LAUTERBACH
DEVELOPMENT TOOLS

IPU Debugger

MANUAL

IPU Debugger

IPU Debugger

Version 06-Jun-2024

Introduction

This manual serves as a guideline for debugging IPU (“Image Processing Unit”) core(s) via TRACE32.

Please keep in mind that only the **Processor Architecture Manual** (the document you are reading at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs supported by Lauterbach. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Training Basic Debugging”** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **“T32Start”** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **“General Commands”** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **“OS Awareness Manuals”** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

PRACTICE Script Language:

- **“Training Script Language PRACTICE”** (training_practice.pdf)
- **“PRACTICE Script Language Reference Guide”** (practice_ref.pdf)

Warning

WARNING:	To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF. Recommendation for the software start: 1. Disconnect the Debug Cable from the target while the target power is off. 2. Connect the host system, the TRACE32 hardware and the Debug Cable. 3. Power ON the TRACE32 hardware. 4. Start the TRACE32 software to load the debugger firmware. 5. Connect the Debug Cable to the target. 6. Switch the target power ON. 7. Configure your debugger e.g. via a start-up script. Power down: 1. Switch off the target power. 2. Disconnect the Debug Cable from the target. 3. Close the TRACE32 software. 4. Power OFF the TRACE32 hardware.

Starting up the debugger is done as follows:

1. Select the device prompt for the ICD Debugger and reset the system.

```
B::

RESet
```

The device prompt `B::` is normally already selected in the TRACE32 command line. If this is not the case, enter `B::` to set the correct device prompt. The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

2. Specify the CPU specific settings.

```
SYStem.CPU <cpu_type>
```

The default values of all other options are set in such a way that it should be possible to work without modification. Please consider that this is probably not the best configuration for your target.

3. Inform the debugger about read-only address ranges (ROM, FLASH).

```
MAP.BOnchip 0x060000000++3FFFF
```

The B(reak)Onchip information is necessary to decide where on-chip breakpoints must be used. On-chip breakpoints are necessary to set program breakpoints to FLASH/ROM.

4. Specify ranges where the access width is restricted.

```
MAP.BUS32 0x060000000++1FFFF
```

If a memory location can only be accessed with a certain bus width you can use Map.BUS8 / BUS16 / BUS32 to force the debugger to use solely the according load or store instructions. This allows for example to have a byte-by-byte dump of a 32 bit wide memory area, where a byte access would cause an exception.

5. Enter debug mode.

```
SYStem.Up
```

This command resets the CPU and enters debug mode. After this command is executed, it is possible to access memory and registers.

6. Load the program.

```
Data.LOAD <file> /LONG              ;load the compiler output.
                                    ;the option /LONG tells the
                                    ;debugger to use 32 bit accesses
```

The format of the **Data.LOAD** command depends on the file format generated by the compiler.

A detailed description of the **Data.LOAD** command and all available options is given in the **"General Commands Reference"**.

A typical start sequence without EPROM simulator is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

```
B::                                 ; Select the ICD device prompt

WinCLEAR                            ; Clear all windows

MAP.BOnchip 0x60000000++0xfffff     ; Specify where FLASH/ROM is

MAP.BUS32 0x50000000++0x1ffff       ; Force the debugger to access this
                                    ; area 32 bit wide

SYStem.Up                           ; Reset the target and enter debug
                                    ; mode

Data.LOAD.elf xtensa_project        ; Load the application

Register.Set PC _ResetVector        ; Set the PC to start point

Register.Set a1 0x63FFFFFC          ; Set the stack pointer to address
                                    ; 0x63FFFFFC

List.Mix                            ; Open source code window         *)

Register.view /SpotLight            ; Open register window            *)

Frame.view /Locals /Caller          ; Open the stack frame with
                                    ; local variables                 *)

Var.Watch %SpotLight flags ast      ; Open watch window for variables *)

Break.Set 0x60100000 /Program       ; Set software breakpoint to address
                                    ; 0x60100000 (address 0x60100000
                                    ; outside of BOnchip range)

Break.Set 0x60001000 /Program       ; Set on-chip breakpoint
                                    ; to address 0x60001000 (address
                                    ; 0x60001000 is within BOnchip range)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command, this may have the following reasons:

- The target has no power.
- The target is in reset.
- The IPU core is not enabled.
- There is logic added to the JTAG state machine.
- There are additional loads or capacities on the JTAG lines.
- There is a short circuit on at least one of the output lines of the core.

FAQ

Please refer to https://support.lauterbach.com/kb.

IPUS and IPUV Core Debugging in Heterogeneous SMP System

There are the following types of IPU cores:

- **IPUS (Scalar Image Processing Unit)**
- **IPUV (Vector Image Processing Unit)**

Although IPUS and IPUV differ in core architecture and their sets of core and debug registers, TRACE32 supports to debug *both* core types simultaneously in *one* TRACE32 PowerView instance (t32mipu.exe). This makes TRACE32 debugger for IPU a heterogeneous debug system.

The type of each core under debug is determined indirectly via the assignment of its respective base address of the debug registers. See **SYStem.CONFIG.IPUSDEBUG.Base** for more details.

Heterogeneous Register Window

Although the two core types have different sets of CPU registers, one instance of TRACE32 PowerView for IPU allows to debug both core types, i.e. IPUS and IPUV, simultaneously. TRACE32 PowerView automatically adapts the registers that are displayed in each **Register.view** window according to the core that it corresponds to.

```
Register.view /CORE 3.    ;display register information from core 3
```

For details, see **CORE.SHOWACTIVE**.

Heterogeneous Disassembler

Although the two core types have different instruction sets, one instance of TRACE32 PowerView for IPU allows to debug both core types, i.e. IPUS and IPUV, simultaneously. TRACE32 PowerView automatically adapts each process of disassembly according to the core that it corresponds to.

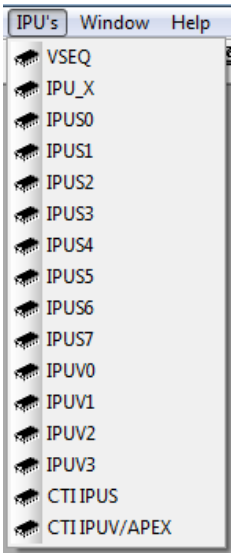
```
List.Mix func1 /CORE 3.   ;display source listing from core 3
```

For details, see **CORE.SHOWACTIVE**.

IPU Specific Peripheral Files

If the CPU **S32V234-IPU** is selected, then the additional menu **IPU's** is added to the TRACE32 main menu bar. The menu options provide quick access to the following IPU specific peripheral files (*.per):

- **IPUS0, IPUS1, ... IPUS7:** Access to debug, control and CPU registers of one of the 8 IPUS cores on the S32V234 board.
- **IPUV0, IPUV1, IPUV2, IPUV3:** Access to debug, control and CPU registers of one of the 4 IPUV cores on the S32V234 board.
- **IPU_X**: Generic peripheral file for all IPUS and IPUV cores on the S32V234 board. The file does have the same structure and functional scope as any of the IPUS and IPUV files. However, the content and structure that gets displayed is automatically adapted to the core that is currently selected (e.g. via **CORE.select** command). So if for example IPUV core #2 is selected, then the IPU_X peripheral file does have the structure of IPUV files and displays the content of IPUV core #2.
- **CTI IPUS / CTI IPUV**: Allows to control the CTI (cross trigger interface) modules of the IPUS or IPUV cores respectively.



For more information about peripheral files in general, refer to the **PER** command group.

Breakpoints

Software Breakpoints

Software breakpoints are currently not supported for IPU.

On-chip Breakpoints for Instructions

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the CPU.

Each IPU core provides one on-chip breakpoint for instructions.

On-chip Breakpoints for Data

On-chip breakpoints are used to stop the CPU after a read or write access to a memory location.

Each IPU core provides one on-chip breakpoint for read- and one on-chip breakpoint for write accesses.

Example for On-Chip Breakpoints

The following is an example for setting standard on-chip breakpoints:.

```
Break.Set 0x40 /Program                ; On-chip instruction breakpoint

Break.Set 0x101000 /Read               ; On-chip data breakpoint (read)

Break.Set 0x102000 /Write              ; On-chip data breakpoint (write)
```

The following ARM specific memory access classes are available.

Memory	Description
P	Program Memory
D	Data Memory

To access a memory class, write the class in front of the address.

Example:

```
Data.dump D:0x0--0x3
```

The following IPU specific memory access class is available:

Memory	Description
DAP	Within the range DAP:0xE0000000--DAP:0xE000FFF, the 12 lower bits get interpreted as a relative offset. The actual absolute address is then calculated by automatically adding up this offset and the base address of the currently selected IPU core. This has the advantage that the base address can be *neglected* as it will be *automatically* added by the debugger. Outside of the range DAP:0xE0000000--DAP:0xE000FFF, the DAP access class works like the D access class. This access class can e.g. be used to observe certain memory ranges in a **Data.dump** window and have the **Data.dump** window automatically switch context when a different core is selected. Furthermore the access class is used throughout the generic peripheral file ~~/peripux.per in order to always address the debug registers of the currently selected core.

Example: Comparison of D and DAP memory accesses:

```
Assume that active core has BaseAddress = 0x40022000
Memory access #1:                      "D:0x40022040"
Memory access #2:                    "DAP:0xE0000040"

Both memory accesses are directed to the same address!
Calculation:
Offset_access_2 = 0xE0000040 & 0xFFF = 0x40
Actual absolute address of access 2 = BaseAddress + Offset_access_2 =
= 0x4002000 + 0x40 = 0x40022040
```

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state** [**/***<tab>*]
<tab>:	**DebugPort** | **Jtag** | **DAP** | **COmponents**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<tab>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, see below.
DebugPort (default)	The **DebugPort** tab informs the debugger about the debug connector type and the communication protocol it shall use. For descriptions of the commands on the **DebugPort** tab, see **DebugPort**.
Jtag	The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. For descriptions of the commands on the **Jtag** tab, see **Jtag**.
DAP	The **DAP** tab informs the debugger about an ARM CoreSight Debug Access Port (DAP) and about how to control the DAP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces. For descriptions of the commands on the **DAP** tab, see **DAP**.
COmponents	The **COmponents** tab informs the debugger (a) about the existence and interconnection of on-chip CoreSight debug and trace modules and (b) informs the debugger on which memory bus and at which base address the debugger can find the control registers of the modules. For descriptions of the commands on the **COmponents** tab, see **COmponents**.

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>: **(DebugPort)**	**CJTAGFLAGS** *<flags>* **CJTAGTCA** *<value>* **CONNECTOR** [**MIPI34** | **MIPI20T**] **CORE** *<core> <chip>* **CoreNumber** *<number>* **DEBUGPORT** [**DebugCable0**] **DEBUGPORTTYPE** [**JTAG** | **SWD**] **Slave** [**ON** | **OFF**] **SWDP** [**ON** | **OFF**] **TriState** [**ON** | **OFF**]
<parameter>: **(JTAG)**	**DAPDRPOST** *<bits>* **DAPDRPRE** *<bits>* **DAPIRPOST** *<bits>* **DAPIRPRE** *<bits>* **Slave** [**ON** | **OFF**] **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**]
<parameter>: **(DAP)**	**AHBACCESSPORT** *<port>* **APBACCESSPORT** *<port>* **AXIACCESSPORT** *<port>* **COREJTAGPORT** *<port>* **DEBUGACCESSPORT** *<port>* **JTAGACCESSPORT** *<port>* **MEMORYACCESSPORT** *<port>*
<parameter>: **(COmponents)**	**CTI.Base** *<address>* **CTI.Config** [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**] **CTI.RESET** **IPUSDEBUG.Base** *<address>* **IPUSDEBUG.RESET** **IPUVDEBUG.Base** *<address>* **IPUVDEBUG.RESET**

The **SYStem.CONFIG** commands inform the debugger about the available on-chip debug and trace components and how to access them.

The **SYStem.CONFIG** command information shall be provided after the **SYStem.CPU** command, which might be a precondition to enter certain **SYStem.CONFIG** commands, and before you start up the debug session e.g. by **SYStem.Up**.

Syntax Remarks

The commands are not case sensitive. Capital letters show how the command can be shortened.
Example: “SYStem.CONFIG.IPUSDEBUG.Base 0x1000” -> “SYS.CONFIG.IPUSDEBUG.B 0x1000”

The dots after “SYStem.CONFIG” can alternatively be a blank.
Example:
“SYStem.CONFIG.IPUSDEBUG.Base 0x1000” or “SYStem.CONFIG IPUSDEBUG Base 0x1000”.

CJTAGFLAGS *<flags>*

Activates bug fixes for "cJTAG" implementations.
Bit 0: Disable scanning of cJTAG ID.
Bit 1: Target has no "keeper".
Bit 2: Inverted meaning of SREDGE register.
Bit 3: Old command opcodes.
Bit 4: Unlock cJTAG via APFC register.

Default: 0

CJTAGTCA *<value>*

Selects the TCA (TAP Controller Address) to address a device in a cJTAG Star-2 configuration. The Star-2 configuration requires a unique TCA for each device on the debug port.

CONNECTOR [**MIPI34** | **MIPI20T**]

Specifies the connector "MIPI34" or "MIPI20T" on the target. This is mainly needed in order to notify the trace pin location.

Default: MIPI34 if CombiProbe is used, MIPI20T if µTrace (MicroTrace) is used.

CORE *<core>* *<chip>*

The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system.

Because of the default setting of this command

debugger#1: *<core>*=1 *<chip>*=1
debugger#2: *<core>*=1 *<chip>*=2
...

each debugger instance assumes that all notified debug and trace resources can exclusively be used.

But some target systems have shared resources for different cores, for example a common trace port. The default setting causes that each debugger instance controls the same trace port. Sometimes it does not hurt if such a module is controlled twice. But sometimes it is a must to tell the debugger that these cores share resources on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board:

debugger#1: *<core>*=1 *<chip>*=1
debugger#2: *<core>*=2 *<chip>*=1

CORE *<core>* *<chip>*

(cont.)

For cores on the same *<chip>,* the debugger assumes that the cores share the same resource if the control registers of the resource have the same address.

Default:
<core> depends on CPU selection, usually 1.
<chip> derives from `CORE=` parameter in the configuration file (config.t32), usually 1. If you start multiple debugger instances with the help of t32start.exe, you will get ascending values (1, 2, 3,...).

CoreNumber *<number>*

Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are core types like ARM11MPCore, CortexA5MPCore, CortexA9MPCore and Scorpion which can be used as a single core processor or as a scalable multicore processor of the same type. If you intend to debug more than one such core in an SMP debug session you need to specify the number of cores you intend to debug.

Default: 1.

DEBUGPORT [**DebugCable0**]

It specifies which probe cable shall be used e.g. “DebugCable0”. At the moment only the CombiProbe allows to connect more than one probe cable.

Default: depends on detection.

DEBUGPORTTYPE [**JTAG** | **SWD**]

It specifies the used debug port type “JTAG”, “SWD”. It assumes the selected type is supported by the target.

Default: JTAG.

Slave [**ON** | **OFF**]

If several debuggers share the same debug port, all except one must have this option active.

JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**.

Default: OFF; ON if `CORE=`... >1 in config file (e.g. config.t32).

SWDP [**ON** | **OFF**]

With this command you can change from the normal JTAG interface to the serial wire debug mode. SWDP (Serial Wire Debug Port) uses just two signals instead of five. It is required that the target and the debugger hard- and software supports this interface.

Default: OFF.

TriState [**ON** | **OFF**]

TriState has to be used if several debug cables are connected to a common JTAG port. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode.
Please note:

- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

Default: OFF.

<parameters> describing the “JTAG” scan chain and signal behavior

With the JTAG interface you can access a Test Access Port controller (TAP) which has implemented a state machine to provide a mechanism to read and write data to an Instruction Register (IR) and a Data Register (DR) in the TAP. The JTAG interface will be controlled by 5 signals:

- nTRST (reset)
- TCK (clock)
- TMS (state machine control)
- TDI (data input)
- TDO (data output)

Multiple TAPs can be controlled by one JTAG interface by daisy-chaining the TAPs (serial connection). If you want to talk to one TAP in the chain, you need to send a BYPASS pattern (all ones) to all other TAPs. For this case the debugger needs to know the position of the TAP it wants to talk to. The TAP position can be defined with the first four commands in the table below.

DAPDRPOST *<bits>*
Defines the DAP (Debug Access Port) TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TDI signal and the TAP you are describing. In BYPASS mode, each TAP contributes one data register bit. See possible TAP types and example below.

Default: 0.

DAPDRPRE *<bits>*
Defines the DAP (Debug Access Port) TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TAP you are describing and the TDO signal. In BYPASS mode, each TAP contributes one data register bit. See possible TAP types and example below.

Default: 0.

DAPIRPOST *<bits>*
Defines the DAP (Debug Access Port) TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between TDI signal and the TAP you are describing. See possible TAP types and example below.

Default: 0.

DAPIRPRE *<bits>*
Defines the DAP (Debug Access Port) TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TAP you are describing and the TDO signal. See possible TAP types and example below.

Default: 0.

NOTE: If you are not sure about your settings concerning **DAPIRPRE**, **DAPIRPOST**, **DAPDRPRE**, and **DAPDRPOST**, you can try to detect the settings automatically with**SYStem.DETECT.DaisyChain** or **SYStem.DETECT.SHOWChain**.

Slave [**ON** | **OFF**] If several debuggers share the same debug port, all except one must have this option active.

JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**.

Default: OFF; ON if `CORE=`... >1 in configuration file (e.g. config.t32).

TAPState *<state>* This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.
0 Exit2-DR
1 Exit1-DR
2 Shift-DR
3 Pause-DR
4 Select-IR-Scan
5 Update-DR
6 Capture-DR
7 Select-DR-Scan
8 Exit2-IR
9 Exit1-IR
10 Shift-IR
11 Pause-IR
12 Run-Test/Idle
13 Update-IR
14 Capture-IR
15 Test-Logic-Reset

Default: 7 = Select-DR-Scan.

TCKLevel *<level>* Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target.
Default: 0.

TriState [**ON** | **OFF**] TriState has to be used if several debug cables are connected to a common JTAG port. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode.
Please note:

- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

Default: OFF.

TAP Types:

Core TAP providing access to the debug register of the core you intend to debug.
-> DRPOST, DRPRE, IRPOST, IRPRE.

DAP (Debug Access Port) TAP providing access to the debug register of the core you intend to debug. It might be needed additionally to a Core TAP if the DAP is only used to access memory and not to access the core debug register.
-> DAPDRPOST, DAPDRPRE, DAPIRPOST, DAPIRPRE.

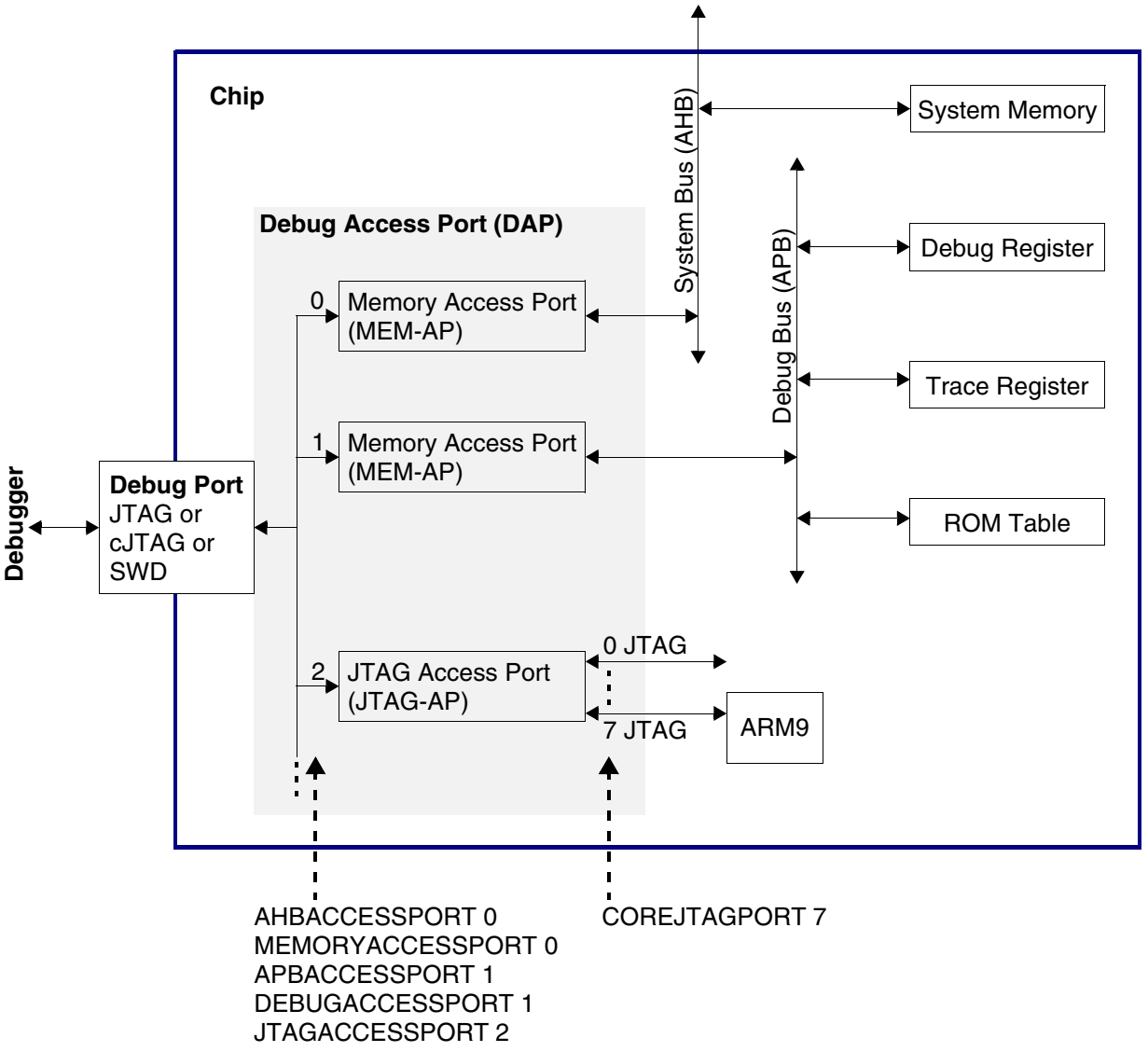
A Debug Access Port (DAP) is a CoreSight module from ARM which provides access via its debugport (JTAG, cJTAG, SWD) to:

1. Different memory busses (AHB, APB, AXI). This is especially important if the on-chip debug register needs to be accessed this way. You can access the memory buses by using certain access es with the debugger commands: "AHB:", "APB:", "AXI:, "DAP", "E:". The interface to these buses is called Memory Access Port (MEM-AP).

2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number between 0 and 7 denotes the JTAG interfaces to be addressed.

3. At emulation or simulation system with using bus transactors the access to the busses must be specified by using the transactor identification name instead using the access port commands. For emulations/simulations with a DAP transactor the individual bus transactor name don't need to be configured. Instead of this the DAP transactor name need to be passed and the regular access ports to the busses.

Example:



AHBACCESSPORT *<port>* DAP access port number (0-255) which shall be used for “AHB:” access class. Default: *<port>*=0.

APBACCESSPORT *<port>* DAP access port number (0-255) which shall be used for “APB:” access class. Default: *<port>*=1.

AXIACCESSPORT *<port>* DAP access port number (0-255) which shall be used for “AXI:” access class. Default: port not available

COREJTAGPORT *<port>* JTAG-AP port number (0-7) connected to the core which shall be debugged.

DEBUGACCESSPORT *<port>*
DAP access port number (0-255) where the debug register can be found (typically on APB). Used for “DAP:” access class. Default: *<port>*=1.

JTAGACCESSPORT *<port>*
DAP access port number (0-255) of the JTAG Access Port.

MEMORYACCESSPORT *<port>*
DAP access port number where system memory can be accessed even during runtime (typically on AHB). Used for “E:” access while running, assuming “SYStem.MemoryAccess DAP”. Default: *<port>*=0.

<parameters> describing debug and trace “Components”

Using the commands on the **COmponents** tab of the **SYStem.CONFIG.state** window , you can add the configurations of base addresses for the IPUS and IPUV cores you want to debug.

IPUSDEBUG.Base {*<address>*}

Base address for debug registers of IPUS (IPU scalar) cores.
This command informs the debugger about the start address of the register block of the component. And this way it notifies the existence of the component. An on-chip debug and trace component typically provides a control register block which needs to be accessed by the debugger to control this component.

Example assuming one IPUS core:
SYStem.CONFIG.IPUSDEBUG D:0x40022000

Meaning: The control register block of the IPUS core #0 starts at address 0x40022000 and is accessible via the data access class D.

For an IPU debugger, the following components are available: IPUSDEBUG, IPUVDEBUG, CTI.

Example assuming four IPUV cores:
SYStem.CONFIG.IPUVDEBUG.Base 0x40042000 0x40043000 0x40044000 0x40045000

IPUVDEBUG.Base {*<address>*}

Base address for debug registers of IPUV (IPU vector) cores.
See **IPUSDEBUG.BASE**

… **.RESET**

Undo the configuration for this component. This does not cause a physical reset for the component on the chip.

CTI.Config *<type>*

Informs about the interconnection of the core Cross Trigger Interfaces (CTI) - ARM CoreSight module. Certain ways of interconnection are common and these are supported by the debugger e.g. to synchronously halt (and sometimes also to start) multiple cores.
For a description of the available types, see **table** below.

CTI.Base *<address>*

Informs the debugger about the start address of the register block of the CTI module.

NONE	The CTI is not used by the debugger.
ARMV1	This mode is used for ARM7/9/11 cores which support synchronous halt, only.
ARMPostInit	Like ARMV1 but the CTI connection differs from the ARM recommendation.
OMAP3	This mode is not yet used.
TMS570	Used for a certain CTI connection used on a TMS570 derivative.
CortexV1	The CTI will be configured for synchronous start and stop via CTI. It assumes the connection of DBGRQ, DBGACK, DBGRESTART signals to CTI are done as recommended by ARM. The CTIBASE must be notified. **CortexV1** is the default value if a Cortex-A/R core is selected and the CTIBASE is notified.
QV1	This mode is not yet used.
ARMV8V1	Channel 0 and 1 of the CTM are used to distribute start/stop events from and to the CTIs. ARMv8 only.
ARMV8V2	Channel 2 and 3 of the CTM are used to distribute start/stop events from and to the CTIs. ARMv8 only.

About the Configuration Examples

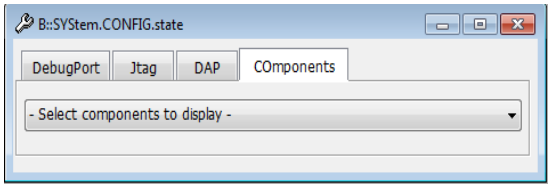
The configuration steps can either be performed via the TRACE32 PowerView GUI, the TRACE32 command line, or a PRACTICE script (*.cmm). Scripting the commands has the advantage that you do not need to manually enter the configuration again for future debug sessions.

Next:

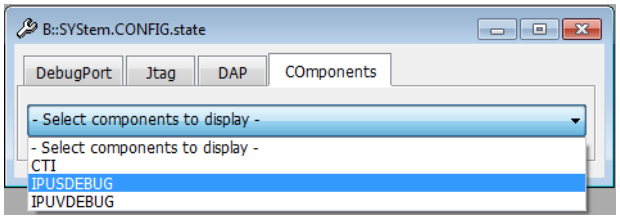
- **"Example - Configuring IPU Components via the GUI"**, page 27
- **"Example - Configuring IPU Components via the TRACE32 Command Line"**, page 28

To configure the base address for one or multiple IPUS cores:

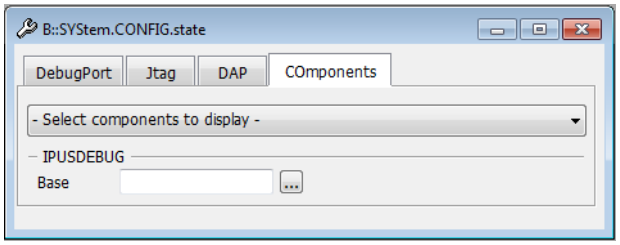
1. Click the **COmponents** tab in the **SYStem.CONFIG.state** window.



2. Select **IPUSDEBUG** from the drop-down list.



A new section named **IPUSDEBUG** appears.

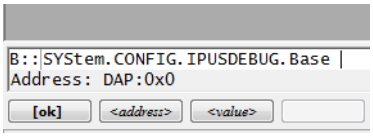


3. Add one or multiple base addresses in the **Base** field.

   - For more details about the syntax, see **"Example - Configuring IPU Components via the TRACE32 Command Line"**, page 28.
   - The button with the three dots copies the corresponding command to the command line. The command can now, for example, be copied and pasted into a PRACTICE script (*.cmm) for re-use.

Example - Configuring IPU Components via the TRACE32 Command Line

Each configuration can be done by a command that is executed in the command line.



You can have several of the following components: IPUSDEBUG and IPUVDEBUG.

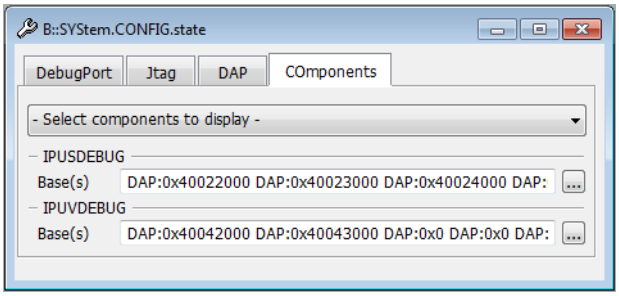
The *<address>* parameter can be just an address (e.g. 0x80001000), or you can add the access class in front (e.g. D:0x40022000). Without access class, the address gets the command specific default access class, which is “DAP:” in most cases.

Configuration of the base addresses of the following IPU cores:

- 4 IPUS cores with base addresses 0x40022000, 0x40023000, 0x40024000, and 0x40025000
- 2 IPUV cores with base addresses 0x40042000 and 0x40043000

```
SYStem.CONFIG.IPUSDEBUG.Base 0x40022000 0x40023000 0x40024000 0x40025000
SYStem.CONFIG.IPUVDEBUG.Base 0x40042000 0x40043000
```

The configuration result can be seen on the **COmponents** tab of the **SYStem.CONFIG.state** window:



Format: **SYStem.CPU** *<cpu>*

<cpu>: **IPU** | **S32V234-IPU**

Selecting S32V234-IPU configures TRACE32 PowerView for debugging IPU cores on an NXP S32V234 board. The debugger automatically loads configuration that is necessary to debug the full cluster of 8 IPUS and 4 IPUV cores.

SYStem.JtagClock Define JTAG frequency

Format: **SYStem.JtagClock** [*<frequency>* | **RTCK**]

<frequency>: **10000.** … **40000000.**
1250000. | **2500000.** | **5000000.** | **10000000.** (on obsolete ICD hardware)

Default frequency: 10 MHz.

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer.

<frequency> The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window.
Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.

ARTCK

The JTAG interface of the IPU does not offer RTCK (**R**eturned **TCK**)**.** However, in multicore applications with ARM, RTCK can be used to control the JTAG clock.

Accelerated method to control the JTAG clock by the RTCK signal **(A**ccelerated **R**eturned **TCK**)**.**

RTCK mode allows theoretical frequencies up to 1/6 (ARM7, ARM9) or 1/8 (ARM11) of the processor clock. For designs using a very low processor clock we offer a different mode (ARTCK) which does not work as recommended by ARM and might not work on all target systems. In ARTCK mode the debugger uses a fixed JTAG frequency for TCK, independent of the RTCK signal. This frequency must be specified by the user and has to be below 1/3 of the processor clock speed. TDI and TMS will be delayed by 1/2 TCK clock cycle. TDO will be sampled with RTCK

CRTCK

The JTAG interface of the IPU does not offer RTCK (**R**eturned **TCK**)**.** However, in multicore applications with ARM, RTCK can be used to control the JTAG clock.

With this option higher JTAG speeds can be reached. The TDO signal will be sampled by the RTCK signal. This compensates the debugger-internal driver propagation delays, the delays on the cable and on the target (**C**ompensation by **RTCK**). This feature requires that the target provides an RTCK signal. In contrast to the **RTCK** option, the TCK is always output with the selected, fixed frequency.

CTCK

The JTAG interface of the IPU does not offer RTCK (**R**eturned **TCK**)**.** However, in multicore applications with ARM, RTCK can be used to control the JTAG clock.

With this option higher JTAG speeds can be reached. The TDO signal will be sampled by a signal which derives from TCK, but which is timely compensated regarding the debugger-internal driver propagation delays (**C**ompensation by **TCK**). This feature can be used with a debug cable versions 3b or newer. If it is selected, although the debug cable is not suitable, a fix JTAG clock will be selected instead (minimum of 10 MHz and selected clock).

RTCK

The JTAG interface of the IPU does not offer RTCK (**R**eturned **TCK**)**.** However, in multicore applications with ARM, RTCK can be used to control the JTAG clock.

On some processor derivatives, there is the need to synchronize the processor clock and the JTAG clock. In this case RTCK shall be selected. Synchronization is maintained, because the debugger does not progress to the next TCK edge until after an RTCK edge is received.

In case you have a processor derivative requiring a synchronization of the processor clock and the JTAG clock, but your target does not provide an RTCK signal, you need to select a fix JTAG clock below 1/6 of the processor clock (ARM7, ARM9), below 1/8 of the processor clock (ARM11), respectively.

When RTCK is selected, the frequency depends on the processor clock and on the propagation delays. The maximum reachable frequency is about 16 MHz.

SYStem.LOCK — Tristate the JTAG port

Format: **SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the JTAG port will be performed by the debugger. While locked the JTAG connector of the debugger is tristated. The intention of the **SYStem.LOCK** command is, for example, to give JTAG access to another tool. The process can also be automated, see **SYStem.CONFIG TriState**.

It must be ensured that the state of the IPU core JTAG state machine remains unchanged while the system is locked. To ensure correct hand-over, the options **SYStem.CONFIG TAPState** and **SYStem.CONFIG TCKLevel** must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

Format: **SYStem.MemAccess** *<mode>*

<mode>: **DAP**
Denied
StopAndGo

Default: Denied.

If **SYStem.MemAccess** is not **Denied**, it is possible to read from memory, to write to memory and to set software breakpoints while the CPU is executing the program.

DAP A run-time memory access is done via a Memory Access Port (MEM-AP) of the Debug Access Port (DAP). This is only possible if a DAP is available on the chip and if the memory bus is connected to it (ARM Cortex, CoreSight). The debugger uses the AXI MEM-AP specified by SYStem.CONFIG AXIACCESSPORT if available, the MEM-AP (typically AHB) specified by SYStem.CONFIG MEMORYACCESSPORT otherwise.

Denied No memory access is possible while the CPU is executing the program.

StopAndGo Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.
For more information, see below.

If specific windows that display memory or variables should be updated while the program is running, select the memory access class E: or the format option **%E**.

```
Data.dump E:0x100

Var.View %E first
```

Format: **SYStem.Mode** <*mode*>

<*mode*>: **Down**
Attach
Up

Down Disables the debugger (default). The state of the CPU remains unchanged. The JTAG port is tristated.

Attach User program remains running (no reset) and the debug mode is activated. After this command the user program can be stopped with the break command or if any break condition occurs.

Up Resets the target, sets the CPU to debug mode and stops the CPU. After the execution of this command the CPU is stopped and all register are set to the default level.

SYStem.Option.AHBHPROT Select AHB-AP HPROT bits

Format: **SYStem.Option.AHBHPROT** <*value*>

Default: 0

Selects the value used for the HPROT bits in the Control Status Word (CSW) of an AHB Access Port of a DAP, when using the AHB: memory class.

SYStem.Option.AXIACEEnable ACE enable flag of the AXI-AP

Format: **SYStem.Option.AXIACEEnable** [**ON** | **OFF**]

Default: OFF.

Enables ACE transactions on the DAP AXI-AP, including barriers. This does only work if the debug logic of the target CPU implements coherent AXI accesses. Otherwise this option will be without effect.

Format: **SYStem.Option.AXICACHEFLAGS** *<value>*

<value>:
DeviceSYStem
NonCacheableSYStem
ReadAllocateNonShareable
ReadAllocateInnerShareable
ReadAllocateOuterShareable
WriteAllocateNonShareable
WriteAllocateInnerShareable
WriteAllocateOuterShareable
ReadWriteAllocateNonShareable
ReadWriteAllocateInnerShareable
ReadWriteAllocateOuterShareable

Default: DeviceSYStem (=0x30: Domain=0x3, Cache=0x0)

This option configures the value used for the Cache and Domain bits in the Control Status Word (CSW[27:24]->Cache, CSW[14:13]->Domain) of an AXI Access Port of a DAP, when using the AXI: memory class.

The below offered selection options are all non-bufferable. Alternatively you can enter a <value>, where value[5:4] determines the Domain bits and value[3:0] the Cache bits.

DeviceSYStem	=0x30: Domain=0x3, Cache=0x0
NonCacheableSYStem	=0x32: Domain=0x3, Cache=0x2
ReadAllocateNonShareable	=0x06: Domain=0x0, Cache=0x6
ReadAllocateInnerShareable	=0x16: Domain=0x1, Cache=0x6
ReadAllocateOuterShareable	=0x26: Domain=0x2, Cache=0x6
WriteAllocateNonShareable	=0x0A: Domain=0x0, Cache=0xA
WriteAllocateInnerShareable	=0x1A: Domain=0x1, Cache=0xA
WriteAllocateOuterShareable	=0x2A: Domain=0x2, Cache=0xA
ReadWriteAllocateNonShareable	=0x0E: Domain=0x0, Cache=0xE
ReadWriteAllocateInnerShareable	=0x1E: Domain=0x1, Cache=0xE
ReadWriteAllocateOuterShareable	=0x2E: Domain=0x2, Cache=0xE

Format: **SYStem.Option.AXIHPROT** *<value>*

Default: 0

This option selects the value used for the HPROT bits in the Control Status Word (CSW) of an AXI Access Port of a DAP, when using the AXI: memory class.

SYStem.Option.DAPNOIRCHECK No DAP instruction register check

Format: **SYStem.Option.DAPNOIRCHECK** [**ON** | **OFF**]

Default: OFF.

Bug fix for derivatives which do not return the correct pattern on a DAP (Arm CoreSight Debug Access Port) instruction register (IR) scan. When activated, the returned pattern will not be checked by the debugger.

SYStem.Option.DAPREMAP Rearrange DAP memory map

Format: **SYStem.Option.DAPREMAP** {*<address_range> <address>*}

The Debug Access Port (DAP) can be used for memory access during runtime. If the mapping on the DAP is different than the processor view, then this re-mapping command can be used

NOTE:	Up to 16 *<address_range>*/*<address>* pairs are possible. Each pair has to contain an address range followed by a single address.

Format: **SYStem.Option.DAPDBGPWRUPREQ** [**ON** | **AlwaysON** | **OFF**]

Default: ON.

This option controls the DBGPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port (DAP) before and after the debug session. Debug power will always be requested by the debugger on a debug session start because debug power is mandatory for debugger operation.

ON Debug power is requested by the debugger on a debug session start, and the control bit is set to 1.
The debug power is released at the end of the debug session, and the control bit is set to 0.

AlwaysON Debug power is requested by the debugger on a debug session start, and the control bit is set to 1.
The debug power is **not** released at the end of the debug session, and the control bit is set to 0.

OFF Only for test purposes: Debug power is **not** requested and **not** checked by the debugger. The control bit is set to 0.

Use case:

Imagine an AMP session consisting of at least of two TRACE32 PowerView GUIs, where one GUI is the master and all other GUIs are slaves. If the master GUI is closed first, it releases the debug power. As a result, a debug port fail error may be displayed in the remaining slave GUIs because they cannot access the debug interface anymore.

To keep the debug interface active, it is recommended that **SYStem.Option.DAPDBGPWRUPREQ** is set to **AlwaysON**.

SYStem.Option.DAPSYSPWRUPREQ Force system power in DAP

Format: **SYStem.Option.DAPSYSPWRUPREQ** [**AlwaysON** | **ON** | **OFF**]

Default: ON.

This option controls the SYSPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port (DAP) during and after the debug session

AlwaysON	System power is requested by the debugger on a debug session start, and the control bit is set to 1. The system power is **not** released at the end of the debug session, and the control bit remains at 1.
ON	System power is requested by the debugger on a debug session start, and the control bit is set to 1. The system power is released at the end of the debug session, and the control bit is set to 0.
OFF	System power is **not** requested by the debugger on a debug session start, and the control bit is set to 0.

SYStem.Option.DEBUGPORTOptions Options for debug port handling

Format:	**SYStem.Option.DEBUGPORTOptions** *<option>*
<option>:	**SWITCHTOSWD.**[**TryAll** | **None** | **JtagToSwd** | **LuminaryJtagToSwd** | **DormantToSwd** | **JtagToDormantToSwd**] **SWDTRSTKEEP.**[**DEFault** | **LOW** | **HIGH**]

Default: SWITCHTOSWD.TryAll, SWDTRSTKEEP.DEFault.

See Arm CoreSight manuals to understand the used terms and abbreviations and what is going on here.

SWITCHTOSWD tells the debugger what to do in order to switch the debug port to serial wire mode:

TryAll	Try all switching methods in the order they are listed below. This is the default. Normally it does not hurt to try improper switching sequences. Therefore this succeeds in most cases.
None	There is no switching sequence required. The SW-DP is ready after power-up. The debug port of this device can only be used as SW-DP.
JtagToSwd	Switching procedure as it is required on SWJ-DP without a dormant state. The device is in JTAG mode after power-up.
LuminaryJtagToSwd	Switching procedure as it is required on devices from LuminaryMicro. The device is in JTAG mode after power-up.

DormantToSwd	Switching procedure which is required if the device starts up in dormant state. The device has a dormant state but does not support JTAG.
JtagToDormantToSwd	Switching procedure as it is required on SWJ-DP with a dormant state. The device is in JTAG mode after power-up.

SWDTRSTKEEP tells the debugger what to do with the nTRST signal on the debug connector during serial wire operation. This signal is not required for the serial wire mode but might have effect on some target boards, so that it needs to have a certain signal level.

DEFault	Use nTRST the same way as in JTAG mode which is typically a low-pulse on debugger start-up followed by keeping it high.
LOW	Keep nTRST low during serial wire operation.
HIGH	Keep nTRST high during serial wire operation

SYStem.state — Display SYStem.state window

Format: **SYStem.state**

Displays the **SYStem.state** window for system settings that configure debugger and target behavior.

The **TrOnchip** command provides low-level access to the on-chip debug register.

TrOnchip.Set.FINISH Set "Break on Finish" on-chip breakpoint

Format: **TrOnchip.Set.FINISH** [**ON** | **OFF**]

Activates the ‘Break on Finish’ on-chip breakpoint.

The CPU stops the application execution at the ‘Break on Finish’ on-chip breakpoint after the processing of a line is finished and a line-done event was generated by the pixel-done instruction.

TrOnchip.Set.POS Set on-chip trigger for total pixel position

Format: **TrOnchip.Set.POS** [**ON** | **OFF**]

Activates the ‘Pixel Position’ on-chip breakpoint.

The CPU stops the application execution at the ‘Pixel Position’ on-chip breakpoint when any instruction is executed at a certain pixel position. The pixel position must be pre-defined by the **TrOnchip.Set.XPOS** and **TrOnchip.Set.YPOS** commands. Application execution is stopped when the values for both XPOS and YPOS match.

TrOnchip.Set.XPOS Set on-chip trigger for horizontal pixel position

Format: **TrOnchip.Set.XPOS** [**ON** | **OFF** | *<value>*]

Activates the on-chip breakpoint for horizontal pixel position.

The CPU stops the application execution when any instruction is executed at a pixel with the number defined by *<value>*.

Format: **TrOnchip.Set.YPOS** [**ON** | **OFF** | *<value>*]

Activates the on-chip breakpoint for vertical pixel position.

The CPU stops the application execution when any instruction is executed in a pixel line with the number defined by *<value>*.

TrOnchip.RESet Set on-chip trigger to default state

Format: **TrOnchip.RESet**

Sets the TrOnchip settings and trigger module to the default settings.

TrOnchip.state Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.