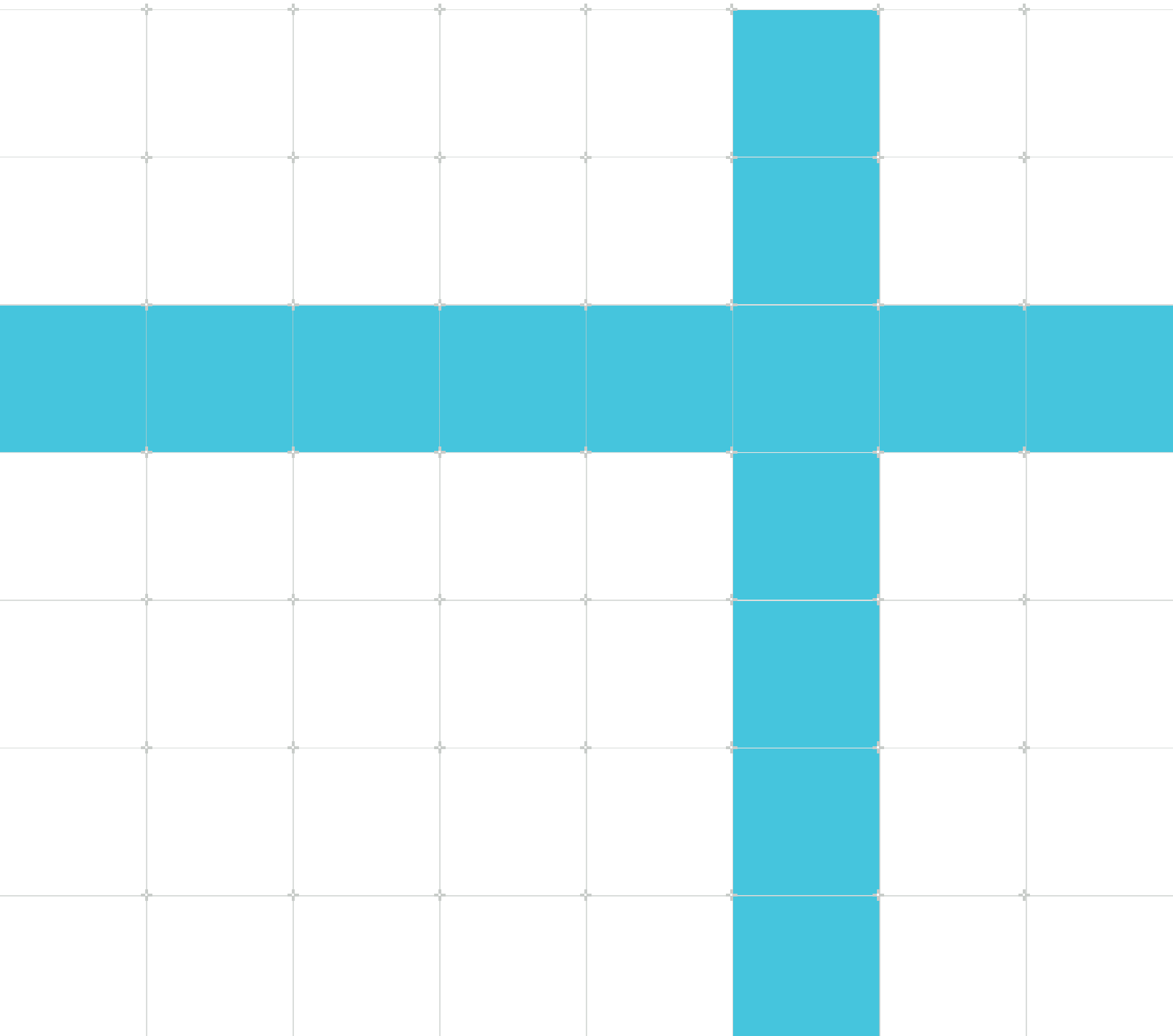
arm

Learn the architecture - Debugger usage on Armv8-A

Version 2.0

Issue 00
102140_0200_00_en



Learn the architecture - Debugger usage on Armv8-A

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	1 April 2019	Non-Confidential	Initial release
0200-00	23 July 2020	Non-Confidential	Updated images and template

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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

Most readers of these guides will have at least some experience with debuggers. No matter which debugger you choose to use, there will always be some common features that every debugger will offer. For instance, every debugger will provide ways to:

- Connect to a target
- Download a program
- Start, stop, and step through program execution
- View memory and registers contents

This guide will not focus on what a debugger offers, or on different debugging methodologies. Instead, we will look at characteristics that are common to bare-metal debuggers that target the Armv8-A architecture. Also, we will explain how these features are achieved, what you need to consider when you work with them, and possible consequences of their use. This guide uses Arm Development Studio, Arm Compiler 6, and GNU Compiler Collection (GCC) to illustrate these concepts.

At the end of this guide you will understand:

- The areas of interest when loading a program and connecting with a debugger
- The problems that might occur when performing a reset
- Different debugger operations, how each operation is performed, and the possible effects of each operation
- The importance of the different memory spaces and register sets used by an Armv8-A processor
- What a user will experience when debugging over processor or core powerdown

Before you begin

This guide assumes that you are familiar with other guides in this series:

- Before debugging on Armv8-A
- Introducing the Arm architecture
- Arm v8-A memory model

You should also be familiar with the material in our Common Tasks guide Building your first embedded image.

2. Program load

Most debugging sessions will involve debugging some type of program. The program may already be on the target when the debugger connects, or the program may be loaded as part of the debugging process. The situation gets more interesting when you consider where the program will be run, and what the program does contain and does not contain.

Program location

The Armv8-A architecture does not specify a set memory map or layout. This means that the arrangement, type, and addresses of memory for an Armv8-A SoC depends on the design of the target. Before trying to load a program, you should check the documentation for your target to see what memory is available, and where that memory is located.

Some targets or execution environments require writing a program to flash memory. Flash memory requires a lot more from the debugger, and the user, than other memory types. This is because writing a program to flash memory usually involves having a flash algorithm written in a C-style language. The algorithm provides a flashing-capable debugger with the information it needs to access the flash memory. Each flash device has its own set of requirements, and each debugger will have its own flash information requirements and procedures.

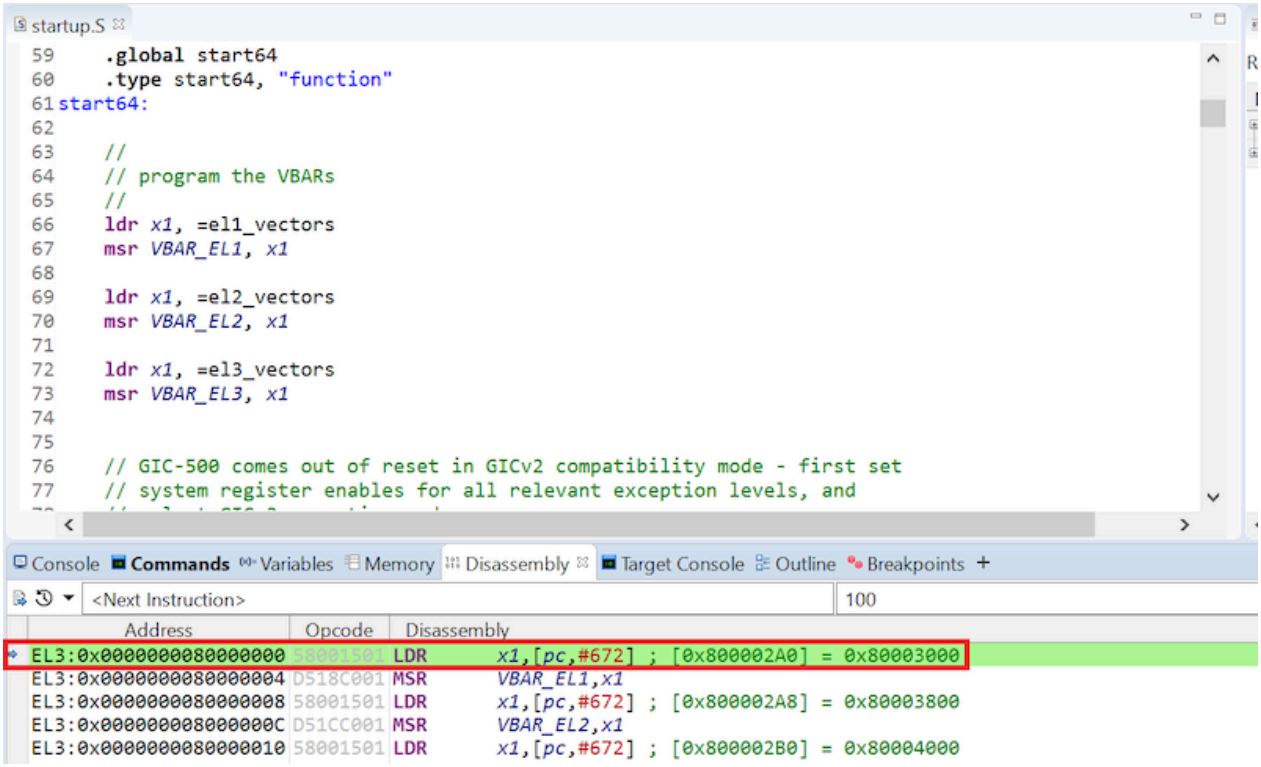
Programs are sometimes executed from *read-only memory* (ROM) types. This only affects a debugger session when setting breakpoints. We will discuss this in detail in Breakpoints.

Program contents

If you work with *GNU Compiler Collection* (GCC) or Arm Compiler 6 when debugging your system, the tools produce an Executable and Linkable Format (ELF) image. An ELF image contains the code and data of the program, and additional metadata, for example debug information. This debug information tells a debugger how to match the binary back to the source when debugging.

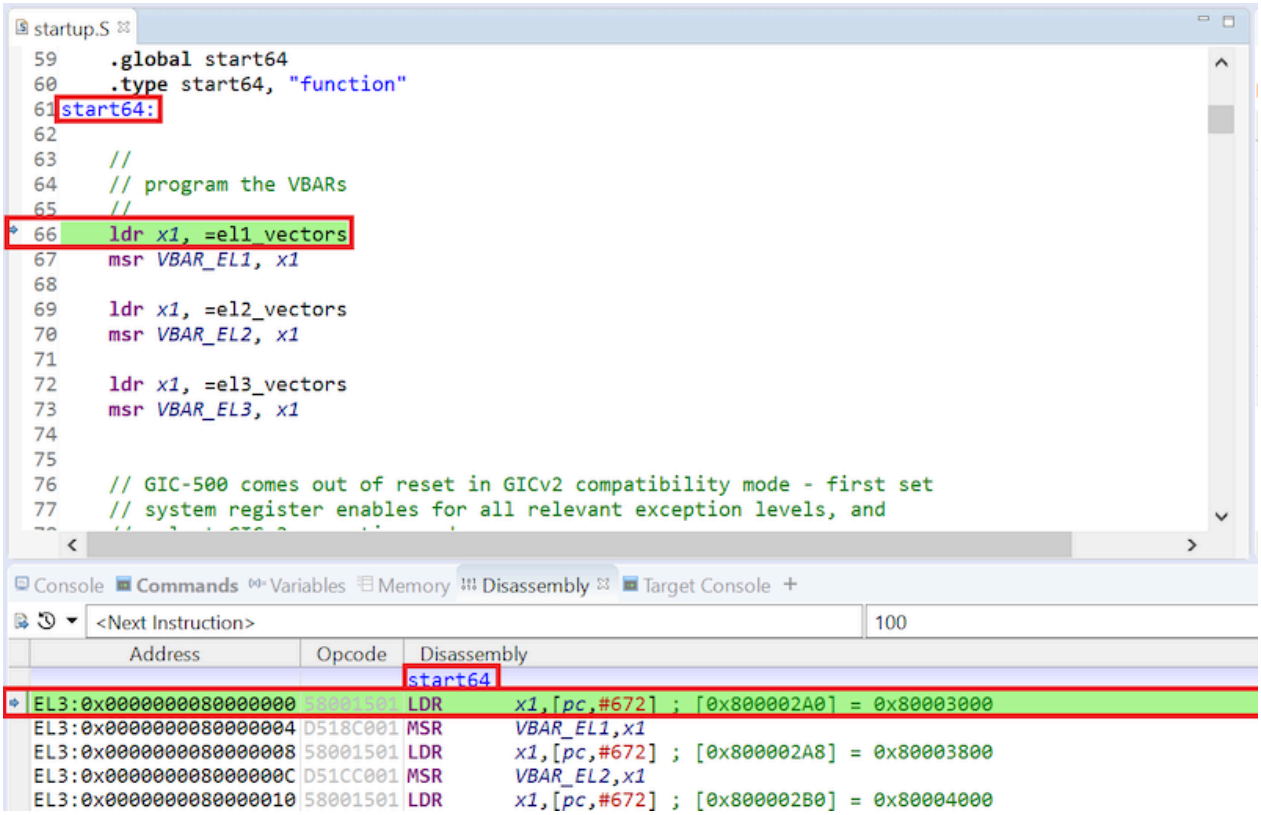
This screenshot shows what happens when you are debugging using Arm Development Studio. You can see, in the area with a red outline, that the *Program Counter* (PC) location is specified in the Disassembly view with a small blue arrow and green highlighted code. However, the function that the code belongs to is not listed in Disassembly view, and the PC location is not indicated in the associated source file: startup.

Figure 2-1: Screenshot from Arm Development Studio showing what happens when you are debugging



If you load the program's debug information, you would see what is shown in the following screenshot:

Figure 2-2: Screenshot from Arm Development Studio showing what you see when you load program information



Now, the PC location and the associated function, start64, are both indicted in the Disassembly view. In addition, the PC location is indicated in the associated source view.

You might work with programs that do not have debug information, or with programs that only have partial debug information. For example, a program may be built against a third-party library that does not have debug information. You can still debug such programs, but the debug process may be more difficult.

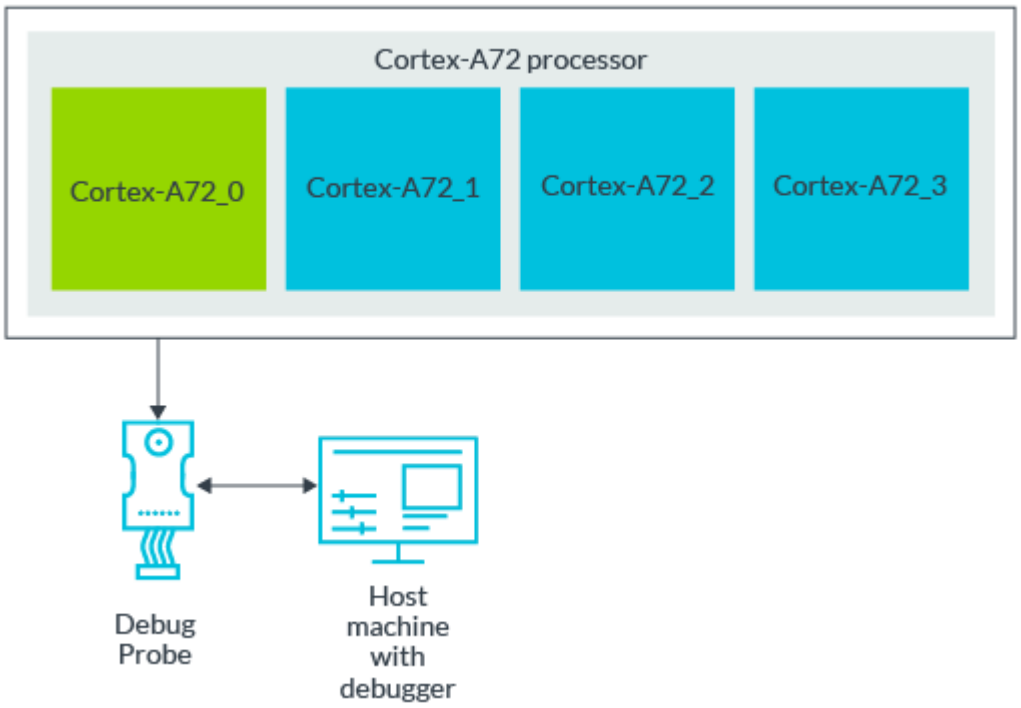
3. Connection types

Most debuggers will give you the option to connect to a target using different methods. Some commonly used connection types are: Single-core, *Symmetric Multiprocessing* (SMP), and *Asymmetric Multiprocessing* (AMP). We will describe each connection type and when you might want to use them.

Single-core connection

A single-core connection usually means that the debugger will connect to only one core of a processor. For example, if you have a four-core Cortex-A72 processor, and you establish a single-core debug connection to the core 0, then the debugger will connect only to Cortex-A72 core 0, which can be written as Cortex-A72_0. The following diagram illustrates a single-core connection to Cortex-A72_0 of a four core Cortex-A72 processor:

Figure 3-1: A single core connection with debug probe



Making a single-core connection usually means that the debugger operations will only affect one core. For example, if you use a debugger to halt the Cortex-A72_0 core, the other cores in the Cortex-A72 processor will not halt.

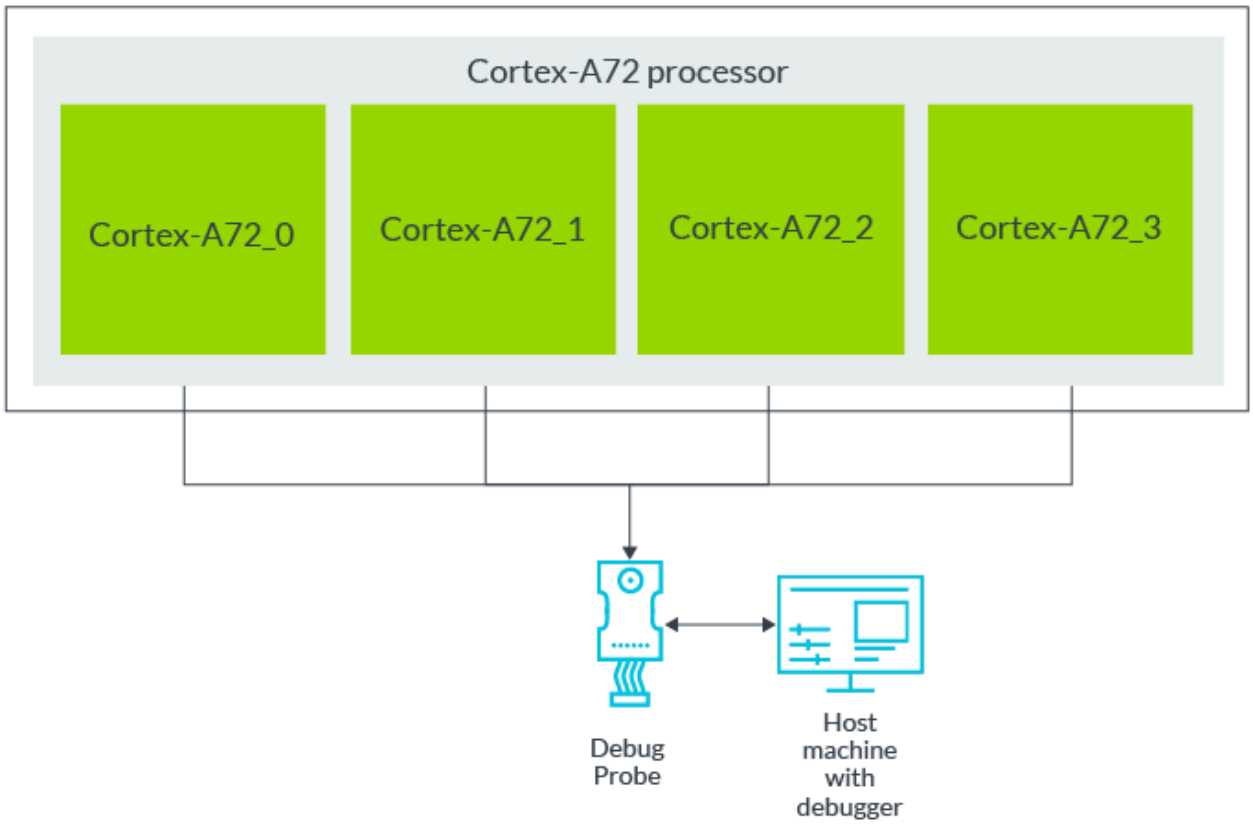
When using a single-core connection on targets that have more than one core, halting the single core may cause issues on the target. This is because, when halted, the core cannot receive or acknowledge any messages from any other core. If the other cores rely on the single core responding, the other cores may enter a problematic state if no responses occur.

Single-core connections are useful if you want to interact with only one core without affecting the other cores or processors on the SoC.

Symmetric Multiprocessing connection

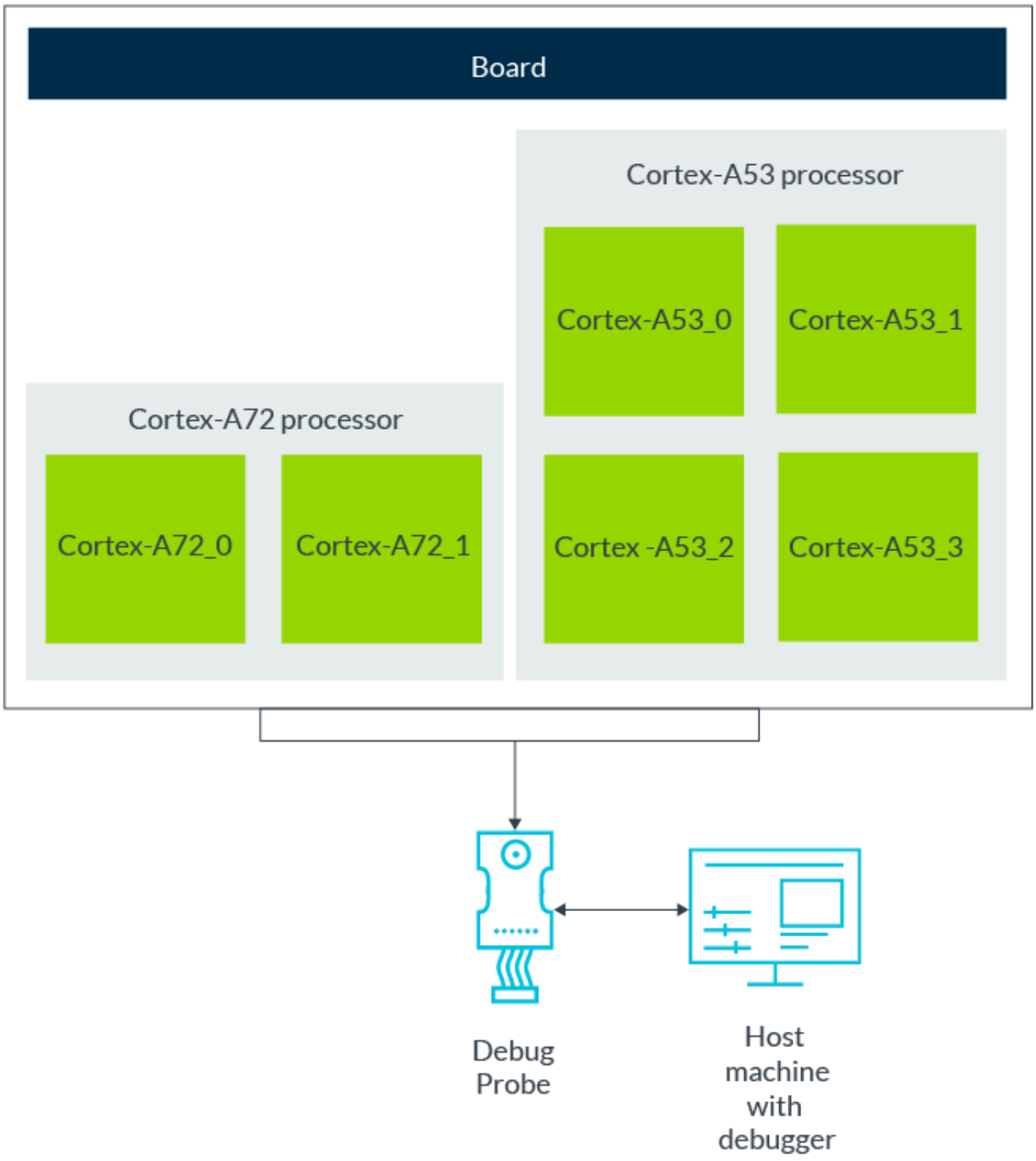
A *Symmetric Multiprocessing* (SMP) connection usually means that the debugger will connect to multiple cores or processors at the same time. For example, if you have a four-core Cortex-A72 processor, and you establish an SMP connection to the processor, then the debugger will connect to all four cores of the Cortex-A72 processor. You can see this example connection in the following diagram:

Figure 3-2: Connecting a debugger to multiple cores



An SMP connection can also take place when you connect two of more different processors at the same time. For example, if you have a two-core Cortex-A72 processor and a four-core Cortex-A53 processor, and you establish an SMP connection to both processors, the debugger will connect to all the Cortex-A72 and Cortex-A53 cores at the same time. The following diagram illustrates this example:

Figure 3-3: Connecting a debugger to multiple processors



For Arm Development Studio, this type of connection is called a big.LITTLE connection.

Making an SMP connection usually means that a debugger operation for one of the cores will affect all the cores. For example, if you use a debugger to halt the Cortex-A72_0 core, the other cores in the Cortex-A72 will also halt.

Depending on the debugger, performing an SMP connection might also imply other factors to a debug environment. For instance, an SMP connection in Arm Development Studio will imply to the debugger that:

- All the cores or processors in the connection have the same device memory map
- All the cores or processors will use the same image during execution
- All the cores or processors have identical debug hardware

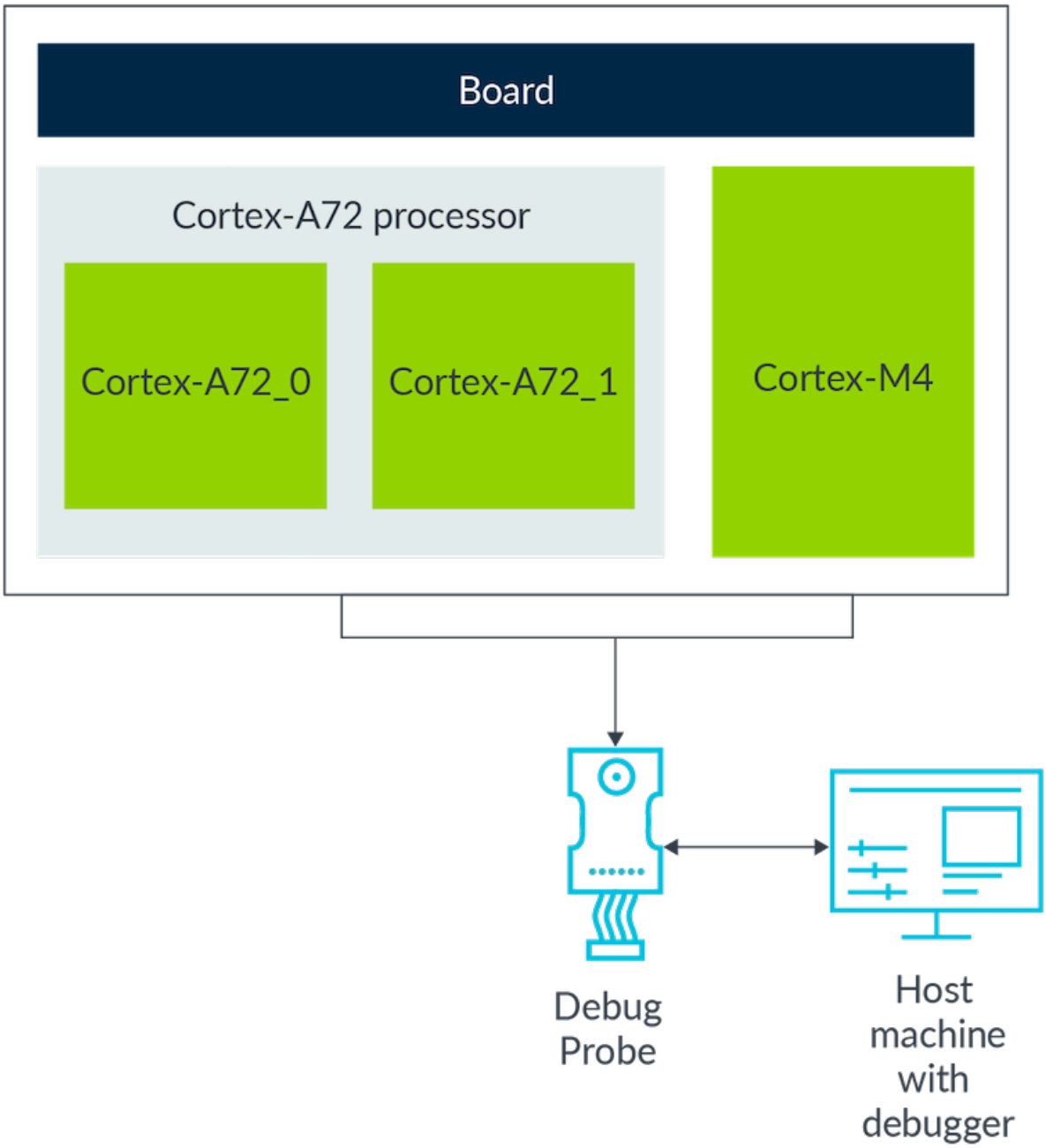
An SMP connection is useful if you want to see the interaction between multiple cores or processors at the same time, and if you want debug operations to affect all processors or cores in the connection.

Asymmetric Multiprocessing connection

Like an SMP connection, an *Asymmetric Multiprocessing* (AMP) connection usually means that the debugger will simultaneously connect to multiple cores or processors. The difference between an SMP connection and an AMP connection is that the cores or processors that the debugger connects to do not operate in the same way.

For instance, let's look at an SoC which contains a Cortex-A72 processor for application execution, and a Cortex-M4 core to control the power and reset logic. Because they are different architectures, the Cortex-A72 and Cortex-M4 in this heterogeneous platform will not run the same program, and do not have the same debug hardware. If you want to view the state of all these cores, you would establish an AMP connection. An AMP connection will provide a connection to both the Cortex-A72 and the Cortex-M4 simultaneously, but will imply a looser coupling between the two devices to the debugger. The following diagram illustrates an AMP connection to a Cortex-A72 and a Cortex-M4 processor:

Figure 3-4: Asymmetric Multiprocessing connection



Depending on the debugger, an AMP connection might cause debug operations performed on one core to affect the other cores in the connection. For example, if you halt the Cortex-A72_0 core with a debugger, the rest of the cores in the Cortex-A72 processor will probably halt, but the Cortex-M4 might not halt.

An AMP connection is useful if you want to see the state of multiple cores, which are performing different operations, at the same time.

4. Reset

Reset is an important aspect of every debugging experience. Reset will put your target in a known working state, accomplish tasks before processor execution begins, or avoid a problematic situation by starting over. The Armv8-A architecture provides very few restrictions on how, when, and where reset is performed.

This lack of restriction in the architecture means that the SoC or board designer determines:

- How the different devices on the SoC are reset
- How the reset requests are distributed and accomplished on the target
- How the reset signals are connected to any physical debug connectors

This lack of restriction in the architecture also means that the debugger determines:

- What reset types are available
- What is invoked when a reset request is made
- Whether the reset is done through hardware or software means

This means that it might be difficult to determine what happens when you perform a reset in a debugger.

For example, when you perform a System Reset on a hardware target in Arm Development Studio, the *System Reset* (nSRST) pin signal on the target debug connector is toggled. The design of the SoC determines whether the nSRST signal is connected to the devices necessary to create the equivalent of a System Reset on the target.

It is a good idea to consult your debugger and target documentation to determine how the desired reset option can be achieved.

5. Run control options

Most debuggers provide run control options, for example stop, run, and step, although the names used for these options may vary. In this section, we will discuss each of the run control options, and provide some of the implications of each.

Stop

The stop option is usually used when you want to inspect the processor or target state for debugging purposes. Stopping or halting a core will put the processor into Debug state. Debug state occurs when:

- The *Processing Element* (PE) stops executing instructions and control is passed over to a connected debugger.
- The debugger determines which instructions, if any, will be executed.
- The *Debug Communications Channel* (DCC) in the debug logic can be used to pass data between the PE and the debugger.

The core cannot service any interrupts when in Debug state.

To perform debug operations, the debugger will need to execute some instructions on the processor. These debug operations generally include:

- Reading and writing memory
- Reading and writing processor registers
- Loading *Memory Management Unit* (MMU) and *Translation Look-aside Buffer* (TLB) table entries
- Moving between *Exception levels* (ELs)

The instructions executed to perform debug operations will behave in the same way as any other instructions executed by the processor. This means that, when these operations are performed, subtle changes might be made to the processor state. For instance, writing to a memory address using a debugger might, in turn, generate the following:

- A new TLB entry for the address translation, which might result in TLB eviction.
- Cache lines pulled in if the address is cached, which might result in cache line evictions.
- A write request to external memory.

In most cases, these subtle changes will not affect the processor or target. However, if you are debugging a problem that is sensitive to changes in timing or caching, you should keep these possible effects in mind.

While a processor is stopped, the rest of the target devices might not stop. For instance, if a watchdog timer is running while a processor is halted, the timer might trigger due to lack of activity and reset the board while you are debugging.

When debugging, the core will usually be clocked at a much slower rate than normal. This means that some operations, like large region memory fills, will take longer to perform with a debugger.

Run

The run option usually means that we want to start, or resume, normal processor execution. If the processor is resuming execution, the debugger needs to remove any changes that no longer apply or are visible to the user. A good example is a debugger replacing a software breakpoint that has already been hit for the original instruction, and then resetting the breakpoint when execution commences. The subtle changes that are introduced when executing instructions to perform debug operations are usually not changed back to their original state.

Step

The step option usually means to move the current execution point down or over one instruction or one source line.

A single instruction step is achieved using a halting step debug event. A debugger uses the following operations to perform a halting step debug event:

1. With the PE in Debug state, the debugger activates halting step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

Exceptions can be generated by the instruction being stepped, or just before or just after the instruction step occurs.

Stepping one source line involves stepping one or more instructions. You may see unusual behavior when stepping in sources files, for example:

- Only seeing a PC indicator in the assembly code but not a source file
- Stepping more source lines than you expected

The common reasons for these types of behavior are:

- Lack of debug information. As mentioned in Program load, a debugger usually needs debug information in order to display the position of the PC in a source file. If you step into a source file for which the debugger does not have debug information, the PC indicator will probably only show up in the assembly code view. This can generally be fixed by loading the source file debug information into the debugger.
- The program was built with a high optimization level. Building with a higher optimization level allows the build tools to be more aggressive about removing code that appears unnecessary. This could mean that the source lines either no longer have any instructions associated with them, or potentially have very few instructions associated with them. If this is the case, you should to investigate why there is an absence of code, or rebuild the program with a lower optimization level.
- Moved Exception levels. Some debuggers will require additional data. for example debug information, to be available when moving between different Exception levels. If the additional data is not present, the debugger may not be able to show the accompanying source files when stepping in a different Exception level.

6. Breakpoints

Armv8-A processors provide the ability to set breakpoints on code or instructions of interest. Setting breakpoints is a common and useful debugging tool to help determine the cause of unexpected or incorrect behavior during execution.

Most debuggers capable of debugging Arm architecture processors will provide two types of breakpoint: software and hardware breakpoints. The following table compares the different breakpoint types:

Debugging	Software breakpoints	Hardware breakpoints
Implemented by	Replacing an existing instruction with a BKPT assembly instruction	Setting up some comparators in the debug logic of the core or processor
Operation when hit	Put the core into debug state	Put the core into debug state
Action on resuming execution	The BKPT instruction will be replaced with the original instruction.	The breakpoint will remain until the breakpoint is disabled or removed.
Number of breakpoints available	A large number (may be limited by the debugger)	Limited by Armv8-A implementation (see core reference manual)

7. Watchpoints

Armv8-A processors provide the ability to set watchpoints on memory values of interest. The behavior and implementation of watchpoints is very similar to hardware breakpoints. A debugger will set watchpoints using some comparators in the debug logic of the core or processor.

When a watchpoint is hit, the debugger will put the core into debug state. Usually, when the core resumes normal execution, the watchpoint will remain until the watchpoint is disabled or removed.

Because they rely on the debug logic, watchpoints can be set on any memory address in any type of memory.

Like hardware breakpoints, the Armv8-A architecture does not specify how many watchpoint comparators are present in an Armv8-A processor implementation. This means that each Armv8-A processor implementation could have a different number of watchpoints comparators available. The reference manual for the core should contain information on how many watchpoint comparators are available.

8. Vector catch

If you have worked with Arm processors that are earlier than the Armv8-A, you may be aware of a mechanism called Vector Catch. Vector Catch allows a debugger to trap exceptions based on an address or exception match. Usually, a debugger will provide a way to enable and disable Vector Catch for each exception allowed. If Vector Catch is enabled for a certain exception, the debugger will typically halt when that exception occurs.

However, the Armv8-A architecture does not allow Vector Catch exceptions when the processor element (PE) is using an AArch64 translation regime. Also, Vector Catch has been deprecated in the architecture, so it may not appear in future Arm architecture versions.

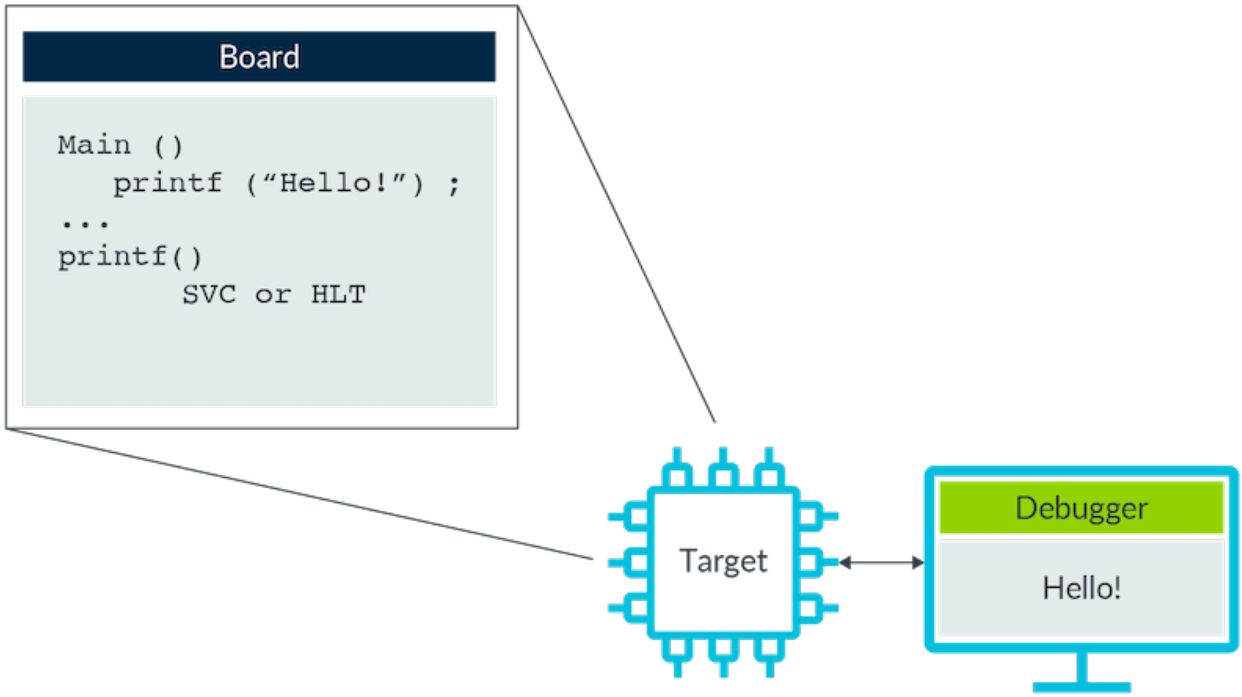
9. Semihosting

Some debuggers that can be used with Armv8-A will provide a mechanism called semihosting. Semihosting enables code that is running on a target to communicate with and use the *Input/Output* (I/O) facilities on a host computer. A good example is a program running on a target that uses the `printf()` function to output a message, and the message appears in a debugger console instead of in a target output device or console.

Semihosting works by building a program against a set of semihosting-enabled libraries which contain special `SVC` or `HLT` instructions in the I/O functions. When a debugger encounters these special `SVC` or `HLTv` instructions, the debugger will take control and perform the I/O operation that the program needs.

The following diagram illustrates how semihosting works if the target program outputs “Hello!” using a `printf()` function:

Figure 9-1: Semihosting for a print() outputting Hello!



Semihosting is useful when you are working with a target that does not have all the I/O functionality, or has only part of the I/O functionality, that is needed to run or validate software.

Because the debugger is performing the I/O operation, using semihosting can cause some differences when compared to executing a program without semihosting. For example, semihosting will effectively halt execution while the semihosting operation is performed. This could cause unwanted timing changes to time-critical systems.

In most cases, using semihosting will not have any noticeable effect on target execution. However, you should consider the possible effects of semihosting if you encounter unexplainable behavior around I/O operations.

10. Memory

Viewing and modifying memory can be a crucial debugging technique. A debugger may have more than one way to access memory on a target. The two most common access methods are using the processor or using the CoreSight debug infrastructure. We will describe each access method, but we will not describe in detail how these access methods are achieved.

Memory access using a processor

Most debuggers will default to performing memory reads and writes by executing instructions on the processor and using the *Debug Communications Channel* DCC to input and retrieve the data. Because the memory accesses are performed by the processor, this method means that:

- The instructions used to perform the access should be an appropriate length size and byte size for the memory being accessed. Most debuggers will default to a certain length and byte size if none are specified.
- The memory accesses are treated in the same way as if software had issued them. This means that they are subject to translation, permission checking, and ordering rules.
- The processor must be in Debug state (stopped or halted) for the accesses to be performed.

As mentioned in Run control options, performing a memory read or write can subtly change the processor state especially if the *memory management unit* (MMU) and caches are enabled. This is worth keeping in mind if you see unexpected behavior involving caches or timing when performing memory accesses with a debugger.

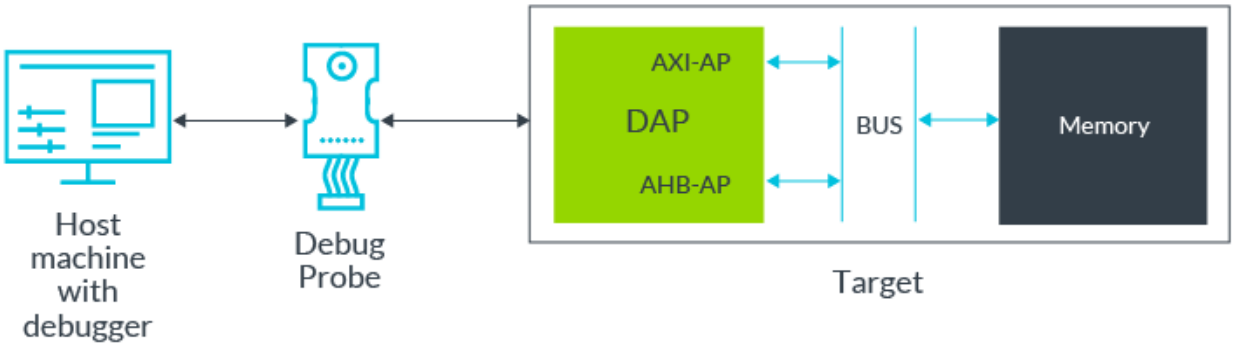
Memory access using the CoreSight debug infrastructure

The Armv8-A architecture uses a debug infrastructure called CoreSight. The CoreSight infrastructure attaches compatible logic to the different *Access Ports* (APs) of a *Debug Access Port* (DAP). A typical debugger setup will perform debug operations in this way:

1. The debugger passes commands to the connected debug adapter
2. The debug adapter will pass commands to the DAP via a debug connector (like Joint Test Action Group (JTAG) or Serial Wire Debug (SWD)
3. The DAP will access the appropriate Access Ports to perform the requested debug operations

Depending on the SoC design, the DAP may contain an *AMBA High-performance Bus Access Port* (AHB-AP) or an *Advanced Extensible Interface Access Port* (AXI-AP). The AHB-APs or AXI-APs connect to buses able to access memory. If these connections exist, some debuggers will allow users to access memory via the DAP AHB-APs or AXI-APs. The following diagram illustrates the typical connections between a debugger, a DAP, and a target's memory:

Figure 10-1: Typical connections between a debugger, a DAP, and a target's memory



Because the reads and writes of the memory directly access memory, this method means that:

- The access will involve doing debugger memory access commands. The access size and length will be entirely dependent on the debugger being used.
- The address used will always be a physical address.
- Because the access is not being done by the processor, the contents of the caches in the processors will not be updated.
- The memory operations are not atomic or locked, so if other devices are accessing the same memory region at the same time, the data may be corrupted.

11. Memory spaces

To access memory successfully, debuggers need to be aware of what memory spaces are available for each processor implementation and any attributes that apply for each space. Most debuggers will have a default memory space setup that the user may need to adapt for their target.

The memory attributes that are available will vary based on the debugger. Some common attributes for memory spaces or regions are read-only, write-only, read/write, and byte access size.

Each Exception level and security level can be associated with different attributes for the memory spaces. For Exception levels, an example is different memory attributes for Non-secure world EL2 addresses and Non-secure world EL1 addresses. For security levels, an example is different memory attributes for Secure world EL1 addresses and Non-secure world EL1 addresses.

Most debuggers will also allow users to define multiple attributes to memory spaces in the same Exception level and security level. For instance, you could have two memory regions for Non-secure world EL1 addresses, in which one region is marked as read-only and the other region is marked as write-only.

12. Registers

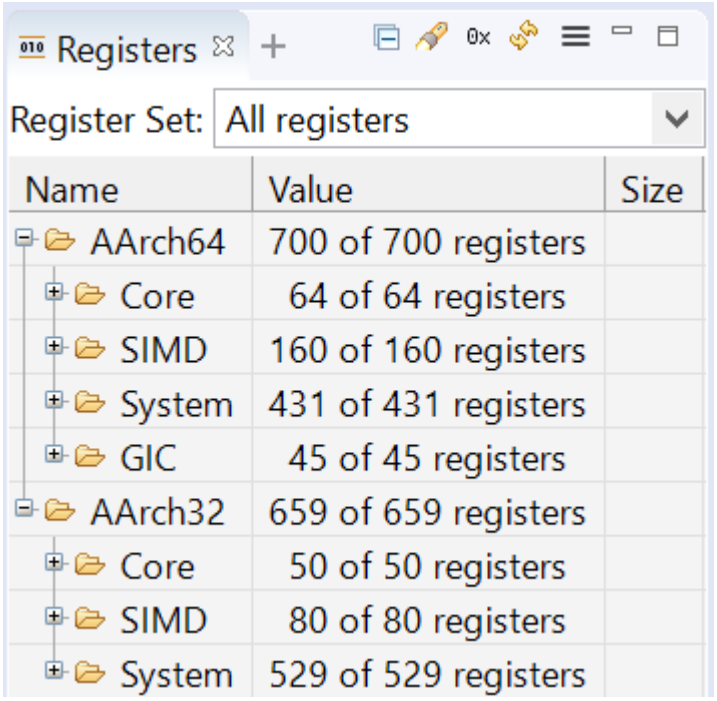
Processors contain various sets of architectural registers, to configure and show the state of the processor. Usually, a debugger will provide some type of window, or view, that shows the current contents of the processor registers. In this section, we will discuss the processor AArch64 and AArch32 architectural registers and debug registers, and explain how they are accessed by a debugger.

AArch64 and AArch32 registers

Armv8-A defines two sets of architectural registers: AArch64 and AArch32. This can sometimes cause confusion, because the AArch64 register names and layout can be very similar to their AArch32 register counterparts.

Most debuggers will provide a way for you to identify which set of registers you are working with. For example, Arm Development Studio lists the architectural register set at the top level of the hierarchy in its Registers view, as shown in the following screenshot:

Figure 12-1: Screenshot from Arm Development Studio showing the architectural register set in Registers view



The method used to access a processor register is very similar to the method used to access memory in a processor. Processor instructions are executed to access the register, and the register data is input and retrieved using the *Debug Communications Channel* (DCC).

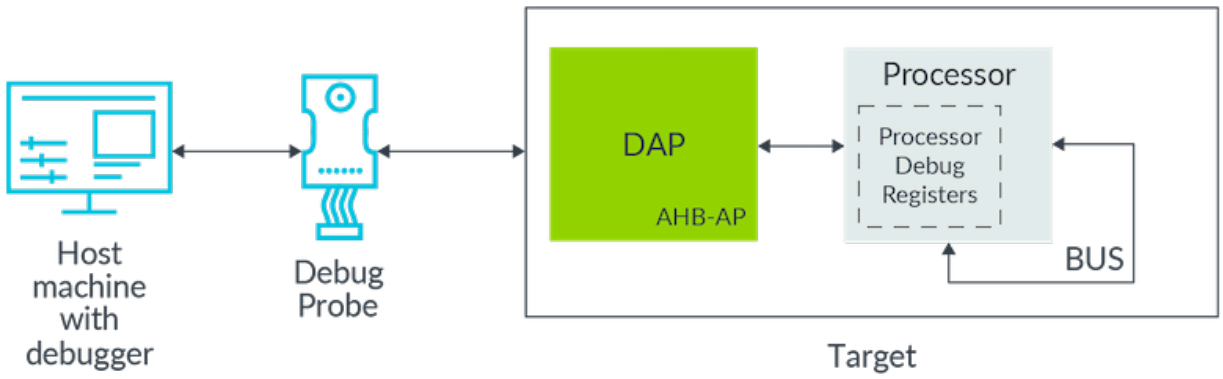
Loading, modifying, and refreshing the register contents in a debugger can take a considerable amount of effort. For instance, just loading a subset of the register contents upon initial connection in Arm Development Studio requires 178 debug operations.

Because the processor is used to perform this operation, subtle changes may occur to the processor state, for example the instruction cache contents or target timing. These subtle changes are unlikely to make any difference, but it is worth remembering that this might happen.

Debug registers

The processor debug registers are used to determine, configure, and monitor the debug state of the processor. For an Armv8-A processor, debuggers will access the debug registers via the CoreSight *Debug Access Port* (DAP), and then through the *AMBA Peripheral Bus Access Port* (APB-AP). Depending on the SoC implementation, the debug registers might be memory mapped so that the processor can access them either during execution or via a debugger memory window or memory access command. The following diagram illustrates the debugger and processor routes to the processor debug registers:

Figure 12-2: Debugger and processor routes to the processor debug registers



Some debuggers will allow users to access the debug registers via the APB-AP directly. You will need to consult your debugger user manual to determine whether this approach is possible.

13. Debugging over powerdown

When using a debugger, you may have connected to or debugged when one or more cores on the target are powered down. Depending on the SoC design or core power state, you may have little or no debugging ability. In this section, we will discuss the architectural facets of debugging over power down, and what power domain separation on a *DynamIQ Shared Unit* (DSU) would look like. We will not discuss how the power states on the DSU are achieved.

Power down architecture information

The Armv8-A architecture implementations usually separate processor logic and processor debug logic into different power domains. Depending on the core power state, this could allow a debugger to continue debugging even when one or more cores are powered down.

The four architected core power states are:

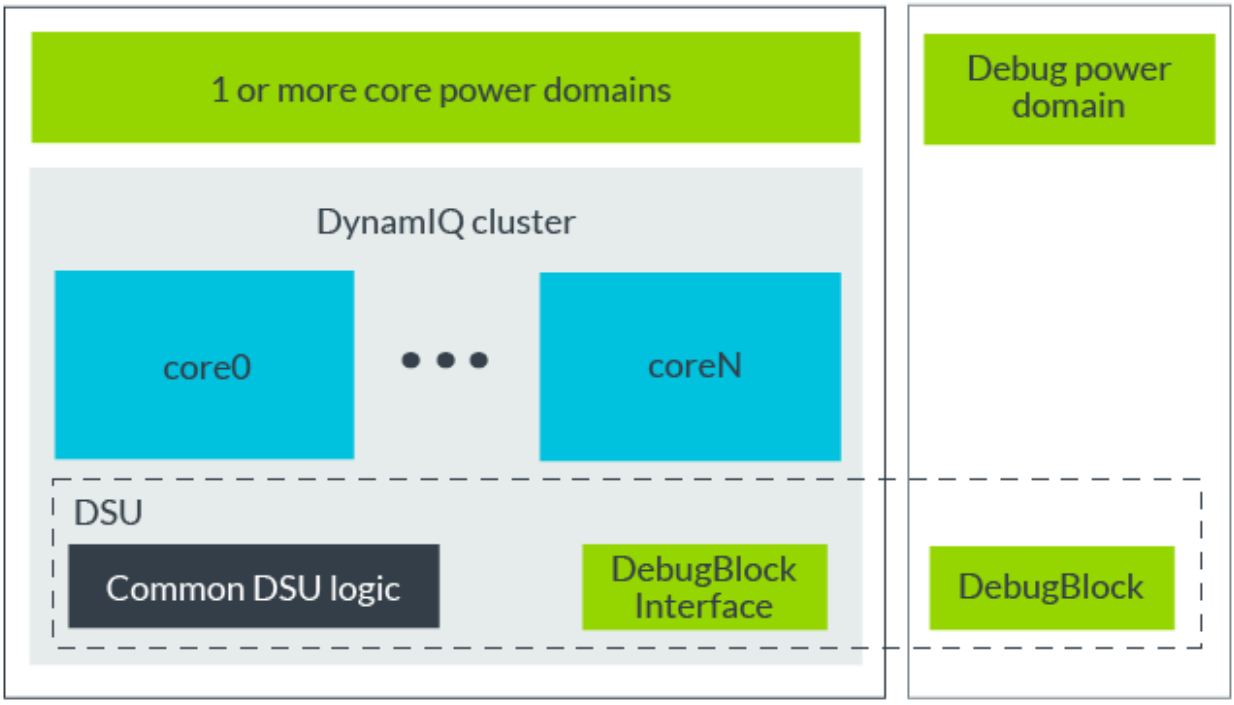
- Normal. The core power domain is fully powered up and the debug registers are accessible.
- Standby. The core power domain is on, but there are measures to reduce energy consumption. In a typical implementation, the *processor element* (PE) enters standby by executing a WFI or WFE instruction, and exits on a wake-up event. The PE preserves the PE state and debug logic state. In standby, the PE is unavailable, and no debug settings are cleared. Receiving a debugger request or accessing the external debug interface will wake up the core from standby. A debugger will not be able to tell if the core is in standby, because standby is transparent. This means that standby is indistinguishable from normal operation.
- Retention. The OS takes some measures to reduce energy consumption. The PE state, including debug settings, is preserved, which allows the core power domain to be at least partially turned off. The saved PE state is restored when it changes from retention state to normal operation. Debugger requests stay pending, and debug registers in the core power domain cannot be accessed. This means that debug capabilities will be very limited when a core is in retention state.
- Powerdown. The OS takes some measures to reduce energy consumption by turning the core power domain off. These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown. Changing from powerdown to normal operation must include:
    - A Cold reset of the PE after the power level has been restored.
    - The OS restoring the saved PE state.
    - Debug events stay pending, and debug registers in the core power domain cannot be accessed. This means that the debugger will probably only be able to access the debug logic if it is a different power domain to the core. This is because no interaction with the powered-down core is possible.
    - OS threads will appear in the debugger as powered off or held in reset state.

Designs with DynamIQ Shared Units

If your SoC design uses DynamicIQ clusters with their accompanying DSUs, the DSU allows for the core-associated logic to be in a separate power domain to the debug logic in the DebugBlock. The

following diagram illustrates the possible power domain divide between the core and DSU logic and the DSU DebugBlock:

Figure 13-1: The possible power domain divide between the core and DSU logic and the DSU DebugBlock



The separate power domains allow the cores and the cluster to be powered down, at the same time maintaining essential state that is required to continue debugging. We saw that different power states result in different types of debugging experiences.

14. Check your knowledge

The following questions will help you test your knowledge.

Which core power state is typically achieved by executing a WFI or WFE instruction: normal, standby, retention, or powerdown?

Standby

To break an instruction in ROM, would you use a hardware breakpoint or a software breakpoint?

Hardware breakpoint

When a user performs a system reset on a target using a debugger, are all the devices on the target reset?

Maybe. What happens when a user performs a system reset on a target depends on the debugger and target used.

15. Related information

Here are some resources related to material in this guide:

- Arm Community (ask development questions, and find articles and blogs on specific topics from Arm experts)
- Arm Development Studio User Guide Documentation

Here are some resources related to topics in this guide:

Program load

- GNU Compiler Collection
- Arm Compiler 6

Memory

- CoreSight Debug and Trace

Debugging over powerdown

- Arm DynamIQ technology

Useful links to training:

- Introduction to Armv8-A
- Memory model overview

16. Next steps

In this guide, you have learned about common features of debuggers that target the Armv8-A architecture and what to consider when using them in various ways.

Other guides in this series introduce different aspects of debugging and related topics in detail and provide commentary and examples. To keep learning about these areas, refer to our guides on:

Before debugging on Armv8-A

Semihosting for AArch32 and AArch64 version 2.0