SYStem.Mode
SYStem.Option BTM
SYStem.Option CFGCLK
SYStem.Option DCFLUSH
SYStem.Option DBGALL
SYStem.Option LocalRESReset
SYStem.Option DTM
SYStem.Option Endianness
SYStem.Option FSS
SYStem.Option ICFLUSH
SYStem.Option IMASKASM
SYStem.Option IMASKHLL
SYStem.Option MMUSPACES
SYStem.Option TOFF
SYStem.Option SYNC

Configuring your FPGA
JTAG.LOADRBF

JTAG Uart Support

On-chip Breakpoints
Program Breakpoints
Read and Write Breakpoints
Data Breakpoints
Trace Control Breakpoints

CPU specific MMU Commands
MMU.DUMP
MMU.List
MMU.SCAN

TrOnchip Commands
TrOnchip.state
TrOnchip.RESet
TrOnchip.CONVert
TrOnchip.VarCONVert

Memory Classes

BDM Connector ICD-NIOS II
NIOS II Trace Connector

Support
Available Tools
Compilers
Target Operating Systems
3rd-Party Tool Integrations
Brief Overview of Documents for New Users

Architecture-independent information:

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

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.

- **“RTOS Debuggers” (rtos_<x>.pdf)**: TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate RTOS manual informs you how to enable the OS-aware debugging.
NOTE: 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.
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

The debugger is accessed via Internet/VPN and the performance is very slow. What can be done to improve debug performance?

The main cause for bad debug performance via Internet or VPN are low data throughput and high latency. The ways to improve performance by the debugger are limited:

In PRACTICE scripts, use "SCREEN.OFF" at the beginning of the script and "SCREEN.ON" at the end. "SCREEN.OFF" will turn off screen updates. Please note that if your program stops (e.g. on error) without executing "SCREEN.OFF", some windows will not be updated.

"SYStem.POLLING SLOW" will set a lower frequency for target state checks (e.g. power, reset, jtag state). It will take longer for the debugger to recognize that the core stopped on a breakpoint.

"SETUP.URATE 1.s" will set the default update frequency of Data.List/Data.dump/Variable windows to 1 second (the slowest possible setting).

prevent unneeded memory accesses using "MAP.UPDATEONCE [address-range]" for RAM and "MAP.CONST [address--range]" for ROM/FLASH. Address ranged with "MAP.UPDATEONCE" will read the specified address range only once after the core stopped at a breakpoint or manual break. "MAP.CONST" will read the specified address range only once per SYStem.Mode command (e.g. SYStem.Up).
### Setting a Software Breakpoint fails

**Ref:** 0276

**What can be the reasons why setting a software breakpoint fails?**

Setting a software breakpoint can fail when the target HW is not able to implement the wanted breakpoint.

Possible reasons:

- The wanted breakpoint needs special features that are only possible to realize by the trigger unit inside the controller.
  
  Example: Read, write and access (Read/Write) breakpoints ("type" in Break.Set window). Breakpoints with checking in real-time for data-values ("Data").
  
  Breakpoints with special features ("action") like TriggerTrace, TraceEnable, TraceOn/TraceOFF.

- TRACE32 can not change the memory.
  
  Example: ROM and Flash when no preparation with FLASH.Create, FLASH.TARGET and FLASH.AUTO was made. All type of memory if the memory device is missing the necessary control signals like WriteEnable or settings of registers and SpecialFunctionRegisters (SFR).

- Contrary settings in TRACE32.
  
  Like: MAP.BOnchip for this memory range. Break.SElect.<breakpoint-type> Onchip (HARD is only available for ICE and FIRE).

- RTOS and MMU:
  
  If the memory can be changed by Data.Set but the breakpoint doesn't work it might be a problem of using an MMU on target when setting the breakpoint to a symbolic address that is different than the writable and intended memory location.

### NIOS

**Instantiating the Off-chip Trace Logic**

**Ref:** 0245

**Which preparations are necessary for off-chip trace in a NIOS II design?**

Using the off-chip trace in a NIOS II design requires special preparation. This application note explains the steps to implement a reliable off-chip trace port for your Nios II system.
<table>
<thead>
<tr>
<th>NIOS</th>
<th>Why is my Nios II stuck at the reset vector and I cannot step away from it?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nios II is stuck at reset vector</td>
<td>Modern Nios II cores have the option to implement a &quot;cpu_resetrequest&quot; input and a &quot;cpu_resettaken&quot; output. By asserting &quot;cpu_resetrequest&quot;, the Nios II core can be reset without resetting anything else in the system. The Nios II core will acknowledge the reset request by asserting &quot;cpu_resettaken&quot;, once the Nios II core has successfully executed the reset. The Nios II debug logic is implemented in such a way, that the debugger can still access a Nios II core while the &quot;cpu_resetrequest&quot; signal is asserted. So you can examine and modify the processor registers and you also can access memory, while the &quot;cpu_resetrequest&quot; signal is asserted. The exception to this behavior is the program counter: As long as the &quot;cpu_resetrequest&quot; signal is asserted, the program counter is stuck at the reset vector and the Nios II core will not execute code if you issue a &quot;step&quot; or &quot;go&quot; command. Also as long as the Nios II core is stopped (so while the debugger shows that the Nios II core is not running), the Nios II core WILL NOT assert the &quot;cpu_resettaken&quot; signal. Any reset logic reacting to the &quot;cpu_resettaken&quot; signal will NOT receive an acknowledge as long as the Nios II core is stopped. Reset logic should be designed to handle the case that a Nios II core is stopped by a debugger and because of that the Nios II core does not assert the &quot;cpu_resettaken&quot; signal for a long time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NIOS</th>
<th>Analyzer.List/Trace.List doesn't show correct data. What's wrong?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace for Nios II doesn't work correctly</td>
<td>If you enable the off-chip trace for a Nios II CPU in the SOPC Builder then we recommend to make some manual modifications to your design to improve the reliability of the trace. This modifications are described in an application note which you can find in your installation directory under pdf/app_nios_trace.pdf. If you want to try without this modifications you HAVE to enable the &quot;FSS&quot; system option. (For example with the command &quot;System.Option FSS ON&quot;).</td>
</tr>
</tbody>
</table>
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. Select the Device Prompt for the ICD Debugger

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

3. 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.
4. 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 `SYStem.JtagClock`. The maximum clock frequency depends on the configuration of your FPGA design. The default clock frequency is 1 MHz.

5. 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.

6. 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
```

7. 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.
8. 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.
```

9. 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)
```

10. 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 (*.cmm, ASCII file format) to be able to do all the necessary actions with only one command. Here is a typical start sequence:

```
B:: ; Select the ICD device prompt
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
Data.List ; Open disassembly window *)
Register /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 “Debugger Basics - Training” (training_developer.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.
| On-chip Breakpoints | 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. |
### SYStem.CONFIG

**Configure multi-core debugger**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.CONFIG</th>
<th>&lt;parameter&gt;</th>
<th>&lt;number_or_address&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SYStem.MultiCore</td>
<td>&lt;parameter&gt;</td>
<td>&lt;number_or_address&gt;</td>
</tr>
</tbody>
</table>

| <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 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> tbd.

Example:

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

Instruction register length of

<table>
<thead>
<tr>
<th>Device</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device A</td>
<td>3 bit</td>
</tr>
<tr>
<td>Device B</td>
<td>5 bit</td>
</tr>
<tr>
<td>Device C</td>
<td>5 bit</td>
</tr>
<tr>
<td>Device D</td>
<td>4 bit</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CONFIG IRPRE  4</td>
<td>IR Device D</td>
</tr>
<tr>
<td>SYStem.CONFIG IRPOST 8</td>
<td>IR Device A + B</td>
</tr>
<tr>
<td>SYStem.CONFIG DRPRE  1</td>
<td>DR Device D</td>
</tr>
<tr>
<td>SYStem.CONFIG DRPOST 2</td>
<td>DR Device A + B</td>
</tr>
</tbody>
</table>
TapStates

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
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 important 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:

| Format: | SYStem.CONFIG.Core <core_number> <chip_number> |
| 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:

| Format: | SYStem.CONFIG.Core 1 1 |
| 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:

| Format: | SYStem.CONFIG.Core 2 1 |
| 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CONFIG IRPRE 0</td>
<td>No device after Cyclone in chain</td>
</tr>
<tr>
<td>SYStem.CONFIG IRPOST 10</td>
<td>IR Stratix</td>
</tr>
<tr>
<td>SYStem.CONFIG DRPRE 0</td>
<td>No device after Cyclone in chain</td>
</tr>
<tr>
<td>SYStem.CONFIG DRPOST 1</td>
<td>DR Stratix</td>
</tr>
</tbody>
</table>

And additionally:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CONFIG.Core 1 2</td>
<td>Connect to Core 1 in Cyclone (Chip 2)</td>
</tr>
</tbody>
</table>
SYStem.CONFIG.state

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.CPU

Select CPU type

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

©1989-2018 Lauterbach GmbH
SYstem.CpuAccess

Run-time memory access (intrusive)

Format: `SYstem.CpuAccess Enable | Denied | Nonstop`

Default: Denied.

**Enable**  
Allow intrusive run-time memory access.  
In order to perform a memory read or write while the CPU is executing the program, the debugger stops the program execution shortly. Each short stop takes 1 … 100 ms depending on the speed of the debug interface and on the number of the read/write accesses required.  
A red S in the state line of the TRACE32 main window indicates this intrusive behavior of the debugger.

**Denied**  
Lock intrusive run-time memory access.

**Nonstop**  
Lock all features of the debugger that affect the run-time behavior.  
Nonstop reduces the functionality of the debugger to:  
- Run-time access to memory and variables  
- Trace display  
The debugger inhibits the following:  
- To stop the program execution  
- All features of the debugger that are intrusive (e.g. action Spot for breakpoints, performance analysis via StopAndGo mode, conditional breakpoints, etc.)

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 lock command is to give debug access to another tool.

### SYStem.MemAccess
Real-time memory access (non-intrusive)

**Format:**

\[
\text{SYStem.MemAccess Denied} \mid <\text{cpu_specific}>
\]

SYStem.ACCESS (deprecated)

**Denied**  
Real-time memory access during program execution to target is disabled.

### SYStem.Mode
Select target reset mode

**Format:**

\[
\text{SYStem.Mode} \ <\text{mode}>
\]

**<mode>:**

- **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.
SYStem.Option BTM  Enable/disable branch trace

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.

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

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 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.
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.

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`.
**SYStem.Option FSS**

Enable/disable FS2 compatibility mode

**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 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

Enable 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.

NOTE: SYStem.Option MMUSPACES should not be used 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 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. tbd.
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 ...**
  - All Stratix, Stratix II, Stratix III, Stratix IV devices.
- **Stratix IV**
- **Cyclone ...**
  - All Cyclone, Cyclone II, Cyclone III, Cyclone IV devices.
- **Cyclone IV**
- **Arria GX ...**
  - All Arria GX, Arria II GX devices.
- **Arria II GX**

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.
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:

```plaintext
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 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.
CPU specific MMU Commands

**MMU.DUMP**

Page wise display of MMU translation table

| Format: | MMU.DUMP <table> [<range> | <addr> | <range> | <root> | <addr> | <root>] |
| --- | --- |
| MMU.<table>dump (deprecated) |

<table>:
- PageTable
- KernelPageTable
- TaskPageTable <magic_number> | <task_id> | <task_name>
- <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.

The optional <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.

**PageTable**
Display the current MMU translation table entries of the CPU.
This command reads all tables the CPU currently uses for MMU translation and displays the table entries.

**KernelPageTable**
Display 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**
Display the MMU translation table entries of the given process.
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.
See also the appropriate OS awareness manuals: RTOS Debugger for <x>.
For information about the parameters, see “What to know about Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf).

**CPU specific tables:**

**ITLB**
Displays the contents of the Instruction Translation Lookaside Buffer.
MMU.List

Compact display of MMU translation table

Format:

MMU.List <table> [ <range> | <addr> | <range> <root> | <addr> <root> ]
MMU.<table>.List (deprecated)

<table>:
PageTable
KernelPageTable
TaskPageTable <magic_number> | <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 optional <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.

PageTable
List the current MMU translation of the CPU. This command reads all tables the CPU currently uses for MMU translation and lists the address translation.

KernelPageTable
List 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
List the MMU translation of the given process. 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. See also the appropriate OS awareness manuals: RTOS Debugger for <x>.

For information about the parameters, see “What to know about Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf).
MMU.SCAN

Load MMU table from CPU

Loads the CPU-specific MMU translation table from the CPU to the debugger-internal translation table. If called without parameters, the complete page table will be loaded. The loaded address translation can be viewed with TRANslatoin.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.

PageTable
Load the current MMU address translation of the CPU.
This command reads all tables the CPU currently uses for MMU translation, and copies the address translation into the debugger-internal translation table.

KernelPageTable
Load 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 translation table.

TaskPageTable <magic_number> | <task_id> | <task_name>
Load the MMU address translation of the given process.
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 translation table.
See also the appropriate OS awareness manual: RTOS Debugger for <x>.
For information about the parameters, see “What to know about Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf).

ALL
Load 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 translation table.
See also the appropriate OS awareness manual: RTOS Debugger for <x>.
**TrOnchip Commands**

### 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 a 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 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
Break.Set 0x100--0x103 /Write ; Allow conversion
Break.Set 0x200--0x203 /Write ; This two breakpoints may be
Break.Set 0x200--0x203 /Write ; converted to single address
Break.Set 0x200--0x203 /Write ; breakpoints
```
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 **ON** the breakpoint will automatically be converted into a single address breakpoint. This is the default setting. Otherwise an error message is generated.

| Format: | `TrOnchip.VarCONVert` [ON | OFF] |

©1989-2018 Lauterbach GmbH
# Memory Classes

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P:</td>
<td>Program. Accesses to this memory class will bypass the data cache.</td>
</tr>
<tr>
<td>D:</td>
<td>Data. Accesses to the memory class will use the cache (if available) to access the memory.</td>
</tr>
<tr>
<td>NC:</td>
<td>No Cache. Accesses to this memory class will bypass the data cache. (This class has the same functionality as the P: class)</td>
</tr>
</tbody>
</table>
This image shows the top view to the male connector on the target board. The meaning of the Pins is as follows:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCK</td>
<td>1</td>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>TDO</td>
<td>3</td>
<td>4</td>
<td>VTREF</td>
</tr>
<tr>
<td>TMS</td>
<td>5</td>
<td>6</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>7</td>
<td>8</td>
<td>RST-</td>
</tr>
<tr>
<td>TDI</td>
<td>9</td>
<td>10</td>
<td>GND</td>
</tr>
</tbody>
</table>

**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.
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.
## Available Tools

<table>
<thead>
<tr>
<th>CPU</th>
<th>ICE</th>
<th>FIRE</th>
<th>ICD DEBUG</th>
<th>ICD MONITOR</th>
<th>ICD TRACE</th>
<th>POWER INTEGRATOR</th>
<th>INSTRUCTION MONITOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIOS-II</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

## Compilers

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Company</th>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GCC</td>
<td>Altera Corporation</td>
<td>ELF/DWARF2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>VX-NIOS</td>
<td>TASKING</td>
<td>ELF/DWARF2</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>GCC</td>
<td>Altera Corporation</td>
<td>ELF/DWARF2</td>
<td></td>
</tr>
</tbody>
</table>

## Target Operating Systems

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>eCosCentric Limited</td>
<td>ECOS</td>
<td>1.3, 2.0 and 3.0</td>
</tr>
<tr>
<td>-</td>
<td>Linux</td>
<td>Kernel Version 2.6, 3.x, 4.x</td>
</tr>
<tr>
<td>MISPO Co. Ltd.</td>
<td>NORTI</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>OSEK</td>
<td>via ORTI</td>
</tr>
<tr>
<td>eSOL Co., Ltd.</td>
<td>prKERNEL</td>
<td></td>
</tr>
<tr>
<td>Elektrobit Automotive GmbH</td>
<td>ProOSEK</td>
<td>via ORTI</td>
</tr>
<tr>
<td>RTEMS</td>
<td>RTEMS</td>
<td>up to v5</td>
</tr>
<tr>
<td>Express Logic Inc.</td>
<td>ThreadX</td>
<td>3.0, 4.0, 5.0</td>
</tr>
<tr>
<td>Micrium Inc.</td>
<td>uC/OS-II</td>
<td>2.0 to 2.92</td>
</tr>
<tr>
<td>-</td>
<td>uCLinux</td>
<td>Kernel Version 2.4 and 2.6, 3.x</td>
</tr>
<tr>
<td>-</td>
<td>uITRON</td>
<td>HI7000, RX4000, NORTi,PrKernel</td>
</tr>
</tbody>
</table>
## 3rd-Party Tool Integrations

<table>
<thead>
<tr>
<th>CPU</th>
<th>Tool</th>
<th>Company</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOWS CE PLATF. BUILDER</td>
<td>-</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CODE::BLOCKS</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C++TEST</td>
<td>-</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>ADENEO</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X-TOOLS / X32</td>
<td>blue river software GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>CODEWRIGHT</td>
<td>Borland Software Corporation</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>CODE CONFIDENCE TOOLS</td>
<td>Code Confidence Ltd</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>CODE CONFIDENCE TOOLS</td>
<td>Code Confidence Ltd</td>
<td>Linux</td>
<td></td>
</tr>
<tr>
<td>EASYCODE</td>
<td>EASYCODE GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>ECLIPSE</td>
<td>Eclipse Foundation, Inc</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>CHRONVIEW</td>
<td>Inchron GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>LDRA TOOL SUITE</td>
<td>LDRA Technology, Inc.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>UML DEBUGGER</td>
<td>LieberLieber Software GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>SIMULINK</td>
<td>The MathWorks Inc.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>ATTOL TOOLS</td>
<td>MicroMax Inc.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>VISUAL BASIC INTERFACE</td>
<td>Microsoft Corporation</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>LABVIEW</td>
<td>NATIONAL INSTRUMENTS Corporation</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>RAPITIME</td>
<td>Rapita Systems Ltd.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>RHAPSODY IN MICROC</td>
<td>IBM Corp.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>RHAPSODY IN C++</td>
<td>IBM Corp.</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>DA-C</td>
<td>RistanCASE</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>TRACEANALYZER</td>
<td>Symtavision GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>ECU-TEST</td>
<td>TraceTronic GmbH</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>UNDODB</td>
<td>Undo Software</td>
<td>Linux</td>
<td></td>
</tr>
<tr>
<td>TA INSPECTOR</td>
<td>Vector</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>VECTORCAST UNIT TESTING</td>
<td>Vector Software</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>VECTORCAST CODE COVERAGE</td>
<td>Vector Software</td>
<td>Windows</td>
<td></td>
</tr>
</tbody>
</table>
## Product Information

<table>
<thead>
<tr>
<th>OrderNo</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
</table>
| LA-2809L  | SIMULATOR-C6000-FL | **1 User Float. Lic. T32 TMS32C6000 Simulator**  
Floating license to use the TRACE32 Instruction Set  
Simulator for automated tests via script language  
PRACTICE or via the TRACE32 Remote API  
supports TMS32C6000  
for Windows32, Windows64, Linux32, Linux64  
and Solaris, other platforms on request  
floating license via RLM (Reprise License Manager)  
Please add the RLM HostID of the license server to your order (please see our FAQ) |
| LA-7837   | JTAG-NIOS-II     | **Debugger for NIOS-II (ICD)**  
supports NIOS-II  
includes software for Windows, Linux and MacOSX  
requires Power Debug Module |
| LA-3863   | CON-ARM-ALTERA   | **Converter ARM-20 to ALTERA-10/RISCV-10**  
Converter from ARM 20-pin connector to  
Altera 10-pin (Byteblaster) connector or  
RISC-V 10-pin  
Target connector pitch can be  
2.54mm or 1.27mm (half size). |
| LA-7837A  | JTAG-NIOS-II-A   | **JTAG Debugger License for NIOS-II Add.**  
supports NIOS-II  
please add the base serial number of your debug cable to your order |
| LA-3801   | PP-NIOSII        | **Preprocessor for NIOS-II Flex Cable**  
Preprocessor for NIOS-II,  
150 MHz DDR clock speed,  
with MICTOR38 connector  
requires PowerTrace |
| LA-3917A  | PP-NIOS-AF-2-A   | **Trace License for Nios II AUTOFOCUS II**  
Supports off-chip trace port for Nios II if applied to an AUTOFOCUS II preprocessor  
please add the serial number of the preprocessor to your order  
AUTOFOCUS II Preprocessors with serial number  
C0806xxxxxx and lower have to be send to Lauterbach Germany for an hardware upgrade |
## Order Information

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-2809L</td>
<td>SIMULATOR-C6000-FL</td>
<td>1 User Float. Lic. T32 TMS32C6000 Simulator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-7837</td>
<td>JTAG-NIOS-II</td>
<td>Debugger for NIOS-II (ICD)</td>
</tr>
<tr>
<td>LA-3863</td>
<td>CON-ARM-ALTERA</td>
<td>Converter ARM-20 to ALTERA-10/RISCV-10</td>
</tr>
<tr>
<td>LA-7837A</td>
<td>JTAG-NIOS-II-A</td>
<td>JTAG Debugger License for NIOS-II Add.</td>
</tr>
</tbody>
</table>

### Additional Options

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-7844A</td>
<td>JTAG-CORTEX_M-A</td>
<td>JTAG Debugger License for Cortex-M Add.</td>
</tr>
<tr>
<td>LA-7960X</td>
<td>MULTICORE-LICENSE</td>
<td>License for Multicore Debugging</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-3801</td>
<td>PP-NIOSII</td>
<td>Preprocessor for NIOS-II Flex Cable</td>
</tr>
<tr>
<td>LA-3917A</td>
<td>PP-NIOS-AF-2-A</td>
<td>Trace License for Nios II AUTOFOCUS II</td>
</tr>
</tbody>
</table>