LAUTERBACH
DEVELOPMENT TOOLS

Andes Debugger

MANUAL

Andes Debugger

Andes Debugger

Version 06-Jun-2024

History

20-Jul-22	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.

Introduction

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.

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known Andes based 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.

You can also manually navigate in the ~~/`demo/andes/` subfolder of the system directory of TRACE32.

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 0x00000000++0x07ffffff
```

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. 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.

5. Load the program.

```
Data.LOAD.ELF sieve.elf      ; .ELF specifies the format
                             ; sieve.elf is the file name
```

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 Reference Guide"**.

A typical start sequence is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

```
WinCLEAR                         ; Clear all windows

SYStem.CPU N1213S                ; Select the core type

MAP.BOnchip 0++0x07ffffff        ; Specify where FLASH/ROM is

SYStem.Up                        ; Reset the target and enter debug mode

Data.LOAD.ELF sieve.elf          ; Load the application

Register.Set pc main             ; Set the PC to function main

Register.Set r31 0x10f00000      ; Set the stack pointer to address
                                 ; 0x10f00000

WinPOS 0. 0.                     ; Position of next window
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 flags ast              ; Open watch window for variables *)

Break.Set 0x10001000 /Program    ; Set software breakpoint to address
                                 ; 10001000 (address outside of BOnchip
                                 ; range)

Break.Set 0x40000 /Program       ; Set on-chip breakpoint to address
                                 ; 40000 (address within BOnchip range)
```

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

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” while executing this command, this may have the reasons below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to see 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 **SYStem.CPU** <*type*>.
- There is an issue with the JTAG interface. See 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.
- 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 JTAG clock speed is too fast, 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.
- The core is used in a multicore system and the appropriate settings for the debugger are missing. See for example **SYStem.CONFIG IRPRE**. This is the case if you get a value IR_Width > 4 when you enter “DIAG 16001” and “AREA”. If you get IR_Width = 4, 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\andes if there is an suitable script file *.cmm for your target.

FAQ

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

AndesCore Specific Implementations

Registers

In addition to the normal register, TRACE32 implements the following pseudo-registers corresponding to the fields of the PSW: CPL, IFCOM, DME, IME, DT, IT, BE, POM, INTL, GIE.

These pseudo-registers can be used to easily inspect or modify the corresponding fields in the PSW via the **Register.Set** command.

```
Register.Set GIE 1   ; set the Global Interrupt Enable flag in PSW
```

Breakpoints

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is patched by a breakpoint code.

There is no restriction in the number of software breakpoints.

On-chip Breakpoints for Instructions

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the CPU. On-chip breakpoints are usually needed for instructions in FLASH/ROM.

With the command **MAP.BOnchip** *<range>* it is possible to tell the debugger where you have ROM / FLASH on the target. If a breakpoint is set into a location mapped as BOnchip one on-chip breakpoint will be used.

On-chip Breakpoints for Data

To stop the CPU after a read or write access to a memory location on-chip breakpoints are required. Overview:

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction breakpoints:** Number of on-chip breakpoints that can be used to set program breakpoints into ROM/FLASH/EPROM.
- **Read/Write breakpoints:** Number of on-chip breakpoints that can be used as Read or Write breakpoints.
- **Data breakpoint:** Number of on-chip data breakpoints that can be used to stop the program when a specific data value is written to an address or when a specific data value is read from an address

On-chip Breakpoints	Instruction Breakpoints	Read/Write Breakpoints	Data Breakpoint
up to 8	up to 8	up to 8 range as bitmask	up to 8

Example for Standard Breakpoints

Assume you have a target with

- FLASH from `0x0--0x07ffffff`
- SDRAM from `0x10000000--0x4fffffff`

The command to configure TRACE32 correctly for this configuration is:

```
Map.BOnchip 0x0--0x07ffffff
```

The following standard breakpoint combinations are possible.

1. Unlimited breakpoints in RAM and one breakpoint in ROM/FLASH

```
Break.Set 0x10000000 /Program          ; Software breakpoint 1
Break.Set 0x10001000 /Program          ; Software breakpoint 2
Break.Set addr /Program                ; Software breakpoint 3
Break.Set 0x100 /Program               ; On-chip breakpoint
```

2. Unlimited breakpoints in RAM and one breakpoint on a read or write access

```
Break.Set 0x10000000 /Program          ; Software breakpoint 1
Break.Set 0x10001000 /Program          ; Software breakpoint 2
Break.Set addr /Program                ; Software breakpoint 3
Break.Set 0x10008000 /Write            ; On-chip breakpoint
```

3. Two breakpoints in ROM/FLASH

```
Break.Set 0x100 /Program               ; On-chip breakpoint 1
Break.Set 0x200 /Program               ; On-chip breakpoint 2
```

4. Two breakpoints on a read or write access

```
Break.Set 0x10008000 /Write            ; On-chip breakpoint 1
Break.Set 0x10008010 /Read             ; On-chip breakpoint 2
```

5. One breakpoint in ROM/FLASH and one breakpoint on a read or write access

```
Break.Set 0x100 /Program               ; On-chip breakpoint 1
Break.Set 0x10008010 /Read             ; On-chip breakpoint 2
```

Runtime Measurement

The command **RunTime** allows run time measurement based on polling the CPU run status by software. Therefore the result will be about few milliseconds higher than the real value.

If the signal DBGACK on the JTAG connector is available, the measurement will automatically be based on this hardware signal which delivers very exact results.

Standby Mode

If the core is in standby mode due to a STANDBY command, the debugger will show 'running'. Manual 'break' is only possible in `no_wake_grant`.

Memory Classes

Memory classes are used to specify which memory to access (e.g. data memory versus instruction memory) and how the access is performed (e.g. via the CPU, via DMA, ...).

The following memory classes are available for AndesCore.

Memory Class	Description
P:	Program Memory
D:	Data Memory
IC:	Program Memory seen through Instruction Cache
DC:	Data Memory seen through Data Cache
NC:	Memory seen with cache switched off
SR:	Access to System Registers. The address is formed from nibbles containing the register's major, minor and extension values. Example: the program status word PSW (major=1, minor = 0, extension = 0) is accessed through the address SR:0x100. Note: The Andes opcodes `mtsr, mfsr` use a slightly different address.
VM:	Virtual Memory (memory on the debug system)
E:	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
ILM:	Instruction local memory, using DMA-access
DLM:	Data local memory, using DMA-access
BUS:	main bus, using DMA-access
EILM:	like ILM:, but performs the access even when core is executing
EDLM:	like DLM:, but performs the access even when core is executing
EBUS:	like BUS:, but performs the access even when core is executing

Interruption Handling in Hardware

In Andes architecture, interruption is handled in hardware based on the “interruption stack level transition” (ISLT). Currently, four interruption stack levels are defined in the architecture, 0-3. And interruption stack level 0 means no interruption. Thus, hardware behaviors for handling an interruption are defined for three interruption stack level transitions, 0/1, 1/2, and 2/3.

Interruption Handling for Interruption Stack Level Transition 0/1 and 1/2

For ISLT 0/1 and 1/2, key interruption states will be saved and restore from and to hardware interruption stack registers. These saving and restoring states are needed since when transition into higher levels of interruption (0 ? 1, and 1 ? 2), the following states will be updated before fetching and executing the first instruction in the interruption handler:

- (PSW) Interruption Stack Level <- Interruption Stack Level ++
- (PSW) Global Interrupt enable <- 0
- (PSW) Privilege Mode <- 1 (Superuser mode)
- (PSW) IT/DT <- 0
- (PSW) BE <- default endian (MMU_CFG.DE)
- (PSW) DRBE <- MMU_CFG.DRDE
- (PSW) IME <- (Instruction Machine Error) ? 1 : IME
- (PSW) DME <- (Data Machine Error) ? 1 : DME
- (PSW) DEX <- (Debug Exception) ? 1 : DEX
- (PSW) HSS <- (PSW.HSS) ? INT_MASK.DSSIM : 0
- Interruption Type
- Exception VA (meaningful for only certain exceptions)
- Program Counter <- Interruption vectored entry point

So before updating these values, the old content of these registers will be saved to the lower level interruption stack registers.

Saving/restoring operations for ISLT 0/1

When entering into interruption stack level 1 from level 0, the saving of registers will be performed as follows:

- IPSW <- PSW
- IPC <- PC

When returning from interruption stack level 1 to level 0, the restoring of registers will be performed as follows:

- IPSW -> PSW
- IPC -> PC

Saving/restoring operations for ISLT 1/2

When entering into interruption stack level 2 from level 1, the saving of registers will be performed as follows:

- P_IPSW <- IPSW <- PSW
- P_IPC <- IPC <- PC
- P_ITYPE <- ITYPE
- P_EVA <- EVA
- P_P0 <- P0
- P_P1 <- P1

When returning from interruption stack level 2 to level 1, the restoring of registers will be performed as follows:

- P_IPSW -> IPSW -> PSW
- P_IPC -> IPC -> PC
- P_ITYPE -> ITYPE
- P_EVA -> EVA
- P_P0 -> P0
- P_P1 -> P1

This register stack design allows nested debug/error exception handling in the code area where software has not prepared itself to handle nested interruption event.

Interruption handling for interruption stack level transition 2/3

When an Andes core transition from interruption stack level 2 to interruption stack level 3, since the hardware register stack has been used up for level 2, only very limited interruption states will be updated. The assumption is that interruption stack level 3 will be entered only when severe error condition happens during debugging or handling of a nested kernel exception. In these cases, the following states will be updated to permit minimum handling:

- (PSW) Interruption Stack Level <- 3 (i.e. maximum stack level)
- (PSW) IME <- (Instruction Machine Error) ? 1 : IME
- (PSW) DME <- (Data Machine Error) ? 1 : DME
- (PSW) DEX <- (EDM_CFG.VER < 0x0030 && Debug Exception) ? 1 : DEX
- Overflow_IPC <- Program Counter
- Program Counter <- Interruption vectored entry point

And, the following hardware behaviors still change based on the fact that the interruption stack level is 3 without updating the corresponding control states.

- Disable interrupt
- Turn off instruction/data address translation
- Use “default endian (MMU_CFG.DE)” as the data access endian
- Use “MMU_CFG.DRDE” as the device register access endian if MMU_CTL.DREE is asserted.
- For EDM_CFG.VER >= 0x0030, disable Hardware Single Stepping

When an Andes core returns from interruption stack level 3 to interruption stack level 2 executing “return from interruption” instructions, the following states will be updated:

- Program Counter <- Overflow_IPC
- Interruption Stack Level <- 2 (i.e. maximum stack level minus 1)

Maximum Interruption Stack Level Option

For implementations to save hardware cost, Andes architecture provides a configuration option to allow an implementation to choose the maximum interruption stack level to be either 3 or 2. The INTLC field in the MISC_CFG system register records this choice.

The previous two sections describes the operations performed at the interruption stack level transition between 0/1, 1/2, and 2/3 when the INTLC field is set to zero (i.e. maximum interruption stack level is 3).

When the INTLC field is set to 1 (i.e. maximum interruption stack level is 2), all the P_* interruption stack system registers will be removed from an implementation. And the operations performed at the interruption stack level transition between 1/2 will be changed to operations equivalent to those ones performed at the interruption stack level transition between 2/3 when the maximum interruption stack level is 3.

Since the level depth of the hardware interruption stack is limited, for software to handle nested interruptions, lowering interruption stack level is needed to allow unlimited number of nested interruptions to happen. The operations of lowering interruption stack level include saving lower-level interruption stack registers, reducing the interruption stack level, and later after the nested interruptions complete, restoring lower-level interruption stack registers, increasing the interruption stack level. These procedures can be shown in the following figures.

Figure 1. Operations of lowering interruption stack level from 2 to 1

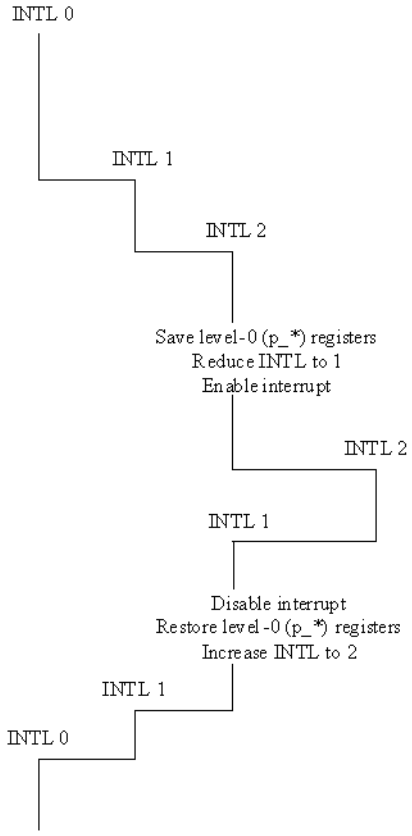
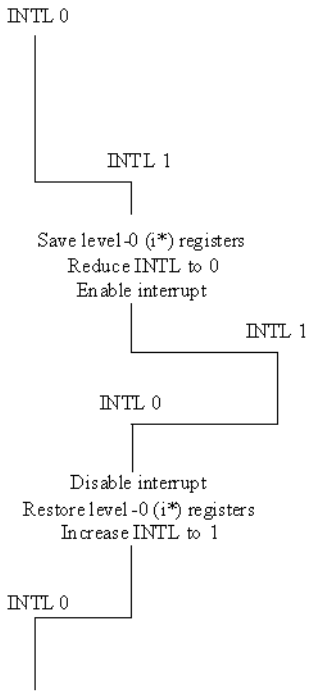


Figure 2. Operations of lowering interruption stack level from 1 to 0



Lowering interruption stack level from 2 to 1

As Figure 1 shown, when in interruption stack level 2, the interruption stack contains level-0 (p_*), level-1 (i*), and level-2 information. Lowering interruption stack level from 2 to 1 means that changing the level-1 (i*) to level-0, changing the level-2 to level-1, and the level-0 (p_*) states are no longer valid/protected. So the level-0 (p_*) information needs to be saved before the lowering happens. The system registers needed to be saved in this case include P_IPSW, P_IPC, P_ITYPE, P_EVA, P_P0, and P_P1.

Lowering interruption stack level from 1 to 0

As Figure 2 shown, when in interruption stack level 1, the interruption stack contains level-0 (i*), and level-1 information. Lowering interruption stack level from 1 to 0 means that changing the level-1 to level-0, and the level-0 (i*) states are no longer valid/protected. So the level-0 (i*) information needs to be saved before the lowering happens. The system registers needed to be saved in this case include IPSW, IPC, ITYPE, EVA, P0, and P1.

Lowering interruption stack level from 3 to 2

Interruption stack level 3 is an overflow interruption level. When in interruption stack level 3, the interruption stack contains level-0 (p_*), level-1 (i*), and level-2 information plus additional level-2 O_IPC state. In this level, the level-2 interruption type and exception VA is not recorded in any register.

If there is a need to lower the interruption stack level from 3 to 2, more steps are required to re-arrange the interruption stack states to the correct stack level hierarchy. Basically, software is required to save the level-0 (p_*) information, and then emulates a stack push operation by (1) moving IPSW, IPC, ITYPE, EVA, P0, and

P1 to their corresponding P_* registers, then (2) moving PSW to IPSW, O_IPC to IPC, and (3) updating PSW to the intended operating values along with the stack level change. Once the update of PSW completes, the CPU will be operating under the interruption stack level 2.

Example codes for ISLT handling:

The following example shows the handling routine for ISLT level 3.

```
Exception entry:
; use $p0 and $p1 to switch stack pointer
        pushm   $r28, $r30
        pushm   $r0, $r25
        mfusr   $r6, $d1.lo
        mfusr   $r14, $d1.hi
        mfusr   $r13, $d0.lo
        mfusr   $r12, $d0.hi
        mfsr    $r11, $P_P1
        mfsr    $r10, $P_P0
        mfsr    $r9, $P_IPC
        mfsr    $r8, $P_IPSW
        mfsr    $r18, $IPC
        mfsr    $r17, $IPSW
        mfsr    $r16, $PSW
        push    $r6
        pushm   $r8, $r14
        andi    $r19, $r16, #PSW_mskINTL
        slti    $r20, $r19, #4              ; check if the INTL is 2
        bnez    $r20, 1f
        addi    $r21, $r16, #-2             ; decrease one level
        mtsr    $r21, $PSW
        isb
```

```
 Exception exit:
        popm    $r16, $r18
        mtsr    $r16, $PSW                  ; use the original one
        popm    $r8, $r14
        pop     $r6
        mtsr    $r17, $IPSW
        mtsr    $r18, $IPC
        mtsr    $r8, $P_IPSW
        mtsr    $r9, $P_IPC
        mtsr    $r10, $P_P0
        mtsr    $r11, $P_P1
        mtusr   $r12, $d0.hi
        mtusr   $r13, $d0.lo
        mtusr   $r14, $d1.hi
        mtusr   $r6, $d1.lo
        popm    $r0, $r25
        popm    $r28, $r30
        ; use $p0, $p1 to change back stack pointer
        iret
```

SYStem.CONFIG Configure debugger according to target topology

Format: **SYStem.CONFIG** *<parameter>*

<parameter>:
state
DRPRE *<bits>*
DRPOST *<bits>*
IRPRE *<bits>*
IRPOST *<bits>*
TAPState *<state>*
TCKLevel *<level>*
TriState [**ON** | **OFF**]
Slave [**ON** | **OFF**]

These commands describe the physical configuration at the JTAG port of a multi core system.

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger about the TAP controller position in the JTAG chain, if there is more than one core in the JTAG chain (e.g. Andes + DSP). The information is required before the debugger can be activated e.g. by a **SYStem.Up**.
See **Daisy-Chain Example**.

For some CPU selections (**SYStem.CPU**) the above setting might be set automatically, since the required system configuration of these CPUs are well-known.

TriState has to be used if several debuggers (“via separate cables”) are connected to a common JTAG port at the same time in order to ensure that always only one debugger drives the signal lines. 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.

state Show selected configuration.

DRPRE (default: 0) *<number>* of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.

DRPOST (default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.

IRPRE (default: 0) *<number>* of instruction register bits in the JTAG chain between the core of interest and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the core of interest and the TDO signal of the debugger.

IRPOST (default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the core of interest. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the core of interest.
See **Daisy-Chain Example**.

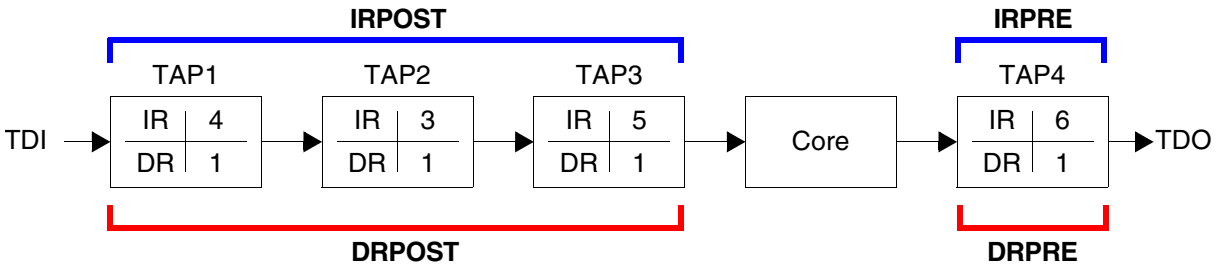
TAPState (default: 7 = Select-DR-Scan) This is the state of the TAP controller when the debugger switches to tristate mode. All **states** of the JTAG TAP controller are selectable.

TCKLevel (default: 0) Level of TCK signal when all debuggers are tristated.

TriState (default: OFF) If several debuggers share the same debug port, this option is required. The debugger switches to tristate mode after each debug port access. Then other debuggers can access the port. This option must be used, if the JTAG line of multiple debug boxes are connected by a JTAG joiner adapter to access a single JTAG chain.

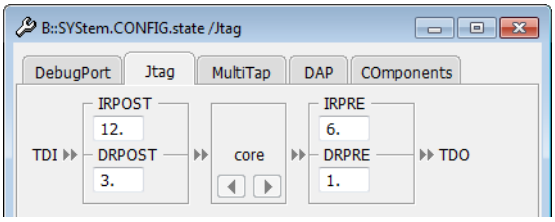
Slave (default: OFF) If more than one debugger share the same debug port, all except one must have this option active. Only one debugger - the ”master” - is allowed to control the signals nTRST and nSRST.

Daisy-Chain Example



IR: Instruction register length **DR**: Data register length **Core**: The core you want to debug

Daisy chains can be configured using a PRACTICE script (*.cmm) or the **SYStem.CONFIG.state** window.



Example: This script explains how to obtain the individual IR and DR values for the above daisy chain.

```
SYStem.CONFIG.state /Jtag     ; optional: open the window

SYStem.CONFIG IRPRE   6.      ; IRPRE: There is only one TAP.
                              ; So type just the IR bits of TAP4, i.e. 6.

SYStem.CONFIG IRPOST 12.      ; IRPOST: Add up the IR bits of TAP1, TAP2
                              ; and TAP3, i.e. 4. + 3. + 5. = 12.

SYStem.CONFIG DRPRE   1.      ; DRPRE: There is only one TAP which is
                              ; in BYPASS mode.
                              ; So type just the DR of TAP4, i.e. 1.

SYStem.CONFIG DRPOST  3.      ; DRPOST: Add up one DR bit per TAP which
                              ; is in BYPASS mode, i.e. 1. + 1. + 1. = 3.
                              ; This completes the configuration.
```

NOTE: In many cases, the number of TAPs equals the number of cores. But in many other cases, additional TAPs have to be taken into account; for example, the TAP of an FPGA or the TAP for boundary scan.

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

SYStem.CPU — Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**N1213S** | **N1213HB0**

Selects the processor type. If your chip is not listed, select the type of the integrated AndesCore.

Default selection: N1213S

Format: **SYStem.JtagClock** [<*frequency*> | **RTCK** | **ARTCK** <*frequency*> | **CTCK** <*frequency*> | **CRTCK** <*frequency*>]
SYStem.BdmClock <*frequency*> (deprecated)

<*frequency*>: **4 kHz**…**80 MHz**

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. 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.

RTCK The JTAG clock is controlled by the RTCK signal (**R**eturned **TCK**)**.** The debugger does not progress to the next TCK edge until after an RTCK edge is received. This mode is not recommended for this debugger since it is not needed here.

ARTCK Accelerated method to control the JTAG clock by the RTCK signal (Accelerated Returned TCK). 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. TDI and TMS will be delayed by 1/2 TCK clock cycle. TDO will be sampled with RTCK. This mode is not recommended for this debugger since it is not needed here.

CTCK 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**).

CRTCK 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 sends back the TCK signal onto the RTCK signal. In contrast to the **RTCK** option, the TCK is always output with the selected, fixed frequency.

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 AndesCore 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. Please note: nTRST and EDBGRQ must have a pull-up resistor on the target.

Format:	**SYStem.MemAccess** <*mode*>
<*mode*>:	**Denied** | **StopAndGo**

Default: Denied.

There's no possibility to access memory while CPU is running. The debugger can access memory only if the CPU is stopped.

SYStem.Mode Establish the communication with the target

Format:	**SYStem.Mode** <*mode*>
	SYStem.Attach **(alias for SYStem.Mode Attach)**
	SYStem.Down **(alias for SYStem.Mode Down)**
	SYStem.Up **(alias for SYStem.Mode Up)**
<*mode*>:	**Down**
	NoDebug
	Go
	Attach
	StandBy
	Up

Down Disables the debugger (default). 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 tristated.

Go Resets the target and enables the debugger and start the program execution. Program execution can be stopped by the break command or external trigger.

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.

StandBy

You need to be in DOWN state when switching to this mode. It resets and starts the program when power is detected. Halt the program execution and set all the breakpoints and trace conditions you need, then re-start the program. Now you can even debug a power cycle, because debug register (breakpoints and trace control) will be restored on power up. This mode is not yet available.

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.ArchVersion Configure version of architecture

Format: **SYStem.Option.ArchVersion** *<arch>*

The option configures the architecture of the AndesCore of the target.

<arch> = 2 Andes V2

<arch> = 3 Andes V3

NOTE: This option is mostly intended for the use with the built-in instruction set simulator. For physical targets the debugger automatically detects the correct value.

SYStem.Option.ArchMcu Configure MCU architecture

Format: **SYStem.Option.ArchMcu** *<mcu_flag>*

The option configures the debugger so it knows whether the target has an MCU architecture or a DSP architecture. This has some implications regarding the instruction set and register file.

<mcu_flag> = 0 DSP architecture

<mcu_flag> = 1 MCU architecture

NOTE: This option is mostly intended for the use with the built-in instruction set simulator. For physical targets the debugger automatically detects the correct value.

Format: **SYStem.Option.ArchRdreg** *<rdreg>*

The option configures the debugger so it knows whether the target has a full or reduced set of general purpose registers.

<rdreg> = 0 full set of general purpose registers (`r0-r31`)

<rdreg> = 1 reduced set of general purpose registers (`r0-r10;r15;r28-r31`)

NOTE: This option is mostly intended for the use with the built-in instruction set simulator. For physical targets the debugger automatically detects the correct value.

SYStem.Option.DIMBR Define base address of debug instruction memory

Format: **SYStem.Option.DIMBR** *<address>*

Default: 0xa0800000.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

ON The Global Interrupt Enable Bits will be cleared during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the Global Interrupt Enable bits will be restored to the value before the step.

OFF A pending interrupt will be executed on a single-step, but it does not halt there. The specific interrupt handler is completely executed even if single steps are done, i.e. step over is forced per default. If the core should halt in the interrupt routine, use **TrOnchip.StepVector ON**.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the Global Interrupt Enable Bit s(SR.IEE and SR.TEE) will be cleared during high-level-language single-step operations. The interrupt routine is not executed during single-step operations. After single step the Global Interrupt Enable bit will be restored to the value before the step.

If disabled, a pending interrupt will be executed on a single-step, but it does not halt there i.e. the interrupt handler is always over stepped. If you want to halt in the interrupt routine, use **TrOnchip.StepVector ON**.

SYStem.Option.MMUSPACES Separate address spaces by space IDs

Format: **SYStem.Option.MMUSPACES** [**ON** | **OFF**]
SYStem.Option.MMUspaces [**ON** | **OFF**] (deprecated)
SYStem.Option.MMU [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **"TRACE32 Concepts"** (trace32_concepts.pdf).

NOTE: **SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target.

If a debug session requires space IDs, you must observe the following sequence of steps:

1. Activate **SYStem.Option.MMUSPACES**.

2. Load the symbols with **Data.LOAD**.

Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

```
;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x012A:
Data.dump D:0x012A:0xC00208A

;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x0203:
Data.dump D:0x0203:0xC00208A
```

SYStem.Option.SCRATCH Define address for dummy reads

Format: **SYStem.Option.SCRATCH** *<address>*

Default: 0xffffffff.

The Andes core has a data cache line buffer. The debugger needs to force a write back into the data cache in order to display the cached data memory properly. To do so it reads from an accessible, data cacheable dummy address which needs to be different from the address hold in the cache line buffer.

On the default value (0xffffffff) it uses the current program counter value. If the program counter value is not in a data cacheable area or if there are constant data next to the program counter location, the user needs to specify a better suitable address with this option.

SYStem.Option.TURBO Speed up memory access

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

Default: OFF.

If TURBO is enabled the debugger will not take care of cache coherency or address translation when accessing memory. But this way the memory access is significantly faster. The user needs to make sure that neither cache nor address translation (MMU) is active, like it is the case after a chip reset. Even if TURBO is activated it is only used for 32-bit accesses.

We recommend to use this option only for a program download when starting-up a debug session. Especially when the local memory is enabled, this option shall not be used.

Example:

```
...
SYStem.Option.TURBO ON
Data.LOAD.ELF <file> /Long
SYStem.Option.TURBO OFF
...
```

SYStem.state — Display SYStem window

Format:	**SYStem.state**

Display the **SYStem.state** window of the AndesCore debugger.

AndesCore Specific TrOnchip Commands

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

TrOnchip.ContextID — Enable context ID comparison

Format: **TrOnchip.ContextID** [**ON** | **OFF**]

Default: OFF.

If the AndesCore debug unit provides breakpoint registers with ContextID comparison capability **TrOnchip.ContextID** has to be set to ON in order to set task/process specific breakpoints that work in real-time.

```
TrOnchip.ContextID ON

Break.Set VectorSwi /Program /Onchip /TASK EKern.exe:Thread1
```

TrOnchip.RESet — Reset on-chip trigger settings

Format: **TrOnchip.RESet**

Resets all TrOnchip settings.

TrOnchip.StepVector — Halt on exception entry when single-stepping

Format: **TrOnchip.StepVector** [**ON** | **OFF**]

Default: OFF.

If StepVector is activated, a breakpoint range will is set on the trap vector table when *single-stepping* through code. This is helpful to check if interrupts, traps or exceptions occur while single-stepping.

NOTE:	For catching exceptions that occur in code that is freely executed (rather than single-stepped), manually set a range of onchip breakpoints on the vector table e.g. by `Break.Set 0x0++0xff /Onchip(`assuming the vector table is located at 0x0).

TrOnchip.state — Display on-chip trigger window

Format:	**TrOnchip.state**

Opens the **TrOnchip.state** window.

MMU.DUMP Page wise display of MMU translation table

Format:	**MMU.DUMP** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.dump** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** *<cpu_specific_tables>*

Displays the contents of the CPU specific MMU translation table.

- If called without parameters, the complete table will be displayed.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: displays the translation table of the specified process • else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	Displays the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and displays its table entries.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer.
DTLB	Displays the contents of the Data Translation Lookaside Buffer.
TLB	Displays the contents of the Translation Lookaside Buffer.

MMU.List — Compact display of MMU translation table

Format: **MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*]
MMU.*<table>***.List** (deprecated)

<table>: **PageTable**
KernelPageTable
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**

Lists the address translation of the CPU-specific MMU table.

- If called without address or range parameters, the complete table will be displayed.
- If called without a table specifier, this command shows the debugger-internal translation table. See **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Lists the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: list the translation table of the specified process • else, this command lists the table the CPU currently uses for MMU translation.

KernelPageTable	Lists the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** [**Clear**] *<cpu_specific_tables>*

Loads the CPU-specific MMU translation table from the CPU to the debugger-internal static translation table.

- If called without parameters, the complete page table will be loaded. The list of static address translations can be viewed with **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, page table entries will only be loaded if their **logical** address matches with the given parameter.

Use this command to make the translation information available for the debugger even when the program execution is running and the debugger has no access to the page tables and TLBs. This is required for the real-time memory access. Use the command **TRANSlation.ON** to enable the debugger-internal MMU table.

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

Pinout of the 20-pin Debug Cable:

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	VSUPPLY (not used)
TRST-	3	4	GND
TDI	5	6	GND
TMS|TMSC|SWDIO	7	8	GND
TCK|TCKC|SWCLK	9	10	GND
RTCK	11	12	GND
TDO|-|SWO	13	14	GND
RESET-	15	16	GND
DBGRQ	17	18	GND
DBGACK	19	20	GND

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints refer to the application note **"Arm Debug and Trace Interface Specification"** (app_arm_target_interface.pdf). It describes a debug cable which is technically the same as the AndesCore debug cable. Only the logical level of the signals DBGRQ/nDBGI and DBGACK/nDBGACK are different. They are active low for AndesCore.