Low Level Debug using CSAT on Armv7-based platforms

Version 1.0

Non-Confidential

Issue 02
102715_0100_02_en

Low Level Debug using CSAT on Armv7-based platforms

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	4 November 2019	Non-Confidential	First release

Proprietary Notice

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 provides information about performing some basic debug operations on an Armv7-based platform using the CoreSight Access Tool (CSAT).

This tutorial focuses on using the CoreSight Access Tool (CSAT) shipped with DS-5 to perform certain debug operations on an Armv7-A platform.

Users who are focused on platform bring-up of Silicon, FPGA, and hardware emulation environments and wish to test certain aspects of their debug design may find this tutorial useful.

The debug operations the tutorial covers are:

- Halting a Core using Debug Registers
- Restarting a Core using Debug Registers
- Stopping and Restarting Cores using CTIs
- Executing Instructions while the Core is Halted
- Setting a hardware breakpoint
- Setting a watchpoint

The different CSAT commands are gone through step by step to aid understanding. This tutorial also provides a .zip file which contains an executable CSAT script for each debug operation.

2. Useful information

While this tutorial assumes the reader has some understanding of the Armv7-A debug architecture, the CoreSight Architectures, and CSAT, there are Terms and Resources sections at the end of this tutorial for reference purposes.

The CSAT scripts use CSAT CoreSight Components to access the different debug devices and registers. More information about CSAT CoreSight Components can be found in the How to use CSAT CoreSight Components tutorial.

3. Test Platform Information

The following testing and debug register information helps you to work through the tutorial.

Testing information

The CSAT scripts in this tutorial have been tested with the Arm Versatile Express with the Coretile Express A15x2 A7x3 (V2P-CA15_A7). CSAT version 2.6.3 which ships with DS-5 5.27.0 was used. Users will need a DS-5 installation and a DSTREAM or DSTREAM-ST unit to work through the tutorial.

The scripts make use of the V2P-CA15_A7 Cortex-A7_0, Cortex-A7_1, and Cortex-A7_2 cores and corresponding CTIs to exercise the CoreSight architecture implemented by the Armv7-A debug registers. The DAP APB-AP index number for the V2P-CA15_A7 board is 1. The DAP ABP-AP index number may be different if you are using a different board.

The below information details what the different addresses used in the below scripts:

V2P-CA15_A7 scripts

Device	CoreSight base address
Cortex-A7_0	`0x82030000`
Cortex-A7_1	`0x82032000`
TPIU	`0x80030000`
ETM_0	`0x8203C000`
ETM_1	`0x8203D000`
CTI for the Cortex-A7_0	`0x82038000`
CTI for the Cortex-A7_1	`0x82039000`
CTI for the Cortex-A7_2	`0x8203A000`

The DDR memory address is at `0x80000000` on the V2P-CA15_A7 board.

Debug Register Information

The CSAT scripts in this tutorial access particular CoreSight debug registers to exercise some CSAT commands. Below is a listing of the components accessed, the different debug registers accessed, the register name used by CSAT to reference the register, and the debug register offset.

Component	Full Debug Register Name	Debug Register Name	CSAT Register Name	Debug Register Offset
Armv7-A/R core	Debug Status and Control Registers	DBGDSCR	`dscr`	`0x088`
Armv7-A/R core	Debug Run Control Register	DBGDRCR	`drcr`	`0x090`
Armv7-A/R core	Device Powerdown and Reset Status	DBGPRSR	`prsr`	`0x314`
Armv7-A/R core	OS Lock Access Register	DBGOSLAR	`oslar`	`0x300`
Armv7-A/R core	OS Lock Status Register	DBGOSLSR	`oslsr`	`0x304`
Armv7-A/R core	Breakpoint Value Register	DBGBVR0	`bvr0`	`0x100`
Armv7-A/R core	Breakpoint Control Register	DBGBCR0	`bcr0`	`0x140`

Component	Full Debug Register Name	Debug Register Name	CSAT Register Name	Debug Register Offset
Armv7-A/R core	Watchpoint Value Register	DBGWVR0	`wvr0`	`0x180`
Armv7-A/R core	Watchpoint Control Register	DBGWCR0	`wcr0`	`0x1c0`
Armv7-A/R core	Instruction Transfer	DBGITR	`itr`	`0x084` (write only)
Armv7-A/R core	Target to Host Data Transfer	DBGDTRTX	`dtrtx`	`0x08C`
Armv7-A/R core	Host to Target Data Transfer	DBGDTRRX	`dtrrx`	`0x080`
CTI	CTI Control Register	CTICONTROL	`ctrl`	`0x000`
CTI	CTI Trigger to Channel Enable Registers 0	CTIINEN0	`inen0`	`0x020`
CTI	CTI Channel to Trigger Enable Registers 0	CTIOUTEN0	`outen0`	`0x0A0`
CTI	CTI Channel to Trigger Enable Registers 7	CTIOUTEN7	`outen7`	`0x0BC`
CTI	CTI Application Pulse Register	CTIAPPPULSE	`apppulse`	`0x01C`
CTI	CTI Interrupt Acknowledge Register	CTIINTACK	`intack`	`0x010`

Below is a list of the core debug register bits that will be looked at in the example scripts:

Please note: The descriptions and values for the different register bits have been paraphrased. You will need to refer to the Arm ARM found in the Reference section for the complete descriptions and values.

Debug Register	Bit name	Bit	Descriptions	Value
DBGDSCR	TXfull	29	DBGDTRTX register full	0 = DBGDTRTX register empty 1 = DBGDTRTX register full
DBGDSCR	InstrCompl_I	24	Signals whether that processor has completed execution of an instruction issued through DBGITR	0 = instruction not completed 1 = instruction completed
DBGDSCR	HDBGen	14	Halting debug-mode enable	0 = halting debug-mode disabled 1 = halting debug-mode enabled
DBGDSCR	ITRen	13	Enables the execution of Arm instructions through the DBGITR	0 = ITR mechanism disabled 1 = ITR mechanism enabled. Processor executes instructions via ITR
DBGDSCR	MOE	5:2	Method of Debug Entry	0 = Halt request debug event 1 = Breakpoint debug event 2 = Asynchronous watchpoint debug event
DBGDSCR	RESTARTED	1	Processor Restarted	0 = processor is exiting Debug state 1 = processor has exited Debug state

Debug Register	Bit name	Bit	Descriptions	Value
DBGDSCR	HALTED	`0`	Processor Halted	`0` = processor is in Non-debug state `1` = processor is in Debug state
DBGOSLSR	OSLK	`1`	OS Lock Status	`0` = OS Lock not set `1` = OS Lock set
DBGOSLAR	OS Lock Access	`31:0`	Locks the debug registers	`0xC5ACCE55` = lock registers `0` = unlock registers
DBGPRSR	OSLK	`5`	OS Lock status	`0` = OS Lock not set `1` = OS lock set
DBGDRCR	HRQ	`0`	Halt request	`0` = No action `1` = Request entry to Debug state
DBGDRCR	RRQ	`1`	Restart request	`0` = No action `1` = Request exit from Debug state
DBGITR	Arm instruction to execute on the processor	`31:0`	The 32-bit of an Arm instruction to execute on the processor	Contains Arm instruction to be executed
DBGDTRTX	Target to host data	`31:0`	One word of data for transfer from the debug target to the debug host	Contains data to be transferred
DBGDTRRX	Host to target data	`31:0`	One word of data for transfer from the debug host to the debug target	Contains data to be transferred
DBGBVR0	Instruction address	`31:2`	Holds an instruction address value for use in breakpoint matching	Contains value for breakpoint comparison
DBGBCR0	MASK	`28:24`	Address range mask	Contains the mask for address range breakpoint comparison
DBGBCR0	BT	`23:20`	Controls the behavior of debug event generation	`0` = Unlinked instruction address match*
DBGBCR0	LBN	`19:16`	If using Linked instruction match or mismatch, contains the number of the breakpoint that holds the Context match needed	`0` = No linked breakpoint *
DBGBCR0	SSC	`15:14`	Security state control	Meaning based on the settings of HMC and PMC bits
DBGBCR0	HMC	`13`	Hyp mode control bit	Meaning based on the settings of SSC and PMC bits
DBGBCR0	BAS	`8:5`	Enables match or mismatch comparisons on only certain bytes of the word address held in the	`0xF` = breakpoint programmed to match on hit*
DBGBCR0	PMC	`2:1`	Privileged mode control	Meaning based on the settings of SSC and HMC bits
DBGBCR0	E	`0`	Breakpoint enable	`0` = breakpoint disabled `1` = breakpoint enabled

Debug Register	Bit name	Bit	Descriptions	Value
DBGWVR0	Data address	`31:2`	Holds a data address value for use in watchpoint matching	Contains value for watchpoint comparison
DBGWCR0	MASK	`28:24`	Address range mask	`0` = No mask*
	WT	`20`	Defines the Watchpoint type	`0` = unlinked data address match `1` = Linked data address match
	LBN	`19:16`	If using Linked data address match, contains the number of the breakpoint that holds the Context match needed	`0` = No linked breakpoint *
	SSC	`15:14`	Security state control	Meaning based on the settings of HMC and PAC bits
	HMC	`13`	Hyp mode control	Meaning based on the settings of SSC and PAC bits
	BAS	`12:5`	Enables the watchpoint to hit only if certain bytes of the addressed word are accessed	`0xF` = 4-bit Byte address select field is implemented*
	LSC	`4:3`	Enables watchpoint matching on the type of access being made	`1` = match on any load `2` = match on any store `3` = match on all types of access
	PAC	`2:1`	Privileged access control	Meaning based on the settings of SSC and HMC bits
	E	`0`	Watchpoint enable	`0` = watchpoint disabled `1` = watchpoint enabled

*marks the values used in this tutorial. There are usually more values available.

Note

Re-reading the DBGPRSR clears the SPD bit could have an effect on v7 debug targets. This does not effect v7.1 debug targets.

Below is a list of the CTI debug registers that will be used in the scripts:

Note

The descriptions and values for the different register bits have been paraphrased. You will need to refer to CoreSight Technical Reference Manual found in the Reference section for the complete descriptions and values.

Debug Register	Bit name	Bit	Definition	Value
CTICONTROL	GLBEN	0	Enables or disables the ECT	`0` = disabled `1` = enabled

Debug Register	Bit name	Bit	Definition	Value
CTIINEN0	TRIGINEN	3:0	Enables a cross trigger event to the corresponding channel. There is one bit for each of the four channels	1 = enables the CTITRIGIN signal to generate an event 0 = disables the CTITRIGIN signal from generating an event
TIOUTEN0 and CTIOUTEN7	TRIGOUTEN	3:0	Enables a channel event for the corresponding channel to generate a CTITRIGOUT output. There is one bit for each of the four channels.	0 = the CTICHIN from the CTM is not routed to the CTITRIGOUT output 1 = the CTICHIN from the CTM is routed to the CTITRIGOUT output
CTIAPPPULSE	APPULSE	3:0	Setting a bit HIGH generates a channel pulse for the selected channel	1 = channel event pulse generated 0 = no effect
CTIINTACK	INTACK	7:0	Acknowledges the corresponding CTITRIGOUT output	1 = CTITRIGOUT is acknowledged 0 = no effect

4. Halting a Core using Debug Registers

The script in this section focuses on the steps necessary to halt an Armv7-A/R processor using the processor's debug registers only.

Below are the list of steps performed by the script:

1. Run CSAT

   ```
   csat
   ```

2. Connect to a DSTREAM

   ```
   con USB
   ```

3. Configure the target connection

   ```
   chain dev=ARMCS-DP clk=10000000
   ```

4. Connect to the DAP

   ```
   dvo 0
   ```

5. Set the CSAT default AP index to 1 (the debug APB-AP)

   ```
   cscomp def_apidx 1
   ```

6. Create a `v7dbg.0` component for the Cortex-A7 we want to halt

   ```
   v7dbg.0 baseaddr 0x82030000
   ```

7. Create an `alias`for the `v7dbg.0` component called `A7`

   ```
   alias v7dbg.0 A7
   ```

8. Read the `DBGOSLSR OSLK bit[1]` to determine the OS Lock state

   ```
   A7 rr oslsr
   ```

9. Read the `DBGPRSR OSLK bit[5]` to determine the OS Lock state

   ```
   A7 rr prsr
   ```

10. Re-read the `DBGPRSR OSLK bit[5]` to determine the OS Lock state

    ```
    A7 rr prsr
    ```

11. If the `OSLK` bit in the `DBGOSLSR` and `DBGPRSR` are `1`, set the `DBGOSLAR OS Lock Access bits[31:0]` to `0` to clear the OS lock

```
A7 rw oslar 0x0
```

12. Read the `DBGOSLSR OSLK` bit to see if the OS Lock is clear (`bit[1]` is `0`)

```
A7 rr oslsr
```

13. Read the `DBGPRSR OSLK` bit to see if the OS Lock is clear (`bit[5]` is `0`)

```
A7 rr prsr
```

14. Re-read the `DBGPRSR OSLK` bit to see if the OS Lock is clear (`bit[5]` is `0`)

```
A7 rr prsr
```

15. Read the `DBGDSCR HALTED` bit to see if processor is already halted (`bit[0]` is `1`)

```
A7 rr dscr
```

16. If the processor is not halted, set the `DBGDRCR HRQ bit[0]` to `1` to request a halt. If the processor is halted, skip to step 18).

```
A7 rw drcr 0x1
```

17. Read the `DBGDSCR HALTED` bit to see if the processor has halted (i.e. `bit[0]` is set = `1` if halted)

```
A7 rr dscr
```

18. Set the `DBGDSCR HDBGen bit[14]` to `1` to enable halting debug-mode

```
A7 rw dscr 0x4000
```

19. Read the `DBGDSCR HDBGen bit[14]` to see if it has been set

```
A7 rr dscr
```

20. Exit CSAT

```
exit
```

5. Restarting a Core using Debug Registers

The script in this section focuses on the steps necessary to restart an Armv7-A/R processor using the processor's debug registers only.

Below are the list of steps performed by the script:

Repeat steps 1) - 14) of the Halting a Core using Debug Registers section, and then do the following steps:

```
csat
con USB
chain dev=ARMCS-DP clk=10000000
dvo 0
cscomp def_apidx 1
v7dbg.0 baseaddr 0x82030000
alias v7dbg.0 A7
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rw oslar 0x0
A7 rr oslsr
A7 rr prsr
A7 rr prsr
```

1. Read the `DBGDSCR HALTED` bit to see if processor is already running (`bit[0]` is `0`)

   ```
   A7 rr dscr
   ```

2. If the processor is not running, set the `DBGDRCR RRQ bit[1]` to `1` to request a restart

   ```
   A7 rw drcr 0x2
   ```

3. Read the `DBGDSCR RESTARTED` bit to see if the processor has restarted (`bit[1]` is `1`)

   ```
   A7 rr dscr
   ```

4. Exit CSAT

   ```
   exit
   ```

6. Stopping and Restarting Cores using CTIs

This script, once the CTIs are setup and enabled, uses the CTIAPPPULSE register to halt and restart Cortex-A7_0, Cortex-A7_1, and Cortex-A7_2.

Below are the list of steps performed by the script:

1. Run CSAT

```
csat
```

2. Connect to a DSTREAM

```
con USB
```

3. Configure the target connection

```
chain dev=ARMCS-DP clk=10000000
```

4. Connect to the DAP

```
dvo 0
```

5. Set the CSAT default AP index to `1`

```
cscomp def_apidx 1
```

6. Create a `v7dbg` component for the Cortex-A7_0

```
v7dbg.0 baseaddr 0x82030000
```

7. Create a `v7dbg` component for the Cortex-A7_1

```
v7dbg.1 baseaddr 0x82032000
```

8. Create a `v7dbg` component for the Cortex-A7_2

```
v7dbg.2 baseaddr 0x82034000
```

9. Create an alias for the `v7dbg.0` component called `A7_0`

```
alias v7dbg.0 A7_0
```

10. Create an alias for the `v7dbg.1` component called `A7_1`

```
alias v7dbg.1 A7_1
```

11. Create an alias for the `v7dbg.2` component called `A7_2`

```
alias v7dbg.2 A7_2
```

12. Create a `cti.0` component for the CTI for Cortex-A7_0

```
cti.0 baseaddr 0x82038000
```

13. Create a `cti.1` component for the CTI for Cortex-A7_1

```
cti.1 baseaddr 0x82039000
```

14. Create a `cti.2` component for the CTI for Cortex-A7_2

```
cti.2 baseaddr 0x8203A000
```

15. Set the Cortex-A7_0 `DBGOSLAR OS Lock Access bits[31:0]` to `0` to clear the OS lock

```
A7_0 rw oslar 0x0
```

16. Read the Cortex-A7_0 `DBGDSCR HALTED` bit to see if the processor is already halted (`bit[0] = 1`)

```
A7_0 rr dscr
```

17. Set the Cortex-A7_1 `DBGOSLAR OS Lock Access bits[31:0]` to `0` to clear the OS lock

```
A7_1 rw oslar 0x0
```

18. Read the Cortex-A7_1 `DBGDSCR HALTED` bit to see if the processor is already halted (`bit[0] = 1`)

```
A7_1 rr dscr
```

19. Set the Cortex-A7_2 `DBGOSLAR OS Lock Access bits[31:0]` to `0` to clear the OS lock

```
A7_2 rw oslar 0x0
```

20. Read the Cortex-A7_2 `DBGDSCR HALTED` bit to see if the processor is already halted (`bit[0] = 1`)

```
A7_2 rr dscr
```

21. If the processors are not halted, set the CTI_0 `CTICONTROL GLBENbit[0]` to `1` to enable ECT

```
cti.0 rw ctrl 0x1
```

22. Set the CTI_0 `CTIINEN0 TRIGINEN bit[0]` to `1` to enable passing DBGACK in on CMT channel 0

```
cti.0 rw inen0 0x1
```

23. Set CTI_0 `CTIOUTEN0 TRIGOUTEN0 bit[0]` to `1` to enable passing EDBGRQ out on CTM channel 0

```
cti.0 rw outen0 0x1
```

24. Set CTI_0 `CTIOUTEN7 TRIGOUTEN0 bit[1]` to `1` to enable passing DBGRESTART out on CTM channel 1

```
cti.0 rw outen7 0x2
```

25. Set the CTI_1 `CTICONTROL GLBENbit[0]` to `1` to enable ECT

```
cti.1 rw ctrl 0x1
```

26. Set the CTI_1 `CTIINEN0 TRIGINEN bit[0]` to `1` to enable passing DBGACK in on CMT channel 0

```
cti.1 rw inen0 0x1
```

27. Set CTI_1 `CTIOUTEN0 TRIGOUTEN0 bit[0]` to `1` to enable passing EDBGRQ out on CTM channel 0

```
cti.1 rw 1 outen0 0x1
```

28. Set CTI_1 `CTIOUTEN7 TRIGOUTEN0 bit[1]` to `1` to enable passing DBGRESTART out on CTM channel 1

```
cti.1 rw outen7 0x2
```

29. Set the CTI_2 `CTICONTROL GLBENbit[0]` to `1` to enable ECT

```
cti.2 rw ctrl 0x1
```

30. Set the CTI_2 `CTIINEN0 TRIGINEN bit[0]` to `1` to enable passing DBGACK in on CMT channel 0

```
cti.2 rw inen0 0x1
```

31. Set CTI_2 `CTIOUTEN0 TRIGOUTEN0 bit[0]` to `1` to enable passing EDBGRQ out on CTM channel 0

```
cti.1 rw 1 outen0 0x1
```

32. Set CTI_2 `CTIOUTEN7 TRIGOUTEN0 bit[1]` to `1` to enable passing DBGRESTART out on CTM channel 1

```
cti.2 rw outen7 0x2
```

33. Set the CTI_0 `CTIAPPPULSE APPULSE bit[0]` to `1` to pulse on CTM channel 0 to halt all the processors

```
cti.0 rw apppulse 0x1
```

34. Read the Cortex-A7_0 `DBGDSCR HALTED` bit to see if the processor has halted (`bit[0] = 1`)

```
A7_0 rr dscr
```

35. Read the Cortex-A7_1 `DBGDSCR HALTED` bit to see if the processor has halted (`bit[0] = 1`)

```
A7_1 rr dscr
```

36. Read the Cortex-A7_2 `DBGDSCR HALTED` bit to see if the processor has halted (`bit[0] = 1`)

```
A7_2 rr dscr
```

37. Set CTI_0 `CTITACK INTACK bit[0]` to `1` to acknowledge CTITRIGOUT[0]

```
cti.0 rw intack 0x1
```

38. Set CTI_1 `CTITACK INTACK bit[0]` to `1` to acknowledge CTITRIGOUT[0]

```
cti.1 rw intack 0x1
```

39. Set CTI_2 `CTITACK INTACK bit[0]` to `1` to acknowledge CTITRIGOUT[0]

```
cti.2 rw intack 0x1
```

40. Set the CTI_0 `CTIAPPPULSE APPULSE bit[1]` to `1` to pulse on CTM channel 1 to restart all the processors

```
cti.0 rw apppulse 0x2
```

41. Read the Cortex-A7_0 `DBGDSCR RESTARTED` bit to see if the processor has restarted (`bit[1] = 1`)

```
A7_0 rr dscr
```

42. Read the Cortex-A7_1 `DBGDSCR RESTARTED` bit to see if the processor has restarted (`bit[1] = 1`)

```
A7_1 rr dscr
```

43. Read the Cortex-A7_2 `DBGDSCR RESTARTED` bit to see if the processor has restarted (`bit[1] = 1`)

```
A7_2 rr dscr
```

44. Exit CSAT

```
exit
```

7. Executing Instructions while the Core is Halted

When debugging problems, it might be advantageous to execute instructions via a core when in debug state. For instance, you might want to execute a barrier instruction while the core is halted to flush previous instruction execution or data accesses.

The Armv7-A architecture provides a set of debug instruction transfer and data transfer registers to accomplish these types of tasks. The registers used in this script are the `DBGITR`, the `DBGDTRTX`, and `DBGDTRRX`. The `DBGITR` is used to execute Arm instructions when a core is in debug state while the `DBGDTRTX` and `DBGDTRRX` registers are used to transfer data to and from the target respectively.

This example script first uses `DBGITR` to move the value of the `PC` register to `R0` and then transfers the `R0` value to `DBGDTRTX` so the `PC` value can be read. Second, it loads `DBGDTRRX` with a value of `0x80000000` and then uses `DBGITR` to move the `DBGDTRRX` value into `R0` and move `R0` into the `PC`.

Below are the lists of steps performed by the script:

Repeat steps 1) - 17) of the Halting a Core using Debug Registers section, and then do the following steps:

```
csat
con USB
chain dev=ARMCS-DP clk=10000000
dvo 0
cscomp def_apidx 1
v7dbg.0 baseaddr 0x82030000
alias v7dbg.0 A7
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rw oslar 0
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rr dscr
A7 rw drcr 1
A7 rr dscr
```

1. Write the `DBGDSCR HDBGen bit[14]` and `ITRen bit[13]` to `1` to enable halting debug-mode and `DBGITR` instruction execution

```
A7 rw dscr 0x6000
```

2. Read the `DBGDSCR HDBGen bit[14]` and `ITRen bit[13]` to see if these have been set

```
A7 rr dscr
```

3. Write the Arm instruction `"MOV R0,PC"` to move the `PC` value into `R0` into the `DBGITR`

```
A7 rw itr 0xE1A0000F
```

4. Write the Arm instruction `"MCR p14,#0,r0,c0,c5,#0"` to transfer the `R0` value to the `DBGTRTX` into the `DBGITR`

```
A7 rw itr 0xEE000E15
```

5. Read the `DBGDSCR InstrCompl_l bit` to determine if the Arm instruction has completed execution (`bit[24] = 1`)

```
A7 rr dscr
```

6. Read the `DBGDTRTX` to get the `PC` value. The `PC` value read will be two instructions off the current `PC` value. For instance, in Arm state, if the current `PC` is `0x80000000`, the `PC` value read from the D`BGDTRTX` will be `0x80000008`.

```
A7 rr dtrtx
```

7. Write `0x80000000` to `DBGDTRRX`.

```
A7 rw dtrrx 0x80000000
```

8. Write the Arm instruction `"MRC p14, 0, R0, c0, c5, 0"` to transfer the `DBGDTRRX` value to `R0` into `DBGITR`.

```
A7 rw itr 0xEE100E15
```

9. Write the Arm instruction `"MOV PC, R0"` to move the `R0` value into the `PC` into `DBGITR`.

```
A7 rw itr 0xE1A0F000
```

10. Read the `DBGDSCR InstrCompl_l` bit to determine if the Arm instruction has completed execution (`bit[24] = 1`)

```
A7 rr dscr
```

11. Exit CSAT

```
exit
```

8. Setting a hardware breakpoint

When a debug environment is not available, it is useful to be able to set hardware breakpoints on instructions to halt core execution in key areas.

This example uses the `"startup_Cortex-A7"` example from `"<DS-5 installation location>\examples\Bare-metal_examples_ARM7.zip\DS-5Examples"` to demonstrate setting an unlinked, PL0 mode only breakpoint at the `"Reset_Handler"` address `S:0x8000005C`. The breakpoint that is set in the `"Reset_Handler"` will cause the core to halt a short period after execution commences from address `S:0x80000000`. The CSAT script assumes that the `startup_Cortex-A7.axf` is in the same directory as the CSAT executable.

Note

The Cortex-A7 does not support address range masking on breakpoints, so the `DBGBCR0 MASK` bits will be `0x0` for this example.

Below are the list of steps performed by the script:

Repeat steps 1 - 17 of the Halting a Core using Debug Registers section, and then do the following steps:

```
csat
con USB
chain dev=ARMCS-DP clk=10000000
dvo 0
cscomp def_apidx 1
v7dbg.0 baseaddr 0x82030000
alias v7dbg.0 A7
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rw oslar 0
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rr dscr
A7 rw drcr 1
A7 rr dscr
```

1. Write the `DBGDSCR HDBGen bit[14]` and `ITRen bit[13]` to `1` to enable halting debug-mode and `DBGITR` instruction execution.

```
A7 rw dscr 0x6000
```

2. Read the `DBGDSCR HDBGen bit[14]` and `ITRen bit[13]` to see if these have been set.

```
A7 rr dscr
```

3. Load the `"startup-Cortex-A7"` example image using the Cortex-A7_0 to target memory starting at address `0x80000000`.

```
dfl v7dbg elf startup_Cortex-A7.axf
```

4. Repeat steps 7 - 10 of the Executing Instructions while the Core is Halted section to load the PC with the start address of the `"startup-Cortex-A7"` image (`0x80000000`)

```
A7 rw dtrrx 0x80000000
A7 rw itr 0xEE100E15
A7 rw itr 0xE1A0F000
A7 rr dscr
```

5. Write the `DBGBCR0` to setup an unlinked, PL0 mode only breakpoint

```
A7 rw bcr0 0x000081E7
```

6. Write the address of the `Reset_Handler` function (`0x8000005C`) to the `DBGBVR0`

```
A7 rw bvr0 0x8000005C
```

7. If the processor is not running, set the `DBGDRCR RRQ bit[1]` to `1` to request a restart

```
A7 rw drcr 0x2
```

8. Read the `DBGDSCR HALTED` and `MOE` bits to see if the processor has halted (`bit[0]` is `1`) due to a Breakpoint debug event (`bit[5:2]` is `0b0001`) respectively. This `CSAT` command may need to be executed many times.

```
A7 rr dscr
```

9. Exit CSAT

```
exit
```

9. Setting a watchpoint

It is sometimes helpful to set a watchpoint on certain addresses to determine access history.

This example uses the `"startup_Cortex-A7"` example from `"<DS-5 installation location>\examples\Bare-metal_examples_ARM7.zip\DS-5Examples"` to demonstrate setting an unmasked, unlinked, PL1 and unprivileged watchpoint on the stack pointer `SP` at address `0x8008FFFC`. The watchpoint will halt the core when the SP reaches address `s:0x8008FFFC` from its initial address value of `s:0x80090000`. The CSAT script assumes that the `Startup_Cortex-A7.axf` is in the same directory as the CSAT executable.

Below are the list of steps performed by the script:

Repeat steps 1 - 17 of the Halting a Core using Debug Registers section

```
csat
con USB
chain dev=ARMCS-DP clk=10000000
dvo 0
cscomp def_apidx 1
v7dbg.0 baseaddr 0x82030000
alias v7dbg.0 A7
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rw oslar 0
A7 rr oslsr
A7 rr prsr
A7 rr prsr
A7 rr dscr
A7 rw drcr 1
A7 rr dscr
```

1. Write the `DBGDSCR HDBGen bit[14]` and `ITRen bit[13]` to `1` to enable halting debug-mode and `DBGITR` instruction execution

   ```
   A7 rw dscr 0x6000
   ```

2. Read the `DBGDSCR HDBGen bit[14]` and ITRen `bit[13]` to see if these have been set

   ```
   A7 rr dscr
   ```

3. Load the `"startup-Cortex-A7"` example image using the Cortex-A7_0 to target memory starting at address `0x80000000`

   ```
   dfl v7dbg elf startup_Cortex-A7.axf
   ```

4. Repeat steps 7) - 10) of the Executing Instructions while the Core is Halted section to load the `PC` with the start address of the `"startup-Cortex-A7"` image (`0x80000000`)

   ```
   A7 rw dtrrx 0x80000000
   A7 rw itr 0xEE100E15
   ```

```
A7 rw itr 0xE1A0F000
A7 rr dscr
```

5. Write the `DBGWCR0` to setup an unmasked, unlinked, PL1 and unprivileged watchpoint

```
A7 rw wcr0 0x0000081FF
```

6. Write the address of the `SP` (`0x8008FFFC`) to the `DBGWVR0`

```
A7 rw wvr0 0x8008FFFC
```

7. If the processor is not running, set the `DBGDRCR RRQ bit[1]` to `1` to request a restart

```
A7 rw drcr 0x2
```

8. Read the `DBGDSCR HALTED` and `MOE` bits to see if the processor has halted (`bit[0]` is `1`) due to a Synchronous watchpoint debug event (`bit[5:2]` is `0b1010`) respectively. This CSAT command may need to be executed many times.

```
A7 rr dscr
```

9. Exit CSAT

```
exit
```

10. Conclusion

The tutorial focuses on performing certain debug operations on an Armv7-A processor using CSAT. The debug register accesses listed in the various scripts can be used as a baseline for performing these operations outside CSAT, such as with C or C++. As in, the necessary register accesses will remain the same no matter the method used to access them.

Note

Additional register accesses may be required or different methods may need to be used to access the registers depending on your execution environment and the target state.

11. Terms

You can find the definitions of the terms used in this tutorial:

Debug state

Refers to the processor state where the processor is halted for debug purposes

Non-debug state

The processor is executing instructions fetched from memory and executing normal application code

Halting-mode debug

Causes the processor to halt when a debug event occurs (such as a debugger halt request, a breakpoint match, a watchpoint match, etc...)

Embedded Cross Trigger (ECT)

Logic which enables Arm/ETM subsystems to interact/cross trigger with each other

Cross Triggering Matrix (CTM)

A logic block which controls the distribution of channel events from CTIs or other CTMs

CTITRIGIN

A signal which passes CTI trigger input to the CTM on a specific channel

CTITRIGOUT

A signal which passes CTI trigger output to the CTM on a specific channel

Channel input (CTICHIN)

The channel input from the CTM

EDBGRQ

Core signal to trigger an External debug request mechanism

DBGACK

Core signal which indicates that that the processor is in Debug State when asserted

DBGRESTART

Core signal to trigger an External Restart request

12. Resources

Here are some resources related to material in this guide:

- Cortex-A7 MPCore Technical Reference Manual
- CoreSight Components Technical Reference Manual
- Arm Architecture Reference Manual Armv7-A and Armv7-R edition Issue C
- CoreSight Access Tool (CSAT) User Guide
- How to use CSAT CoreSight Components