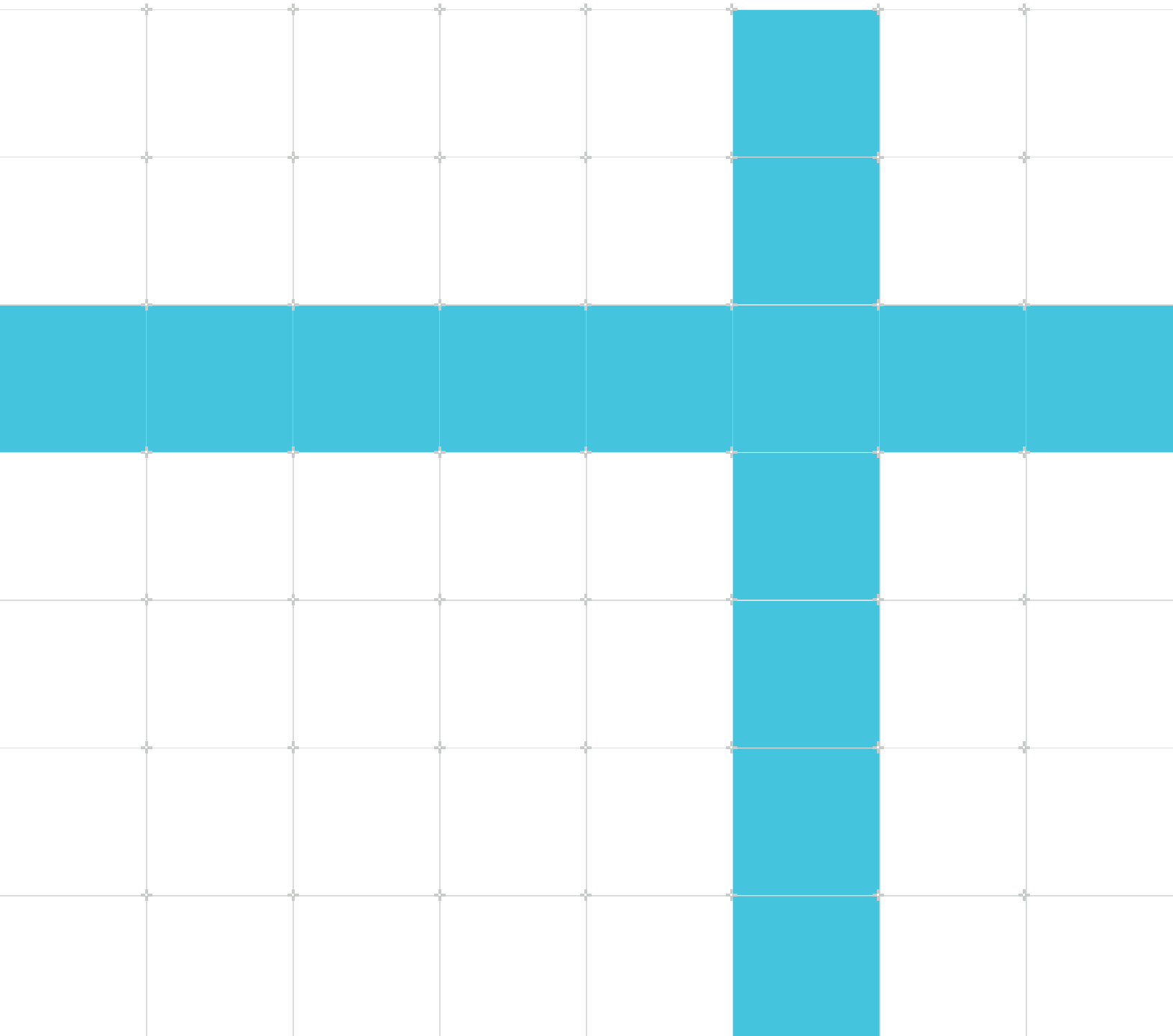
Learn the architecture - Before debugging on Armv8-A

Version 1.0

Issue 03
102408_0100_03_en



Learn the architecture - Before debugging on Armv8-A

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	1 February 2021	Non-Confidential	Initial release
0100-03	14 June 2023	Non-Confidential	Minor edit to fix error

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. Overview

This guide describes concepts that are useful to know before debugging an Armv8-A processor. To illustrate these concepts, we mention debuggers, for example GDB, the GNU Project Debugger and the Arm Debugger, which is part of Arm Development Studio.

At the end of this quide, you should be able to:

- Understand the differences between external, self-hosted, invasive, and non-invasive debug.
- Apply the correct target and application considerations when working with a debugger.
- Understand the state of the target when it is connecting to a debugger.
- Understand that some target initialization may be necessary before establishing a debug connection.

Before you begin

This guide assumes a familiarity with the following guides:

- Exception model
- Memory model

2. Different types of debug

The Armv8-A architecture can support two types of debug: external debug and self-hosted debug. Two types of debug operation can be performed: non-invasive debug and invasive debug.

We will describe external and self-hosted debug, and invasive versus non-invasive debug. We are providing an overview only, and do not provide detail about how to program an Armv8-A processor to use any particular type of debug.

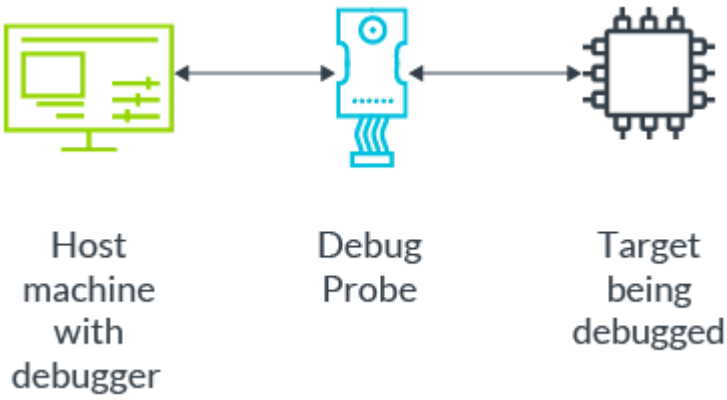
External debug

External debug occurs when the debugger is external to the processor being debugged. For example, external debug is taking place if a debugger that is running on a host machine is debugging an Armv8-A processor that is available on a separate development board. Another example of external debug is a debugger that is running on one processor and debugging another processor on the same SoC.

External debug is usually used when doing chip validation and bring-up, or when working with bare-metal environments. Generally, external debug relies on a physical connector, for example JTAG, or Serial Wire Debug (SWD) on the target being debugged.

External debug also requires a debug probe to be connected between the target being debugged and the host machine running the debugger. The following diagram shows the debug probe connection between a host machine running a debugger and a debug target:

Figure 2-1: External Debug

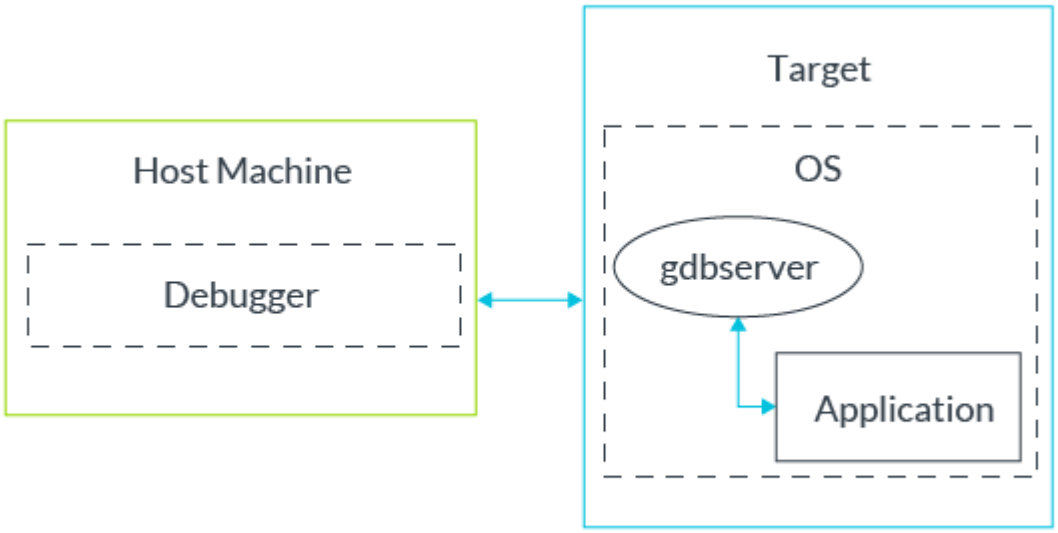


Self-hosted debug

Self-hosted debug occurs when the debugger and the software being debugged run on the same processor. A good example is gdbserver, which is used for Linux application debug. Gdbserver is a program that runs alongside a Linux application, and allows a debug environment like GDB or the Arm Debugger to debug the application.

This diagram shows how gdbserver interacts with an Operating System (OS) application and a debugger:

Figure 2-2: Self-hosted Debug



Even though the debugger attached to gdbserver usually runs off-target, gdbserver is still classed as self-hosted debug.

Self-hosted debug is usually used with debugging applications that are running under an OS, or when physical debug connectors are not available.

Invasive debug

When using invasive debug, the act of debugging changes the processor state in a subtle way. For example, when a breakpoint is set using a debugger, the processor stops when the breakpoint is hit, and the timing of the program being debugged changes.

Some familiar debug methods are invasive debug methods. These include stepping through code, viewing memory, setting breakpoints and watchpoints, and viewing registers.

Usually, the subtle changes introduced using invasive debug techniques do not affect the general execution of the program. However, some bugs, for example problems related to timing, might be sensitive to these subtle changes. This means that invasive debug techniques are not suitable for investigating this type of bug.

Non-invasive debug

In contrast to invasive debug, non-invasive debug does not change the processor state in any way.

For example, generating and collecting trace data from a target will usually not affect the processor, so trace is usually classified as non-invasive debug. Other operations that are usually classified as

non-invasive debug are use of the Performance Monitoring Unit (PMU) and performing Program Counter (PC) sampling.

3. Target types

When debugging, it is important to consider the capabilities and relevancy of the target. Sometimes a target, during the development cycle, bears only a cursory resemblance to the final SoC design. This means that areas like speed, functionality, or performance might differ between earlier targets and the final design.

Here is a list of different targets that you might work with during a project, and some things to remember about each of them:

RTL simulation or emulation

- Accurate model of the processor but a complete representative of the final design.
- Debugging is very slow.

FPGA

- Can be true to final design.
- Debugging is faster than RTL simulation, but slower than final target.
- May require more steps to establish a debug connection.

Software model

- Can be true to layout of final design, but not necessarily timing accurate.
- There may be aspects of the model which do not function the same way that the hardware functions.
- Debugging is faster than RTL simulation, but slower than an FPGA.
- Need to consider whether the model is functionally accurate or timing accurate.

Development board

- If built in-house, the target is similar to, or the same as, the final design. If built by a third party, the target is only similar to the final design.
- May require more steps to establish a debug connection.
- Debugging is faster than a model.

Final silicon

- The target is the final design.
- May not have physical connectors for debugger connection.
- Debugging is faster.
- Typically available late in project development.

4. Program being debugged

Before debugging, another thing to consider is the type of program that is being debugged. The image being debugged determines the level of complexity for both the debug operations and the information that you obtain from a debugger.

Here is a list of different types of programs that you might work with, and some things to remember about each of them:

Bare-metal and boot code

- Directly uses the architectural features of a processor
- Usually contains a mix of source and assembly files
- Usually less overhead when performing debug operations
- Easy to determine the order of execution
- Debug information is usually not complex
- Program memory layout is usually easy to understand
- Memory map generally does not change during execution
- Usually has one execution thread
- May cross multiple Execution levels (ELs)

OS kernel

- Directly uses the architectural features of a processor
- Usually contains a mix of source and assembly files
- Harder to determine the order of execution
- Debug information is usually more complex
- Program memory layout is more complex
- Memory map can change during execution
- May have more than one execution thread

OS application and OS module or driver

- More difficult to see how processor architecture features are used
- Usually made up of source files
- May rely on library for which you do not have the source code
- Harder to determine the order of execution
- Debug information is usually very complex
- Program memory layout is very complex
- Memory map can change during execution

- Usually has more than one execution thread
- Usually requires the debugger to have some awareness of the OS being used

5. Target state

On connection to a target, the debugger shows the current state of the processor. Because of the way that the architecture is configured, the processor is probably just coming out of reset in Secure EL3 when you halt the debugger. However, your SoC might not do this.

We will now describe some common states that the target might be in upon connection:

Core or processor is powered down

Some targets contain hardware elements (for example switches), firmware, boot code, or an OS that power down certain cores or processors unless steps are taken. Also, cores or processors that are not being used for execution, or that have not been used recently, may be powered down by the power controller or OS of the SoC.

Processors contain hardware called debug probe, which allows you to see and configure the debug setup and operation for the processor. Because the debug probe and the processor logic are usually in different power domains, the processor logic might be powered up, while the debug probe is powered down.

Some debuggers cannot connect to powered down logic. If a debugger connection is successful, you probably cannot do much with the debugger until the core, processor, or debug probe is powered up.

Core or processor is held in reset

Some targets contain hardware elements, firmware or boot code that hold a core or processor in reset unless steps are taken. Also, some SoCs contain a reset controller or processor that must be configured to release another core or processor from reset.

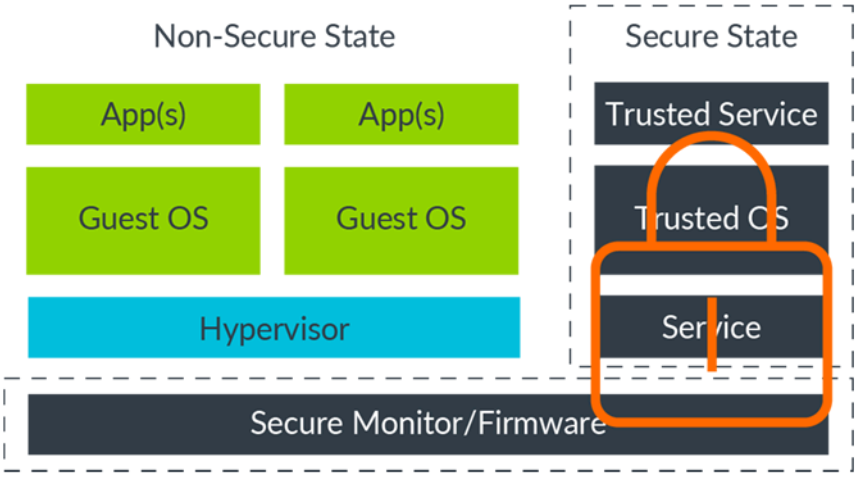
Some debuggers cannot connect to cores or processors that are held in reset. If a connection is successful, debug probe cannot be performed until the core or processor is released from reset.

Core or processor is in a different security state

An Armv8-A architecture processor or core can have two security states, Secure state and Non-secure state. If the Secure state is implemented, privileged or secret information for the SoC is stored or handled through the Secure state. This means that, after a certain point in the development cycle of an SoC, the hardware or software will lock untrusted users out of the Secure state. This step is taken to prevent access to the protected data in the Secure state.

This diagram shows hardware or software locking a user out of the Secure state:

Figure 5-1: Secure State



If you try to connect a debugger to an SoC that uses this operational model, then the debugger will only allow you to connect to the Non-secure state.

Core or processor is in a different Exception level

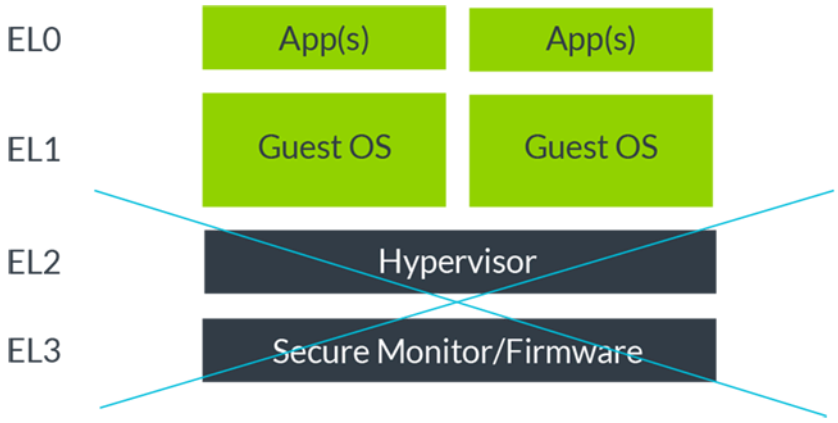
There are two common reasons why EL3 is not entered when connecting a debugger to a target.

The first reason is that the Exception level is not implemented.

In the Armv8-A architecture, certain Exception levels are optional. In some Armv8-A processor or core implementations, an Exception level is not accessible because it is not present in the design.

This diagram shows a design in which EL3 and EL2 are not implemented:

Figure 5-2: Target State Exception level not implemented



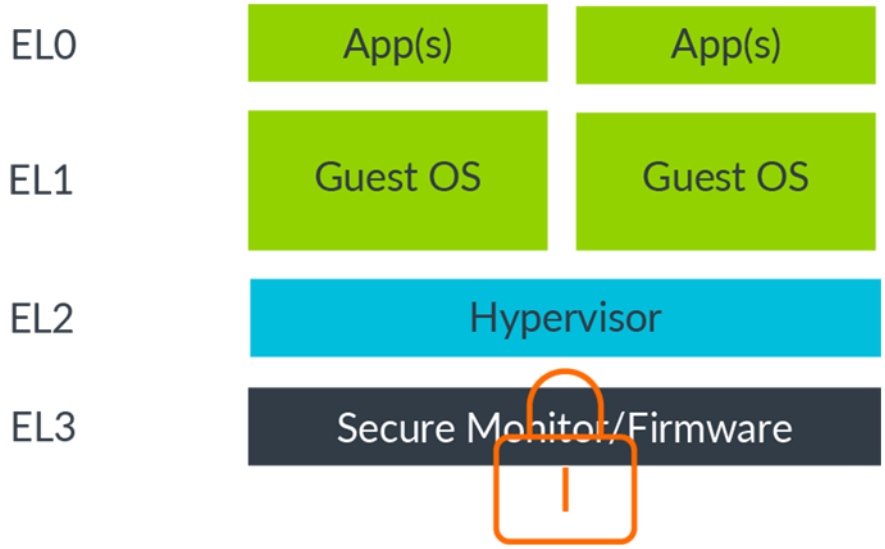
If you connected a debugger to an SoC that uses this operational model, you can only debug Exception levels EL1 and EL0.

The second reason why EL3 is not entered when connecting a debugger to a target is that the hardware or software locks users out of an Exception level.

SoC designs can lock users out of certain Exception levels either through hardware or software. This locking usually occurs late in the development cycle of the SoC, so that unprivileged users cannot access aspects of the code or SoC design.

This diagram shows an SoC in which EL3, which is usually where the secure monitor and firmware reside, is locked:

Figure 5-3: Target State Exception level locked



If you connected a debugger to an SoC that uses this operational model, then you could debug EL2, EL1 and EL0, but not EL3.

6. Initializing a target

Some targets require more steps before establishing a debug connection. The exact nature of these steps will depend on the target that you are using. If you are working with an unfamiliar target, look through the user manual for the target, or consult the designers of the target, to determine the requirements for establishing a debug connection.

Here are some of the more common target initialization steps:

Powering up or taking a core or processor out of reset

As explained in Target state, there are various reason why a core or processor might be powered down or held in reset.

Here are some common steps that you can follow to put the core or processor into a better state for debug:

- Perform reads and writes to memory or specific devices
- Modify OS build or run attributes
- Change firmware or target settings
- Manipulate hardware elements, for example switches or SD cards

Unlocking scan chains or devices

Some SoC designers add locking devices or manipulate the debug hardware, so that users without certain information cannot access the debug probe on the target. These locking techniques prevent unprivileged users from debugging the SoC. These techniques are usually implemented late in the SoC development cycle. It is possible that earlier revisions of a target do not require unlocking, but later revisions do.

Here are some common steps that you can use to unlock the debug logic:

- Feed certain patterns into the lock device on the scan chain
- Perform reads and writes to memory and/or specific devices
- Manipulate hardware elements, for example switches or jumpers

Multiplexed signals

Pin space on an SoC is usually limited. This means that less frequently used devices might have their signals multiplexed with other devices on the SoC. For example, the signals of the off-chip trace port, the Trace Port Interface Unit (TPIU), might be multiplexed with the General Purpose I/O (GPIO) signals.

Here are two common steps that you can follow to demultiplex signals:

- Perform reads or writes to memory or registers
- Manipulate hardware elements, for example switches or jumpers

Missing connectors

Because pin space on an SoC is generally limited, debug output lines on an SoC might be available, but the physical connector is not. If there is pinout space on the PCB, you can overcome this issue by soldering a connector to the board.

In the late stages of the development cycle, the debug socket for an SoC might not be physically available. This might be because the SoC designer does not want users to be able to debug the target, or, or because debug is available through a non-physical method, for example gdbserver. If no socket is available, consult the user manual for the target, or other resources, for example tutorials or community postings.

7. Check your knowledge

The following questions will help you test your knowledge:

In a debugger, which type of debug is stepping code? Invasive, non-invasive, or neither

Invasive

If a processor is being held in reset, which debugger operations would you be about to perform? Stepping code, updating the registers, executing code, or none of the above

None of the above

If the Secure state is locked on an SoC, what is the highest possible Exception level that you could debug? EL3, EL2, EL1, or EL0

EL2

8. Related information

Here are some resources related to material in this guide:

Exception model guide

Memory model guide

Debugger usage on Armv8-A guide

Ask development questions, and find articles and blogs on specific topics from Arm experts at the Arm community.