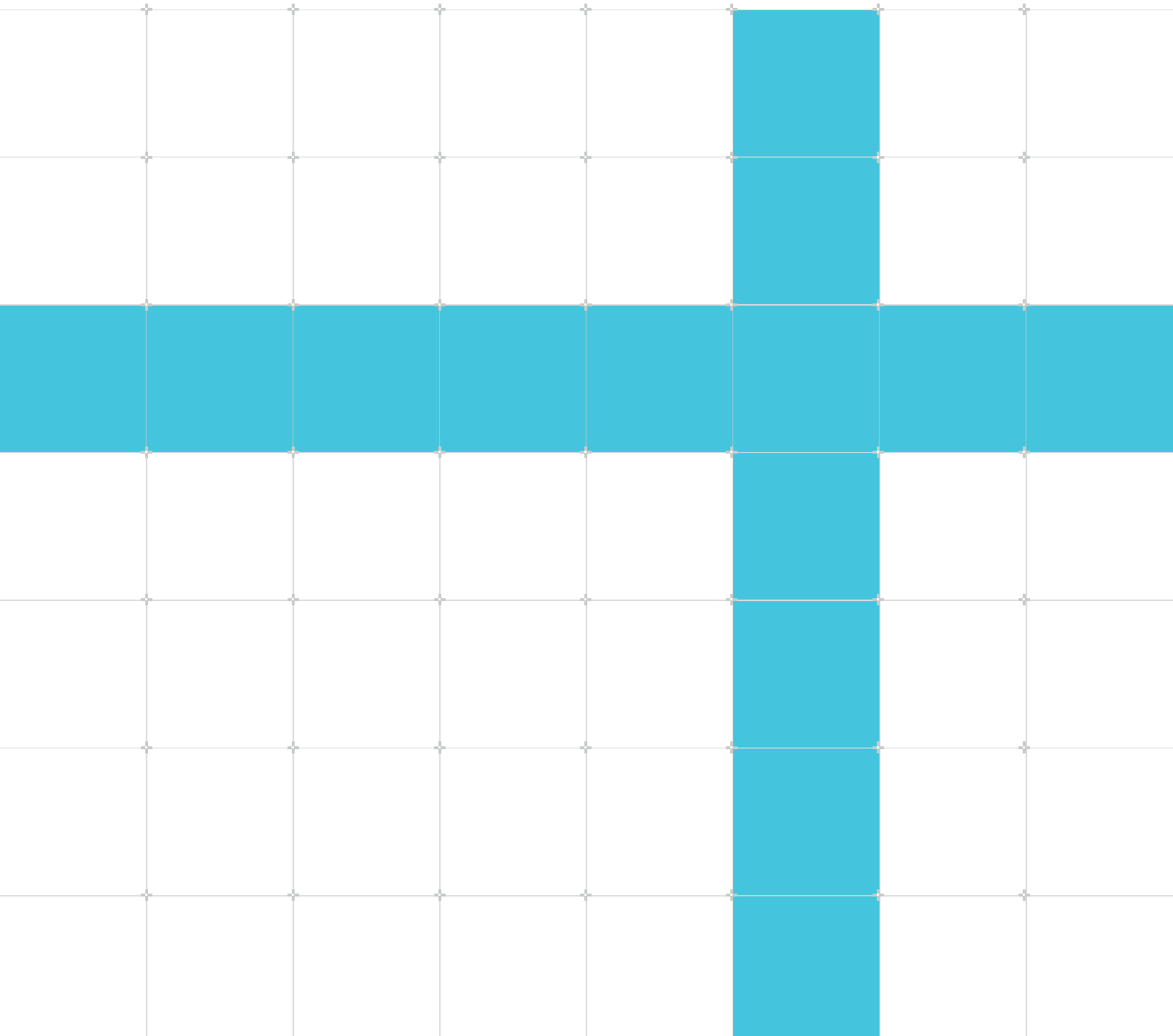
Low-level debug using CSAT on Armv8-based platforms

Version 1.0

Non-Confidential

Issue 02
102706_0100_02_en



Low-level debug using CSAT on Armv8-based platforms

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	1 January 2021	Non-Confidential	First release

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 tutorial will give 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, and perform functions such as halting and restarting the core, setting breakpoints and watchpoints and reading the ROM table.

CSAT includes something called a CoreSight component to provide a convenient way to access and control particular CoreSight devices on the debug bus.We have going to look at CSAT CoreSight components, in particular, the `v7dbg` and `v8cs` CoreSight components in this tutorial.

Below is a listing of the different CSAT CoreSight components, which CoreSight devices you use the different CSAT CoreSight components with, the architecture of the system or core the CSAT component is appropriate for, whether you can use the CSAT CoreSight component to access the CoreSight devices registers, and whether you can use the CSAT CoreSight component to access memory via the CoreSight device.

2. Armv8 topology

This chapter gives a brief description about the CoreSight ROM table and the CoreSight components.

What is a CoreSight ROM table?

A CoreSight ROM (read-only memory) table is a section of memory that stores the locations of all the debug components available. The values listed in the ROM table can be read as consecutive entries, starting at the base address of the ROM table, which can be found on the memory map in the target's technical reference manual (TRM). Alternatively, the base address can be found by reading the MEM-AP register BASE at offset `0xf8` on the access port you are using.

The position of the debug components as seen by the processor, listed on the memory map, is often different from the position seen by the external debugger. The external debugger is used to access the debug registers to bypass locks that could restrict software. For example, on Arm's Cortex-A53x2 SMM, the target's TRM lists the debug component at address `0x20000000`, but using the external debugger, the registers are accessed at `0x80000000`. For more information, look at the 'Memory system design' section of the ARM CoreSight Architecture Specification.

The ROM table is located at the beginning of the debug region and occupies 4KB of memory. For targets with more than one cluster of processors, there will often be a ROM table for each cluster as well as a ROM table that covers the entire debug region.

What are CoreSight components?

CoreSight components are used to provide the debug and trace infrastructure of a SoC (System-on-Chip). The components include the control and access components such as the DAP and the CTI, trace sources, links and sinks.

A Debug Access Port (DAP) is a collection of components used to access the on-chip debug and trace tools. The DAP includes debug ports, to access the DAP from the external debugger, and access ports, to access the on-chip system resources.

An Access Port (AP) links the debug port interface with one or more of the debug components present within the system. In this tutorial, we will be using the APB access port, a memory access port, which can be used to access the memory-mapped debug registers we will be reading from and writing to.

The Cross Trigger Interface (CTI) is a CoreSight component that allows other CoreSight components to interact with each other.

The CTI has input triggers that respond to internal changes in the core, such as a core restarting or an interrupt request from the PMU, and output triggers that can cause changes such as halting or restarting the core. These triggers can be connected to channels, which can be used to connect triggers on multiple cores' CTIs together via the Cross Trigger Matrix (CTM). An input trigger can generate a channel event on the channel or channels that it is connected to, or the user can generate their own channel event. This channel event then triggers the output trigger event. If

more than one core is connected to the channel, then one core halting can be used to trigger a synchronous halt of all the connected cores.

To find out the specific functions of each trigger, look in the 'Trigger inputs and outputs' section in the 'Cross Trigger' chapter of the implementation's TRM. It should be noted that in Armv8, the CTI is now part of the architecture specification as opposed to an optional extra. For an Armv8 core, the CTI must be used to halt or restart the core.

3. Useful debug registers

To perform low level debug using CSAT, you need to read and write registers that are accessible via the external memory mapped interface. Some of the useful registers are:

- EDSCR - External Debug Status and Control Register - offset `0x088` This register is the main control register for an Armv8 core.
- OSLAR_EL1 - OS Lock Access Register - offset `0x300` This register controls the OS Lock. Unlocking the OS Lock allows you to access the debug registers. This is a write only register.
- EDPCSRlo - External Debug Program Counter Sample Register[31:0] and EDPCSRhi - External Debug Program Counter Sample Register [63:32] - offsets `0x0a0` and `0x0ac` These optional registers allow you to read the program counter whilst externally debugging. These are read only registers. As these registers are optional, consult the implementation's TRM to see if they are implemented.
- EDPRCR - External Debug Power/Reset Control Register - offset `0x310` This register controls the powerup, reset and powerdown functionality of the CPU.
- EDPRSR - External Debug Processor Status Register - offset `0x314` This register holds information regarding the reset and powerdown state of the CPU.

Each hardware breakpoint has a pair of registers that you must use. The number of hardware breakpoints available is implementation specific.

- DBGBVR<n>_EL1 - Debug Breakpoint Value Register - offset `0x400 + 16n` This register holds the address of hardware breakpoint `n`.
- DBGBCR<n>_EL1 - Debug Breakpoint Control Register - offset `0x408 + 16n` This register is used to control hardware breakpoint `n`. This includes enabling the hardware breakpoint.

Similarly, each watchpoint has a pair of registers that you must use. The number of watchpoints available is implementation specific.

- DBGWVR<n>_EL1 - Debug Watchpoint Value Register - offset `0x800 + 16n` This register holds the address of watchpoint `n`.
- DBGWCR<n>_EL1 - Debug Watchpoint Control Register - offset `0x808 + 16n` This register is used to control watchpoint `n`. This includes enabling the watchpoint.

The offsets of other debug registers can be found in the 'External Debug Register Descriptions' section of the Arm Architecture Reference Manual for Armv8-A. These offsets are relative to the address of the debug region of the core, which can be found in the target's TRM.

For example, if the base address of the target's debug and trace region is `0x80000000`, the cluster's offset is `0x02000000`, the individual core's debug region offset is `0x00010000` and the EDSCR has an offset of `0x088`, the address of the EDSCR of that core is `0x80000000 + 0x02000000 + 0x00010000 + 0x088 = 0x82010088`.

4. Useful CTI registers

To halt/restart the core, we need to use the CTI. Useful registers for CTI include:

- CTICONTROL - CTI Control Register - offset `0x000` This register is used to enable or disable CTI.
- CTIGATE - CTI Channel Gate Enable Register - offset `0x140` This register controls whether the internal channels are connected to the CTM, allowing channel events on specific cores to propagate to other components.
- CTIOUTEN<n> - CTI Input Channel to Output Trigger Enable Registers - offset `0x0a0 + 4n` These registers connect input channels to output trigger `n`, allowing channel events on these channels to generate trigger events on output trigger `n`. The number of output triggers available is implementation specific.
- CTIINEN<n> - CTI Input Trigger to Output Channel Enable Registers - offset `0x020 + 4n` These registers connect input trigger `n` to output channels, allowing input trigger `n` to generate channel events on the connected channels. The number of input triggers available is implementation specific.
- CTIAPPPULSE - CTI Application Pulse Register - offset `0x01c` This register can be used to generate channel events on a specific channel. This is a write only register.
- CTIINTACK - CTI Output Trigger Acknowledge Register - offset `0x010` This register can be used to create soft acknowledges of output triggers. It can be used to deassert a trigger event by writing 1 to bit 0. This is a write only register.
- CTITRIGOUTSTATUS - CTI Trigger Out Status Register - offset `0x134` This register gives the status of the trigger outputs. To confirm an output trigger has been deasserted, the debugger must poll bit 0 of this register until it reads 0. This must be done before attempting to generate another trigger event. This is a read only register.

The offsets of other CTI registers can be found in the ‘External Debug Register Descriptions’ section of the Arm Architecture Reference Manual for Armv8-A. These offsets are relative to the address of the CTI region of the core, which can be found in the ‘CoreSight debug and trace’ section of the target’s TRM.

For a more detailed register descriptions, have a look at the Arm Architecture Reference Manual for Armv8-A.

5. Worked examples

In this chapter you can find worked examlpes for setting up CSAT to use on your target, reading the ROM table, halting multiple cores, restarting a single core, restarting multiple cores, setting hardware breakpoints, setting watchpoints or automating the process.

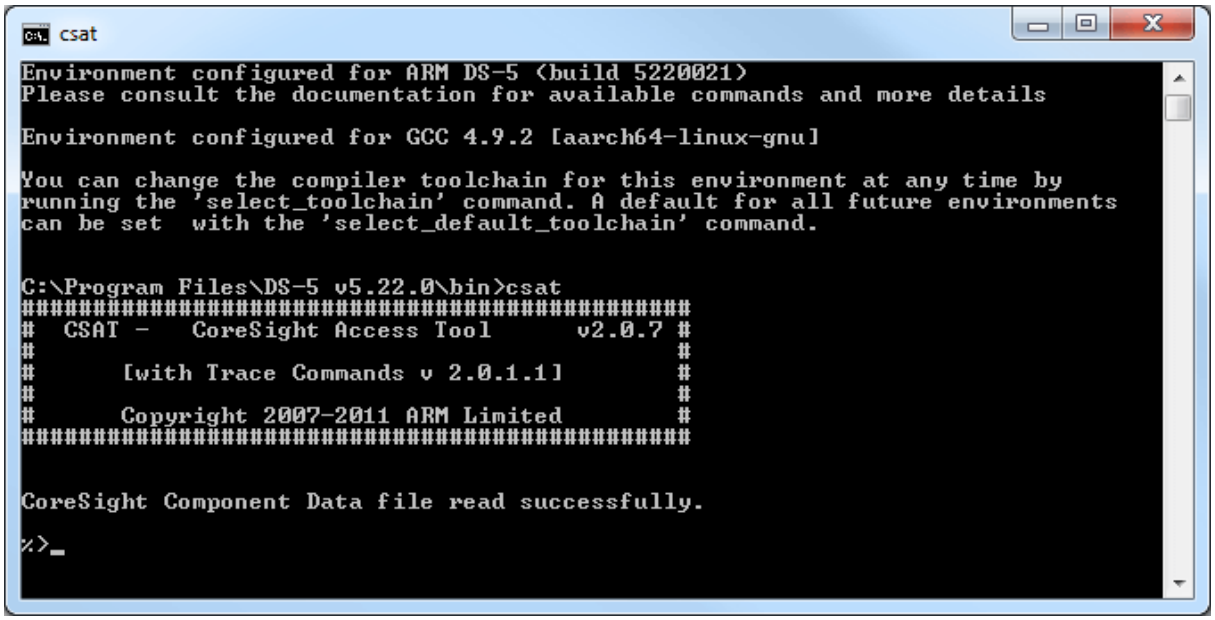
Description of the example target

This tutorial was completed on a Versatile Express with a Cortex-A53x2 SMM implemented on a LogicTile Express 20MG V2F-1XV7 FPGA board.

Setting up CSAT to use on your target

- Turn on the target and make sure it is in a state which can be connected to using a bare metal connection.
- Connect DSTREAM to the host via USB or TCP.
- Connect the DSTREAM to the target and check TARGET indicator light is on.
- Open a DS-5 command prompt and type CSAT.

```
csat
```



- Connect your DSTREAM to CSAT.

```
con USB (if connected by USB) or con TCP:<IP address or host name of your target>
 (if connected by TCP)
```

- Auto detect the scan chain of your device.

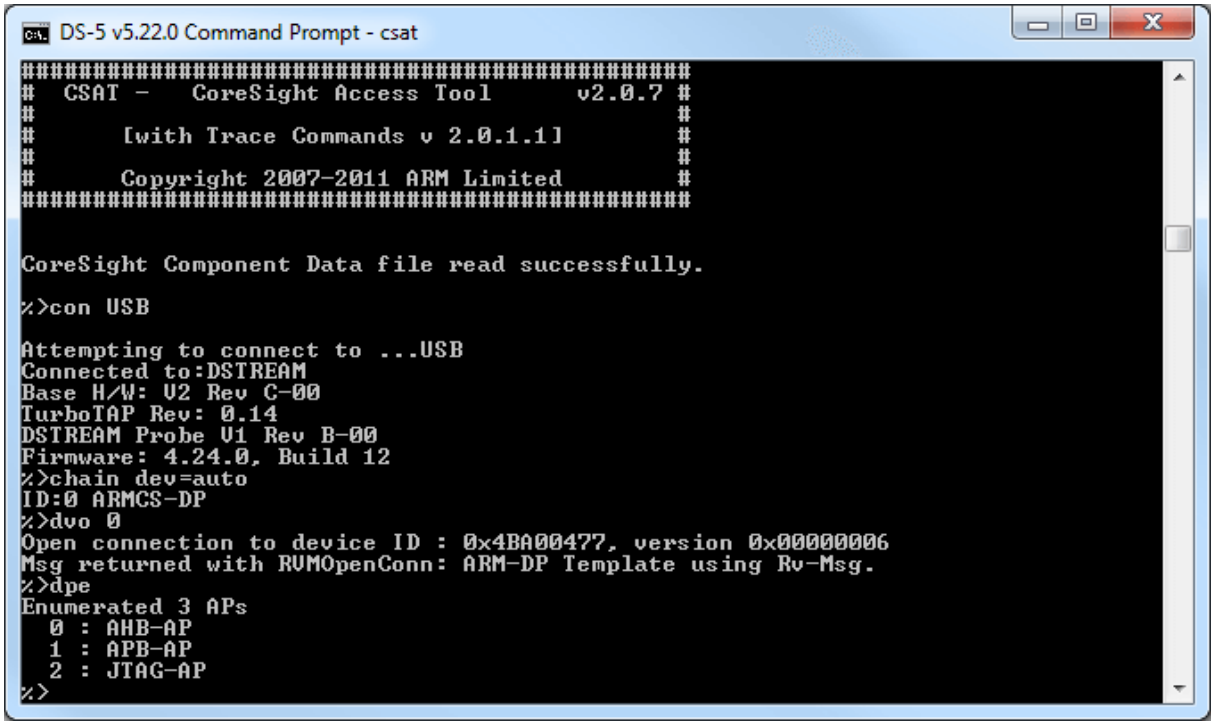
```
chain dev=auto
```

- You must now open a device on the scan chain. You can only have one device open at a time. For the example target, the DAP was device 0, but may be another number, for example if your target has secure JTAG mode enabled.

```
dvo <device number>
```

- Enumerate the access ports available on the Debug Access Ports (DAPs).

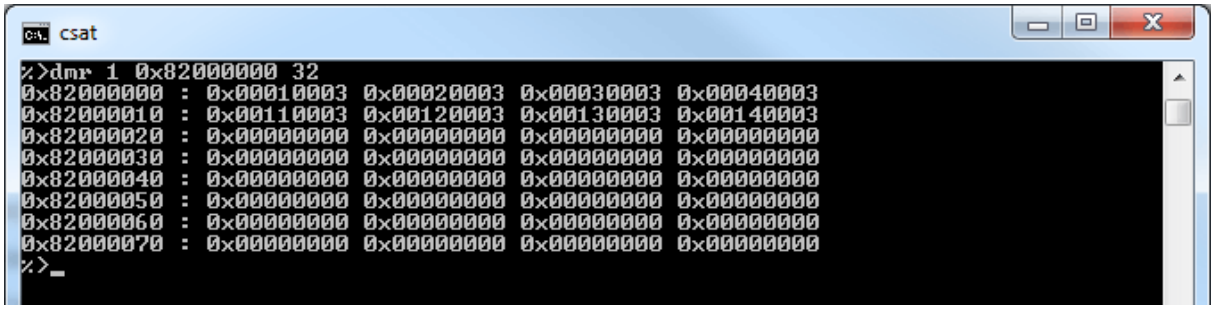
```
dpe
```



Reading the ROM table

- Locate the base address of the ROM table you wish to read. This is listed on the memory map in the target's TRM or can be found by reading the MEM-AP register BASE at offset `0xf8` on the access port you are using.
- Once you have the base address of the ROM table, you can read it using the `dmr` command. You will need to specify the number of entries you want to read. ROM tables can be quite big so start with a large value like 32. All the consecutive non-zero entries above the base address are ROM table entries. You will also need to specify the access port number from the available access ports listed when you used the `dpe` command. In this case, we will use the APB access port, AP1.

```
dmr 1 0x82000000 32
```



The values printed are the offsets of the available components relative to the base address of the ROM table. You will notice that the bottom two bits of all the ROM table entries are set to 1. On a Cortex-A53, this indicates that the component is present and the ROM table entry is in 32-bit format. Therefore, in this instance, the offset of the component is only represented by the top 20 bits (bits 11 to 2 are reserved). For example, a ROM table entry `0x00010003` read using the command `dmr 1 0x82000000 32` means that there is a component at address `0x82000000` + `0x00010000` = `0x82010000`. For information on reading the ROM table entries, look at the implementation's TRM.

For the following examples, 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 we use the CTI to trigger debug request events. There are multiple channels available for CTI. In this tutorial, we will use channel 2 to generate a channel event that generates a debug request trigger on output trigger 0. On a Cortex-A53 core, output trigger 0 is the trigger that causes the processor to enter debug state, halting the core.

- First, unlock the OS Lock by writing 0 to bit 0 of the OSLAR.

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

- Read the value of the EDPRSR to see the status of the processor.

```
dmr 1 0x82010314 1
```

- Write 1 to bit 0 of the CTICONTROL register to enable CTI.

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

- Write 0 to bit 2 of the CTIGATE register so the channel event is not passed on internal channel 0 to the CTM.

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

- Write 1 to bit 2 of the CTIOUTEN0 register to generate a debug request trigger event using trigger 0 when a channel event happens on channel 2.

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

- Write 1 to bit 2 of the CTIAPPPULSE register to generate a channel event on channel 2.

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

At this point, the core should halt.

- To check that the core is halted, read the EDPRSR again. You will notice that bit 4 is now set. This indicates that the core has halted.

```
dmr 1 0x82010314 1
```

Halting multiple cores

The procedure for halting all cores is similar to halting a single core, but the setup steps must be replicated on every core you wish to halt. You also need to use the CTIGATE register to connect channel 2 of every core you wish to halt to the CTM so that when core 0 generates a channel event on channel 2, it can propagate via the CTM to all the other cores connected to channel 2. This may look something like this:

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

#Read the unhalted 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 CTIAPPPULSE[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, the process is similar to halting the core. This time, we will use channel 1 to generate a channel event that causes a debug request trigger on output trigger 1. On a Cortex-A53 core, output trigger 1 is the trigger that causes the processor to exit debug state, restarting the core.

- Unlock the OS Lock by writing 0 to bit 0 of the OSLAR.

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

- Read the EDPRSR to see the status of the processor. Bit 4 should be set to 1, indicating that the core has halted.

```
dmr 1 0x82010314 1
```

- Write 1 to bit 0 of the CTICONTROL register to enable CTI.

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

- Write 1 to bit 0 of the CTIINTACK register to clear any debug request trigger event that halted the core. Then, read the CTITRIGOUTSTATUS register 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 the CTIGATE register so the channel event is not passed on internal channel 1 to the CTM.

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

- Write 1 to bit 1 of the CTIOUTEN1 register to generate a restart request trigger event using trigger 1 when a channel event happens on channel 1.

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

- Write 1 to bit 1 of the CTIAPPPULSE register to generate a channel event.

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

At this point, the core should restart.

- To check that the core has restarted, read the EDPRSR again. Bit 4 should now equal 0, indicating that the processor has restarted.

```
dmr 1 0x82010314 1
```

Restarting multiple cores

The procedure for restarting multiple cores is similar to restarting only one core, but the setup steps must be replicated on every core you wish to halt. You also need to use the internal channel 1 to connect the cores to the CTM so that when core 0 generates a channel event on channel 1, it can propagate via the CTM to all the other cores connected to channel 1. This may look something like this:

```
#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 CTIAPPPULSE[1] = 1 to generate channel event on channel 1 on core 0
dmw 1 0x8202001c 0x2

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

Setting hardware breakpoints

A hardware breakpoint is used to halt the execution at a particular instruction in the code. There are a limited number of hardware breakpoints available on each core. Each breakpoint has a value register, which holds the address of the instruction to break on, and a control register, which dictates the type of breakpoint, as well as if it is enabled or not.

For this tutorial, we are going to setup hardware breakpoint 0.

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

```
dmr 1 0x82010314 1
```

- If bit 14 of the EDSCR is not already set, write 1 to it to enable halting debug.

```
dmw 1 0x82010088 0x03007f13 (for example)
```

- Write the address of the instruction you wish to break on in your image to the DBGBVR0_EL1. If the address is longer than 32 bits, write bits 32 to 63 to DBGBVR0_EL1[63:32]. If the address is only 32 bits, it is a good idea to write zeros to this register.

```
dmw 1 0x82010400 0x80000008 (for example)
dmw 1 0x82010404 0x0
```

- Write `0x000021e7` to the DBGBCR0_EL1 to set a basic breakpoint: an enabled, unlinked address match breakpoint that will cause the execution to break, at any exception level, on both AArch64 and AArch32 instructions. For more information on different breakpoint variations, look at the **DBGBCR<n>_EL1** register description in the Arm Architecture Reference Manual.

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

- Restart the core and your program should break on the address you have specified.
- To check the program has broken on the breakpoint, you can read the EDPRSR's value. Bit 4 should be set to 1, indicating that the core has stopped.

```
dmr 1 0x82010314 1
```

The STATUS bits[5:0] of the EDSCR give the debug status and can tell you why the core has halted. If the core has stopped on a breakpoint, the STATUS bits should read `0b000111`. You can check that execution has broken on the correct address by reading the program counter registers 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 settings, write 0 to bit 0 of the breakpoint's control register.

```
dmw 1 0x82010408 0x000021e6 (for example)
```

- To delete the breakpoint entirely, simply clear the breakpoint's control and value registers.

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

```
dmw 1 0x82010404 0x0
dmw 1 0x82010408 0x0
```

Setting watchpoints

A watchpoint is used to halt the execution when a particular value in memory is accessed. Unlike a breakpoint, you do not need to know the exact location the data is used within the code, only the address in memory that that data is stored at. Watchpoints are sometimes called data breakpoints. There are a limited number of watchpoints available on each core. Each watchpoint has a value register, which holds the address value for comparison, and a control register, which dictates the type of watchpoint, as well as if it is enabled or not.

For this tutorial, we are going to set watchpoint 0.

- Halt the core. *To check if the core is halted, read the EDPRSR.

```
dmr 1 0x82010314 1
```

- If bit 14 of the EDSCR is not already set, write 1 to it to enable halting debug.

```
dmw 1 0x82010088 0x03007f13 (for example)
```

- Write the address of the memory location you want to 'watch' to the DBGWVR0_EL1. If the address is longer than 32 bits, write bits 32 to 63 to DBGWVR0_EL1[63:32]. If the address is only 32 bits, it is a good idea to write zeros to this register.

```
dmw 1 0x82010800 0x80000100 (for example)
dmw 1 0x82010804 0x0
```

- Write `0x00003fff` to the DBGWCR0_EL1 to set a basic watchpoint for testing: an enabled, unlinked data address match watchpoint that causes the execution to break, at any exception level, on both loads and stores to the memory location being watched. More information on the different watchpoint options available (break on load only, break on store only) can be found in the **DBGWCR<n>_EL1** register description in the Arm Architecture Reference Manual.

```
dmw 1 0x82010808 0x00003fff (for example)
```

- Restart the core and your program should break on a load or store to the address you have specified.
- To check the program has broken on the watchpoint, you can read the EDPRSR's value. Bit 4 should be set to 1, indicating that the core has stopped.

```
dmr 1 0x82010314 1
```

The STATUS bits[5:0] of the EDSCR give the debug status and can tell you why the core has halted. If the core has stopped on a watchpoint, the STATUS bits should read `0b101011`. You can

check where execution has broken in the code by reading the program counter registers EDPCSRlo and EDPCSRhi.

```
dmr 1 0x82010088 1
dmr 1 0x820100a0 1
dmr 1 0x820100ac 1
```

- To disable a watchpoint without losing the settings, write 0 to bit 0 of the watchpoint's control register.

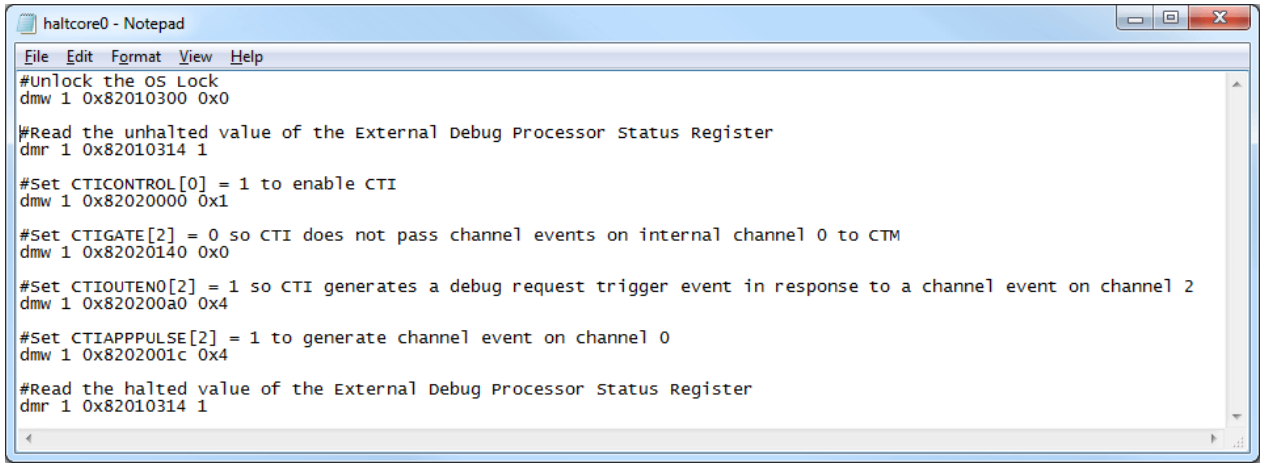
```
dmw 1 0x82010808 0x00003ffe (for example)
```

- To delete the watchpoint entirely, simply clear the watchpoint's control and value registers.

```
dmw 1 0x82010800 0x0
dmw 1 0x82010804 0x0
dmw 1 0x82010808 0x0
```

Automating the process

Any of the examples mentioned above can be achieved by creating `.cst` files listing all the commands and saving them in the directory where you are running CSAT. These scripts can be run using the command `batch <script name>.cst`.



You can download the example .cs scripts for this tutorial, as well as the equivalent DS-5 .ds scripts. For information on running scripts whilst debugging in Eclipse, have a look at the Arm DS-5 Debugger User Guide.

6. Related information

This tutorial has provided an overview of how to use CSAT with an Armv8 target to perform tasks such as halting the core, restarting the core and setting breakpoints and watchpoints.

For more information on CSAT commands, have a look at the CSAT User Guide. To learn more about the CoreSight components, look at the CoreSight SoC-400 TRM.

Low level debug can also be done in simulation or, if DS-5 Debug Configuration files have already been created, through Eclipse for DS-5 using similar scripts, or via the Memory view.

The Arm Architecture Reference Manual for Armv8-A also has a section on external debug. Interesting chapters for this topic include:

- Breakpoints/Watchpoints - Chapter H2
- CTI Examples - Chapter H5
- Register Description - Chapter H9