LAUTERBACH
DEVELOPMENT TOOLS

# NIOS II Debugger and Trace

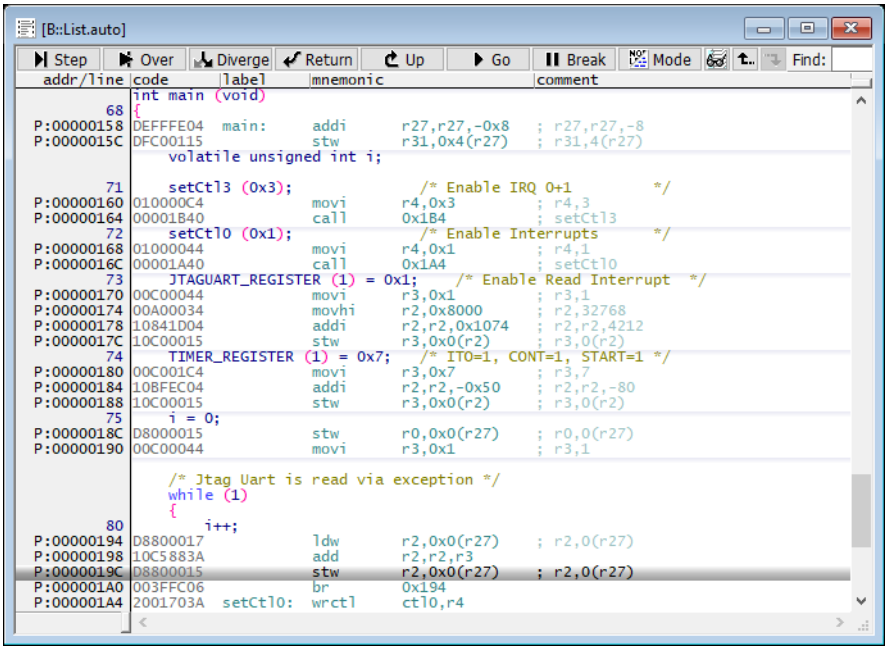
MANUAL

# NIOS II Debugger and Trace

**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 NIOS II 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/nios/` 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.<br><br>Recommendation for the software start:<br><br>1. Disconnect the Debug Cable from the target while the target power is off.<br>2. Connect the host system, the TRACE32 hardware and the Debug Cable.<br>3. Power ON the TRACE32 hardware.<br>4. Start the TRACE32 software to load the debugger firmware.<br>5. Connect the Debug Cable to the target.<br>6. Switch the target power ON.<br>7. Configure your debugger e.g. via a start-up script.<br><br>Power down:<br><br>1. Switch off the target power.<br>2. Disconnect the Debug Cable from the target.<br>3. Close the TRACE32 software.<br>4. Power OFF the TRACE32 hardware. |

# 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 FPGA which should hold a Nios II Core with debugging interface isn't configured, or the design doesn't contain a Nios II Core with debugging interface.
- There is a short-circuit on at least one output line of the CPU.
- There is a problem with the electrical connection between ICDNIOS and the target - check if the BDM connector is plugged correctly and if the target is built corresponding to the **definition** of the used BDM connector.

## Trace Errors

To use an off-Chip trace for a Nios II CPU we strongly recommend to follow the application note **"NIOS II Debugger and Trace"** (debugger_nios.pdf).

If you don't follow this application note, you have to enable the **FSS** option.

# FAQ

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

# Quick Start of the ICD Debugger for Nios II

This chapter should help you to prepare your Debugger for Nios II. Depending on your application not all steps might be necessary.

For some applications additional steps might be necessary, that are not described in this Quick Start section.

## 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 next to the nose of the target **connector.**

Power up your TRACE32 system (This is not necessary on PODPC).

Start the TRACE32 Debugger Software.

Power up your Target!

To prevent damage please take care to follow this sequence all the time you are preparing a start.

## 2. Configure your FPGA with a Nios II Core (optional)

Before you can start debugging, the FPGA has to contain a design with a Nios II Core with a debugging interface. On some targets the FPGA will be automatically configured at PowerUp. If you want to use your own design, you can configure the FPGA by using the commands **JTAG.PROGRAM.JAM** or **JTAG.PROGRAM.JBC** or **JTAG.LOADRBF**.

## 3. Select the Clock for the JTAG Communication

You can select the JTAG clock frequency, which the Debugger uses to communicate with the target. This can be either done in the JtagClock field in the SYStem Window, or by using the command line with the command **SYStem.JtagClock**. The maximum clock frequency depends on the configuration of your FPGA design. The default clock frequency is 1 MHz.

## 4. Configure the Debugger According to the Needs of the Application

Depending on the variant of the debugged Nios II core, different cache handling strategies can be used. All of the available settings, can be configured with the SYStem Window. Set the **SYStem Options** in this window according to your FPGA configuration and application program. Generally the SYStem Options can remain at the default values for the first start.

## 5. Tell the Debugger where it should use On-chip Breakpoints (optional)

By default the In Circuit Debugger for Nios II modifies the code to realize program breakpoints. This will not work for ROM or FLASH memory locations. If the used Nios II core provides on-chip breakpoints, these breakpoints can be used for ROM/FLASH areas instead. With the command **MAP.BOnchip** *<range>* you can specify where the debugger has to use on-chip breakpoints.

```
MAP.BOnchip 0x1000--0x0ffff      ; activates the on-chip breakpoints
                                 ; within the range from 0x1000 to
                                 ; 0xffff
```

## 6. Enter Debug Mode

```
SYStem.Up
```

This command asserts a reset to the Nios II core. While the reset is asserted, the machine code for a standard monitor will be downloaded. After the reset is deasserted, the Nios II will enter debug mode and jump to the break address of the debugged core.

## 7. Load the Program

Depending on your FPGA configuration, the Nios II core may have access to many different variants of memory, including on-chip memory, external SDRAM or FLASH memory.

When the core is prepared the code can be downloaded. This can be done with the command **Data.Load.<file_format> <file>**. The debugger knows about various file formats. If you use the GNU C compiler provided by Altera, you will usually have an ELF file. The typical command to load such an executable is:

```
Data.Load.Elf <file>.elf /verify    ; Load application file generated
                                    ; with the gcc compiler, provided by
                                    ; Altera. Verify that the application
                                    ; is written correctly to memory.
```

## 8. Initialize Program Counter and Stackpointer

In a ready-to-run compiled ELF file, these settings are in the start-up code of the ELF file. In this case nothing has to be done. You can check the contents of Program Counter and Stack Pointer in the Register Window, which provides the contents of all CPU Registers. Use CPU Registers in the CPU menu to open this window or use the command **Register**.

The Program Counter and the Stackpointer and all other registers can be set with the commands **Register.Set PC** *<value>* and **Register.Set SP** *<value>*. Here is an example of how to use these commands:

```
Register.Set PC 0xc000              ; Set the Program Counter to address
                                    ; 0xC000

Register.Set SP 0xbff               ; Set the Stack Pointer to address
                                    ; 0xbff

Register.Set PC main                ; Set the PC to a label (here:
                                    ; function main)
```

## 9. View the Source Code

Use the command **Data.List** to view the source code at the location of the Program Counter.

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

```
WinCLEAR                          ; Clear all windows

SYStem.Reset                      ; Set all options in the SYStem window
                                  ; to default values

MAP.BOnchip 0x01080--0x0ffff      ; Select on-chip breakpoints for the
                                  ; FLASH and ROM areas

SYStem.Up                         ; Reset the target and enter debug mode

Data.LOAD.Elf demo.elf            ; Load the application

List.Mix                          ; Open disassembly 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 0x400                   ; Set software breakpoint to address
                                  ; 0x400 (address 0x400 is outside the
                                  ; range, where on-chip breakpoints are
                                  ; used)

Break.Set 0x8024                  ; Set on-chip program breakpoint to
                                  ; address 0x8024 (address 0x8024 is
                                  ; within the range, where on-chip
                                  ; breakpoints are used)
```

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

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

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. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

## Restrictions

| | |
|---|---|
| **On-chip Break-points** | Because the Nios II is a completely configurable soft core, not all variants support on-chip breakpoints. The debugger will check the number of available on-chip breakpoints, when the SYStem.Up command is executed. If more on-chip breakpoints are used than the core supports, the debugger will report an invalid breakpoint configuration. |

Format: **SYStem.CONFIG** *<parameter> <number_or_address>*
**SYStem.MultiCore** *<parameter> <number_or_address>* (deprecated)

*<parameter>* (JTAG):
**state**
**CORE** *<core> <chip>*
**DRPRE** *<bits>*
**DRPOST** *<bits>*
**IRPRE** *<bits>*
**IRPOST** *<bits>*
**TAPState** *<state>*
**TCKLevel** *<level>*
**TriState** [**ON** | **OFF**]
**Slave** [**ON** | **OFF**]
**InstanceNR** *<value>*

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger about the system configuration if there is more than one JTAG compatible device in the JTAG chain (e.g. Stratix FPGA + Cyclone FPGA). The information is required before the debugger can be activated e.g. by a **SYStem.Up**.

TriState has to be used if more than one debugger are connected to the common JTAG port at the same time. 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..

**state** Show multicore settings.

**CORE** *<core>* *<chip>* For multicore debugging one TRACE32 PowerView GUI has to be started per core. To bundle several cores in one processor as required by the system this command has to be used to define core and processor coordinates within the system topology.
Further information can be found in **SYStem.CONFIG.CORE**.

**DRPRE** Default: 0.
*<number>* of data register bits in the JTAG chain between the data register of the core and the TDO signal (usually one data register bit per JTAG device which is in BYPASS mode).

**DRPOST** Default: 0.
*<number>* of data register bits in the JTAG chain between the TDI signal and the data register of the core (one data register bit per JTAG device which is in BYPASS mode).

**IRPRE** Default: 0.
*<number>* of instruction register bits of all JTAG devices in the JTAG chain between the instruction register of the core and the TDO signal.

**IRPOST** (default: 0) *<number>* of instruction register bits of all JTAG devices in the JTAG chain between TDI signal and the instruction register of the core.

**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 more than one debugger share the same JTAG port, this option is required. The debugger switches to tristate mode after each JTAG access. Then other debuggers can access the port.

**Slave** (default: 0) If more than one debugger share the same JTAG port, all except one must have this option active. Only one debugger - the 'master' - is allowed to control the optional reset signal.

**InstanceNR** *<value>* Instance number.

**Example:**

TDI ---> Device A ---> Device B ---> Device C ---> Device D ---> TDO

Instruction register length of

```
Device A : 3 bit

Device B : 5 bit

Device C : 5 bit

Device D : 4 bit
```

Now to debug Device C you will need the following settings:

```
SYStem.CONFIG IRPRE 4                ; IR Device D

SYStem.CONFIG IRPOST 8               ; IR Device A + B

SYStem.CONFIG DRPRE 1                ; DR Device D

SYStem.CONFIG DRPOST 2               ; DR Device A + B
```

| | |
|---|---|
| 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 |

Format: **SYStem.CONFIG.CORE** *<core_number> <chip_number>*
**SYStem.MultiCore.Core** *<core_number> <chip_number>* (deprecated)

With the Nios II the more common case is that you have only one FPGA device, which has several cores in it. The Nios II cores in one FPGA device use a multiplexing scheme, which means that they are *not* daisy chained in a JTAG chain. To select which core you want to debug in one FPGA device, you use the above command.

The **core number** specifies which core you want to debug in one FPGA device. The **chip number** is only needed, if you have several FPGA devices on your JTAG chain, and you want to debug them in parallel. In this case you should enumerate your FPGA devices so that each FPGA device has a unique **chip number**; it is recommended to start with **chip number** 1.

All cores which are in the **same** FPGA should get the **same chip number**. Which FPGA gets which **chip number** can be chosen arbitrarily. Example configuration:

TDI ---> Stratix with 2 Nios II cores ---> Cyclone with 1 Nios II core ---> TDO.

In this example we will give the Stratix **chip number** 1 and the Cyclone **chip number** 2. As mentioned, it is not importand how you enumerate your FPGAs, so it would also be possible to exchange this chip numbers (so that the Stratix is 2 and the Cyclone is 1).

Now to debug the two cores in the Stratix you'll need the following JTAG Multicore settings:

```
SYStem.CONFIG IRPRE 10          ; IR Cyclone
SYStem.CONFIG IRPOST 0          ; No device before Stratix in chain
SYStem.CONFIG DRPRE 1           ; DR Cyclone
SYStem.CONFIG DRPOST 0          ; No device before Stratix in chain
```

The debugger for the first core in the Stratix device additionally needs the following setting

```
SYStem.CONFIG.Core 1 1          ; Connect to Core 1 in Stratix (Chip 1)
```

And the debugger for the second core in the Stratix device needs the setting:

```
SYStem.CONFIG.Core 2 1          ; Connect to Core 2 in Stratix (Chip 1)
```

The debugger for the core in the Cyclone device needs the following JTAG Multicore settings:

```
SYStem.CONFIG IRPRE 0          ; No device after Cyclone in chain
SYStem.CONFIG IRPOST 10        ; IR Stratix
SYStem.CONFIG DRPRE 0          ; No device after Cyclone in chain
SYStem.CONFIG DRPOST 1         ; DR Stratix
```

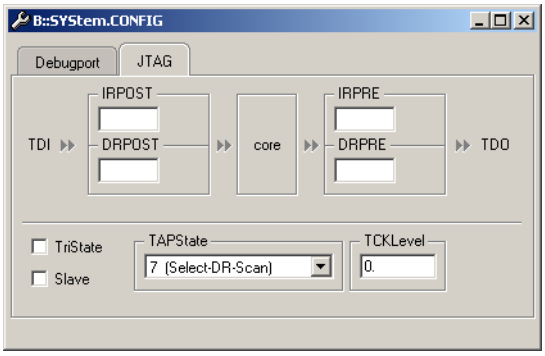
And additionally:

```
SYStem.CONFIG.Core 1 2         ; Connect to Core 1 in Cyclone (Chip 2)
```

| | |
|---|---|
| Format: | **SYStem.CONFIG.state** [*/<tab>*]<br>**SYStem.MultiCore.view** (deprecated) |
| *<tab>*: | **DebugPort** \| **Jtag** |

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.




| | |
|---|---|
| **DebugPort** | n/a |
| **Jtag** | 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. |

## SYStem.CONFIG.CPUID Tell the debugger to which CPU it should connect

Format: **SYStem.CONFIG.CPUID** *<value>*

Tells the debugger to which CPU core it should connect. Refer to **SYStem.DETECT.ScanCpuIDS** for more information.

## SYStem.CPU Select CPU type

At the moment the only CPU type which can be selected is “Nios II”.

## SYStem.CONFIG.JtagUartNR Specify JTAG UART component number

Format: **SYStem.CONFIG.JtagUartNR** [*<value>*]

This option is only relevant, if you have an FPGA design with

- multiple “Altera JTAG UART” IP components in it
- multiple “Nios II” cores in it, but it's not clear which Nios II CPU should access the JTAG UART

Per default the PowerView software matches JTAG UART components to Nios II CPU cores, by enumerating both kinds of components. That means the first found Nios II CPU core will be associated with the first found JTAG UART, the second found Nios II CPU core will be associated with the second found JTAG UART and so on.

In an FPGA with just a single Nios II core, this matching works fine; but in an FPGA design with multiple Nios II cores or multiple JTAG UART components, the matching might not be what you need.

In these scenarios, you might specify which JTAG UART should be accessed, with the **SYStem.CONFIG JtagUartNR** command. For a specific FPGA design unfortunately the Quartus II FPGA design software, does not tell you how the JTAG UART components are enumerated. You might not know immediately what is the first JTAG UART component, what is the second JTAG UART component and so on; some experimentation might be necessary.

To undo the setting use **SYStem.CONFIG.JtagUartNR** without specifying a value.

Format: **SYStem.DETECT.ScanCpuIDS**

If you have an FPGA design, which contains multiple Nios II CPU cores, then you have to tell the debugger which CPU core should be debugged.

One way of specifying this is to use the **SYStem.CONFIG.CORE <nr>** command.

However, the Nios II FPGA Design Software, does not make it obvious, which CPU core gets which number. The Nios II CPU core has a CPUID register; the value of this register is specified by the user in the Nios II FPGA Design Software for each CPU core. If you give each CPU core a unique CPUID register value, then you might use the CPUID register value to unambiguously tell to which CPU core TRACE32 is connected.

If you want to do that manually, you can iterate through all existing CPU cores:

- Start with **SYStem.CONFIG.CORE 0**
- Execute the command **SYStem.Up**
- Display the CPUID register (by looking into the **Register** window)
- Go back into the **SYStem.Down** state
- Increment the number for **SYStem.CONFIG.CORE <nr>** until you have iterated through all CPU cores

This way you can manually find out which CPUID register value belongs to which **SYStem.CONFIG.CORE <nr>**.

TRACE32 offers some help in automating this process:

1. Make sure you are in **SYStem.Down** state
2. Execute **SYStem.DETECT.ScanCpuIDS**. This will scan, which CPUIDs exist in the FPGA design. TRACE32 will print a list of found CPUIDs in the **AREA** message window.
3. Tell TRACE32 to which CPU core it should connect by using the command **SYStem.CONFIG.CPUID <value>**.
4. When you now execute a **SYStem.Up**, TRACE32 will connect to the CPU core with the specified CPUID register value. If no such CPUID was found an error is reported.

Please note that The command **SYStem.DETECT.ScanCpuIDS** is **intrusive**: it will iterate through all CPU cores and stops each of them for a short period of time. This unfortunately is unavoidable, because the Nios II hardware debug interface does not offer a way to read out the CPUID register value without stopping the CPU core.

The command **SYStem.DETECT.ScanCpuIDS** only works if you are in the **SYStem.Down** state.

If you want to un-set the CPUID (and instead use the normal **SYStem.CONFIG.CORE <nr>** mechanism to select the CPU core), you might execute **SYStem.CONFIG.CPUID** and leave out the CPUID value. This will un-set the CPUID and switch back to select the CPU core with the number specified via **SYStem.CONFIG.CORE <nr>**.

## SYStem.JtagClock — Select clock for JTAG communication

This command selects the frequency of the JTAG clock, which is used to communicate with the Nios II core inside the FPGA. The maximum reachable frequency is dependent on the design in the FPGA. In general 10 MHz should work properly. To be on the safe side, the default frequency, which is selected when the debugger is started is set to 1 MHz.

## 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.

Format: **SYStem.MemAccess StopAndGo | Denied**
**SYStem.ACCESS** (deprecated)

**Denied** Memory access during program execution to target is disabled.

**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.

## SYStem.Mode

Select target reset mode

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**
**Attach**
**Up**

**Down** Stops communicating with the Nios II core over JTAG.

**NoDebug** Not implemented.

**Go** Not Implemented.

**Attach** User program remains running (no reset) and the debug interface is initialized. After this command the user program can be stopped with the break command or if a break condition occurs.

**Up** Resets ALL cores, enters debug mode, starts to execute monitor code.

**StandBy** Not implemented.

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

This option controls if the trace unit (when available) of the Nios II produces Branch Trace Messages or not. If you disable this option, then you don't get any program flow information from the trace. This option configures the behavior for the on-chip and off-chip trace.

## SYStem.Option.CFGCLK Set clock frequency for configuration

Format: **SYStem.Option.CFGCLK** *<frequency>*

*<frequency>*: **10MHz**
**5MHz**
**2.5MHz**
**1.25MHz**
**612kHz**

When you want to configure your FPGA a fixed frequency is used to send the configuration data to the FPGA. This frequency can be set by this option.

## SYStem.Option.DCFLUSH Flush data cache before "Go"

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

Default: ON.

If this option is enabled the data cache will be flushed (written back to memory and invalidated), before the debugger executes a **Go** command. On Nios II cores, which have a data cache, this is might be necessary, to ensure that program code, which was written to the data cache, gets transferred into the memory.

## SYStem.Option.DBGALL Enable/disable debug mode for all cores

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

A **System.Up** command will reset all cores in one FPGA. With this option you can select, which cores are also put into debug mode. If this option is enabled, then all cores in the chip will be put into debug mode. If this option is disabled, then only cores, which are connected to a TRACE32 PowerView GUI will be put into debug mode.

## SYStem.Option.LocalRESet Assert a local JTAG reset at SYStem.Up

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

This option is intended for multi-core debugging of QSYS based Systems: With QSYS each Nios II CPU gets a local JTAG reset output port. Depending on your configuration this local JTAG reset might need to be asserted to reset the corresponding Nios II CPU. If you debug in a multi-core environment (with multiple instances of the t32mnios executable in parallel connected to different CPUs in your FPGA), then you might need to enable this option to reset all CPUs at the same time when executing a **System.Up**. If this option is enabled for a CPU to which a t32mnios executable is connected, then the local JTAG reset output port will be asserted when a **System.Up** is executed.

Format: **SYStem.Option.DTM** <*mode*>

<*mode*>: **OFF**
**ReadAddress**
**WriteAddress**
**ReadWriteAddress**
**ReadData**
**Read**
**Write**
**ReadWrite**

This option controls the data trace if available. The Nios II supports several different modes for data tracing:

| | |
|---|---|
| **OFF** | Don't record any data trace information. |
| **ReadAddress** | Record addresses of read accesses. |
| **WriteAddress** | Record addresses of write accesses. |
| **ReadWriteAddress** | Record addresses of read and write accesses. |
| **ReadData** | Record data of read accesses without addresses. |
| **Read** | Record data and addresses of read accesses. |
| **Write** | Record data and addresses of write accesses. |
| **ReadWrite** | Record data and addresses of read and write accesses. |

## SYStem.Option.Endianness Select endianness of core

Format: **SYStem.Option.Endianness** [**AUTO** | **Little** | **Big**]

Default: AUTO.

This option tells the debugger if you use a Little- or Big-Endian Nios II core. If you select AUTO, the endianness will be determined automatically, when you execute a **System.Up**.

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

If you implement an off-chip trace port on your FPGA, we highly recommend to follow the application note about the off-chip trace. If you don't follow the application note, than you have to enable this option to put the trace into a compatibility mode, which works with the original behavior of the off-chip trace port.

## SYStem.Option.FPH Enable the disassembly of floating point instructions

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

Default: OFF.

Enables/disables the mnemonics of floating point instructions in the disassembly (**List** window).

## SYStem.Option.ICFLUSH Flush instruction cache before "Go"

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

Default: ON.

If enabled, the instruction cache will be flushed, before the debugger executes a **Go** or a **Step** command. On Nios II cores, which have an instruction cache, this is necessary to ensure that software breakpoints work correctly and to ensure that code, which is downloaded to the target, will get executed correctly.

If you debug a Nios II processor, which includes an instruction cache, and you turn this option OFF, software breakpoints won't work correctly. You have to use on-chip breakpoints in this case!

## SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, the debug core will disable all interrupts for the CPU, when single stepping assembler instructions. No hardware interrupt will be executed during single-step operations. When you execute a Go command, the hardware interrupts will be enabled again, according to the system control registers.

## SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the debug core will disable all interrupts for the CPU, during HLL single-step operations. When you execute a Go command, the hardware interrupts will be enabled again, according to the system control registers. This option should be used in conjunction with **IMASKASM**.

## 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.<br><br>If a debug session requires space IDs, you must observe the following sequence of steps:<br><br>1. Activate **SYStem.Option.MMUSPACES**.<br><br>2. Load the symbols with **Data.LOAD**.<br><br>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.PIDWidth — Specify size of PID field in the TLB

Format: **SYStem.Option.PIDWidth** *<bits>*

Default: 10.

This setting is only needed, if:

- You use a Nios II CPU with MMU
- You use a Nios II instruction trace (onchip or offchip)

To decode a Nios II (with MMU) instruction trace, the TRACE32 software needs to know the size of the PID (Process Identifier) field in the TLB (Translation Lookaside Buffer) of the Nios II MMU.

Use this option to specify the width of the PID bit field.

Format: **SYStem.Option.QUARTUS GENeric** | **13_0**

The Quartus II FPGA design software version 13.0 produces Nios II CPU cores, which require a workaround for successful debugging.

Use this command to turn on the workaround for FPGA Designs, generated with this particular Quartus II version.

## SYStem.Option.TOFF Enable/disable tracetrigger input

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

If you use an off-chip trace port and if you don't connect the trigger pin of the trace connector, the trigger input of the off-chip trace floats. In this case our trace hardware will detect a lot of false triggers, which will disturb your regular trace recording. You can turn on this system option to disable the trigger input of the off-chip trace, to get rid of the false triggers.

## SYStem.Option.SYNC Specify frequency of SYNC messages

Format: **SYStem.Option.SYNC** <*mode*>

<*mode*>: **ALL**
**4**
**16**
**64**

This option is only relevant if the trace unit generates Branch Trace Messages. There are two kinds of Branch Trace Messages: Compressed messages and SYNC messages. The compressed messages can only be decompressed by analyzing the surrounding SYNC messages. So without SYNC messages, compressed messages can't be decompressed. This option controls how often the trace produces SYNC messages. *ALL* means that the trace only uses SYNC messages and no compressed messages; in this case the Branch Trace uses more trace memory. 64 means that each 64th Branch Trace Message will be a SYNC message; in this case the Branch Trace uses less trace memory, but decompressing the trace is harder.

# Configuring your FPGA

Before you can start debugging, your FPGA has to contain a valid design. The design has to include a Nios II core, for which JTAG debugging is enabled. For instructions how to create such a design, please refer to the technical documentation about the SOPC Builder, provided by Altera.

You can use the debugger to configure your FPGA, if you provide a suitable JBC (Jam Byte Code) or JAM file. Both file formats can be produced for any design with the Quartus II software from Altera. Instructions can be found in the online help of Quartus II.
You can also use a raw binary file (RBF file), which can also be produced by Quartus II. Using a raw binary file is currently the fastest and most flexible configuration method.

You should ensure that the debugger is in SYStem.down mode, before configuring your FPGA. Configuring the FPGA will break the communication link between the debugger and the Nios II core, if your debugger is in **SYStem.Up** mode.

## JTAG.LOADRBF — Configure FPGA with RBF file

Format: **JTAG.LOADRBF** *<file>*

This command will use a raw binary file to configure your FPGA with the debugger.

The raw binary file **must not** contain a compressed bitstream. So you have to deactivate this option in Quartus II, when you generate your raw binary file.

Not all FPGA families from Altera are supported. Currently the following devices are supported:

| | |
|---|---|
| **Stratix ...**<br>**Stratix IV** | All Stratix, Stratix II, Stratix III, Stratix IV devices. |
| **Cyclone ...**<br>**Cyclone IV** | All Cyclone, Cyclone II, Cyclone III, Cyclone IV devices. |
| **Arria GX ...**<br>**Arria II GX** | All Arria GX, Arria II GX devices. |

The used programming algorithm might also work for more recent devices, but this is not guaranteed.

Using a raw binary file currently is the fastest method to configure your FPGA. There is also another advantage:
JAM and JBC files have to contain a complete description of the JTAG chain. So if you have several devices

in your JTAG chain, your JAM and JBC files have to match this configuration.
With a RBF the device, which will be configured, is selected by the **MULTICORE** settings in the debugger.
So the RBF file is independent of the layout of your JTAG chain.

# JTAG Uart Support

Altera provides a JTAG Uart module with its Quartus II software, which can be used as a terminal for applications. The TRACE32 software allows to connect a terminal window to such an UART, with the commands:

```
term.method DCC       ; For Nios II debuggers the "DCC" method will use
                      ; the Jtag UART.

term.                 ; Open up terminal window.
```

# On-chip Breakpoints

The Nios II core can be configured to support up to four on-chip program breakpoints and up to four on-chip read/write breakpoints.

## Program Breakpoints

Generally the In Circuit Debugger for Nios II uses Software Breakpoints to realize Program Breakpoints. Software Breakpoint means that the code at the desired memory location is modified by the debugger to make the CPU break when the program counter hits this address. After a break the original contents of the memory location are restored.

This mechanism can not work in Read Only Memory. To provide breakpoints in ROM areas the CPU' s on-chip breakpoints can be used. The memory ranges, where on-chip breakpoints should be used, must be defined with the command **MAP.BOnchip**.

```
MAP.BOnchip 0x1080--0xffff          ; In the address range 0x1080--0xffff
                                    ; on-chip breakpoints will be used.
```

With the command **Break.List** the actual breakpoint configuration can be checked.

## Read and Write Breakpoints

Read and Write Breakpoints always use the CPU' s on-chip breakpoints regardless of the ranges defined with **MAP.BOnchip**.

Read and Write Breakpoints can be set with the Break window or with the command **Break.Set**:

```
Break.Set 0x4738 /Write             ; The CPU will be stopped if there is a
                                    ; write access to address 0x4738

Break.Set 0xb223 /Read              ; The CPU will be stopped if there is a
                                    ; read access to address 0xB223
```

It is also possible to break on an access to an addresses range. In this case two on-chip breakpoints will be combined to realize the Breakpoint:

```
Break.Set 0x1000--0x10FF /Write     ; The CPU will be stopped if there is a
                                    ; write access to an address in the
                                    ; range 0x1000--0x10FF
```

## Data Breakpoints

Data Breakpoints always use the CPU' s on-chip breakpoints regardless of the ranges defined with **MAP.BOnchip**. All Read/Write Breakpoints can be combined with a 32 Bit data value. If only a 16 or 8 Bit data value is used, or if a data mask is used instead of a data value, two on-chip breakpoint resources are necessary to realize the breakpoint.

```
Break.Set 0x100 /Write /DATA 0x12345678    ; CPU will stop, if the 32 bit
                                           ; value 0x12345678 is written
                                           ; to address 0x100

Break.Set 0x110 /Read /DATA.Byte 0x55      ; CPU will stop, if data is
                                           ; read from address 0x110 and
                                           ; the byte at address 0x110
                                           ; contains the value 0x55.
```

## Trace Control Breakpoints

You can use the on-chip breakpoints to turn the trace on and off and to generate a trigger on the trigger output of the off-chip trace port. This works for the on-chip and off-chip trace. You simply have to add one of the following options to your breakpoint definition:

| | |
|---|---|
| **TraceON** | Turns the collection of trace data on, when the breakpoint is reached. |
| **TraceOff** | Turns the collection of trace data off, when the breakpoint is reached. |
| **TraceEnable** | Only for read/write breakpoints: Will generate a single Data Transfer Message, for the access which matched the breakpoint. |
| **TraceTrigger** | Send a trigger to the off-chip trace via the trigger output of the off-chip trace port of the Nios II core (TRIGA on the mictor connector). |

**Example**:

```
Break.Set 0x9C0 /Onchip /Program /TraceOn    ; Will turn the trace on,
                                             ; when the program reaches
                                             ; address 0x9C0.

Break.Set 0x9D0 /Onchip /Program /TraceOff   ; Will turn the trace off,
                                             ; when the program reaches
                                             ; address 0x9D0.
```

**Restrictions:** TraceEnable breakpoints only work as expected, when the whole trace is turned off. In this case data accesses will be only traced, when the breakpoint condition is met. If the trace is turned on (by hitting a TraceON breakpoint), then the trace will record all data accesses, regardless of any TraceEnable breakpoints.

## MMU.DUMP Page wise display of MMU translation table

| | |
|---|---|
| Format: | **MMU.DUMP** *<table>* [*<range>* \| *<address>* \| *<range>* *<root>* \| *<address>* *<root>*]<br>**MMU.***<table>***.dump** (deprecated) |
| *<table>*: | **PageTable**<br>**KernelPageTable**<br>**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>*<br>*<address>* | Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*.<br><br>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.<br>• if *<range>* or *<address>* have a space ID: displays the translation table of the specified process<br>• else, this command displays the table the CPU currently uses for MMU translation. |

| | |
|---|---|
| **KernelPageTable** | Displays the MMU translation table of the kernel.<br>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.<br>• For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).<br>• See also the appropriate **OS Awareness Manuals**. |

## CPU specific tables in MMU.DUMP <table>

| | |
|---|---|
| **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. |

| | |
|---|---|
| Format: | **MMU.List** *<table>* [*<range>* \| *<address>* \| *<range>* *<root>* \| *<address>* *<root>*]<br>**MMU.***<table>***.List** (deprecated) |
| *<table>*: | **PageTable**<br>**KernelPageTable**<br>**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>*<br>*<address>* | Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*.<br><br>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.<br>• if *<range>* or *<address>* have a space ID: list the translation table of the specified process<br>• else, this command lists the table the CPU currently uses for MMU translation. |
| **KernelPageTable** | Lists the MMU translation table of the kernel.<br>If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation. |
| **TaskPageTable**<br>*<task_magic>* \|<br>*<task_id>* \|<br>*<task_name>* \|<br>*<space_id>***:0x0** | Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want.<br>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.<br>• For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).<br>• See also the appropriate **OS Awareness Manuals**. |

| | |
|---|---|
| Format: | **MMU.SCAN** *<table>* [*<range>* *<address>*]<br>**MMU.***<table>***.SCAN** (deprecated) |
| *<table>*: | **PageTable**<br>**KernelPageTable**<br>**TaskPageTable** *<task_magic>* \| *<task_id>* \| *<task_name>* \| *<space_id>***:0x0**<br>**ALL** [**Clear**]<br>*<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.<br>• if *<range>* or *<address>* have a space ID: loads the translation table of the specified process<br>• else, this command loads the table the CPU currently uses for MMU translation. |

| | |
|---|---|
| **KernelPageTable** | Loads the MMU translation table of the kernel.<br>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.<br>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.<br>• For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).<br>• See also the appropriate **OS Awareness Manual**. |
| **ALL** [**Clear**] | Loads all known MMU address translations.<br>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.<br>See also the appropriate **OS Awareness Manual**.<br>**Clear:** This option allows to clear the static translations list before reading it from all page translation tables. |

## TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

## TrOnchip.RESet — Set on-chip trigger to default state

Format: **TrOnchip.RESet**

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

## TrOnchip.CONVert — Adjust range breakpoint in on-chip resource

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

By default a read/write breakpoint to a 16- or 32-bit value in memory will be realized as an on-chip read/write breakpoint for an address range. For example to break on a write access to the 32 Bit Word starting at address 0x100 an on-chip breakpoint for the address range 0x100--0x103 will be used. When the **TrOnchip.CONVert** option is set to **ON** and there are not enough on-chip breakpoint resources available to realize all on-chip breakpoints, the debugger will try to convert these special cases to single address Read/Write Breakpoints, to use the on-chip breakpoint resources more efficiently.

```
TrOnchip.CONVert On                ; Allow conversion
Break.Set 0x100--0x103 /Write      ; This two breakpoints may be
Break.Set 0x200--0x203 /Write      ; converted to single address
                                   ; breakpoints
```

Format: **TrOnchip.VarCONVert** [**ON** | **OFF**] (deprecated)
**Use Break.CONFIG.VarConvert instead**

The on-chip breakpoints can only cover specific ranges. If you want to set a marker or breakpoint to a complex variable, the on-chip break resources of the CPU may be not powerful enough to cover the whole structure. If the option **TrOnchip.VarCONVert** is set to **ON**, the breakpoint will automatically be converted into a single address breakpoint. This is the default setting. Otherwise an error message is generated.

| Memory Class | Description |
|---|---|
| P: | Program. Accesses to this memory class will bypass the data cache. |
| D: | Data. Accesses to the memory class will use the cache (if available) to access the memory. |
| NC: | No Cache. Accesses to this memory class will bypass the data cache. (This class has the same functionality as the P: class) |

For Nios II cores which don't have a data cache all three memory classes have the same behavior.

| Signal | Pin | Pin | Signal |
|---|---|---|---|
| TCK | 1 | 2 | GND |
| TDO | 3 | 4 | VTREF |
| TMS | 5 | 6 | N/C |
| N/C | 7 | 8 | RST- |
| TDI | 9 | 10 | GND |

This image shows the top view to the male connector on the target board. The meaning of the Pins is as follows:

**TCK** Jtag Clock. It is recommended to put a pull-DOWN to GND on this signal.

**TMS** Jtag TMS. It is recommended to put a pull-UP to VCC on this signal.

**TDI** Jtag TDI. It is recommended to put a pull-UP to VCC on this signal.

**TDO** Jtag TDO. (No pull-up, or pull down is needed for this signal.)

**VTREF** Reference voltage. This voltage should indicate the nominal HIGH level for the JTAG pins. So for example, if your signals have a voltage swing from 0 … 3.3 V, the VTREF pin should be connected to 3.3 V.

**RST-** Optional. This pin is not used at the moment and is intended for future use:
If your board has a low active CPU reset signal, you can connect this low active reset signal to this pin. The debugger can drive this pin to GND to hold the CPU in the reset state. The debugger drives this pin as open-drain, so a pull-up is mandatory.

| Signal | Pin | Pin | Signal |
|---|---|---|---|
| N/C | 1 | 2 | N/C |
| N/C | 3 | 4 | N/C |
| N/C | 5 | 6 | CLK |
| N/C | 7 | 8 | TRIGB |
| RST- | 9 | 10 | TRIGA |
| TDO | 11 | 12 | VTREF |
| N/C | 13 | 14 | N/C |
| TCK | 15 | 16 | D11 |
| TMS | 17 | 18 | D10 |
| TDI | 19 | 20 | D09 |
| N/C | 21 | 22 | D08 |
| N/C | 23 | 24 | D07 |
| N/C | 25 | 26 | D06 |
| D17 | 27 | 28 | D05 |
| D16 | 29 | 30 | D04 |
| D15 | 31 | 32 | D03 |
| D14 | 33 | 34 | D02 |
| D13 | 35 | 36 | D01 |
| D12 | 37 | 38 | D00 |

The pins have the following meaning:

**TCK** Jtag Clock. It is recommended to put a pull-DOWN to GND on this signal.

**TMS** Jtag TMS. It is recommended to put a pull-UP to VCC on this signal.

**TDI** Jtag TDI. It is recommended to put a pull-UP to VCC on this signal.

**TDO** Jtag TDO. (No pull-up, or pull down is needed for this signal.)

**VTREF** Reference voltage. This voltage should indicate the nominal HIGH level for the JTAG and trace pins. So for example, if your signals have a voltage swing from 0V - 3.3V, the VTREF pin should be connected to 3.3V.

**RST-** Optional. This pin is not used at the moment and is intended for future use:
If your board has a low active CPU reset signal, you can connect this low active reset signal to this pin. The debugger can drive this pin to GND to hold the CPU in the reset state. The debugger drives this pin as open-drain, so a pull-up is mandatory.

**CLK** Trace Clock.

**D00-D17** Trace Data.

| | |
|---|---|
| **TRIGA** | Optional. Trace Trigger. At the moment the trace logic of the Nios II core supports one trigger output. This output can be used to trigger actions of the external trace (for example stopping a trace recording). |
| **TRIGB** | Optional. Trace Trigger. At the moment the trace logic of the Nios II core only supports one trigger output, so this pin is intended for future use. You might leave it unconnected, if you have not enough pins available on your FPGA. |

If possible the PCB trace lengths of **CLK** and **D00-D17** should have the same lengths, since this signals carry high frequency data.

It is possible to use the 10-pin connector for the JTAG signals (**TCK**, **TMS**, **TDI**, **TDO** and **RST-**) and to use the mictor connector for the trace signals (**CLK**, **D00-D17**, **TRIGA** and **TRIGB**) exclusively (you should leave the JTAG signals on the mictor connector unconnected in this case). In this case you can use different voltage levels for the trace signals and the JTAG signals. You have to provide the correct voltage levels on the **VTREF** pins for both connectors in this case.