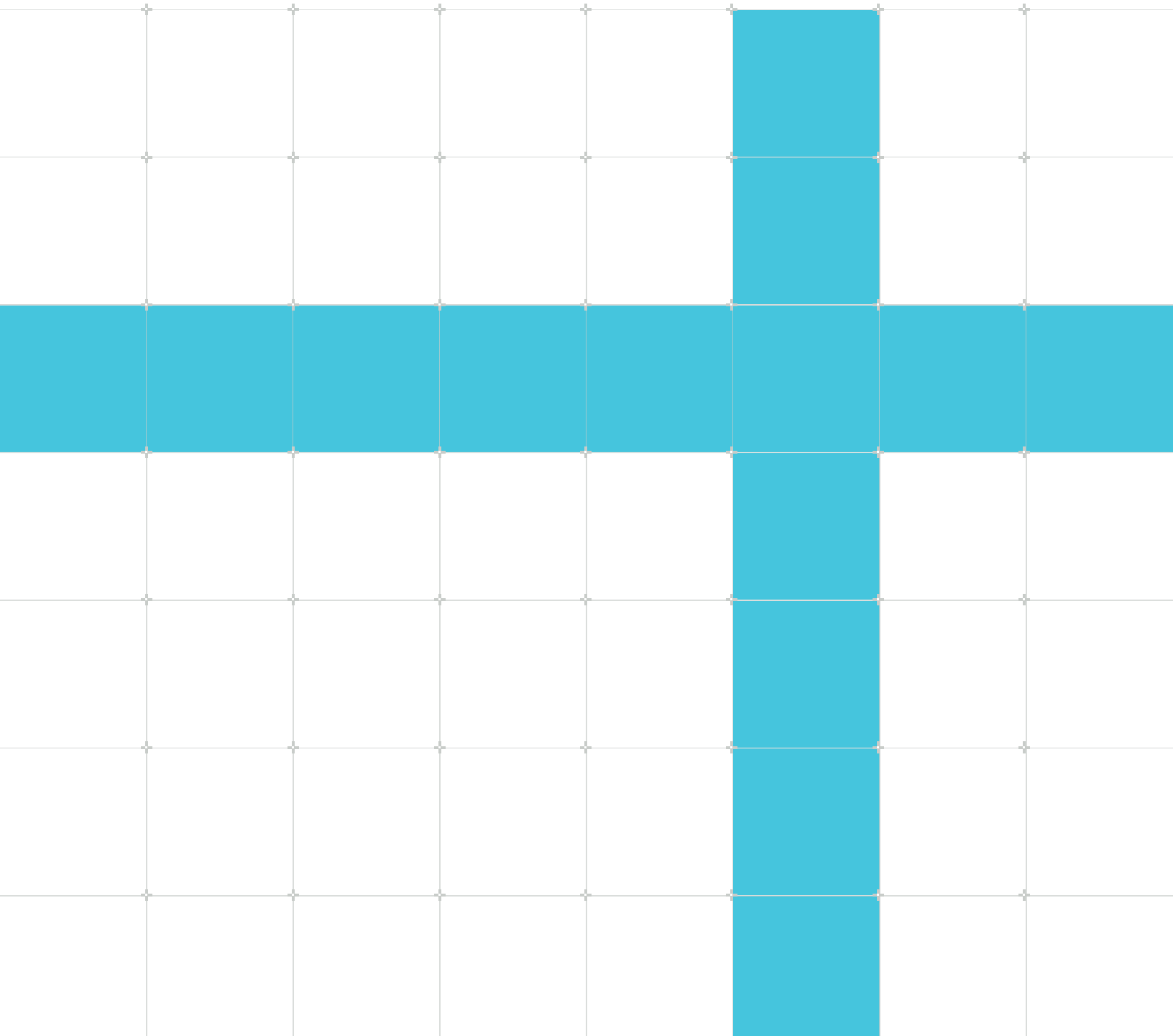
Using the ELA-500 with Arm DS

Version 2.0

guide

Issue 01
102588_2_01_en



Using the ELA-500 with Arm DS guide

Release information

Document history

Issue	Date	Confidentiality	Change
0200-01	12 February 2026	Non-Confidential	Added information for Arm Development Studio 2025.0-2
0100-01	8 April 2021	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

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 to using the ELA-500 with Arm DS

The Arm CoreSight ELA-500 Embedded Logic Analyzer (ELA-500) provides low-level signal visibility into Arm IP and third-party IP. When used with a processor, the ELA-500 provides visibility of:

- Load
- Stores
- Speculative fetches
- Cache activity
- Transaction life cycle

None of the previous items are available through instruction tracing.

CoreSight ELA-500 enables fast hardware-assisted debug of hard-to-trace issues, including data corruption and dead or live locks. The ELA-500 also accelerates debug cycles during complex IP bring-up and assists with post deployment debug.

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

The problem with traditional debug methods

Processors can stop functioning because they are locked-up, also known as deadlocked. One common deadlock scenario happens when a processor initiates memory transactions to a location in the system that cannot response or handle the request. A deadlock might happen if there is no bus completer or the bus completer has limitations like it does not support Burst transactions.

In a perfect world, systems are designed so the entire physical memory map is fully populated. A fully populated memory map means that all memory transactions to all addresses correctly respond with either a valid transaction result or a bus fault. However, for certain designs, this memory model is not implemented.

Places in the memory map that are not populated are referred to as holes. Aggressive speculation and prefetching performed by Arm processors means memory map holes are more likely to be exposed during execution. This exposure can happen even if the software does not explicitly reference the memory holes.

Software can prevent memory hole-related issues by correctly configuring the Memory Management Unit (MMU) translation tables to accurately describe the physical memory map. Software can configure any memory map holes as being invalid. Configuring the MMU this way prevents the processor from making any physical bus transactions to a hole which prevents a deadlock scenario.

Debugging memory hole-related deadlock scenarios cause an issue when debugging using traditional methods like with external debug and instruction and data trace. If an incomplete memory transaction occurs, a processor might not enter halt mode debug. If the core does not enter halt mode debug, the external debugger is unable to halt the processor and inspect its internal state. In this situation, trace capture might still be available. However, trace does not provide a record of the speculative or prefetched transactions that could be responsible for the deadlock.

When deadlock scenarios occur, to trace the external bus transactions made by the processor, use the ELA-500. Depending on the board implementation, the ELA-500 allows you to trace both explicit and speculative transactions. This guide shows how to work with the use case scripting capabilities of Arm Development Studio (Arm DS). In particular, showing the ELA-500 use case scripts shipped with Arm DS.

About the ELA-500

The ELA-500 implements up to 12 Signal Groups, each containing 64-bit, 128-bit, or 256-bit signals. The connections between the signals in the Signal Groups depend on the system and the IP that it is connected to. The available signal interfaces are documented in the relevant IP documentation. A JSON file documents and annotates the signal group connections for a particular IP in a machine-readable format. Arm DS interprets the JSON file to allow seamless debugging of a piece of IP using Arm DS and the ELA-500.

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

The example target

To explore a lock-up scenario, this guide uses an example target with a Cortex-A72, an ELA-500, and a LAK-500A. The LAK-500A is an add-on integration kit for the ELA-500. The LAK-500A exposes some pre-defined debug observation ports of the Cortex-A72 to the ELA signal groups. As part of the LAK-500A, a debug observation port exposes the physical read address signal bus ARADDR and an address valid signal ARVALID.

Note

For this guide, these address signal names are obfuscated.

These signals are required to determine the read addresses issued by the core before the lock-up scenario. In this guide, while a memory copy routine is executed, we monitor these signals with the ELA-500 so we can do a post analysis of the core read transactions. Analyzing the read transactions helps us identify which transaction might have caused the core lock-up.

2. Before you begin

Arm DS ships with use case scripts to allow Arm DS to configure and use the ELA-500.

You must have the following before you begin using the ELA-500 with Arm DS:

- An installation of Arm Development Studio.
- A target with a CoreSight ELA-500 implemented.
- An Arm DS platform configuration for the board.
- A JSON signal mapping file listing the signals coming into the ELA-500 signal groups.

Further information about using the ELA-500 with Arm DS is available in the "ELA-500" section of the Arm Development Studio User Guide.

3. Importing the ELA-500 DTSL use case scripts

Depending on your Arm DS version, you might need to import and add the DTSL ELA-500 use case scripts to Arm DS.

For Arm DS 2025.0-2 or later, you do not need to import nor add the ELA-500 use case scripts. The ELA-500 use case scripts are included as part of the configuration database (configdb) shipped with Arm DS. This inclusion means that, after connecting to any target, the ELA-500 use case scripts are available in the **Scripts** view. If the **Scripts** view is not present in your Arm DS Workspace, open the **Scripts** view by selecting **Window** > **Show View** > **Scripts**.


Note

To see the ELA-500 use case scripts, you must have connected to 1 target at least once.

Importing and adding the DTSL ELA-500 use case scripts for older Arm DS versions

For Arm DS versions earlier than 2025.0-2, you must import the DTSLELA-500 project that contains the scripts and add the scripts to the **Scripts** view.

Import the ELA-500 DTSL use case scripts using the following steps:

1. Launch the **Arm DS IDE**.
2. If prompted, select a Workspace for your Arm DS projects. The default workspace is fine.
3. Select **File** > **Import...** to open the **Import** dialog.
4. Select **Arm Development Studio** > **Examples and Programming Libraries**.
5. Click **Next**
6. Select **Examples** > **Debug and Trace Services Layer (DTSL)** > **DTSLELA-500**.
7. Click **Finish**.

Result: The **Project Explorer** view shows a **DTSLELA-500** project.

Add the DTSLELA-600 project use case scripts to the **Scripts** view using the following steps:

1. Connect to a target with Arm DS. To learn how to connect to a target with Arm DS, read the "Debugging code" section of the Arm Development Studio Getting Started Guide.
2. If the **Scripts** view is not open, click **Window** > **Show View** > **Scripts**.
3. In the **Scripts** view, click **Import a Script or Directory** > **Add Use Case Script Directory...**.
4. In the **Select Folder** dialog, browse to the DTSLELA-500 project in your Arm DS Workspace and click **Select Folder**.

Result: In the **Scripts** view, the DTSLELA-500 use case scripts appear under **Use Case** > **DTSLELA-500**.

For **Decode trace data** of **ela_example.py** to decode the generated trace data correctly, the JSON file for the ELA must be named **example_ela_connection.json**. Also, the **example_ela_connection.json** JSON file must be available in the same directory as the DTSLELA-500 use case scripts. If you are using an Arm DS version earlier than 2025.1, the **example_ela_connection.json** JSON file must be in the DTSLELA-500 project directory.

If you are using an Arm DS version earlier than 2025.1, all the Arm DS DTSL ELA-500 use case scripts assume the ELA-500 is named **ELA-500** in the SDF file in the platform configuration. To use your platform configuration with the Arm DS use case scripts, you must do 1 of the following:

- If you are only using 1 ELA, in the SDF file, change the ELA-500 component **Device Name** to **ELA-500**. Save the change with **File** > **Save**.
- If are using more than 1 ELA, manually configure the **ELA-500 device name** field for each use case script to the **Device Name** for the ELA specified in the SDF file. Save the use case script changes by clicking **Apply** and **OK**.

4. Configuring the ELA-500 use case scripts

To configure the ELA-500, you can use the configuration dialog provided by the **DTSLELA-500** > **ela_lowlevel.py** use case script. The dialog allows you to script a specific debug recipe. The debug recipe is used to debug a specific debug scenario with the ELA-500.

The configuration dialog represents the ELA-500 registers and register bit assignments as fields, tick boxes, or drop-down items. For example, the ELA-500 Actions registers bit assignments are shown as the following in the configuration dialog:

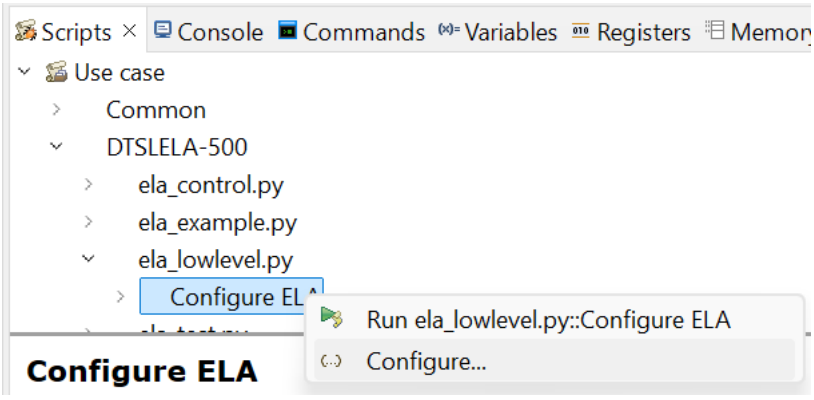
Actions register bit assignments	ELA-500 GUI configuration utility representation
`ELAOUTPUT[7:4]`	**Value to drive on ELAOUT[3:0]** field
`TRACE[3]`	**Enable trace** tick box
`STOPCLOCK[2]`	**Value to drive on STOPCLOCK** tick box
`CTTRIGOUTPUT[1:0]`	**Value to drive on CTTRIGOUTPUT[1:0]** field

In this guide, we use the ELA-500 configuration dialog to configure the ELA-500 to debug a deadlock situation.

The following steps show how to use the configuration dialog provided by the **ela_lowlevel.py** use case script to configure the ELA-500 for our deadlock scenario:

1. If you have not already, connect to a target. You must connect to a target to configure and run the ELA-500 use case scripts.
2. Go to **Scripts view** > **Use case** > **DTSLELA-500** > **ela_lowlevel.py** > **Configure ELA**.
3. Right-click **Configure ELA** and select **Configure**.

Figure 4-1: Selecting Configure ELA



If using Arm DS 2025.0-2 or later, the following 2 new fields are available in the **Common** tab:

- Dropdown option available from **ELA-500 device name**.
- **Signal Mapping File**

You can select which ELA you are configuring using the dropdown option available from **ELA-500 device name**. The ELA list that is shown in the dropdown is based on the ELAs that are listed in the SDF file for the target. The icons and space on the left side of the dialog allows you to easily manage your ELA use case script configurations. To learn more about ELA configurations, read the "Manage ELA configurations" section of the Arm Development Studio User Guide.

Setting **Signal Mapping File** causes the Trigger State tabs to show signal names associated with the ELA signal mapping specified by the provided JSON file. Contact your board designer for the JSON file for your specific ELA signal mapping.


Note

If you choose to not set **Signal Mapping File**, delete the JSON path and file shown in the field. You must delete the path and file shown so no signal mapping is shown in the Trigger State tabs.

Now we configure the common controls using the following steps:

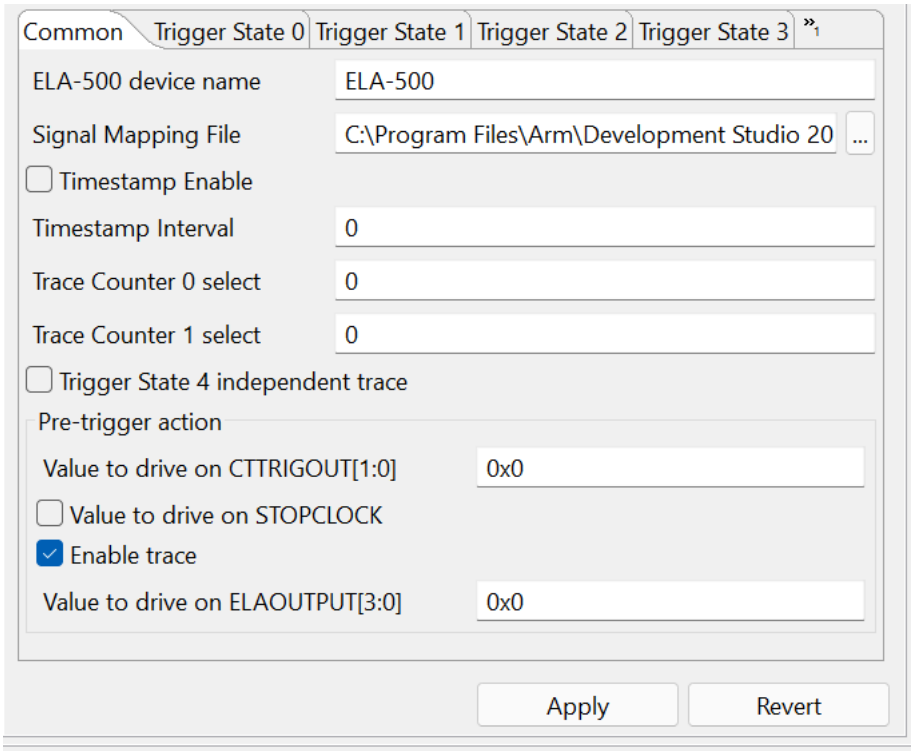
1. In the **Common** tab, if using Arm DS 2025.0-2 or later, select **Signal Mapping File** and set the file path to your signal mapping JSON file.
2. In the **Pre-trigger action** section, select **Enable trace**.

   This setting configures the ELA to start tracing when it is enabled. This field sets TRACE[3] of the Pre-trigger Action register so that trace is active when the ELA-500 is enabled. When trace is active, 1 of the following controls trace capture:

   - Each ELA clock cycle
   - A Trigger Signal Comparison match
   - A Trigger Counter Comparison match
3. Click Apply.

When the previous steps are completed, the **Common** tab shows the following for Arm DS 2025.0-2 and later:

Figure 4-2: Configuring Common settings for the ELA-500



Then we configure our trigger state:

1. Open the **Trigger State 0** tab.
2. Set **Select Signal Group** to `0x1`.

   On our example target, the RVALID signal goes into Signal Group 0. To locate the RVALID signal location for other targets, check the JSON file or documentation for your IP. Configuring **Select Signal Group** to `0x1` sets SIGSEL0[11:0] of the Signal Select 0 register (SIGSEL0) to `0x1`. The ELA-500 uses a one hot encoding for the Signal Group in the Signal Select registers. Trigger State 0 is now associated with the signals coming into Signal Group 0.
3. Set **Signal Comparison (COMP)** to **Equal**.

   This step sets the Trigger Signal Comparison type select COMP[2:0] of the Trigger Control 0 register (TRIGCTRL0). In this case, we want to trigger when the ARVALID signal is valid or active-HIGH.
4. Set the **Next state** value to **Trigger State 0** or `0x1`.

   Here we set the Next state. If the Trigger Condition is met, the ELA enters the state assigned to the Next State 0 register (NEXTSTATE0). As the NEXTSTATE0 register uses a one hot encoding, this configuration sets NEXTSTATE0 of the register to Trigger State 0. In our case, we want to capture on each ARVALID assertion.
5. We must set SIGMACK[127:0] of the Signal Mask 0 register (SIGMASK0) and SIGCOMP[127:0] of the Signal Compare 0 register (SIGCOMP0) to monitor the ARVALID

signal. The bit position of the ARVALID signal is documented in the JSON file or documentation for your IP.

If using Arm DS 2025.0-2 or later, in the signal mapping table, for the **ARVALID** signal, set **Mask** and **Comparison Value** to `0x1`.

For Arm DS versions earlier than 2025.0-2, to configure bit 83, set both the **Signal Mask [95:64]** and the **Signal Compare [95:64]** fields to `0x00080000`.

6. Click **Apply** > **OK**.

When the previous steps are completed, the **Trigger State 0** tab shows the following for Arm DS 2025.0-2 and later:

Figure 4-3: Configuring Trigger State 0 part 1

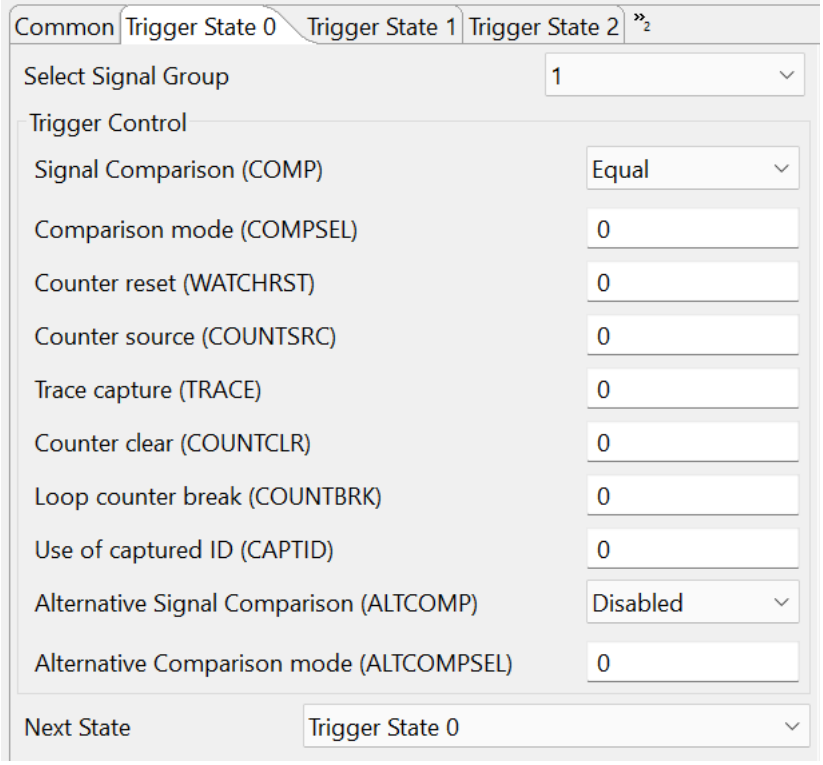
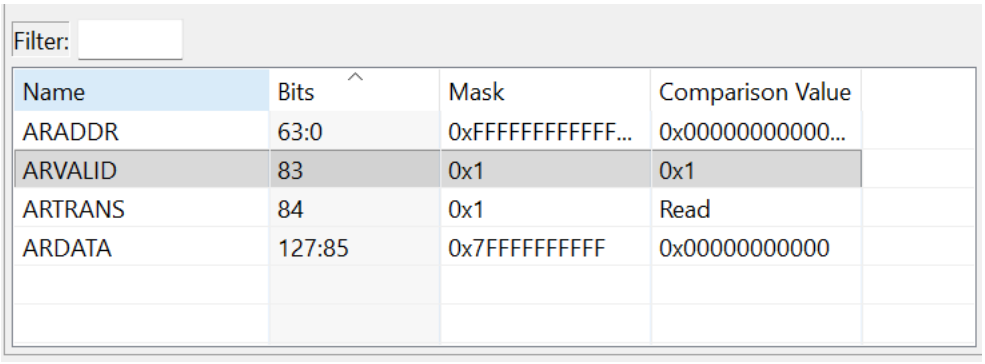


Figure 4-4: Configuring Trigger State 0 part 2



5. Running the ELA test scripts

To run the ELA test scripts, follow these steps:

1. To program the ELA configuration registers, navigate to **Scripts view** > **Use case** > **DTSLELA-500** > **ela_lowlevel.py** > **Configure ELA**.
2. Right-click **Configure ELA** and select **Run ela_lowlevel.py::Configure ELA**.
3. To run the ELA, navigate to **Scripts view** > **Use case** > **DTSLELA-500** > **ela_control.py** > **Run ELA-500**.
4. Right-click **Run ELA-500** and select **Run ela_control.py::Run ELA-500**.
5. Using the **Debug Control** view, run the target.

Result: The target runs and the ELA monitors the input Signal Group 0 for the configured trigger condition.

6. Capturing the ELA trace data

In this debug scenario, the core is unable to enter halt mode debug, so we must stop the ELA-500. After stopping the ELA-500, we dump and decode the trace data that is generated by the ELA-500.

Following these steps to stop the ELA-500 and capture the trace data:

1. Navigate to **Scripts view** > **Use case** > **DTSLELA-500** > **ela_control.py** > **Stop ELA-500**.
2. Right-click **Stop ELA-500** and select **Run ela_control.py::Stop ELA-500**.
3. If you have not done so already, name the JSON file for the ELA **example_ela_connection.json**. Also, the **example_ela_connection.json** JSON file must be available in the same directory as the DTSLELA-500 use case scripts. If you are using an Arm DS version earlier than 2025.1, the **example_ela_connection.json** JSON file must be in the DTSLELA-500 project directory.
4. To dump and decode the ELA trace, navigate to **Scripts view** > **Use case** > **DTSLELA-500** > **ela_example.py** > **Decode trace data**.
5. Right-click **Decode trace data** and select **Configure...**.
6. Ensure the value for **ELA-500 device name** matches the name of the ELA-500 in the platform configuration.
7. Under **Signal groups**, set **State 0** to **0** and click **OK**.
8. Right-click **Decode trace data** and select **Run ela_example.py::Decode trace data**.

Result: Arm DS stops the ELA-500 and then collects, decodes, and outputs the trace data.

7. Analyzing the ELA trace capture

After performing all the previous steps, the ELA traces each read transaction and stores 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 came from 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 following example trace capture shows several accesses, explicitly called which were outside the bounds of the run memory copy routine:

```
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
```

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 target SRAM. Performing accesses outside the internal target SRAM can cause execution issues like deadlocks. To fix any potential issues, we could configure addresses outside the internal SRAM in the translation tables as invalid. Configuring the translations tables as invalid prevent the prefetcher from prefetching from problematic regions of memory.

8. Resources

The following resources are related to this guide:

- CoreSight ELA-500 Product Support
- Arm CoreSight ELA-500 Embedded Logic Analyzer Technical Reference Manual (TRM)
- "ELA-500" section of the Arm Development Studio User Guide