SDMA Debugger

©1989-2020 Lauterbach GmbH

©1989-2020 Lauterbach GmbH

Version 18-Nov-2020

19-Mar-20	Draft of new manual.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

Introduction

This manual serves as a guideline for debugging SDMA cores and describes all processor-specific TRACE32 settings and features.

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.

Currently SDMA cores are only implemented in several i.MX Chips. Before debugging SDMA cores, the chip's main core must ensure that the clock signal is available to the SDMA(OnCE) core.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Basics - Training"** (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:

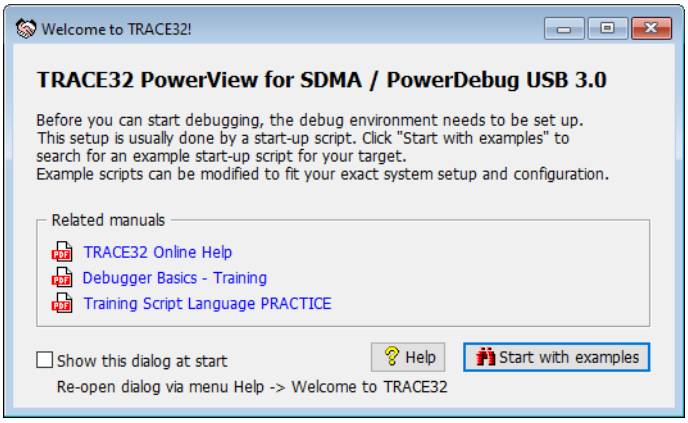
- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:

  - 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.

To get started with the most important manuals, use the **Welcome to TRACE32!** dialog (**WELCOME.view**):

©1989-2020 Lauterbach GmbH



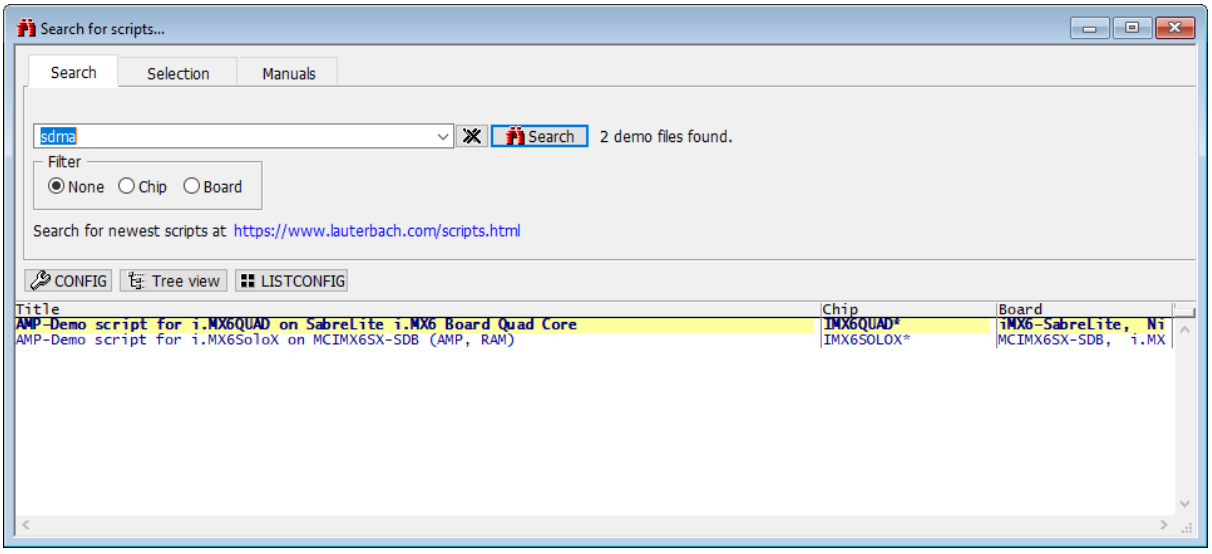
Demo and Start-up Scripts

Lauterbach provides ready-to-run PRACTICE start-up scripts for public known architecture hardware.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

  You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software:



©1989-2020 Lauterbach GmbH

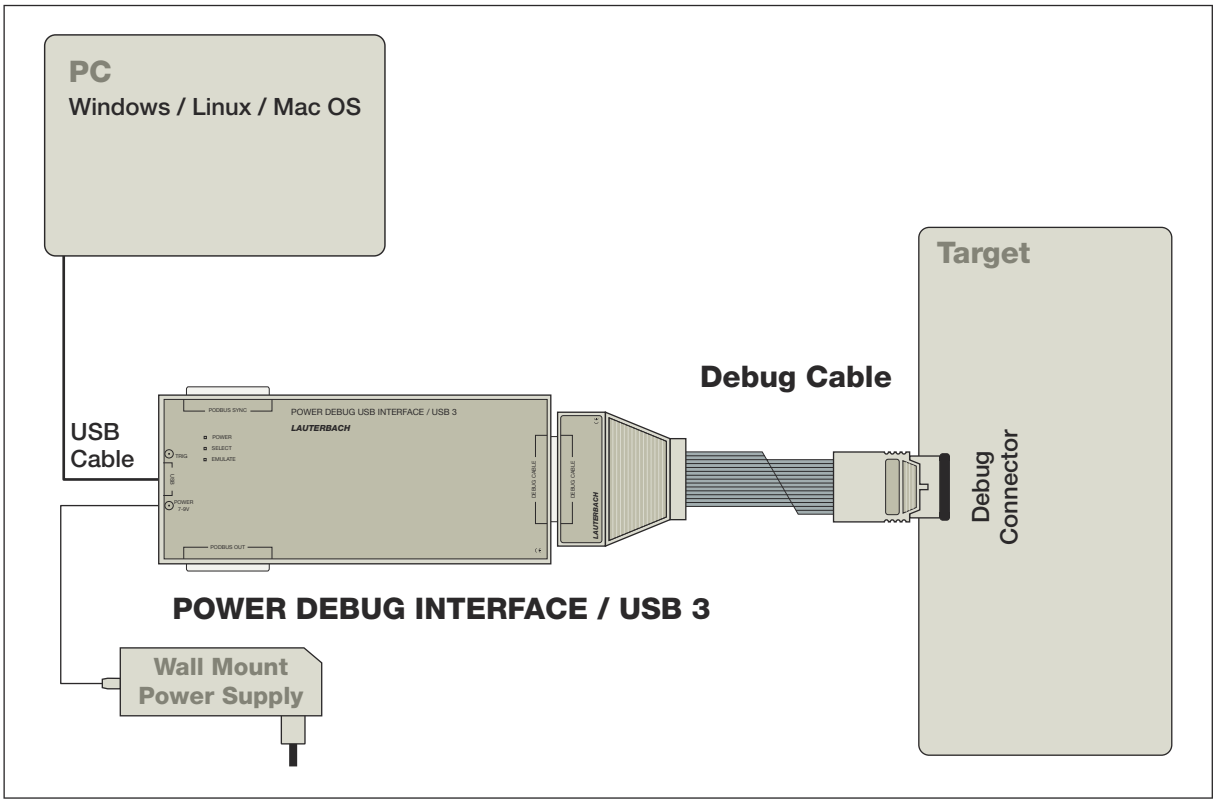
You can also inspect the demo folder manually in the system directory of TRACE32.
The `~~/demo/sdma/` folder contains:

hardware/	Ready-to-run debugging demos for evaluation boards. **Recommended for getting started!**

©1989-2020 Lauterbach GmbH

System Overview

Example configuration for a single core debugger.



Please consider the tips given in the chapter **"Connector Type and Pinout"**, page 34.

©1989-2020 Lauterbach GmbH

Quick Start of the Debugger

This chapter helps you to prepare your Debugger for SDMA. Depending on your application not all steps might be necessary. It is assumed that you are using an i.MX 6SoloX on an NXP SABRE Board.

For some applications additional steps might be necessary that are not described here. See **Demo and Start-up Scripts** for more details.

1. Prepare the Start.

   Connect the Debug Cable to your target. Check the orientation of the connector. Pin 1 of the Debug Cable is marked with a small triangle or the number 1.

   Start a PowerView instance for the host-core (Cortex-A9 in this example) and for SDMA:

   - Start the TRACE32 Debugger Software for Cortex-A9. Make sure the configuration file for the ARM instance (e.g. `config_cortex.t32`) contains the line `CORE=1`.
   - Start the TRACE32 Debugger Software for SDMA. Make sure the configuration file for the

©1989-2020 Lauterbach GmbH

SDMA instance (e.g. `config_sdma.t32`) contains the line `CORE=2`.

Power up your target.

2. Configure the master core ICD for Debugging

   Refer to **"ARMv7-A/-R Debugger"** (debugger_arm.pdf) for information on how to do this.

   Since SDMA needs to be initialized by the master core, execute your application so that SDMA code and data is loaded into SDMA memory. This is often done by executing the function `sdma_init` or similar.

   Remember that these steps have to be performed on the master core instance.

3. Select the Device prompt B: for the ICD Debugger

   ```
   B::
   ```

   On all TRACE32 tool configurations except for the emulator device B:: is already selected.

4. Select the CPU Type to load the CPU specific Settings.

   ```
   SYStem.CPU IMX6SOLOX-SDMA
   ```

   It is strongly recommended to select the specific CPU instead of SDMA which is only dedicated for hardware configurations not known by TRACE32. In such cases the user has to create a target-specific **SYStem.CONFIG.MULTITAP.JtagSEQuence** to add the SDMA core to the JTAG chain.

5. Establish the communication to the device.

   ```
   SYStem.Attach
   ```

   Enter debug mode. After this command is executed, it is possible to access memory and registers.

   ICD SDMA requires an ICD ARM which properly configures the system's clock signals. Access to the SDMA core will fail otherwise.

6. Load symbols for your Application Program. (optional)

   If available, a file containing symbol information can be loaded now. Since the SDMA code is normally written to the RAM by the main core, only the symbols have to be loaded.

   ```
   Data.LOAD.auto <filename> /NoCODE /NOREG
   ```

   The options of the **Data.LOAD** command depend on the file format generated by the compiler. A detailed description of the **Data.LOAD** command is given in **"General Commands Reference"**.

7. Write a Start-up script.

   Now the quick start is done. If you were successful you can start to debug. It is recommended to prepare a PRACTICE script file (*.cmm, ASCII format) to be able to do all the necessary actions with only one command.

©1989-2020 Lauterbach GmbH

Here is a typical start sequence:

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

System.CPU IMX6SOLOX-SDMA           ; select CPU

SYStem.Attach                       ; Establish communication to device

Data.LOAD.auto script.elf           ; Load the symbol information
/NoCODE /NOREG

WinCLEAR                            ; Clear all windows

List.Mix                            ; Open source window   *)

Register.view                       ; Open register window *)
```

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

For information about how to create a PRACTICE script file (*.cmm file), refer to **"Debugger Basics - Training"** (training_debugger.pdf). There you can also find some information on basic actions with the debugger.

©1989-2020 Lauterbach GmbH

Error Message	Event	Reason
Target power fail	**SYStem.Mode.Up**	See below.
No clock signal detected.	**SYStem.Mode.Up**	See below.
Target processor in reset	**SYStem.Down**	See below.
The number of *<number>* accessed bytes in memory is not a multiple of the access size *<size>* bytes.	No special event	Internal error, please consult your Lauterbach representative.
Memory address *<address>* is not aligned to access size *<size>*.	No special event	Internal error, please consult your Lauterbach representative.
Invalid memory access size: *<size>* bytes (@ address *<address>*)	No special event	Internal error, please consult your Lauterbach representative.
Memory access timeout: Reading from address *<address>*	No special event	Corrupted debug connection. Check debug hardware and settings.
Emulation running	**Break**	The master core has set the SDMA to Sleep mode (no clock) so the SDMA core seems to be continuously running.

Communication between Debugger and Processor can not be established

Typically the **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages like “debug port fail” or “debug port time out” hile executing this command, this may have the reasons described below. “target processor in reset” is just a follow-up error message.

©1989-2020 Lauterbach GmbH

Open the **AREA** window to view all error messages.

- The target has no power or the debug cable is not connected to the target. This results in the error message “target power fail”.
- You did not select the correct core type via **SYStem.CPU**.
- The target to **Debug Cable** connection is not valid for the selected CPU. See **SYStem.CPU**.
- There is an issue with the JTAG interface. See **“ARM JTAG Interface Specifications”** (app_arm_jtag.pdf) and the manuals or schematic of your target to check the physical and electrical interface. Maybe there is the need to set jumpers on the target to connect the correct signals to the JTAG connector.
- The target core is not part of the JTAG chain. If CPU type SDMA is selected, the user has to create a target-specific **SYStem.CONFIG.MULTITAP.JtagSEQuence** which adds the SDMA core to the JTAG chain.
- There is the need to enable (jumper) the debug features on the target. It will e.g. not work if nTRST signal is directly connected to ground on target side.
- The target is in an unrecoverable state. Re-power your target and try again.
- The target can not communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by “Break” instead of **SYStem.Up**.
- The default frequency of the JTAG/SWD/cJTAG debug port is too high, especially if you emulate your core or if you use an FPGA-based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- Your core needs adaptive clocking. Use the RTCK mode: **SYStem.JtagClock RTCK**.
- The core is used in a multicore system and the appropriate multicore settings for the debugger are missing. See for example **SYStem.CONFIG IRPRE**. This is the case if you get a value IR_Width > 5 when you enter “DIAG 3400” and “AREA”. If you get IR_Width = 4 (ARM7, ARM9, Cortex) or IR_Width = 5 (ARM11), then you have just your core and you do not need to set these options. If the value can not be detected, then you might have a JTAG interface issue.
- The core has no clock.
- The core is kept in reset.
- There is a watchdog which needs to be deactivated.

Your target needs special debugger settings. Check the directory ~~\demo\arm\hardware if there is an suitable script file *.cmm for your target.

FAQ

Please refer to our Frequently Asked Questions page on the Lauterbach website.

©1989-2020 Lauterbach GmbH

Memory Classes

Though the SDMA architecture uses the same address space for data, program and peripheral memory, the addressing and access for both types is different. Therefore, following memory access classes are available:

Access Class	Description
D	Data
P	Program
PER	Peripheral devices

To access a memory class, write the class in front of the address. For example, use D to access the data memory:

```
Data.dump D:0x00
```

The following examples return different results, since the dsPIC architecture uses different addressing modes.

```
Data.dump D:0x100
```

```
Data.dump P:0x100
```

NOTE: To access the peripheral memory space (data addresses > 0x1000), the respective peripheral devices must be provided with the system clock by the master core. Otherwise the System will be reset.

Breakpoints

TRACE32 uses two techniques to implement breakpoints: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are only available for program breakpoints. If a program breakpoint is set to an instruction, the original instruction at the breakpoint location is patched by a break code. This patching is the reason why software breakpoints are usually only used in RAM areas.

©1989-2020 Lauterbach GmbH

On-chip Breakpoints

On-chip breakpoints are only available for data breakpoints. They are used to analyze the read and write accesses to global variables. The data breakpoints can be triggered with respect to the data address or access type, i.e. read, write or both, or the data value. The data breakpoints are especially useful to find out when a global variable is written with a certain value. It is not possible to implement a similar breakpoint in software without affecting the real-time behavior of the system. Since the load and store instructions work on RAM, data breakpoints always use the Data memory class. Up to two on-chip breakpoints are available on SDMA cores.

On-chip Trace

The SDMA core devices are equipped with an on-chip trace buffer. This allows to analyze the most recent program branches since the last halt. On-chip tracing requires no extra Lauterbach hardware, it can be configured and read out with a regular Debugger.

The program flow trace has no influence on the performance of program execution and is always active when the core is running.

Special Hints, Restrictions, and Known Problems

Special Hints

- Peripheral memory locations are only accessible if the respective peripheral is activated by the master core.

Restrictions

- Assembler loops containing only 1 instruction are executed completely when stepping through the code.
- The SDMA core can only be reset by the master core.

Known Problems

NOTE:	All problems will be fixed in one of the next SW versions without notice!

©1989-2020 Lauterbach GmbH

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state** [/*<tab>*]
<tab>:	**DebugPort | JTAG | MultiTap**

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 — 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 (default) — 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**.

MultiTap — Informs the debugger about the existence and type of a System/Chip Level Test Access Port. The debugger might need to control it in order to reconfigure the JTAG chain or to control power, clock, reset, and security of different chip components.

For descriptions of the commands on the **MultiTap** tab, see **Multitap**.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>:	**CORE** <core> <chip>
(DebugPort)	**DEBUGPORT [DebugCable0 | DebugCableA | DebugCableB]**
	DEBUGPORTTYPE [JTAG | SWD]
	Slave [ON | OFF]
	SWDPIdleHigh [ON | OFF]
	SWDPTargetSel <value>
	TriState [ON | OFF]
<parameter>:	**DRPOST** <bits>
(JTAG)	**DRPRE** <bits>
	IRPOST <bits>
	IRPRE <bits>
<parameter>:	**DAPDRPOST** <bits>
(JTAG cont.)	**DAPDRPRE** <bits>
	DAPIRPOST <bits>
	DAPIRPRE <bits>
<parameter>:	**Slave [ON | OFF]**
(JTAG cont.)	**TAPState** <state>
	TCKLevel <level>
	TriState [ON | OFF]
<parameter>:	**MULTITAP [NONE | JtagSEQuence** <sub_cmd>**]**
(Multitap)	
<parameter>:	**COREJTAGPORT** <port>
(AccessPorts)	**JTAGACCESSPORT** <port>

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

Ideally you can select with **SYStem.CPU** the chip you are using which causes all setup you need and you do not need any further **SYStem.CONFIG** command.

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**.

©1989-2020 Lauterbach GmbH

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

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> derived 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,...).

DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]

It specifies which probe cable shall be used e.g. “DebugCableA” or “DebugCableB”. 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” or “SWD”. It assuems the selected type is supported by the target.

Default: JTAG

©1989-2020 Lauterbach GmbH

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 ON**.

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

SWDPIdleHigh [**ON** | **OFF**]

Keep SWDIO line high when idle. Only for Serialwire Debug mode. Usually the debugger will pull the SWDIO data line low, when no operation is in progress, so while the clock on the SWCLK line is stopped (kept low).

You can configure the debugger to pull the SWDIO data line high, when no operation is in progress by using
SYStem.CONFIG SWDPIdleHigh ON
Default: OFF.

SWDPTargetSel <value>

Device address in case of a multidrop serial wire debug port.

Default: none set (any address accepted).

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.

©1989-2020 Lauterbach GmbH

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.

… **DRPOST** *<bits>*	Defines the 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.
… **DRPRE** *<bits>*	Defines the 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.
… **IRPOST** *<bits>*	Defines the 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.
… **IRPRE** *<bits>*	Defines the 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 **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**, you can try to detect the settings automatically with the **SYStem.DETECT.DaisyChain** command.

©1989-2020 Lauterbach GmbH

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.
Default: ON if `CORE=... >1` in the 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.

©1989-2020 Lauterbach GmbH

TAP types:

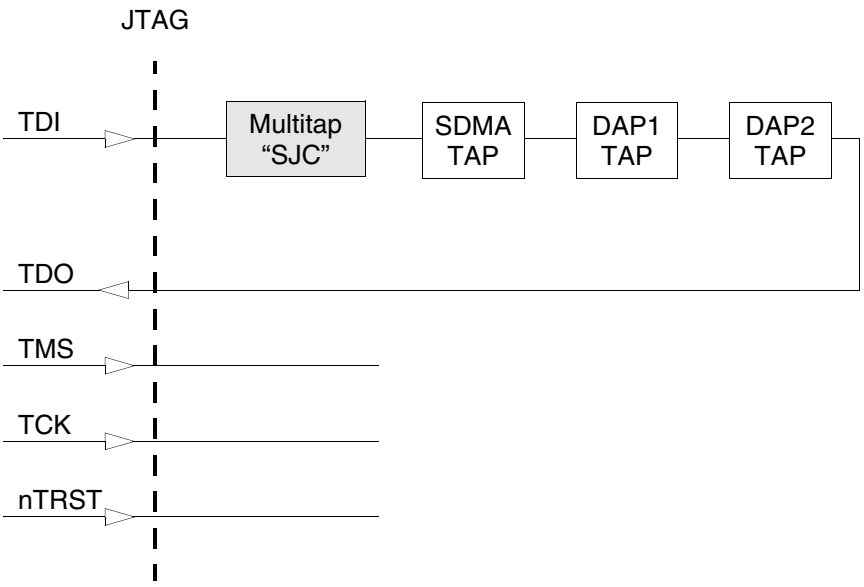
Core TAP providing access to the debug register of the core you intend to debug.
-> DRPOST, DRPRE, IRPOST, IRPRE.

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.

©1989-2020 Lauterbach GmbH

A “Multitap” is a system level or chip level test access port (TAP) in a JTAG scan chain. It can for example provide functions to re-configure the JTAG chain or view and control power, clock, reset and security of different chip components.

Example:



MULTITAP
[**NONE** |
JtagSEQuence <*sub_cmd*>]

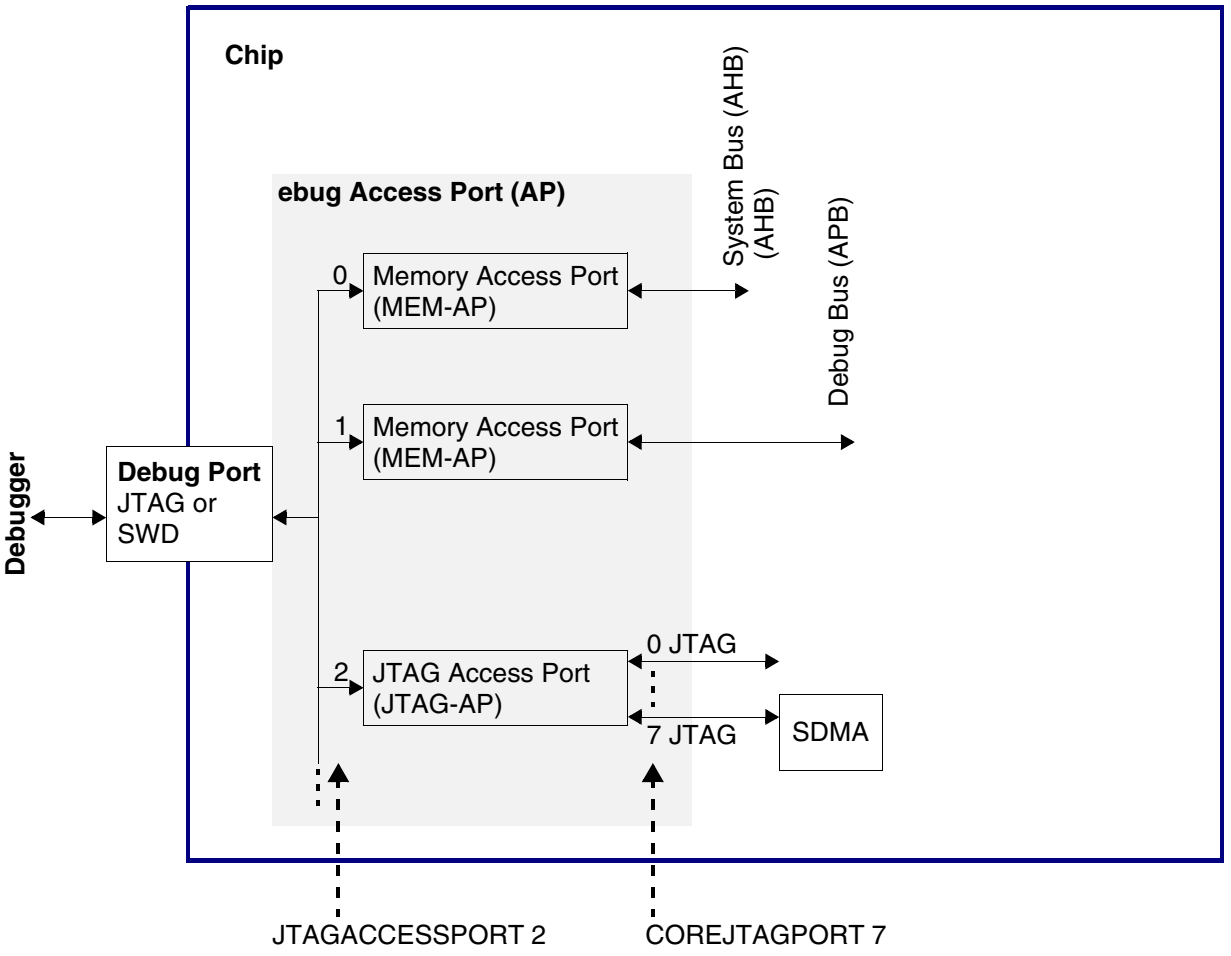
Selects the type and version of the MULTITAP.

For a description of the **JtagSEQuence** subcommands, see **SYStem.CONFIG.MULTITAP JtagSEQuence**.

©1989-2020 Lauterbach GmbH

In some Controllers with SDMA cores, a Debug Access Port (DAP) is used instead of a JTAG scan chain to communicate with the different cores. A DAP is a CoreSight module from ARM which e.g. provides access via its debugport (JTAG, SWD) to other, chip-internal JTAG interfaces. The module controlling such an interface 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.

Example:



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

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

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**SDMA | IMX6ULL-SDMA | IMX6ULTRALITE-SDMA | ...**

Default: SDMA

Selects the processor type and corresponding instruction set for disassembler and in-line assembler. In addition, the command selects the JTAG configurations and configures the JTAG sequence for adding the SDMA core to the chain.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.JtagClock** <*frequency*>
<*frequency*>:	**4 kHz**…**100 MHz**

Default frequency: 1.0 MHz.

Selects the frequency (TCK/SWCLK) used by the debugger to communicate with the processor in JTAG mode. 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 debug port signals or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer. Therefore we recommend to use the default setting if possible.

<*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.

SYStem.LOCK Lock and tristate the debug port

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

Default: OFF.

If the system is locked, no access to the debug port will be performed by the debugger. While locked, the debug connector of the debugger is tristated. The main intention of the **SYStem.LOCK** command is to give debug access to another tool. The command has no effect for the simulator.

©1989-2020 Lauterbach GmbH

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

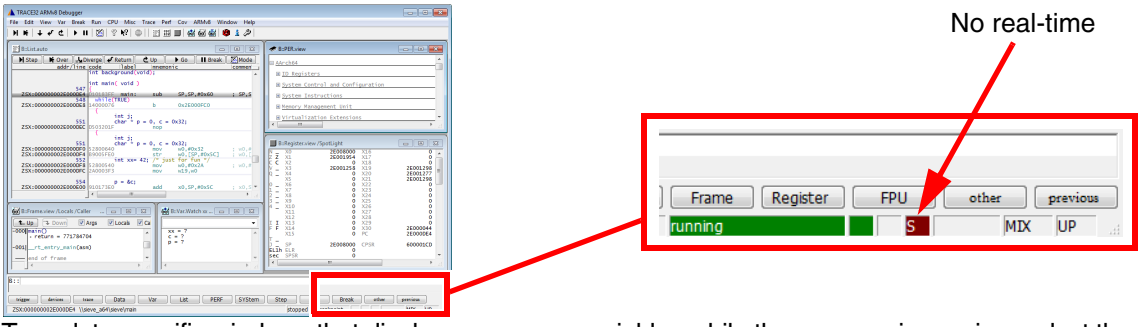
<mode>: **Denied**
StopAndGo

Default: Denied.

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

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

If **SYStem.MemAccess StopAndGo** is set, it is possible to read from memory, to write to memory and to set software breakpoints while the CPU is executing the program. To make this possible, the program execution is shortly stopped by the debugger. Each stop takes some time depending on the currently active debug port clock speed and the operations that should be performed. A white S against a red background in the TRACE32 state line warns you that the program is no longer running in real-time:



To update specific windows that display memory or variables while the program is running, select the memory class **E:** or the format option **%E**.

```
Data.dump E:0x100

Var.View %E first
```

©1989-2020 Lauterbach GmbH

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

<mode>: **Down**
NoDebug
Go
Attach
StandBy
Up

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

NoDebug Disables the debugger. The state of the CPU remains unchanged. The JTAG port is not tristated.

Go Resets the target via the reset line, initializes the debug port and starts the program execution. For a reset, the reset line has to be connected to the debug connector.
Program execution can, for example, be stopped by the **Break** command.

Attach No reset happens, the mode of the core (running or halted) does not change. The debug port will be initialized. After this command has been executed, the user program can, for example, be stopped by the **Break** command.

StandBy Keeps the target in reset via the reset line and waits until power is detected. For a reset, the reset line has to be connected to the debug connector.

Once power has been detected, the debugger restores as many debug registers as possible (e.g. on-chip breakpoints) and releases the CPU from reset to start the program execution.

When a CPU power-down is detected, the debugger switches automatically back to the **StandBy** mode. This allows debugging of a power cycle because debug registers will be restored on power-up.

NOTE: usually only on-chip breakpoints and vector catch events can be set while the CPU is running. To set a software breakpoint, the CPU has to be stopped.

Up Resets the target via the reset line, initializes the debug port, stops the CPU, and enters debug mode.
For a reset, the reset line has to be connected to the debug connector.
The current state of all registers is read from the CPU.

©1989-2020 Lauterbach GmbH

The **SYStem.Option** commands are used to control special features of the debugger or emulator or to configure the target. It is recommended to execute the **SYStem.Option** commands **before** the emulation is activated by a **SYStem.Up** or **SYStem.Mode** command.

SYStem.Option DAPDBGPWRUPREQ Force debug power in DAP

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**.

©1989-2020 Lauterbach GmbH

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.

SYStem.Option DAPSYSPWRUPREQ Force system power in DAP

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

Default: ON.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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.Option DUALPORT Implicitly use run-time memory access

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

All TRACE32 windows that display memory are updated while the processor is executing code (e.g. **Data.dump**, **Data.List**, **PER.view**, **Var.View**). This setting has no effect if **SYStem.MemAccess** is disabled.

If only selected memory windows should update their content during runtime, leave **SYStem.Option DUALPORT OFF** and use the access class prefix **E** or the format option **%E** for the specific windows.

SYStem.Option IMASKASM Disable interrupts while single stepping

[SYStem.state window > IMASKASM]

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

©1989-2020 Lauterbach GmbH

SYStem.Option IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

©1989-2020 Lauterbach GmbH

Target Adaption

Probe Cables

For debugging two kind of probe cables can be used to connect the debugger to the target: “Debug Cable” and “CombiProbe”.

Though the probe cables support several standards and the host core additionally supports cJTAG and SWD, the debugport type JTAG is mandatory if the SDMA core should be debugged.

Connector Type and Pinout

Debug Cable

Adaption for ARM Debug Cable: See **http://www.lauterbach.com/adarmdbg.html**.

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints refer to **“ARM JTAG Interface Specifications”** (app_arm_jtag.pdf).

CombiProbe

Adaption for ARM CombiProbe: See **http://www.lauterbach.com/adarmcombi.html**.

If you use more than one CombiProbe cable (twin cable is no standard delivery) you need to specify which one you want to use by **SYStem.CONFIG DEBUGPORT [DebugCableA | DebugCableB]**. The CombiProbe can detect the location of the cable if only one is connected.

©1989-2020 Lauterbach GmbH