tutorial.

These signals are required to determine the read addresses issued by the core, prior to the “lock-up”. Post analysis of these read transactions will help identify which transaction may have caused the fault.

2. Before you begin

Make sure you have installed DS-5 v5.26 or later.

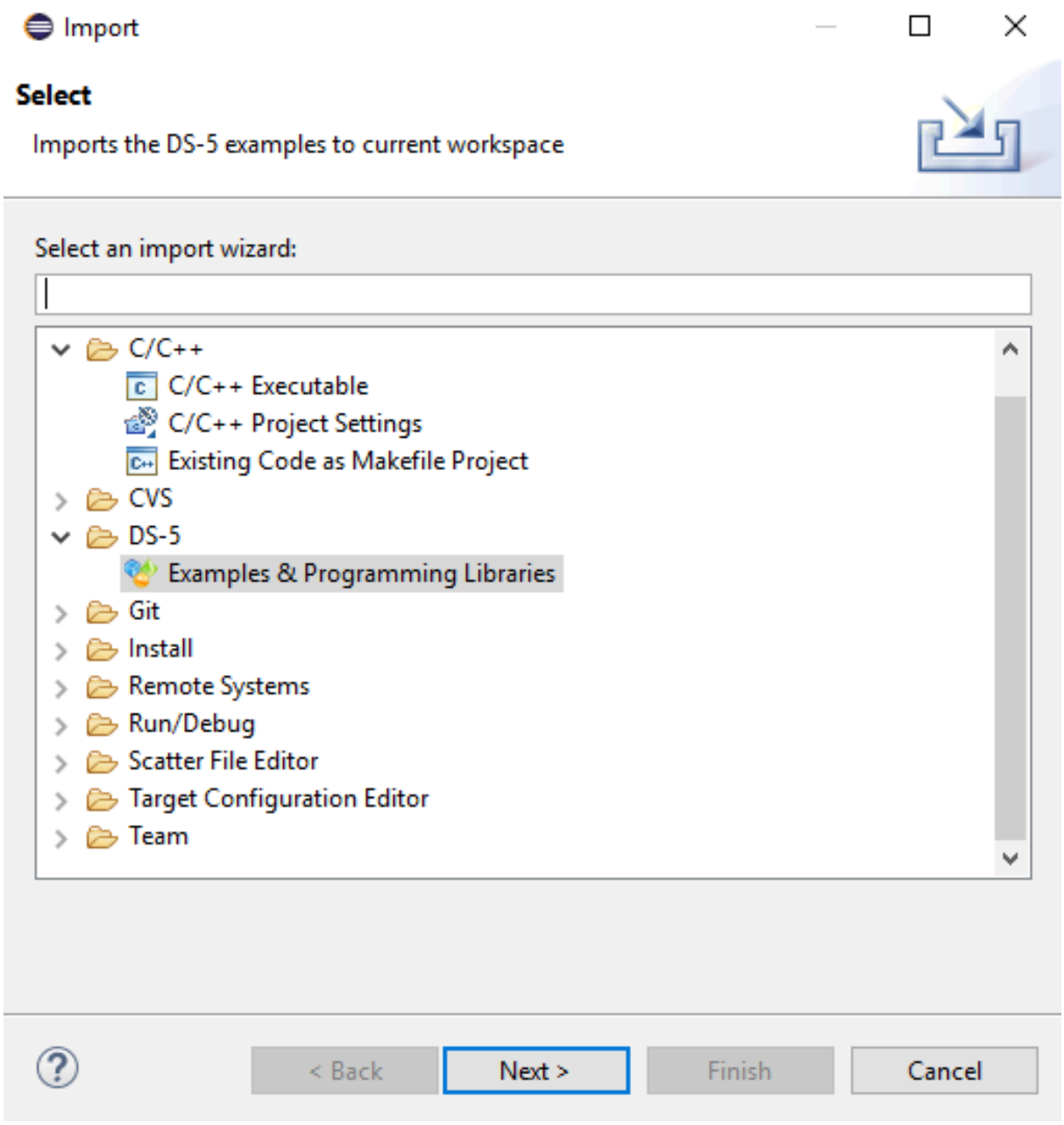
The scripts that enable DS-5 to work with the ELA were added to DS-5 v5.25. However the Use case scripts import method that we refer to below, was added in DS-5 v5.26.

3. Importing the CoreSight ELA-500 DTSL Use case scripts

With the following steps, you can import the CoreSight ELA-500 DTSL Use case scripts:

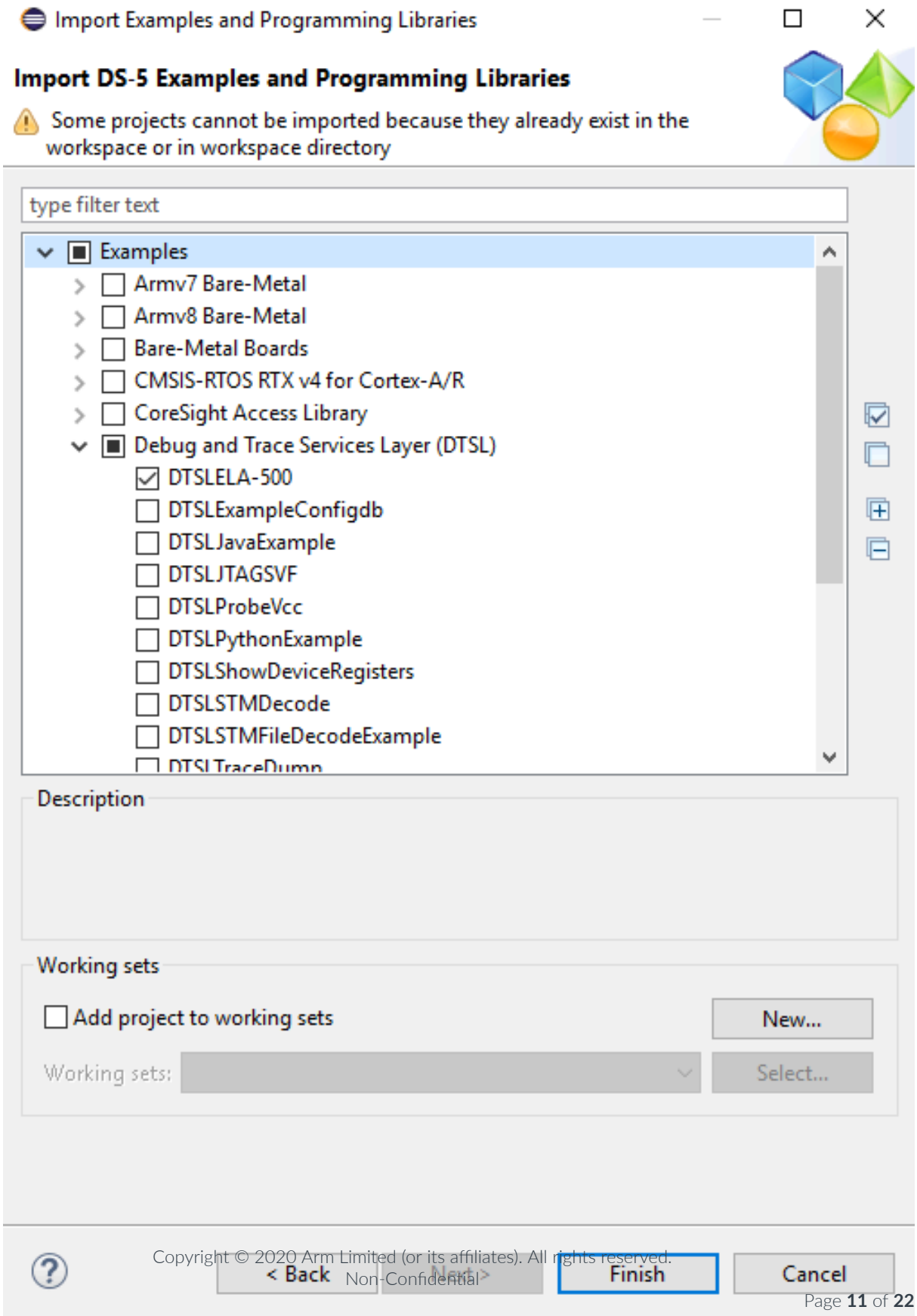
1. Launch Eclipse for DS-5 from the Start Menu.
2. Select a Workspace for your DS-5 projects. The default workspace is fine.
3. Close the Welcome screen, if it appears.
4. Select **File** > **Import...** to open the Import Selection dialog.
5. Expand the DS-5 group and select Examples and Programming Libraries, click Next.

Figure 3-1: Import Examples and Programming Libraries



6. Expand the **Examples** group, then expand the **Debug and Trace Services Layer (DTSL)** group.
7. Select **DTSLELA-500**.

Figure 3-2: DTSLELA-500 check-box under Debug and Trace Services Layer (DTSL)



8. Click **Finish**.

   Result: The **Project Explorer** view populates with the project.

4. Configuring the CoreSight ELA-500 DTSL Use case scripts

To configure the ELA-500, you can either edit a use case script or use the configuration GUI interface. The application specific use case script allows you to script a specific debug recipe. The debug recipe is used to debug a specific debug scenario with the ELA-500. An example of this can be found by navigating to the following use case script:

Scripts window > **Use case** > **DTSLELA-500** > **ela_example.py** > **Configure ELA**

For this demonstration, we will use the GUI ELA-500 Configuration Utility to configure the ELA-500 for our specific debug scenario.

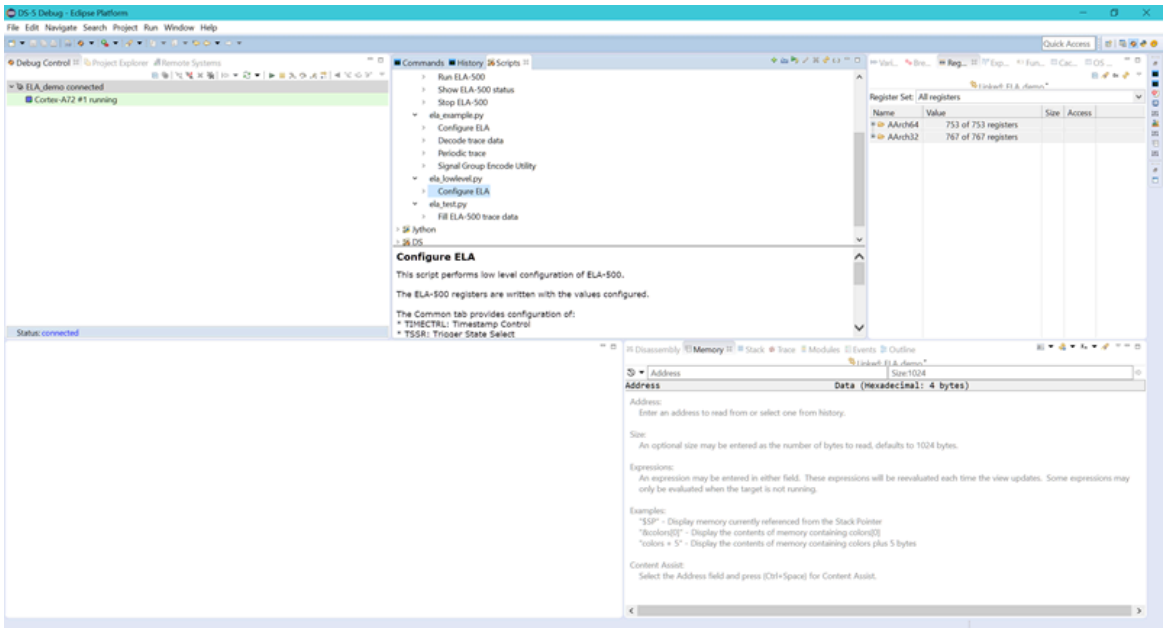
Note

DS-5 must be connected to the target SoC before starting configuration.

---

1. Connect to the target.
2. Open the GUI ELA-500 configuration utility:
   a. Navigate to: **Scripts window** > **Use case** > **DTSLELA-500** > **ela_lowlevel.py** > **Configure ELA**
   b. Right click **Configure ELA** and select **Configure**

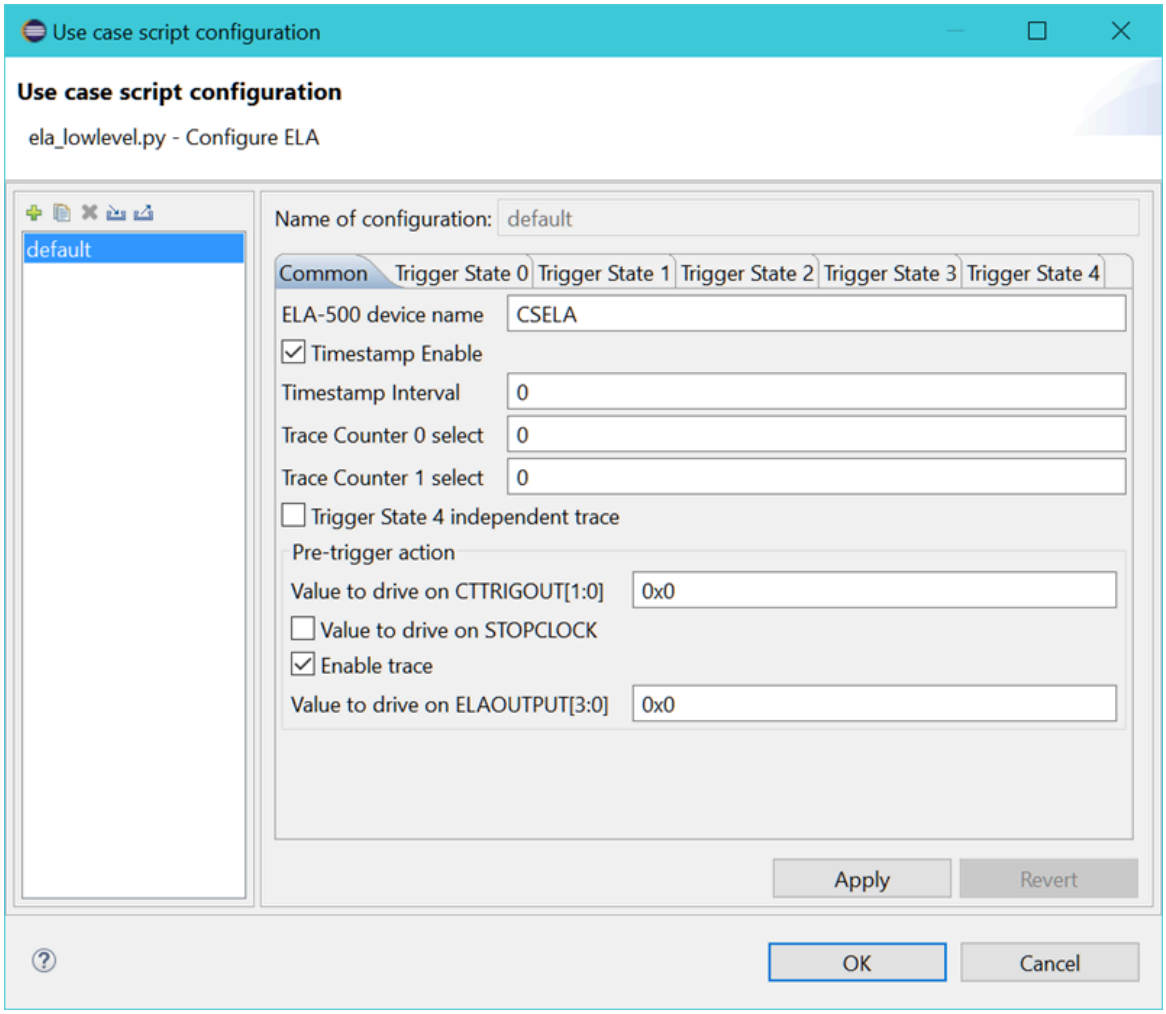
Figure 4-1: Configure option under the scripts menu



3. Configure the common controls:
   a. Under the **Common** tab, in the **Pre-trigger action** section, select **Enable trace**.

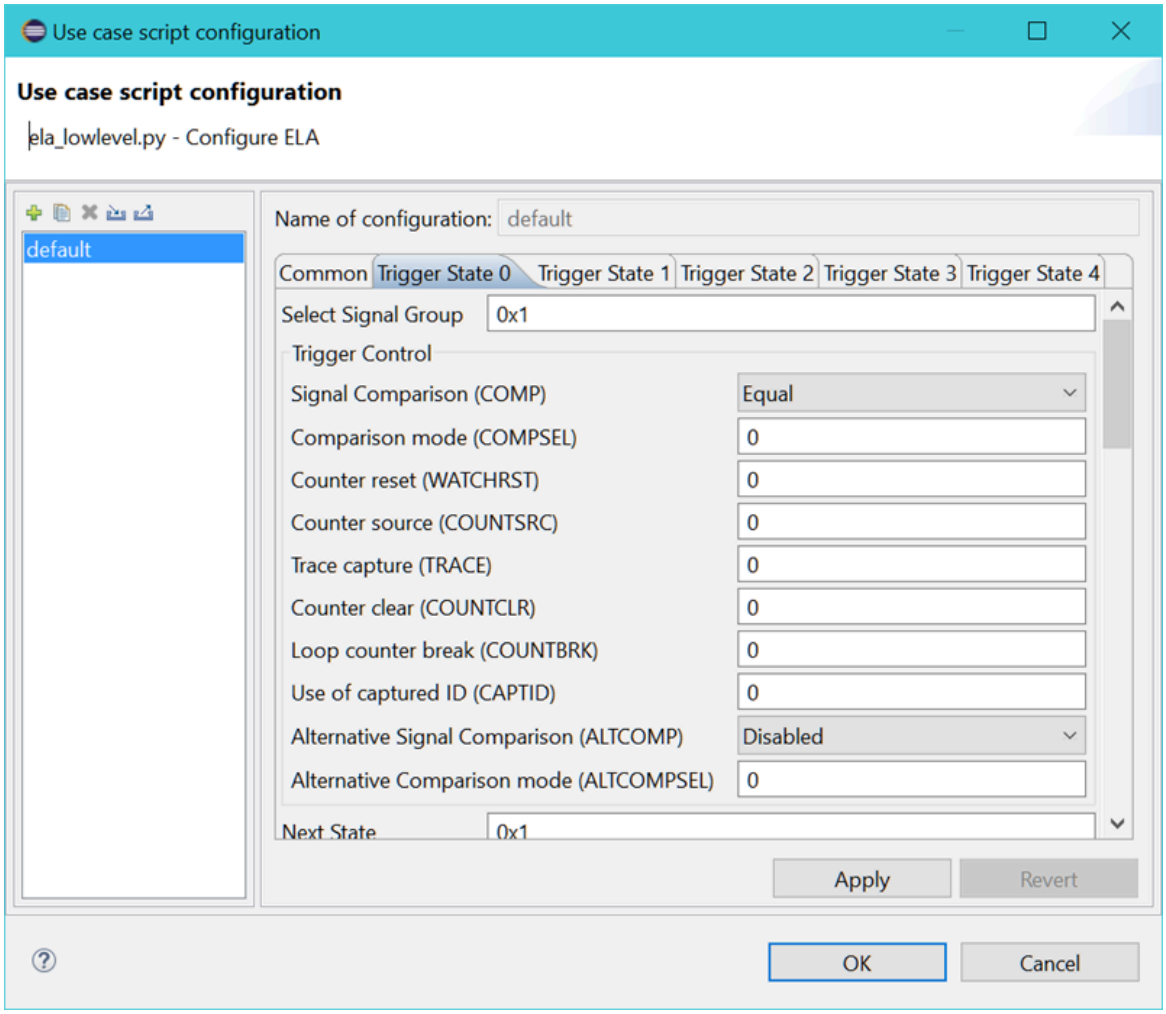
      This configures the ELA to start tracing when it is enabled - it sets `PTACTION.TRACE` so that trace becomes active when the ELA-500 is enabled. When trace is active, trace capture can be controlled to capture on either each ELA clock cycle, a trigger Signal Comparison match, or a trigger Counter Comparison match.

      Figure 4-2: Enable trace check-box under the Common tab



   b. Click **Apply**
4. We now need to configure our initial trigger:
   a. Open the **Trigger State 0** tab.

Figure 4-3: Trigger State 0 tab



b. Set **Select Signal Group** value to `0x1`.

   This selects the Signal Group we want to trigger on, which includes the qualifier signal(s), and sets `SIGSEL0 == 0x1`. This means that Trigger State 0 will be associated with the trigger signals in Signal Group 0. The ELA-500 uses a 'ones hot' encoding for the Signal Group in the Signal Select registers. In our example, Cortex-72 + ELA-500 + LAK-500A, the `RVALID` signal resides in Signal Group 0. To locate the `RVALID` signal location for other targets, check your IP's corresponding JSON file or documentation.

c. Set **Signal Comparison (COMP)** to `Equal`.

   This sets the Signal Comparison condition. In this case, we want to trigger when the `ARVALID` signal is valid (ACTIVE HIGH).

d. Set the **Next state** value to `0x1`.

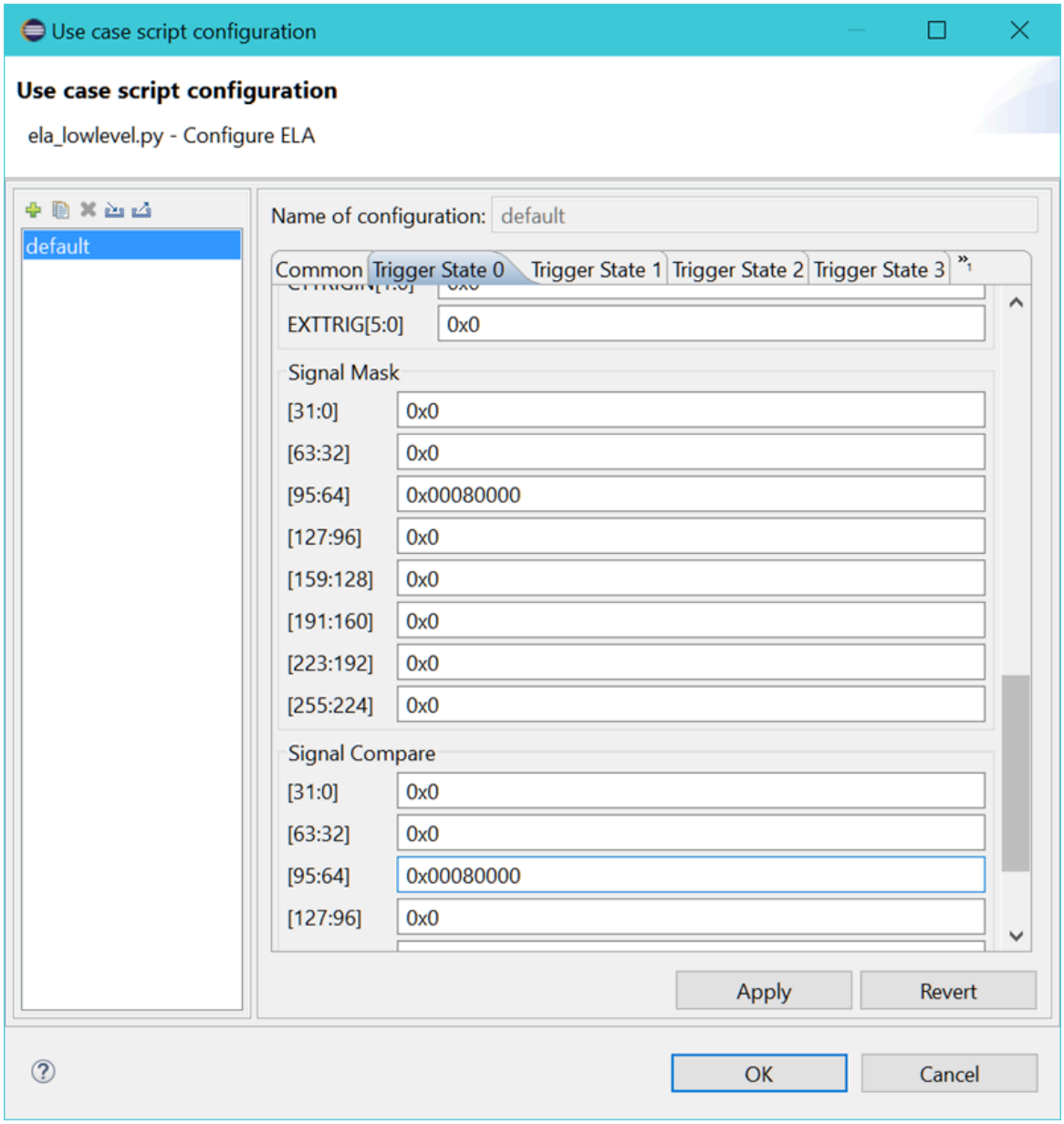
Here we set the Next state. This is the ELA state we will enter when we meet the trigger condition. In our case we want to capture on each ARVALID assertion, which uses the 'ones hot' for Trigger state 0.

e. Set both the **Signal Mask [95:64]** and **Signal Compare [95:64]** fields to `0x00080000`.

We need to set Trigger State 0's Signal Compare and Signal Mask values for Signal Group 0 to monitor the ARVALID signal. The bit position of the ARVALID signal is documented in your IPs corresponding JSON file or documentation.

In our example, ARVALID is mapped to bit 83, so we need to specify `0x00080000` in both the Signal Mask [95:64] and Signal Compare [95:64] fields.

Figure 4-4: Signal Mask and Signal Compare fields set to `0x00080000`



f. Click **Apply** > **OK**.

5. Running the DS-5 ELA use case scripts

Running the DS-5 ELA use case scripts:

1. Program the ELA configuration registers:
   a. Navigate to: **Scripts window** > **Use case** > **DTSLELA-500** > **ela_lowlevel.py** > **Configure ELA**
   b. Right-click **Configure ELA** and select **Run...**
2. Run the ELA:
   a. Navigate to: **Scripts window** > **Use case** > **DTSLELA-500** > **ela_control.py** > **Run ELA-500**
   b. Right click **Run ELA-500** and select **Run...**
3. Run the target.

   Result: The target will run and the ELA will be monitoring the input Signal Group for the trigger conditions.

6. Capturing the ELA trace data

Capturing the ELA trace data:

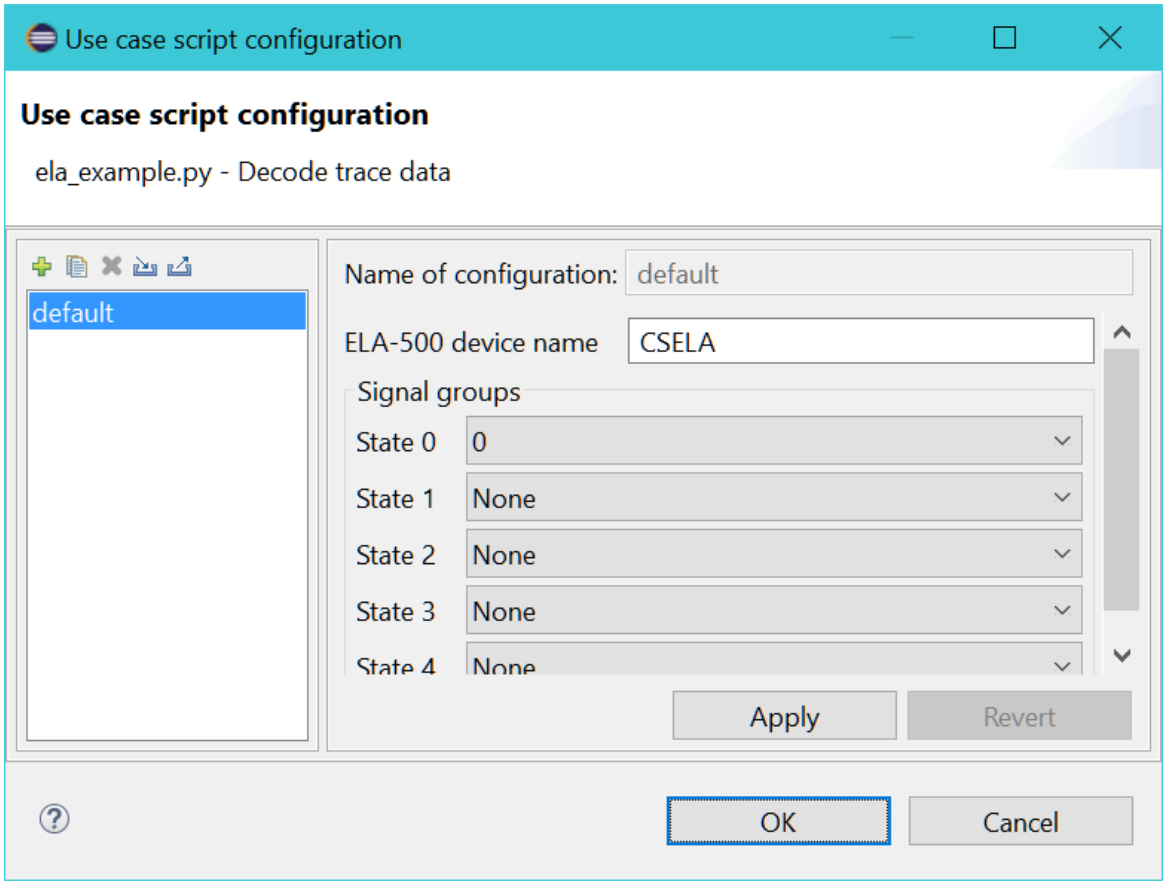
1. The core is unable to enter halt mode debug in our debug scenario, so we need to stop the ELA:
   a. Navigate to **Scripts window** > **Use case** > **DTSLELA-500** > **ela_control.py** > **Stop ELA-500**
   b. Right-click **Stop ELA-500** and select **Run...**
2. Dump and decode the ELA trace:
   a. Navigate to: **Scripts window** > **Use case** > **DTSLELA-500** > **ela_example.py** > **Decode trace data**

   
   Note

   The **Decode trace data** script requires the corresponding JSON file to be named `example_ela_connection.json`. The location of this file can be found in the **DTSLELA-500** directory.

   b. Right click **Decode trace data** and select **Configure**
   c. Check that **Signal group 0** is selected for **State 0** and click **OK**

Figure 6-1: Signal group 0 selected for State 0



d. Right click **Decode trace data** and select **Run...**

7. Analyzing the ELA trace capture

As a result of our work above, the ELA traced each read transaction and stored them into a circular buffer. The circular buffer holds `x` number of read transactions, where `x` relates to the size of the ELA-500 SRAM and number of signals. The read transactions were generated by both explicit reads and speculative reads. You can identify rogue accesses to the potential holes in the memory map by analyzing the read transactions.

The snippet of our trace capture below shows there were several accesses, explicitly called, which were outside the bounds of the memory copy routine. The last address explicitly read by the core was `0x01001fc0`. The processor prefetcher continued to read memory from `0x01002000`, `0x01002040` and `0x01002080`. These memory accesses were to addresses that were outside the internal SRAM. To fix this, we would configure these addresses in the translation tables as `Invalid`, to prevent the prefetcher from prefetching from this region of memory.

```
Address read valid              = 0x1
Shareability                    = Inner Shareable
Execution state                 = AARCH64
Cache Attr                      = Write-back, read/write allocate
Access size                     = 64 bytes
Read address                    = 0x01001fc0

Address read valid              = 0x1
Sharability                     = Inner Shareable
Execution state                 = AARCH64
Cache Attr                      = Write-back, read/write allocate
Access size                     = 64 bytes
Read address                    = 0x01002000

Address read valid              = 0x1
Shareability                    = Inner Shareable
Execution state                 = AARCH64
Cache Attr                      = Write-back, read/write allocate
Access size                     = 64 bytes
Read address                    = 0x01002040

Address read valid              = 0x1
Shareability                    = Inner Shareable
Execution state                 = AARCH64
```

```
Cache Attr                    = Write-back, read/write allocate
Access size                   = 64 bytes
Read address                  = 0x01002080
```