



Arm Debugger Tutorial

Revision: NA

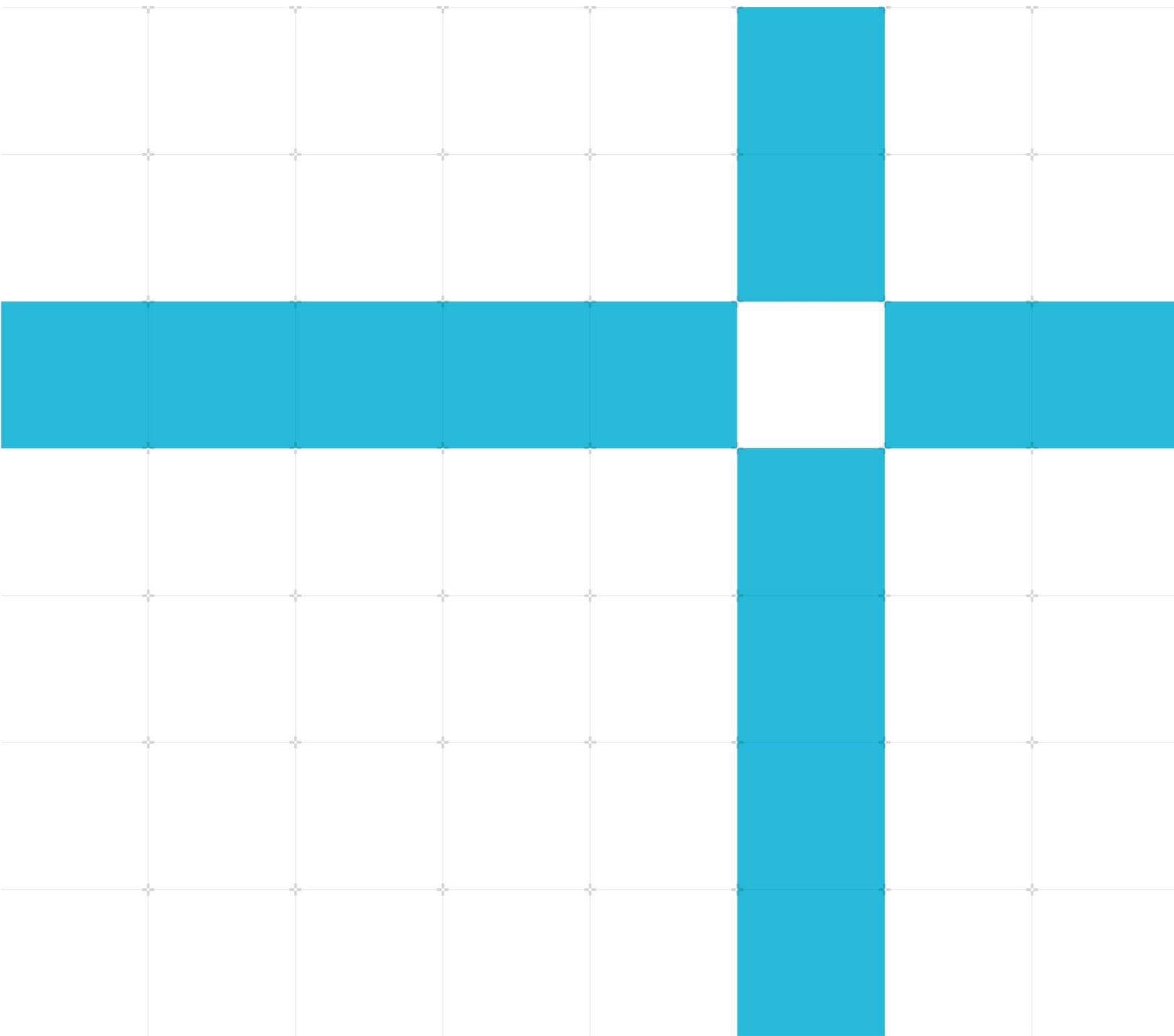
Using the ELA-500 with Arm DS

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 0.0

NA



Arm Debugger Tutorial

Using the ELA-500 with Arm DS

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0.0	8 th of April 2021	Non- Confidential	First version

Confidential Proprietary Notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorized by Arm to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: **(i)** for the purposes of determining whether implementations infringe any third party patents; **(ii)** for developing technology or products which avoid any of Arm's intellectual property; or **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant 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.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is release quality.

Web Address

<http://www.arm.com>

Contents

1 Introduction	5
1.1.1 The problem with traditional debug methods.....	5
1.1.2 About the CoreSight ELA-500	6
1.1.3 The example board	6
2 Before you begin	7
3 Importing the ELA-500 DTSL use case scripts	8
4 Configuring the ELA-500 use case scripts.....	10
5 Running the ELA use case scripts	16
6 Capturing the ELA trace data	17
7 Analyzing the ELA trace capture.....	19

1 Introduction

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 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 and proprietary IP blocks. Program trigger conditions over standard debug interfaces either by an on-chip processor or an external debugger.

1.1.1 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 can be 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 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 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 break 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 CoreSight ELA-500. Depending on the board implementation, the ELA-500 allows you to trace both explicit and Speculative transactions. This tutorial shows how to work with the use case scripting capabilities of [Arm Development Studio](#) (Arm DS). In particular, demonstrating the example CoreSight ELA-500 use case scripts shipped with Arm DS.

1.1.2 About the CoreSight 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 specific signal interfaces are documented in the relevant IP documentation. These documents might only be available to Arm IP licensees. Arm IP connected to an ELA is supplied with a JSON file. The JSON file documents and annotates the signal group connections for that 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.

1.1.3 The example board

This tutorial uses a board with a Cortex-A72 and an ELA-500 to explore a lock-up scenario. This Cortex-A72 and ELA-500 system utilizes the LAK-500A. The LAK-500A is an Integration Kit for the ELA-500 and the Cortex-A72. The Integration Kit is an add-on to the ELA-500. The LAK-500A exposes some pre-defined debug observation ports to the Cortex-A72 ELA Signal Groups, and provides the corresponding JSON signal-mapping file.

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

Note: For this tutorial, these signal names are obfuscated.

These signals are required to determine the read addresses issued by the core, before the lock-up scenario. In this tutorial, 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.

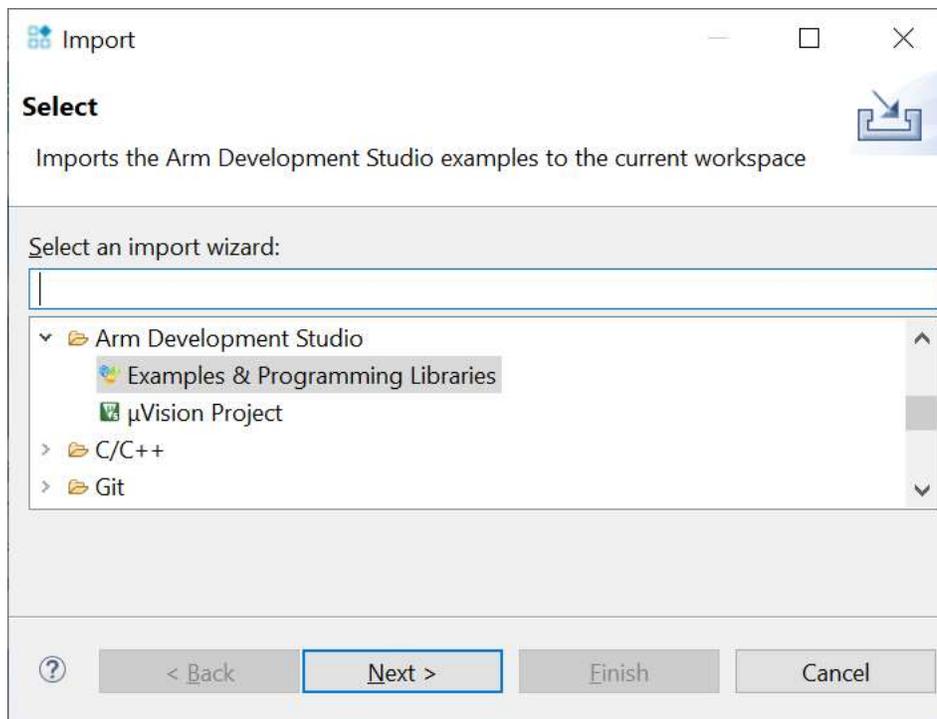
You must have the following before you begin using the ELA-500 with Arm DS:

- An installation of [Arm Development Studio](#).
- A target with an CoreSight ELA-500 implemented.
- An Arm DS [platform configuration](#) for the board.
 - In the platform configuration, you must name the CoreSight ELA-500 device as **ELA-500**.
- A JSON signal-mapping file listing the signals coming into the ELA-500 Signal Groups.
 - To run the **Decode trace data** script, name the JSON file for the ELA `example_ela_connection.json`.
 - The `example_ela_connection.json` JSON file must be available in the `DTSLELA-500` project directory.

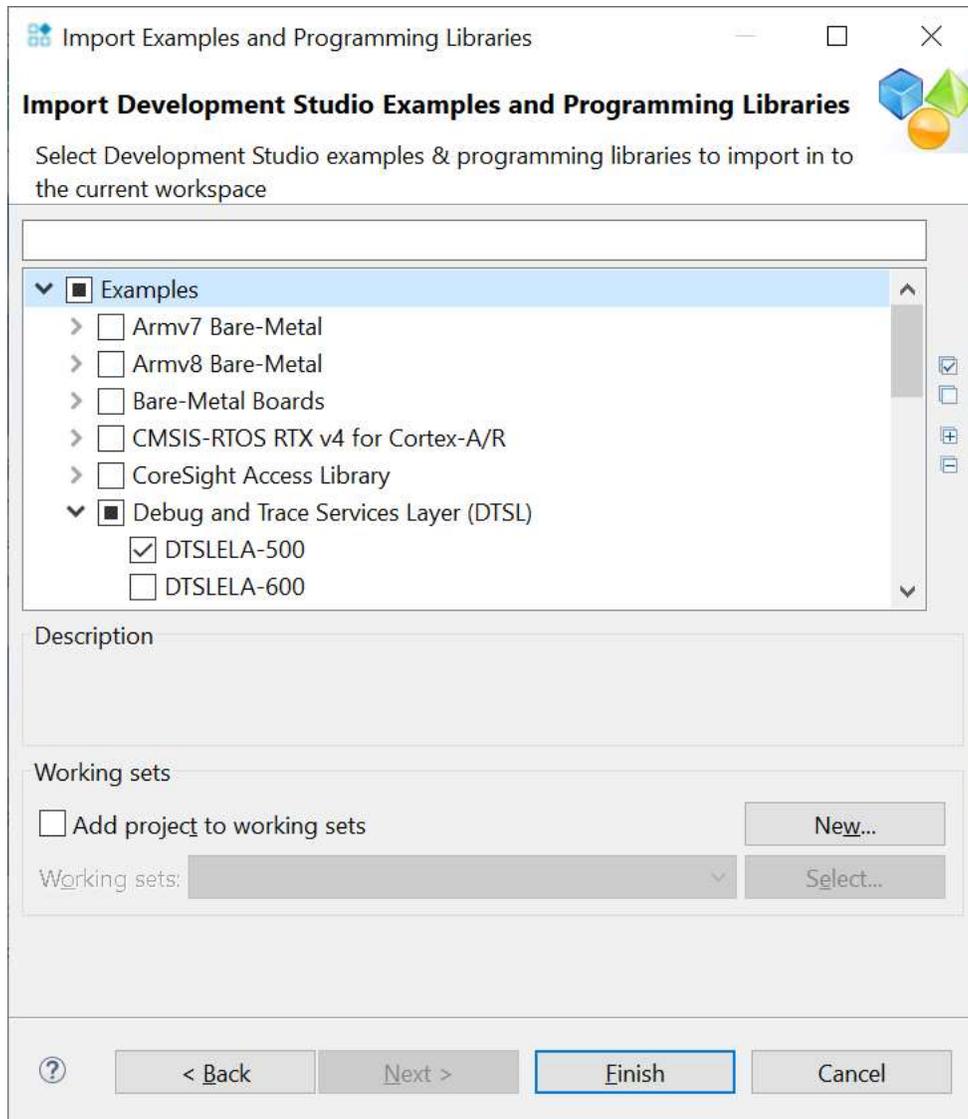
Further information about using the ELA-500 with Arm DS is available in the "[Embedded Logic Analyzer \(ELA\)](#)" section of the [Arm Development Studio User Guide](#).

3 Importing the ELA-500 DTSL use case scripts

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.

4 Configuring the ELA-500 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.

The GUI configuration utility represents the [ELA-500 registers](#) and register bit assignments as fields, tickboxes, or drop-down items. For example, the ELA-500 [Actions registers](#) bit assignments are shown as the following in the GUI configuration utility:

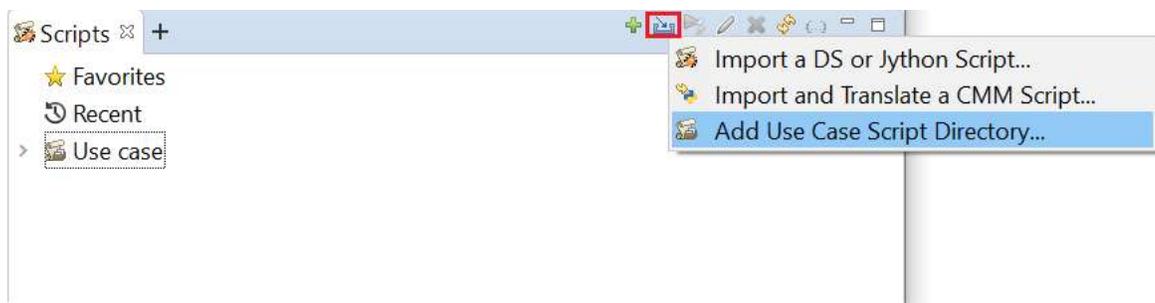
Actions register bit assignments	ELA-500 GUI configuration utility representation
ELAOUTPUT [7 : 4]	Value to drive on ELAOUT[3:0] field
TRACE [3]	Enable trace tickbox
STOPCLOCK [2]	Value to drive on STOPCLOCK tickbox
CTTRIGOUTPUT [1 : 0]	Value to drive on CTTRIGOUTPUT[1:0] field

In this tutorial, we use the ELA-500 GUI configuration utility to configure the ELA-500 to debug a deadlock situation.

To add the DTSLELA-500 project use case scripts to the **Scripts** view in Arm DS, do the following:

1. Connect to a target with Arm DS.

Learn how to connect to a target with Arm DS, read the "[Configuring debug connections in Arm Debugger](#)" section of the [Arm Development Studio User 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...**



- In the **Select Folder** dialog, browse to the DTSLELA-500 project in your Arm DS Workspace and click **Select Folder**.

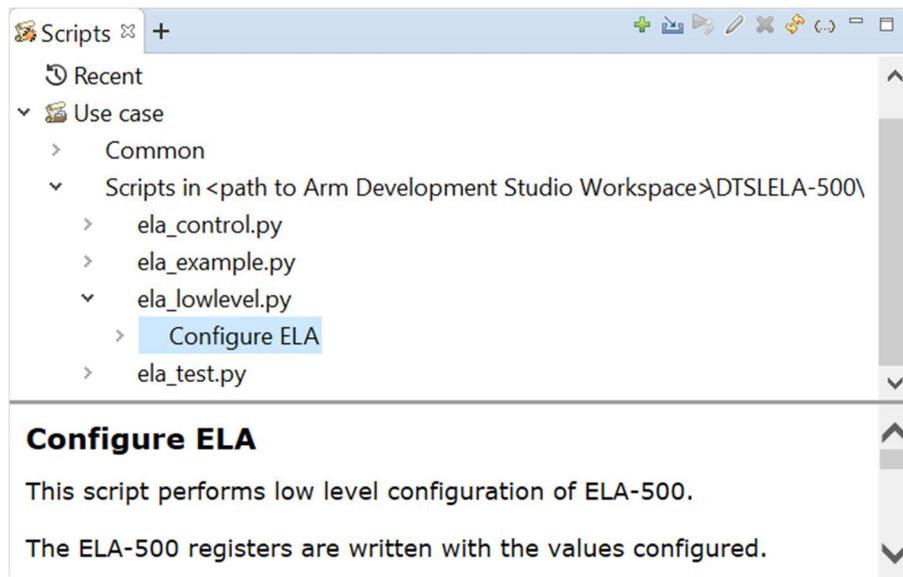
In the **Scripts** view, the DTSLELA-500 use case scripts appear under **Use Case > Scripts in <path to Arm Development Studio Workspace>\DTSLELA-500**.



The following steps show how to use the GUI ELA-500 Configuration Utility to configure the ELA-500 for our deadlock scenario:

- If you have not already, connect to a target.

Note: You must connect to a target to configure and run the ELA-500 use case scripts.
- Open the GUI ELA-500 configuration utility:
 - Go to **Scripts view > Use case > Scripts in <path to Arm Development Studio Workspace>\DTSLELA-500 > ela_lowlevel.py > Configure ELA**.
 - Right-click **Configure ELA** and select **Configure...**

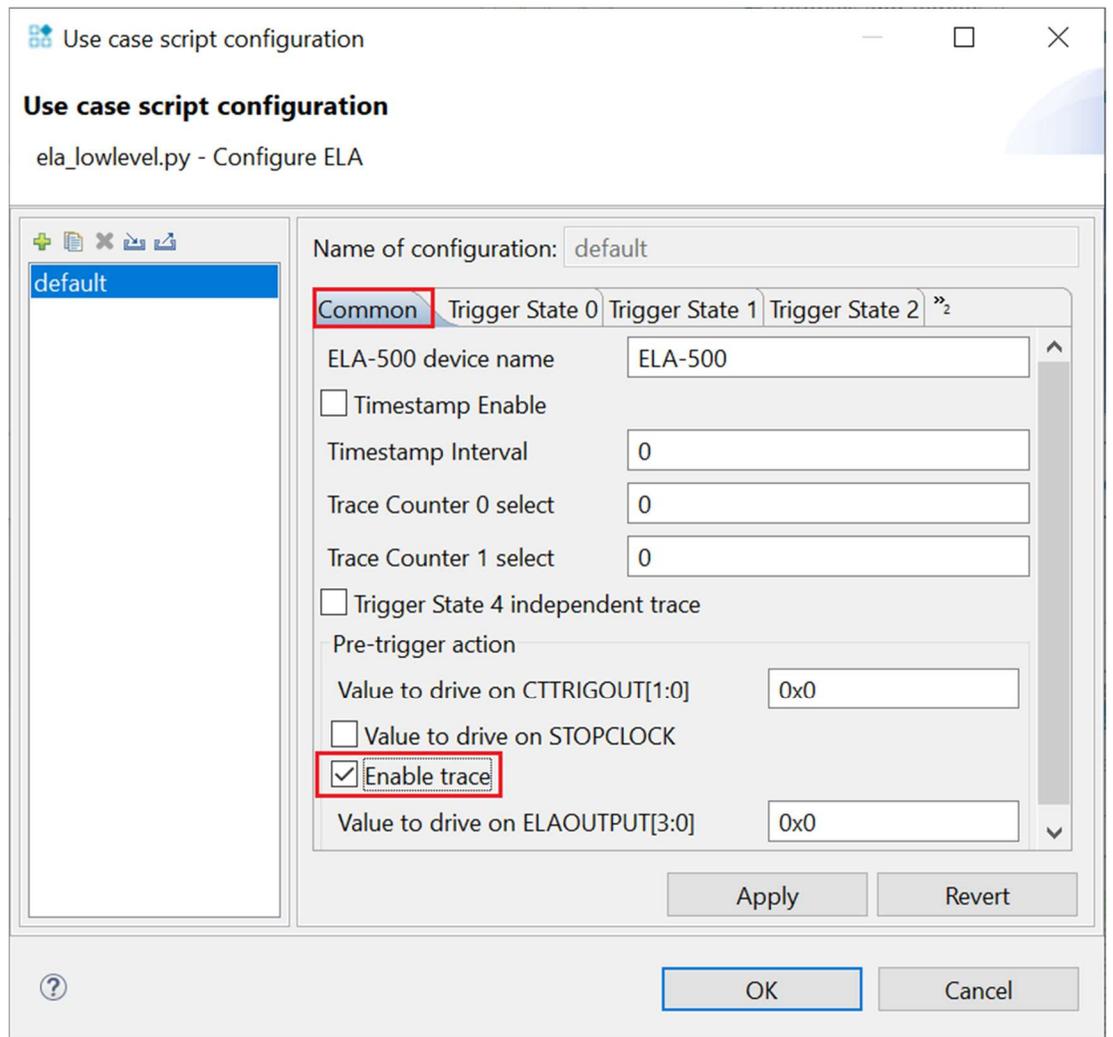


- Configure the common controls:

- a. In the **Common** tab, in the **Pre-trigger action** section, select **Enable trace**.

This setting configures the ELA to start tracing when it is enabled. This field sets `PTACTION.TRACE` so that trace is active when the ELA-500 is enabled. When trace is active, one of the following controls trace capture:

- i. Each ELA clock cycle
- ii. A Trigger Signal Comparison match
- iii. A Trigger Counter Comparison match.



- b. Click **Apply**.

4. We now must configure our initial trigger:

- a. Open the **Trigger State 0** tab.

- b. Set **Select Signal Group** to 0x1.

On our example board, Cortex-72 + ELA-500 + LAK-500A, the `RVALID` signal goes into Signal Group 0. To locate the `RVALID` signal location for other targets, check your IPs corresponding JSON file or documentation.

To trigger on the `RVALID` signal in Signal Group 0, we set the Trigger State 0 **Select Signal Group** value to 0x1. This step sets the ELA-500 [Signal Select register 0](#) (`SIGSEL0`) to 0x1. The ELA-500 uses a 'ones hot' encoding for the Signal Group in the Signal Select registers.

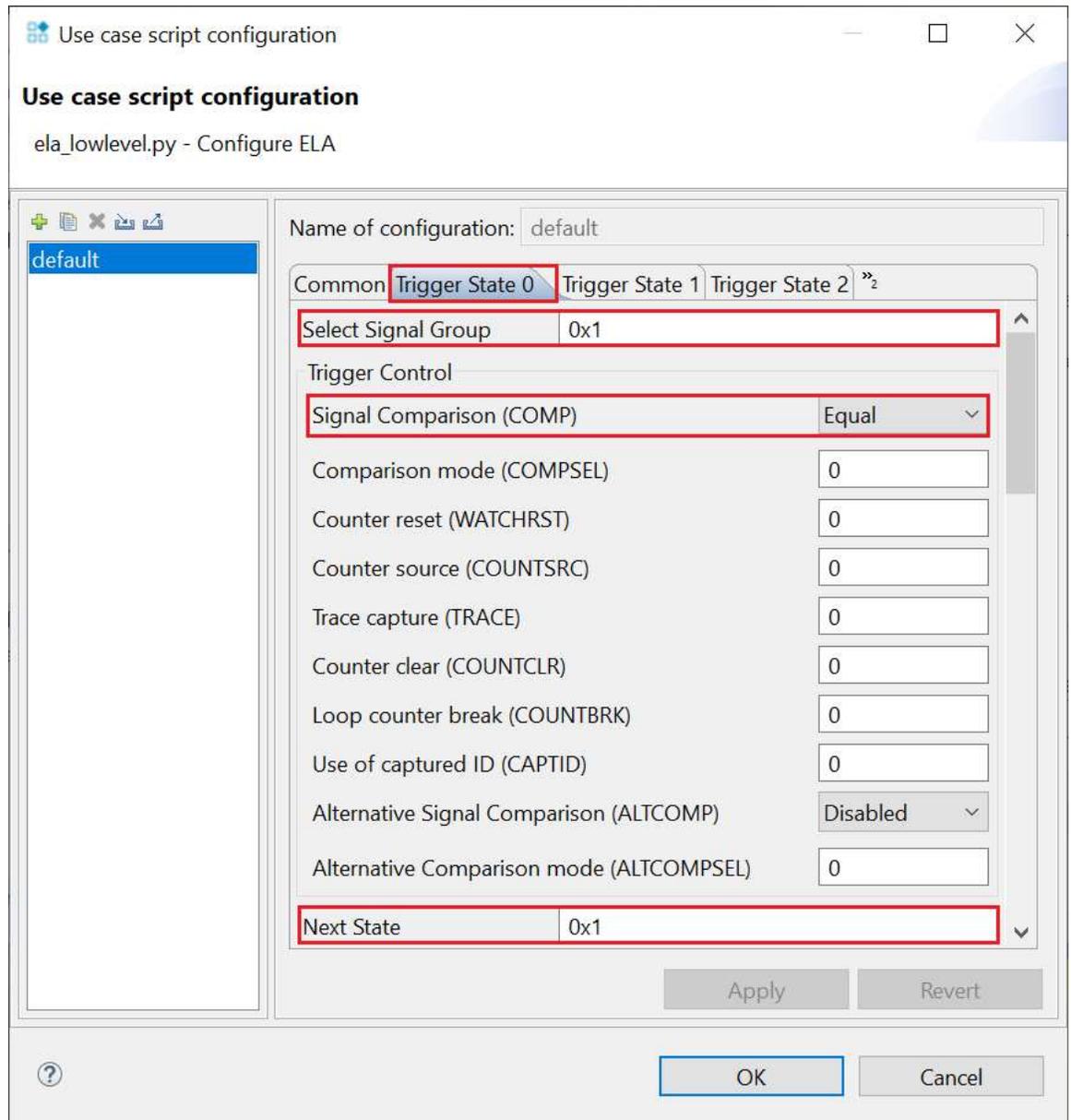
Trigger State 0 is now associated with the signals coming into Signal Group 0.

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

This step sets the Trigger Signal Comparison type select (`COMP`) of the Trigger State 0 [Trigger Control register 0](#) (`TRIGCTRL0`). In this case, we want to trigger when the `RVALID` signal is valid (active-HIGH).

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

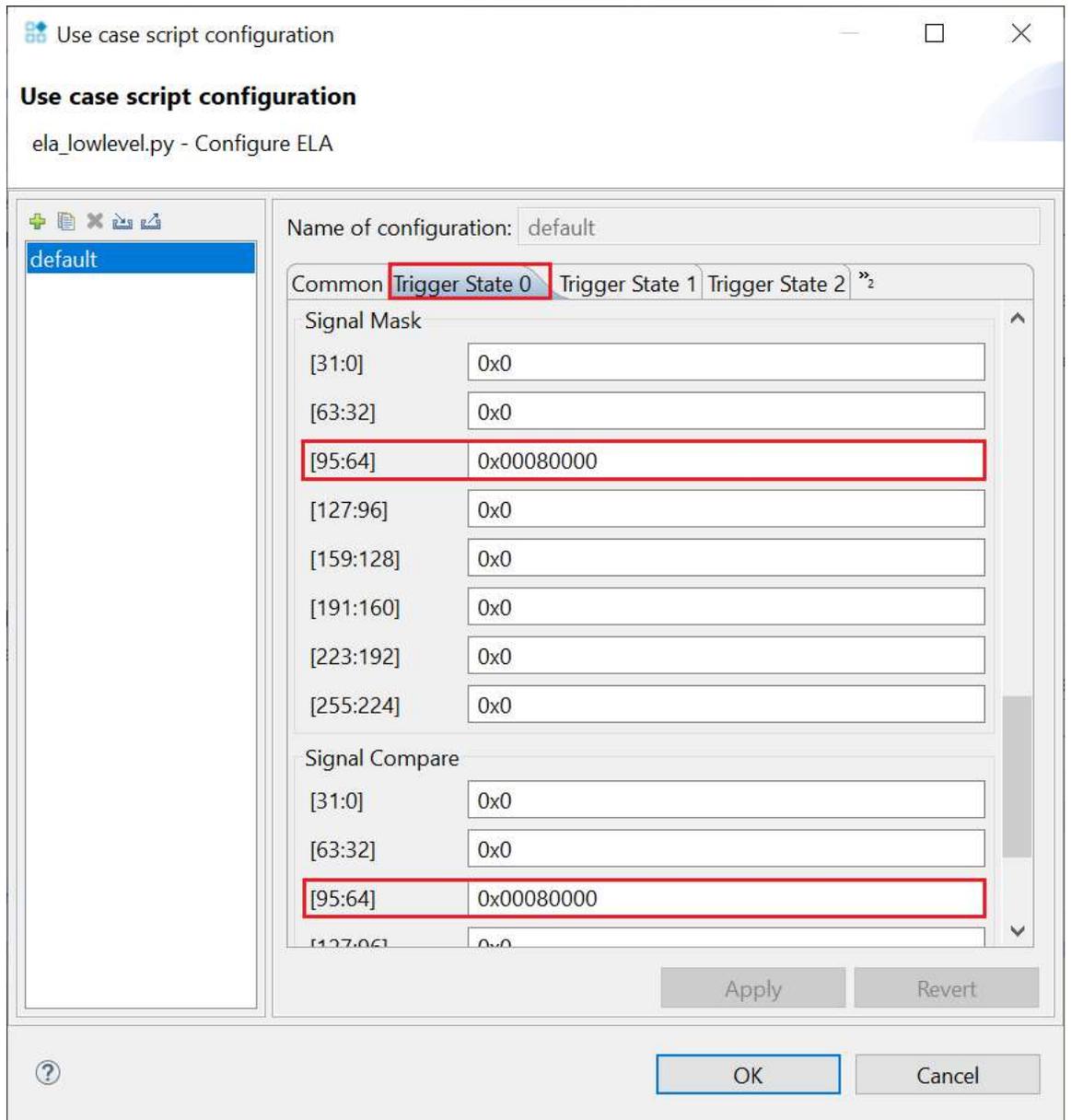
Here we set the Next state. If the Trigger Condition is met, the ELA enters the state assigned to the [Next State register](#). This configuration sets the Trigger State 0 Next state register 0 (`NEXTSTATE0`) to Trigger State 0. In our case, we want to capture on each `RVALID` assertion, which uses the 'ones hot' for Trigger State 0.



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

We must set Trigger State 0 **Signal Compare 0 (SIGCOMP0)** and **Signal Mask 0 (SIGMASK0)** registers 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 must specify 0x00080000 in both the **Signal Mask [95:64]** and **Signal Compare [95:64]** fields.



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

5 Running the ELA use case scripts

1. Program the ELA configuration registers:
 - a. Navigate to: **Scripts** view > **Use case** > **Scripts** in <path to Arm Development Studio Workspace>\DTSLELA-500 > **ela_lowlevel.py** > **Configure ELA**.
 - b. Right-click **Configure ELA** and select **Run ela_lowlevel.py::Configure ELA**.
2. Run the ELA:
 - a. Navigate to: **Scripts** view > **Use case** > **Scripts** in <path to Arm Development Studio Workspace>\DTSLELA-500 > **ela_control.py** > **Run ELA-500**.
 - b. Right-click **Run ELA-500** and select **Run ela_control.py::Run ELA-500**.
3. In Development Studio, run the target.

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

6 Capturing the ELA trace data

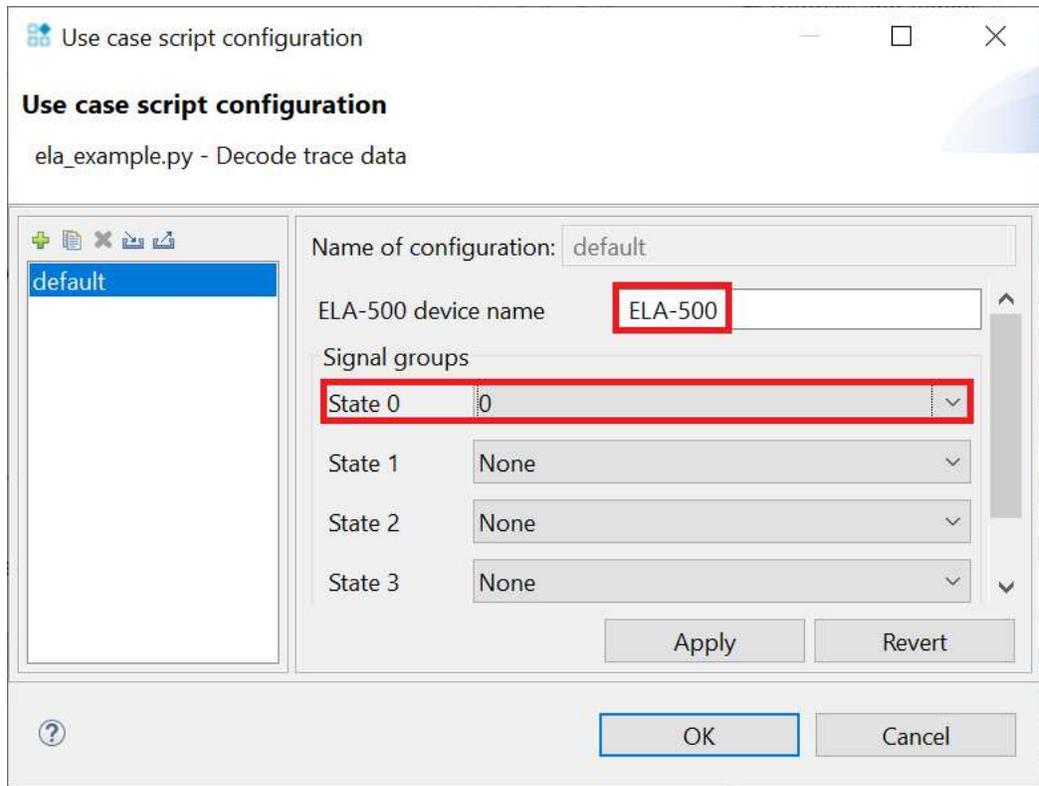
1. The core is unable to enter halt mode debug in our debug scenario, so we must stop the ELA:
 - a. Navigate to **Scripts** view > **Use case** > **Scripts** in <path to Arm Development Studio Workspace>\DTSLELA-500 > **ela_control.py** > **Stop ELA-500**.
 - b. Right-click **Stop ELA-500** and select **Run ela_control.py::Stop ELA-500**.
2. Dump and decode the ELA trace:

- a. Navigate to: **Scripts** view > **Use case** > **Scripts** in <path to Arm Development Studio Workspace>\DTSLELA-500 > **ela_example.py** > **Decode trace data**.

Note: To run the **Decode trace data** script, name the JSON file for the ELA `example_ela_connection.json`. Also, the `example_ela_connection.json` JSON file must be available in the **DTSLELA-500** project directory.

- b. Right-click **Decode trace data** and select **Configure...**
- c. Under **Signal groups**, set **State 0** to **0** and click **OK**.

Note: In the platform configuration, you must name the CoreSight ELA-500 device as **ELA-500**. This naming is done so the ELA-500 device name in the platform configuration matches the **ELA-500 device name** in the **Decode trace data** script.



- d. Right-click **Decode trace data** and select **Run ela_example.py::Decode trace data**.

Result: Arm DS collects the captured ELA trace data, decodes it, and outputs into Development Studio.

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. 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.

```

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

```