



Debugging Armv8 platforms with CSAT

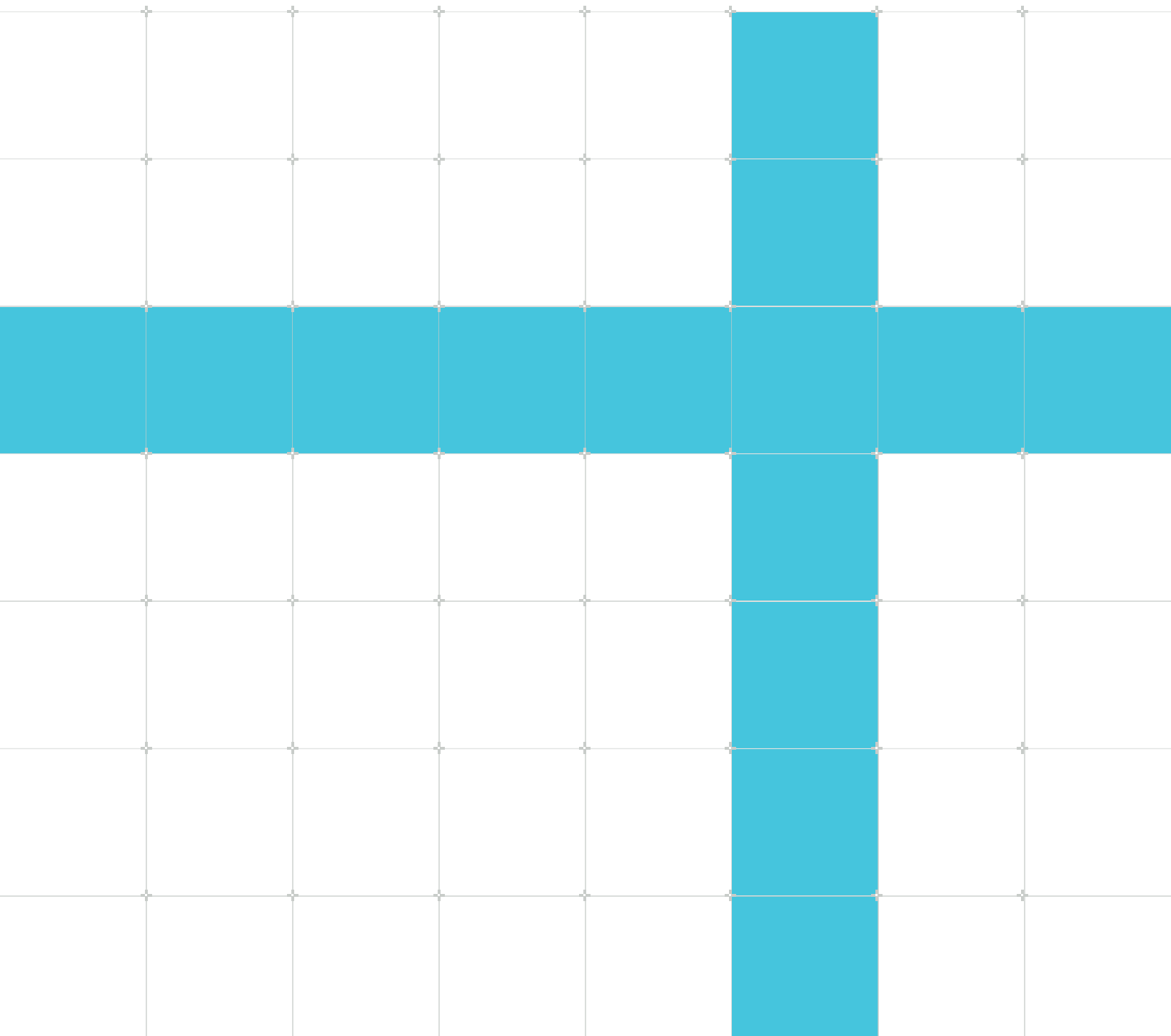
Version 1.0

Non-Confidential

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

Issue 01

102589_0100_01_en



Debugging Armv8 platforms with CSAT

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

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	8 March 2021	Non-Confidential	First release

Proprietary Notice

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 for the purposes of determining whether implementations infringe any third party patents.

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, 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.

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 affiliates) 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 <https://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-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction.....	6
2. Armv8 Topology.....	7
3. Worked Examples.....	11
4. Conclusion.....	22
5. Further Information.....	23

1. Introduction

This tutorial gives an overview of performing low-level debug using the CoreSight Access Tool (CSAT) with an Armv8 target. Low-level debug allows you to:

- Manipulate individual registers, including Debug registers that are not normally accessible to an application-level debugger.
- Perform functions such as halting and restarting the core, setting breakpoints, and watchpoints, and reading the ROM Table.

CSAT only supports [CoreSight SoC-400](#) targets. If you are using a [CoreSight SoC-600](#) target, use [CoreSight Access Tool for SoC600 \(CSAT600\)](#) instead.

CSAT is included in installations of [Arm Development Studio \(Arm DS\)](#).

2. Armv8 Topology

Armv8 platforms use a debug infrastructure called CoreSight to provide visibility into a SoC (System on Chip) for debug purposes. CSAT directly interacts with the CoreSight infrastructure to debug issues at a lower level than most debug tools. To understand how to use CSAT, you must have a basic understanding of the CoreSight infrastructure and its components.

To prepare you to work with CSAT, this section includes information on the following CoreSight topics:

- [What are CoreSight components?](#)
- [What is a CoreSight ROM Table?](#)
- [Useful Debug Registers](#)
- [Useful CTI registers](#)

What are CoreSight components?

CoreSight components provide the debug and trace infrastructure of a SoC. The components include control and access devices like the following:

- Debug Access Port (DAP)
- Trace sources
- Trace links
- Trace sinks.

A DAP is a Debug Port (DP) that is connected to one or more Access Ports (APs). A DP provides a connection from outside the SoC to one or more APs. An external debugger connects to a DAP to access the CoreSight infrastructure.

An AP provides a bridge into another system on the SoC. A Memory Access Port (MEM-AP) provides a window into a memory system. This window allows memory-mapped accesses to debug resources like Debug registers. Read [Understanding the CoreSight DAP](#) for more information on DAPs and APs.

An example of a MEM-AP is an Advanced Peripheral Bus Access Port (APB-AP). In this tutorial, with CSAT, we use the target APB-AP to access the memory-mapped Debug registers.

One type of trace link is a cross-trigger network that consists of Cross Trigger Interfaces (CTIs) and Cross Trigger Matrices (CTMs). CTIs enable the distribution of events to and from sources and destinations in the system. CTIs are connected to each other using one or more CTMs through channels.

The cross-trigger network works by input triggers or the user generating channel events on a channel or connected channels. This channel event triggers an output trigger event like a core halt or restart request. For example, if multiple cores are connected to a channel and one core halts, a synchronous halt of all connected cores occurs. For an Armv8 core, core halt and restart must

happen using CTIs. Read your core Technical Reference Manual (TRM) for more information on the use of CTI channels and triggers.

Read [Understanding trace](#) for more information on how trace works in Arm systems.

What is a CoreSight ROM Table?

A CoreSight ROM Table stores the locations of all the debug components accessible through a DP or MEM-AP. ROM Table values are usually read consecutively starting at the base address of the ROM Table. To find the ROM Table base address, look at the memory map in your target Technical Reference Manual (TRM). Alternatively, read the MEM-AP register BASE at offset 0x F8 to find the ROM Table base address for the MEM-AP.

The processor might see the debug components at a different address to the external debugger. This difference might occur so the external debugger can bypass locks used to restrict software access to the Debug registers. For example, on the Arm Cortex-A53x2 SMM target, the TRM states the debug components start at address 0x20000000. The external debugger accesses the debug components starting at address 0x80000000. For more information, read the Memory system design section of the [Arm CoreSight Architecture Specification](#).

A ROM Table is usually at the beginning of a memory system and is 4KB in size. For targets with more than one cluster of processors, there is usually a top-level ROM Table and a ROM Table for each cluster. Read [Understanding the CoreSight DAP](#) for more information on ROM Tables and the CoreSight architecture.

Useful Debug Registers

To perform low-level debug using CSAT, you must read and write registers that are accessible VIA the external memory mapped interface. The following are some useful Debug registers:

Register name	Memory-mapped offset	Register description
External Debug Status and Control Register (EDSCR)	0x088	Main debug control register for an Armv8 core.
OS Lock Access Register (OSLAR_EL1)	0x300	Controls the OS Lock. Unlocking the OS Lock allows you to access the Debug registers. Write-only register.
External Debug Program Counter Sample Register [31:0] (EDPCSRlo) And External Debug Program Counter Sample Register [63:32] (EDPCSRhi)	0x0a0 And 0x0ac	Optional registers to read the Program Counter when externally debugging. Read-only registers. Not present in all implementations.
External Debug Power/Reset Control Register (EDPRCR)	0x310	Controls the powerup, reset, and powerdown functionality of the CPU.
External Debug Processor Status Register (EDPRSR)	0x314	Contains status information on the reset and powerdown state of the CPU.
Debug Breakpoint Value Register (DBGBVR<n>_EL1)*	0x400 + 16n	Contains address of hardware breakpoint n.
Debug Breakpoint Control Register (DBGBCR<n>_EL1)*	0x408 + 16n	Controls and enables hardware breakpoint n.

Register name	Memory-mapped offset	Register description
Debug Watchpoint Value Register (DBGWVR<n>_EL1)**	0x800 + 16n	Contains address of watchpoint n.
Debug Watchpoint Control Register (DBGWCR<n>_EL1)**	0x808 + 16n	Controls and enables watchpoint n.

*Each hardware breakpoint uses a pair of registers to control, set, and enable the breakpoint. The number of hardware breakpoints available is IMPLEMENTATION SPECIFIC.

**Each watchpoint has a pair of registers to control, set, and enable the watchpoint. The number of watchpoints available is IMPLEMENTATION SPECIFIC.

To find the offsets of other Debug registers in an Armv8-A system, read the External Debug Register Descriptions section of the [Arm Architecture Reference Manual Armv8-A](#). These offsets are relative to the address of the debug memory system of the core. The core debug memory system address is found in the target TRM. For example, calculate the address of the EDSCR register for a core with the following information:

- Base address of the target debug and trace system is 0x80000000.
- Cluster offset is 0x02000000.
- Individual core debug region offset is 0x00010000.
- EDSCR at offset 0x088.

Add the previous information together

0x80000000 + 0x02000000 + 0x00010000 + 0x088

to get an EDSCR address of 0x82010088.

Useful CTI registers

To halt or restart the core, use one or more target CTIs. The following are useful CTI registers:

Register name	Memory-mapped offset	Register description
CTI Control Register (CTICONTROL)	0x000	Used to enable or disable CTIs.
CTI Channel Gate Enable Register (CTIGATE)	0x140	Controls whether the internal channels are connected to the CTM. Internal channels allow channel events on specific cores to propagate to other components.
CTI Input Channel to Output Trigger Enable Registers (CTIOUTEN<n>#)	0x0a0 + 4n	Connects input channels to output trigger n. This connection allows channel events on these channels to generate trigger events on output trigger n. The number of output triggers available is IMPLEMENTATION SPECIFIC.
CTI Input Trigger to Output Channel Enable Registers (CTIINEN<n>#)	0x020 + 4n	Connects input trigger n to output channels. This connection allows input trigger n to generate channel events on the connected channels. The number of input triggers available is IMPLEMENTATION SPECIFIC.
CTI Application Pulse Register (CTIAPPULSE)	0x01c	Can generate channel events on a specific channel. Write-only register.

Register name	Memory-mapped offset	Register description
CTI Output Trigger Acknowledge Register (CTIINTACK)	0x010	Can create soft acknowledgements of output triggers. Can deassert a trigger event by writing 1 to bit[0]. Write-only register.
CTI Trigger Out Status Register (CTITRIGOUTSTATUS)	0x134	Gives the status of the trigger outputs. To confirm an output trigger has been deasserted, the debugger must poll bit[0] until it reads as 0. Bit[0] must read as 0 before attempting to generate another trigger event. Read-only register.

To find the offsets of other CTI registers in an Armv8-A system, read the External Debug Register Descriptions section of the [Arm Architecture Reference Manual Armv8-A](#). These offsets are relative to the address of the CTI region for a core. The CTI addresses are available in the CoreSight debug and trace section of the target TRM.

For more detailed CTI register descriptions for an Armv8-A system, read the [Arm Architecture Reference Manual Armv8-A](#).

3. Worked Examples

This section focuses on using CSAT to perform low-level debug activities. The following activities are performed:

- [Setting up CSAT to use on your target](#)
- [Reading the ROM Table](#)
- [Halting a single core](#)
- [Halting multiple cores](#)
- [Restarting a single core](#)
- [Restarting multiple cores](#)
- [Setting hardware breakpoints](#)
- [Setting watchpoints](#)

Also, this section covers [Alternatives to using CSAT](#) and instructions on how to [Automate the debug activities](#) performed in this section.

Description of the example target and tools

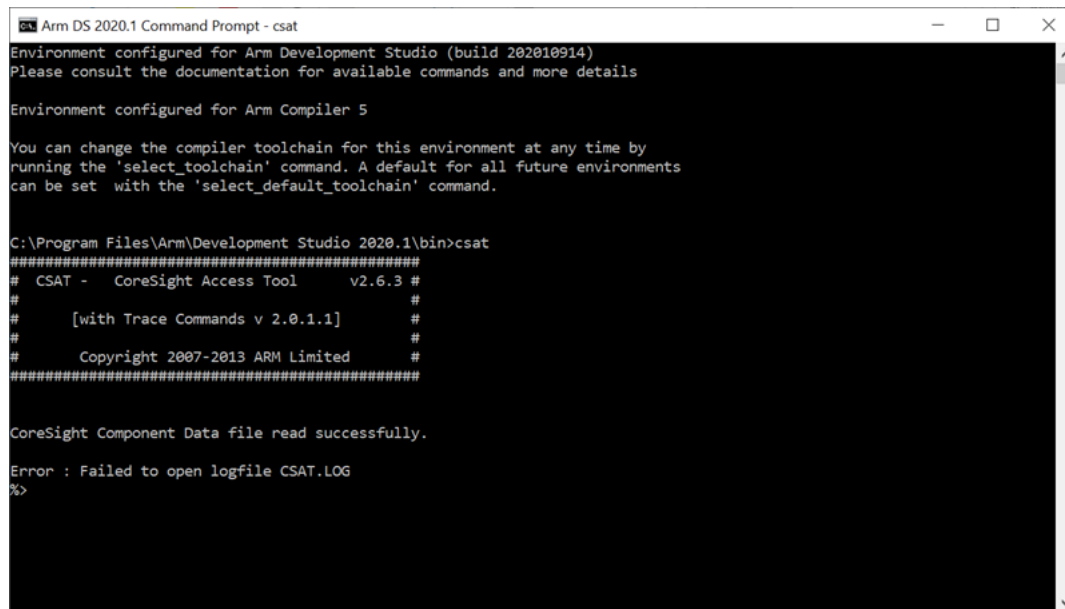
This tutorial uses a Versatile Express with a Cortex-A53x2 SMM implemented on a LogicTile Express 20MG V2F-1XV7 FPGA board. CSAT shipped with Arm Development Studio is used. CSAT is connected to the target using a [DSTREAM](#) debug probe.

Setting up CSAT to use on your target

With the following steps you can set up the CSAT to use on your target:

1. Turn on the target. Ensure the target is in a stable state that allows a bare-metal debug connection.
2. Connect DSTREAM to the host through USB or TCP.
3. Connect the DSTREAM to the target and check the TARGET LED is on.
4. Open an Arm Development Studio command prompt and type csat.

```
csat
```

Figure 3-1: Opening CSAT in a command-prompt


```

Arm DS 2020.1 Command Prompt - csat
Environment configured for Arm Development Studio (build 202010914)
Please consult the documentation for available commands and more details

Environment configured for Arm Compiler 5

You can change the compiler toolchain for this environment at any time by
running the 'select_toolchain' command. A default for all future environments
can be set with the 'select_default_toolchain' command.

C:\Program Files\Arm\Development Studio 2020.1\bin>csat
#####
# CSAT - CoreSight Access Tool v2.6.3 #
# [with Trace Commands v 2.0.1.1] #
# Copyright 2007-2013 ARM Limited #
#####

CoreSight Component Data file read successfully.

Error : Failed to open logfile CSAT.LOG
%>

```

5. Connect your DSTREAM to CSAT. If using a USB DSTREAM connection:

```
con USB
```

If using a TCP DSTREAM connection:

```
con TCP:<IP address or host name of your target>
```

6. Auto detect the scan chain of your device.

```
chain dev=auto
```

7. Open the DAP on the scan chain. The device number used is taken from the output of chain dev=auto.

```
dvo <device number>
```

On the example target, the DAP is device number 0.

```
dvo 0
```

8. Enumerate the access ports available on the DAP.

```
dpe
```

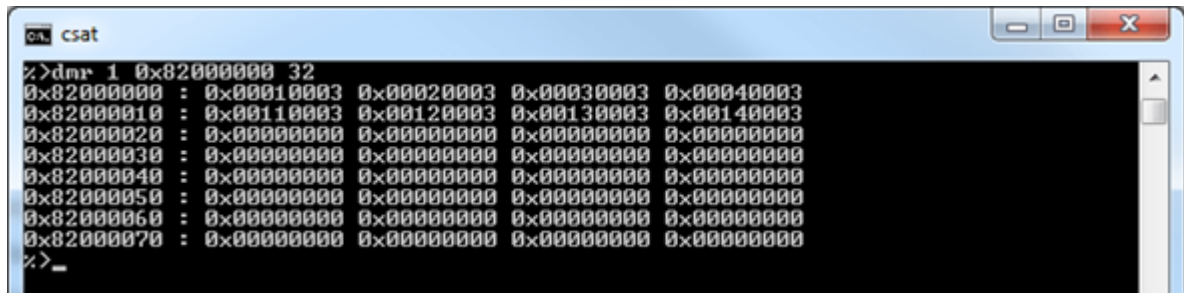
Reading the ROM Table

Following these steps, you can read the ROM table:

1. Locate the base address of the ROM Table. Look at the memory map in your target TRM. Alternatively, read the MEM-AP register BASE at offset 0xF8 for the MEM-AP.
2. After finding the base address of the ROM Table, read the ROM Table using the `dmr` command. As part of the command, specify the access port number using the output from the `dpe` command. On the example target, use the APB-AP access port 1 (AP1). Also, specify the number of ROM Table entries to read. The following `dmr` command reads 32 words from APB-AP 1 at address 0x82000000:

```
dmr 1 0x82000000 32
```

Figure 3-2: ROM table contents



The command output shows the offsets of the available components relative to the base address of the ROM Table. Notice that bits[1:0] of all the ROM Table entries are set to 1. This value indicates that the component is present in the system and the ROM Table entries are 32-bits wide. Bits[1:0] are ignored when calculating the component address.

To calculate a component address, add the ROM Table base address to the component offset. For example, a ROM Table entry at 0x00010003 means there is a component at address.

$$0x82000000 + 0x00010000 = 0x82010000$$

For information on reading ROM Table entries, look at the component TRM.

For the example target, the base addresses of the components are:

- Core 0 debug region - 0x82010000
- Core 0 CTI region - 0x82020000
- Core 1 debug region - 0x82110000
- Core 1 CTI region - 0x82120000

Halting a single core

To halt a core, use the CTI to trigger debug request events. There are multiple channels available for a CTI. In this tutorial, we use channel 2 to generate a channel event. This channel event generates a debug request trigger on output trigger 0. On a Cortex-A53 core, output trigger 0

causes the core to enter Debug state, halting the core. The channel and output trigger might be different on your target.

1. Unlock the OS Lock by writing 0 to bit[0] of OSLAR.

```
dmw 1 0x82010300 0x0
```

2. To see the status of the processor, read EDPRSR.

```
dmr 1 0x82010314 1
```

3. To enable the CTI, write 1 to bit[0] of CTICONTROL.

```
dmw 1 0x82020000 0x1
```

4. Write 0 to bit[2] of CTIGATE so the channel event is not passed on internal channel 0 to the CTM.

```
dmw 1 0x82020140 0x0
```

5. To generate a debug request trigger event using trigger 0 when a channel event happens on channel 2, write 1 to bit[2] of CTIOUTENO.

```
dmw 1 0x820200a0 0x4
```

6. To generate a channel event on channel 2, write 1 to bit[2] of CTIAPPULSE.

```
dmw 1 0x8202001c 0x4
```

At this point, the core halts.

7. To check that the core is halted, read EDPRSR. Notice that bit[4] is now set. If this bit is set, the core has halted.

```
dmr 1 0x82010314 1
```

Halting multiple cores

To halt all cores, repeat the single core halt steps for every core you want to halt. Also, use the CTIGATE register to connect channel 2 of every core you want to halt to the CTM. This connection transmits a channel event from core 0 to all the other cores connected through channel 2 VIA the CTM.

For the example target, using the CTIs and CTM, execute the following CSAT commands to halt core 0 and core 1:

```
#Unlock OS Lock
dmw 1 0x82010300 0x0
dmw 1 0x82110300 0x0
```

```
#Read the unhalting value of the External Debug Processor Status Register
dmr 1 0x82010314 1
dmr 1 0x82110314 1

#Set CTICONTROL[0] = 1 to enable CTI
dmw 1 0x82020000 0x1
dmw 1 0x82120000 0x1

#Set CTIGATE[2] = 1 so CTI passes channel events on internal channel 2 to
#CTM
dmw 1 0x82020140 0x4
dmw 1 0x82120140 0x4

#Set CTIOUTEN0[2] = 1 so CTI generates a debug request trigger event 0 in
#response to a channel event on channel 2
dmw 1 0x820200a0 0x4
dmw 1 0x821200a0 0x4

#Set CTIAPPULSE[2] = 1 to generate channel event on channel 2 on core 0
dmw 1 0x8202001c 0x4

#Read the halted value of the External Debug Processor Status Register
dmr 1 0x82010314 1
dmr 1 0x82110314 1
```

Restarting a single core

To restart the core, use a similar process to halting the core. For the example target, use channel 1 to generate a channel event. This channel event causes a debug request trigger on output trigger 1. On a Cortex-A53 core, output trigger 1 causes the processor to exit Debug state, restarting the core. The channel and output trigger might be different on your target.

1. Unlock the OS Lock by writing 0 to bit[0] of OSLAR.

```
dmw 1 0x82010300 0x0 to see the status of the processor
```

2. To see the status of the processor, read EDPRSR. Bit[4] is 1 indicating that the core has halted.

```
dmr 1 0x82010314 1
```

3. To enable the CTI, write 1 to bit[0] of CTICONTROL.

```
dmw 1 0x82020000 0x1
```

4. To clear any debug request trigger events that halted the core, write 1 to bit[0] of CTIINTACK. Then, read CTITRIGOUTSTATUS until bit[0] is 0 to confirm that the output trigger event has been deasserted.

```
dmw 1 0x82020010 0x1
```

```
dmr 1 0x82020134 1
```

- Write 0 to bit[1] of CTIGATE so the channel event is not passed on internal channel 1 to the CTM.

```
dmw 1 0x82020140 0x0
```

- To generate a restart request trigger event using trigger 1 when a channel event happens on channel 1, write 1 to bit[1] of CTIOUTEN1.

```
dmw 1 0x820200a4 0x2
```

- To generate a channel event, write 1 to bit[1] of CTIAPPPULSE.

```
dmw 1 0x8202001c 0x2
```

At this point, the core restarts.

- To check that the core has restarted, read EDPRSR. Bit[4] is 0 indicating that the processor has restarted.

```
dmr 1 0x82010314 1
```

Restarting multiple cores

To restart all cores, repeat the single core restart steps for every core you want to restart. Use the internal channel 1 to connect the cores to the CTM. This connection transmits a channel event from core 0 to all the other cores connected through channel 1 through the CTM.

For the example target, using the CTIs and CTM, execute the following CSAT commands to restart core 0 and core 1:

```
#Unlock OS Lock
dmw 1 0x82010300 0x0
dmw 1 0x82110300 0x0

#Read the halted value of the External Debug Processor Status Register
dmr 1 0x82010314 1
dmr 1 0x82110314 1

#Set CTICONTROL[0] = 1 to enable CTI
dmw 1 0x82020000 0x1
dmw 1 0x82120000 0x1

#Set CTIINTACK[0] = 1 to clear the debug request trigger event
dmw 1 0x82020010 0x1
dmw 1 0x82120010 0x1

#Read CTITRIGOUTSTATUS[0] to check trigger event is deasserted.
dmr 1 0x82020134 1
dmr 1 0x82120134 1

#Set CTIGATE[1] = 1 so CTI passes channel events on internal channel 1 to
#CTM
dmw 1 0x82020140 0x2
dmw 1 0x82120140 0x2
```



```
#Set CTIOUTEN1[1] = 1 so CTI generates a debug request trigger event 0 in
#response to a channel event on channel 1
dmw 1 0x820200a4 0x2
dmw 1 0x821200a4 0x2

#Set CTIAPPULSE[1] = 1 to generate channel event on channel 1 on core 0
dmw 1 0x8202001c 0x2

#Read the unhalting value of the External Debug Processor Status Register
dmr 1 0x82010314 1
dmr 1 0x82110314 1
```

Setting hardware breakpoints

A hardware breakpoint halts core execution at a particular instruction in the code. There is a limited number of hardware breakpoints available on each core. Each breakpoint has a value and control register. The value register contains the address of the instruction to break on. The control register defines the breakpoint type and whether the breakpoint is enabled.

Execute the following to set up hardware breakpoint 0:

1. Halt the core.
2. To check if the core is halted, read the EDPRSR. If bit[4] is 1, the core is halted.

```
dmr 1 0x82010314 1
```

3. If bit[14] of the EDSCR is not set, write 1 to it to enable Halting debug. If bit[14] is 1, skip this step.

```
dmw 1 0x82010088 0x03007f13
```

4. Write the address of the instruction you want to halt on to DBGBVR0_EL1. DBGBVR0_EL1 is a 64-bit register. On a Cortex-A53, DBGBVR0_ELO uses two register offsets, 0x400 and 0x404. Offset 0x400 contains bits[31:0] of the breakpoint address. Offset 0x404 contains bits[63:32] of the breakpoint address. The target example halts on instruction address 0x00000000_80000008.

```
dmw 1 0x82010400 0x80000008
```

```
dmw 1 0x82010404 0x0
```

5. Write 0x000021e7 to DBGBCR0_EL1 to set a basic breakpoint with the following attributes:
 - Enabled breakpoint
 - Unlinked address match
 - Execution halts at any Exception level
 - Works for both AArch64 and AArch32 instructions.

- For more information on the different breakpoint attributes, look at the DBGBCR<n>_EL1 register description in the [Arm Architecture Reference Manual Armv8-A](#).

```
dmw 1 0x82010408 0x000021e7
```

- Restart the core. Halt occurs on the address you specified.
- To check the program has halted on the breakpoint, read EDPRSR. Bit[4] is 1 indicating that the core has stopped.

```
dmr 1 0x82010314 1
```

- To get the debug status and the reason for core halt, read the STATUS bits[5:0] of EDSCR. If the core has stopped on a breakpoint, the STATUS bits read as 0b000111. To check that execution has stopped on the correct address, read EDPCSRlo and EDPCSRhi and see if their values match the breakpoint address.

```
dmr 1 0x82010088 1
```

```
dmr 1 0x820100a0 1
```

```
dmr 1 0x820100ac 1
```

- To disable a breakpoint without losing the breakpoint settings, write 0 to bit[0] of DBGBCR0_EL1.

```
dmw 1 0x82010408 0x000021e6
```

- To delete the breakpoint, clear DBGBCR0_EL1 and DBGVRO_EL1.

```
dmw 1 0x82010400 0x0
```

```
dmw 1 0x82010404 0x0
```

```
dmw 1 0x82010408 0x0
```

Setting watchpoints

A watchpoint halts execution when a particular value in memory is accessed. Unlike a breakpoint, for a watchpoint, you only must know the memory address of the data. Watchpoints are sometimes called data breakpoints. There are a limited number of watchpoints available on each core. Each watchpoint has a value register and a control register. The value register contains the address value for comparison. The control register defines the watchpoint type and whether the watchpoint is enabled.

Executed the following to set up watchpoint 0:

1. Halt the core.
2. To check if the core is halted, read EDPRSR.

```
dmr 1 0x82010314 1
```

3. If bit[14] of the EDSCR is not set, write 1 to it to enable Halting debug. If bit[14] is 1, skip this step.

```
dmw 1 0x82010088 0x03007f13
```

4. Write the address of the data to halt on when accessed to DBGWVRO_EL1. DBGWVRO_EL1 is a 64-bit register. On a Cortex-A53, DBGWVRO_EL1 uses two register offsets, 0x800 and 0x804. Offset 0x800 contains bits[31:0] of the watchpoint address. Offset 0x804 contains bits[63:32] of the watchpoint address. The target example halts on data address 0x00000000_80000100.

```
dmw 1 0x82010800 0x80000100
```

```
dmw 1 0x82010804 0x0
```

5. Write 0x00003fff to the DBGWCR0_EL1 to set a basic watchpoint for testing:
 - Enabled watchpoint
 - Unlinked data address
 - Execution halts at any Exception level
 - Applies to both loads and stores to the memory address.
 - For more information on the different watchpoint attributes, look at the DBGWCR<n>_EL1 register description in the [Arm Architecture Reference Manual Armv8-A](#).

```
dmw 1 0x82010808 0x00003fff
```

6. Restart the core. Halt occurs on a load or store to the watchpoint address.
7. To check the program has halted on the watchpoint, read EDPRSR. Bit[4] is set to 1 indicating that the core has halted.

```
dmr 1 0x82010314 1
```

8. To get the debug status and the reason for the core halt, read the STATUS bits[5:0] of EDSCR. If the core has stopped on a watchpoint, the STATUS bits read as 0b101011. To check where execution halted, read EDPCSRlo and EDPCSRhi.

```
dmr 1 0x82010088 1
```

```
dmr 1 0x820100a0 1
```

```
dmr 1 0x820100ac 1
```

9. To disable a watchpoint without losing the watchpoint settings, write 0 to bit[0] of DBGWCR0_EL1.

```
dmw 1 0x82010808 0x00003ffe
```

10. To delete the watchpoint, clear DBGWCR0_EL1 and DBGWVR0_EL1.

```
dmw 1 0x82010800 0x0
```

```
dmw 1 0x82010804 0x0
```

```
dmw 1 0x82010808 0x0
```

Alternatives to using CSAT

Low-level debug is also done in:

- Simulation
- With a debugger like Arm DS

If using Arm DS, low-level debug is done by creating scripts that access debug functionality or doing Debug register accesses through the Memory view.

To learn more about accessing debug functionality, read the external debug sections of the [Arm Architecture Reference Manual Armv8-A](#).

Automate the debug activities

To run the previous examples using a script, do the following steps:

1. Create a file with a .cst files extension in the CSAT executable directory.
2. Copy the example commands to the .cst file.
3. Run the scripts using

```
batch <script name>.cst
```

Attached to this tutorial are two script directories:

- [CSAT Scripts](#)
 - Scripts containing CSAT commands from this tutorial.
 - Run these scripts using the previous steps.
- [Arm DS scripts](#)
 - Arm DS scripts that do the same actions as the previous CSAT scripts.
 - Run the scripts using information from the Running a script section of the [Arm Development Studio User Guide](#).

4. Conclusion

In this tutorial, you learned how to use CSAT with an Armv8 target to:

- Halt one or more cores.
- Restart one or more cores.
- Set a breakpoint.
- Set a watchpoint.

5. Further Information

Read the following documents for further information on topics related to this tutorial:

- [Arm Architecture Reference Manual Armv8-A](#)
 - Architecture reference manual for Armv8-A implementations.
- [Arm CoreSight Architecture Specification](#)
 - Specification containing information on the architecture of CoreSight systems.
- [Arm CoreSight SoC-400 Technical Reference Manual](#)
 - CoreSight TRM containing information on CoreSight Components.
- [Arm Development Studio User Guide](#)
 - Arm DS user guide containing information on running and using Arm DS.
- [CSAT User Guide](#)
 - User guide containing CSAT usage instructions, examples, and commands.
- [Understanding the CoreSight DAP](#)
 - Guide containing information on DAPs, APs, and ROM Tables.
- [Understanding trace](#)
 - Guide containing more information on how trace works in Arm systems.
- [What is the difference between CSAT and CSAT600?](#)
 - KBA that shows the differences between CSAT and CSAT600.