ICD Debugger User’s Guide

Version 30-Apr-2021

History

06-Aug-19	Added an for the command line option --t32-screen=.

WARNING:

To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.

Recommendation for the software start:

1. Disconnect the debug cable from the target while the target power is off.
2. Connect the host system, the TRACE32 hardware and the debug cable.
3. Power ON the TRACE32 hardware.
4. Start the TRACE32 software to load the debugger firmware.
5. Connect the debug cable to the target.
6. Switch the target power ON.
7. Configure your debugger e.g. via a start-up script.

Power down:

1. Switch off the target power.
2. Disconnect the debug cable from the target.
3. Close the TRACE32 software.
4. Power OFF the TRACE32 hardware.

Important Information Concerning the Use of the TRACE32 Development System

Due to the special nature of the TRACE32 development system, the user is advised that it can generate higher than normal levels of electromagnetic radiation which can interfere with the operation of all kinds of radio and other equipment.

To comply with the European Approval Regulations therefore, the following restrictions must be observed:

1. The development system must be used only in an industrial (or comparable) area.
2. The system must not be operated within 20 metres of any equipment which may be affected by such emissions (radio receivers, TVs etc).

Installation

In this section the installation for the ICD Debugger connected to the PC is described. The configuration of the ICD debug system is done in the file CONFIG.T32 which must reside in the TRACE32 system directory.

Software Installation

See **"TRACE32 Installation Guide"** (installation.pdf)**.**

Command Line Arguments for Starting TRACE32

TRACE32 can be started with and without command line arguments. If you do not pass any command line arguments, the following default will be used:

- For the configuration file, the first available file from the following locations will be used:
  - The file specified by the environment variable **T32CONFIG**
  - The file **config.t32** from the working directory (from where TRACE32 was started).
  - The file **config.t32** from the TRACE32 system directory (usually C:\t32)
- After the start of a TRACE32 instance, the PRACTICE script **autostart.cmm** from the TRACE32 system directory will be executed, which then calls the following scripts:
  - **system-settings.cmm** (from the TRACE32 system directory, usually C:\t32)
  - **user-settings.cmm** (from the user settings directory, on Windows `%APPDATA%\TRACE32` or `~/.trace32` otherwise)
  - **work-settings.cmm** (from the current working directory)

If you pass command line arguments, you can adapt the start and the environment of TRACE32 to your project-specific needs. You can define a user-specific name and location for the configuration file and your own PRACTICE start-up script. In addition, you can pass user-defined parameters to your configuration file and to your start-up script.

The command line syntax for Windows, Linux, and Unix is as follows:

Format:	**t32m***<arch>*[**.***<x>*] [*<options>*] [**-c** *<config_file>* [*<c_args>*]] [**-s** *<startup_script>* [*<s_args>*]]

-c *<config_file>*	By default, TRACE32 expects to find the configuration file config.t32 in the same folder as the TRACE32 executable. An error message is displayed if the configuration file config.t32 does not exist. **-c** *<config_file>* allows you to define a user-specific name and location for the TRACE32 configuration file. For information about the configuration options in the configuration file, type at the TRACE32 command line: `HELP.Index "config.t32"` **NOTE**: • **-c** is case sensitive, i.e. **-C** results in an error. • The name of the *<config_file>* must **not** start with a hyphen and must **not** contain any commas.

-s *<startup_ script>*	When a TRACE32 instance starts, the PRACTICE script **autostart.cmm** is executed, which then calls the following scripts: • **system-settings.cmm** (from the TRACE32 system directory, usually C:\t32) • **user-settings.cmm** (from the user settings directory: on Windows %APPDATA%\TRACE32 or ~/.trace32 otherwise) • **work-settings.cmm** (from the current working directory) With the command line option **-s** *<startup_script>* you can specify an additional PRACTICE script (*.cmm) which is automatically started afterwards. **NOTE**: If you use command line option **-s** *<startup_script>* but don't have the file **autostart.cmm** in your TRACE32 system directory, only the file specified by the command line option **-s** will be executed. If you don't use the command line option **-s** *<startup_script>* and don't have the file **autostart.cmm** either, TRACE32 will fall back to a legacy mode and execute the script **t32.cmm** from the working directory or from the TRACE32 system directory if the **t32.cmm** does not exist in the working directory.
<arch>	Architecture, e.g. ARM in `t32m`**`arm`**`.exe` stands for the ARM architecture.
<c_args>	Sequence of white-space separated arguments passed to the configuration file. The individual command line arguments are assigned to the configuration options in the configuration file using `${n}` where `n` is the number of the command line argument. The numbering starts at `1` **Example**: `HEADER=${2}` means that the second *<c_arg>* is assigned to the configuration option `HEADER=` **NOTE**: The *<c_args>* must **not** start with a hyphen and must **not** contain any commas.

<options>	The following command line options are available for all architectures: • **--t32-help** Lists all available command line options and suppresses the automatic execution of any PRACTICE script after starting TRACE32. • **--t32-help-diag** Prints all found unknown command line options into the message **AREA** window. • **--t32-cmdline-quote-all** Passes all parameters to the start-up script enclosed in double quotes. • **--t32-cmdline-quote-esc** Passes all parameters to start-up script enclosed in double quotes. Additionally the command line option escapes already included double quotes in parameters, so that you get the same character sequence when accessed with **PRACTICE.ARG()**. • **--t32-area-size-lines=***<lines>* Sets the initial number of lines in the message **AREA** window. • **--t32-safestart** Suppresses the automatic execution of any PRACTICE script after starting TRACE32. • **--t32-logautostart** Internally executes the **LOG.DO** command to generate an autostart log in the temporary directory of TRACE32. The resulting autostart log file lists all PRACTICE scripts which were called during start-up of TRACE32. For an example and a description of the file name convention, see below. • **--t32-screen=***<screendriver>* Set the screen driver used to display all PowerView windows. Possible values area: auto, cde, qt, qt4, qt5
<s_args>	Sequence of white-space separated arguments passed to the PRACTICE start-up script (*.cmm). All characters are permissible. However, if an *<s_arg>* is enclosed in quotation marks (e.g. `"Hello"`), then the *<s_arg>* must not be interrupted by another quotation mark. An *<s_arg>* that contains one or more blanks must be enclosed in quotation marks (e.g. `"Hello World"`).
<x>	The file name extension of the executable is optional on Windows. (no extension for Linux and Unix)

Examples

- Example 1: Logging PRACTICE script calls with --t32-logautostart during start-up of TRACE32
- Example 2: Passing command line arguments via a Windows shortcut to TRACE32
- Example 3: Passing command line arguments to a TRACE32 configuration file (*.t32)
- Example 4: Passing command line arguments to a PRACTICE start-up script (*.cmm)
- Example 5: Returning environment variables in a PRACTICE start-up script (*.cmm)

Example 1: Logging PRACTICE script calls during start-up of TRACE32 with --t32-logautostart

[Back to Top]

In this example, a Windows batch file (*.bat) sets the folder **C:\T32\project_c** as the working directory for TRACE32. The next script line starts the TRACE32 executable for ARMv8 by using the configuration file **config.t32** file and the PRACTICE start-up script **sieve.cmm** from the working directory.

The command line option **--t32-logautostart** causes the autostart log file to be generated.

```
C:
cd C:\T32\project_c
start C:\T32\bin\windows64\t32marm64.exe --t32-logautostart   ^
                                         -c config.t32        ^
                                         -s sieve.cmm
```

The caret sign ^ serves as a line continuation character in Windows batch files (*.bat). White space characters after ^ are NOT permissible.

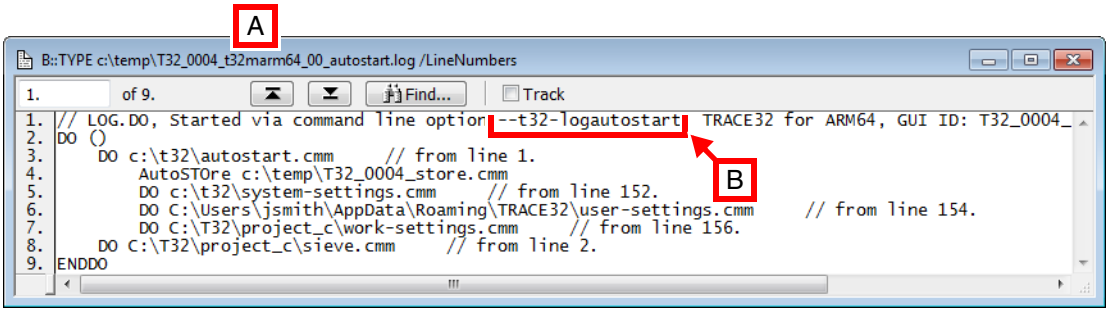
To access the autostart log file in TRACE32:

1. Execute your *.bat file to start TRACE32.

   The autostart log file is generated automatically.

2. Choose **File** menu > **Automatic Scripts on Start** > **View Autostart log**.

   The file opens in the **TYPE** window. The screenshot shows an example of an autostart log file:



A The file name convention of the autostart log is described below.

B The log file header tells you how the autostart log was generated. For alternatives, see **"Logging the Call Hierarchy of PRACTICE Scripts"** (practice_user.pdf).

File name convention of the autostart log file: ~~~/*<id>*_t32m*<arch>*_*<xx>*_autostart.log

~~	Path prefix of the temporary directory of TRACE32. See also **OS.PresentTemporaryDirectory()**.
<id>	ID of the PowerView GUI that was started. See also **OS.ID()**.
t32m*<arch>*	Name of the PowerView executable (without file extension), e.g. "t32marm64"
<xx>	The instance number of the PowerView executable.

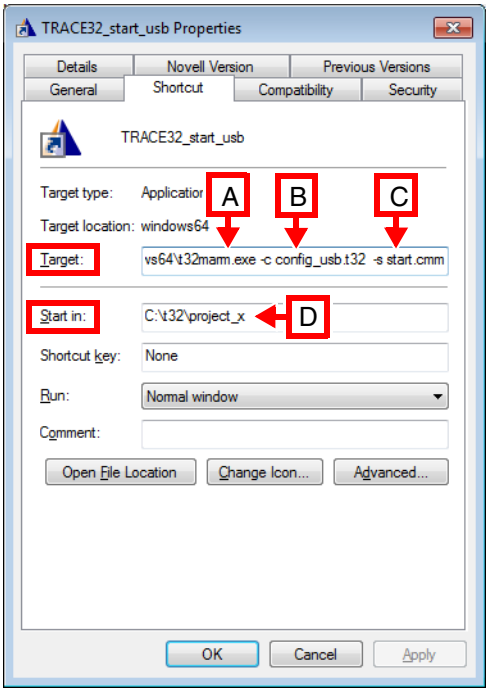
Example 2: Passing command line arguments via a Windows shortcut to TRACE32

[Back to Top]

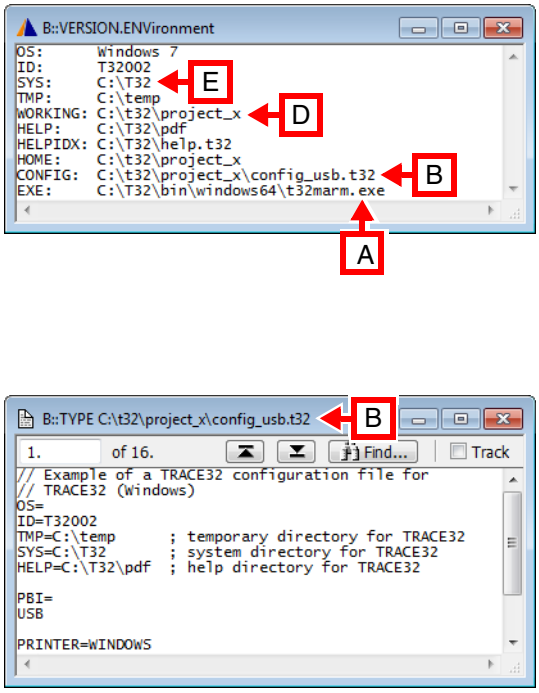
This example shows how to pass TRACE32 command line arguments via a Windows shortcut to TRACE32. The command line arguments are:

- A user-defined configuration file called `config_usb.t32`
- A user-defined PRACTICE start-up script called `start.cmm`

Operating System side



TRACE32 side



A Path and name of a TRACE32 executable.
In a default installation, all TRACE32 executables are located in C:\t32\bin\<*os*>\

B The option **-c** <*config_file*> allows you to define a user-specific name and location for the TRACE32 configuration file (*.t32).

C The option **-s** <*startup_script*> allows you to define a user-specific name and location for your PRACTICE start-up script (*.cmm).

D User-defined working directory.
In the above **Properties** dialog box, the **Start in** text box sets the path for the **Target** text box - unless different paths are specified in the **Target** text box.
This means for our example that the `t32m`**`arm`**`.exe` searches for the files `config_usb.t32` and `start.cmm` in `C:\t32\project_x`.

E TRACE32 system directory (by default `c:\t32`). It is specified during the installation of TRACE32. Normally, you do not need to change anything here.

Example 3: Passing command line arguments to a TRACE32 configuration file (*.t32)

[Back to Top]

The following example shows how to pass TRACE32 command line arguments from a batch file (*.bat) to the configuration file (*.t32). The command line arguments are:

- *<c_arg1>*: A user-defined window title for TRACE32
- *<c_arg2>*: A network folder path containing the pdf files of the TRACE32 online help

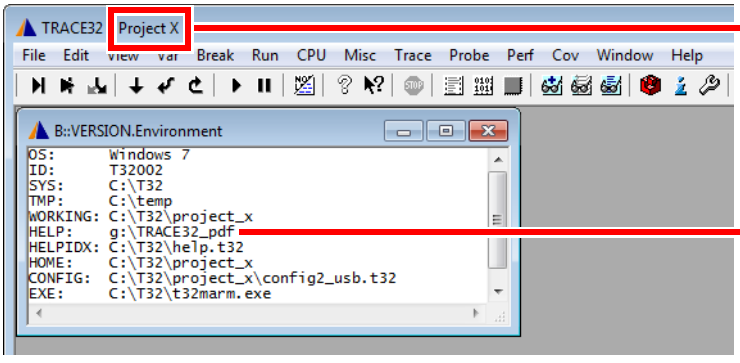
Batch file start2_usb.bat:

```
rem                                                  <c_arg1>      <c_arg2>
start C:\T32\t32marm.exe -c config2_usb.t32 " Project X" "g:\TRACE32_pdf"
```

TRACE32 configuration file config2_usb.t32:

```
// Example of a TRACE32 configuration file
OS=
ID=T32002
TMP=C:\temp       ; temporary directory for TRACE32
SYS=C:\T32        ; system directory for TRACE32
HELP=${2}         ; help directory for TRACE32

PBI=
USB

PRINTER=WINDOWS

SCREEN=
HEADER=TRACE32 ${1}
```

The values passed as command line arguments to the user-defined configuration file are now used by TRACE32:



Value of argument `${1}`

Value of argument `${2}`
TRACE32 now retrieves the pdfs of the help system from this network folder.

NOTE: The `help.t32` file of the online help must reside in the TRACE32 system directory (by default `C:\t32`).

Example 4: Passing command line arguments to a PRACTICE start-up script (*.cmm)

[Back to Top]

The following example shows how to pass TRACE32 command line arguments from an MS batch file (*.bat) to a PRACTICE start-up script (*.cmm). The command line arguments are:

- *<startup_script>*: A user-defined PRACTICE start-up script (*.cmm)
- *<s_args>*: The arguments passed to the PRACTICE start-up script are `0x1` and `0x2` and a string with a blank `"Hello World!"`

Batch file start3.bat:

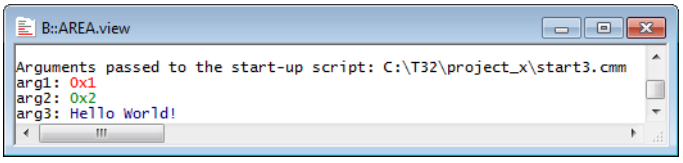
```
rem                                                <startup_script> <s_args>
start C:\T32\t32marm.exe  -c config3_usb.t32   -s start3.cmm     0x1  ^
                                                                  0x2  ^
                                                          "Hello World!"
```

The caret sign ^ serves as a line continuation character in Windows batch files (*.bat). White space characters after ^ are NOT permissible.

PRACTICE start-up script start3.cmm:

```
LOCAL &arg1 &arg2 &arg3      ;Declare local PRACTICE macros

&arg1=PRACTICE.ARG(0)        ;Assign the values of the command line
&arg2=PRACTICE.ARG(1)        ;arguments to the PRACTICE macros
&arg3=PRACTICE.ARG(2)

AREA.view                    ;Open an AREA.view window
PRINT "Arguments passed to the start-up script: "
OS.PresentPracticeFile()
PRINT "arg1: " %COLOR.RED   "&arg1"
PRINT "arg2: " %COLOR.GREEN "&arg2"
PRINT "arg3: " %COLOR.NAVY  "&arg3"
```

The values passed as command line arguments to the PRACTICE start-up script are printed to the **AREA.view** window.



Example 5: Returning environment variables in a PRACTICE start-up script (*.cmm)

[Back to Top]

The first two tables show how to create the user-defined environment variable `T32P1` and start TRACE32 under Windows and Linux. After starting TRACE32, the parameter of `T32P1` is returned in the PRACTICE start-up script start4.cmm with the **OS.ENV()** function and printed to the TRACE32 message line, see 3rd table.

Windows: start4.bat

```
SET T32P1=C:\ain't this\a complicated\path\myprog.elf
START t32marm.exe   -c config4_sim.t32   -s start4.cmm
```

Linux: dash/bash script

```
T32P1="/ain't this/a complicated/path/myprog.elf"
START t32marm.exe   -c config4_sim.t32   -s start4.cmm
```

TRACE32: PRACTICE start-up script start4.cmm

```
;<your_code>

PRINT OS.ENV(T32P1)  ;prints C:\ain't this\a complicated\path\myprog.elf

;...
```

General Information

Multicore debugging in general means that there are multiple cores on one chip which use the same JTAG interface. In contrary, if there is more than one chip with different JTAG interfaces to be debugged simultaneously, we use the term multiprocessor debugging.

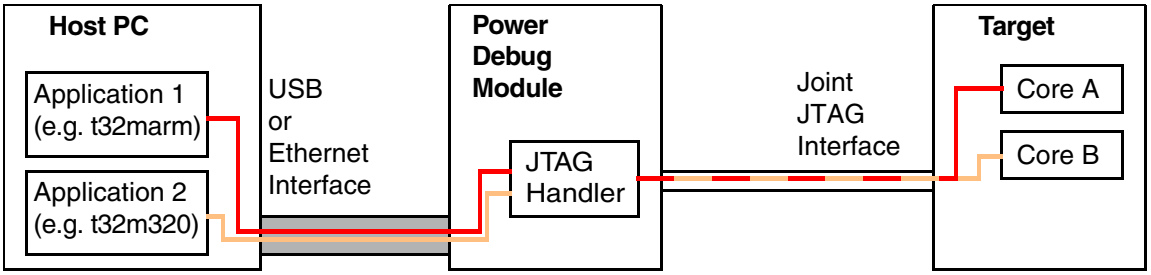
The following chapters describe which hardware and licenses are needed and how to set up the TRACE32 software for multicore debugging. For detailed core specific settings, please refer to the particular Processor Architecture Manuals. In addition, there is a number of PRACTICE sample scripts (*.cmm) for different multicore systems available in the **~~/demo/…** folder or can be received from LAUTERBACH support (for contact possibilities, see www.lauterbach.com).

There are two general ways for multicore debugging:

- single device solution: one debug module (debug box) for several cores
- multi device solution: several debug modules (debug boxes), one for each core

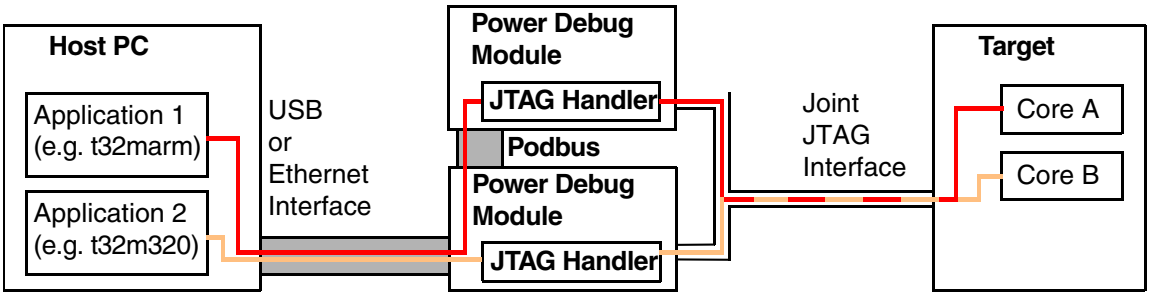
In any case, each core has its own application software on the host PC, but those applications use different ways to access the LAUTERBACH Debug Hardware, and, in the end, their particular core.

Tool Configuration for Single Device Solution



When using single device solution, the applications do not care about availability of the joint JTAG interface, the different accesses are arranged by a JTAG handler. The application executables (t32marm and t32m320 in the diagram above) register themselves at the JTAG handler of the common Power Debug Module. Therefore, the configuration files (default file name: config.t32) of the applications have to contain **`core=`** settings (as specified below).

The master application (those belonging to the base license of the debug cable) has to be started first to install the proper JTAG handler. Nevertheless, the setting to define which core is addressed actually is done later on.

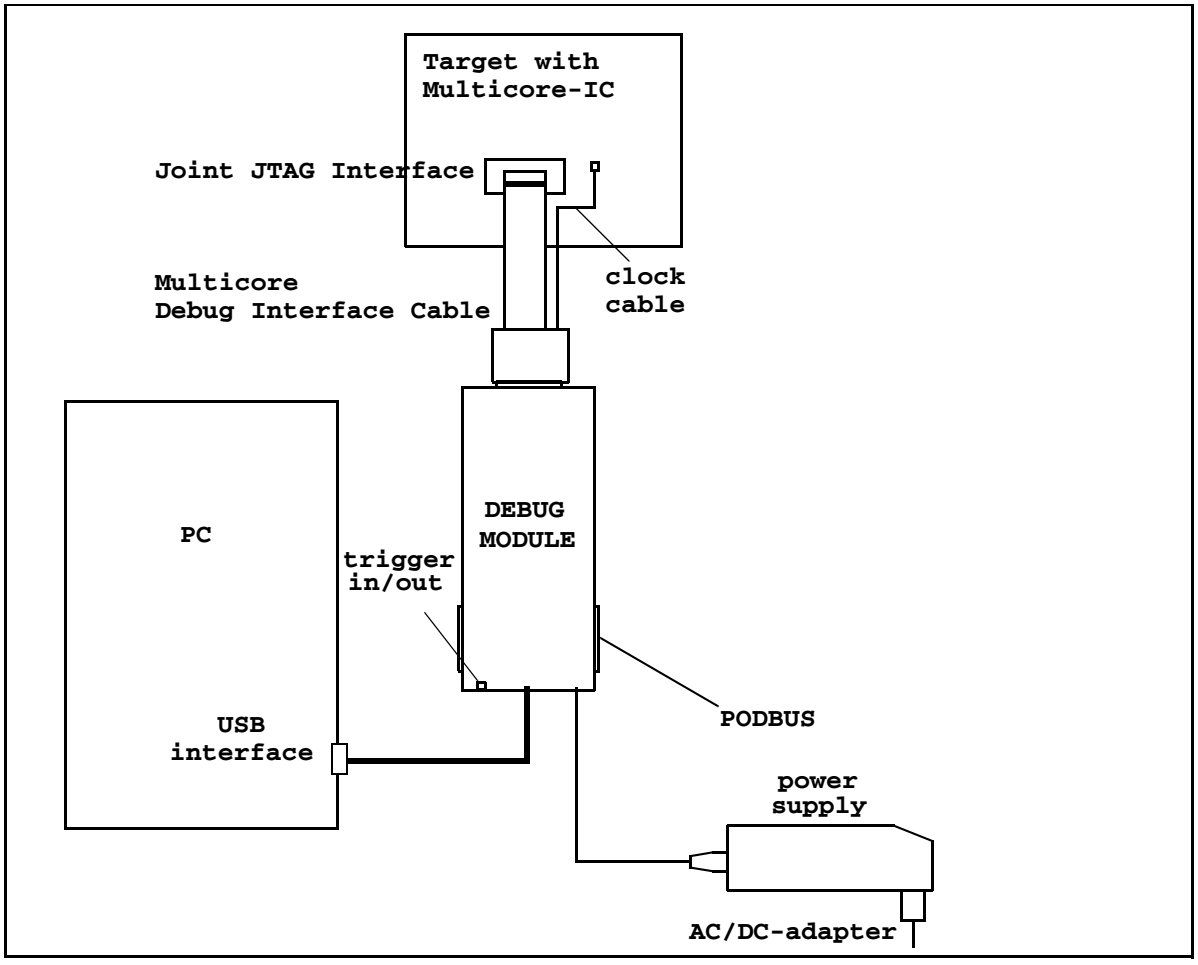


The multi device alternative demands self-administration of the access to the joint JTAG interface for each application. The application reserves the JTAG port for usage, and releases it afterwards transferring its Debug Module into tristate mode. The applications (t32marm and t32m320 in diagram above) have to be configured to use their own separate debug box. Therefore, the configuration files (default file name: config.t32) of the applications have to contain **`use=`** settings (as specified below).

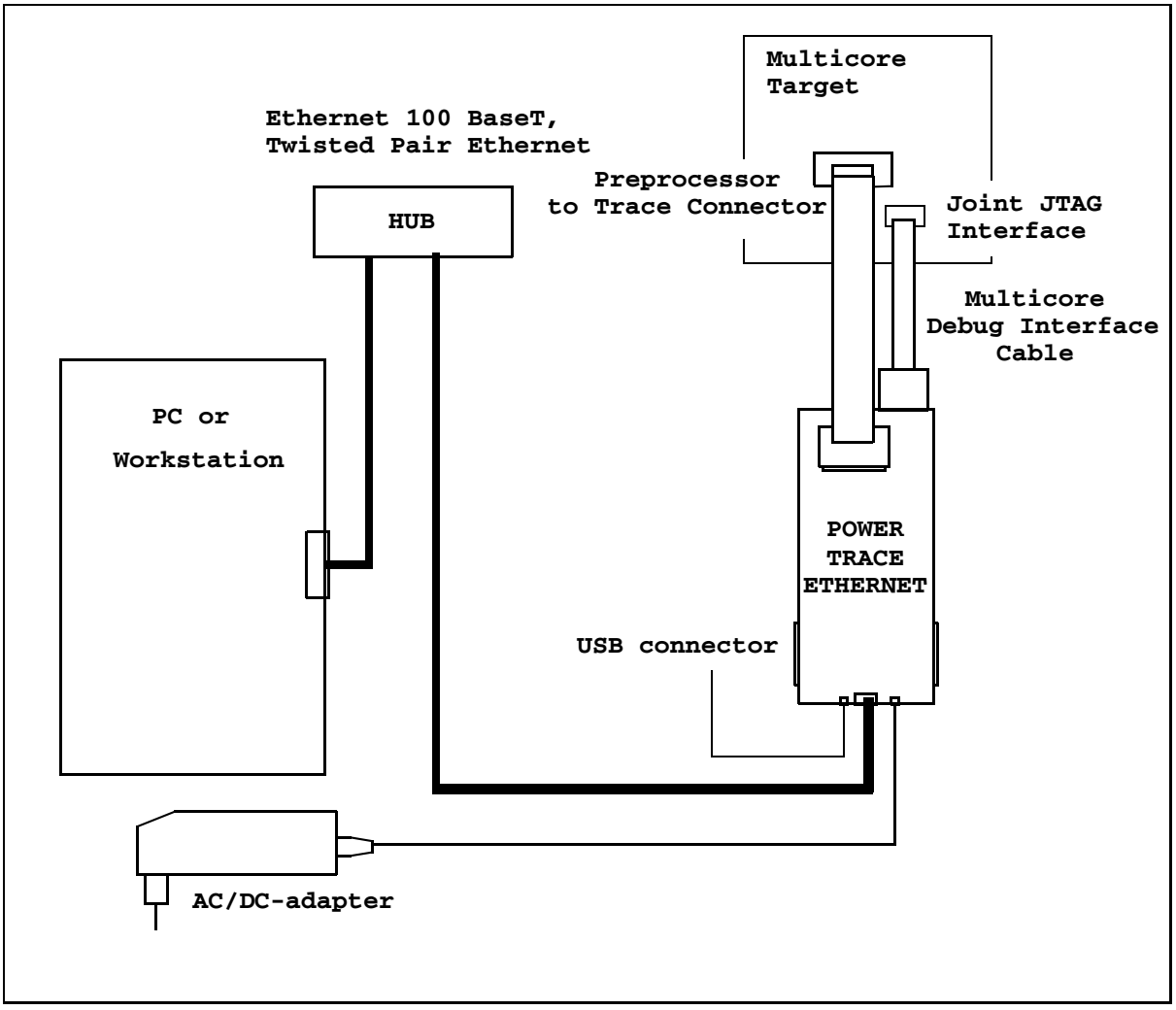
	In this case it does not matter which application is started first. As well as in single device solution, the setting to define which core is addressed at the end is done later on.

The following configuration examples provide an overview on usable LAUTERBACH debugging hardware for multicore debugging. To see which multicore processor ICs are supported by specific configurations, please contact LAUTERBACH sales department or local distributor.

Configuration 1: Single PowerDebug-Module

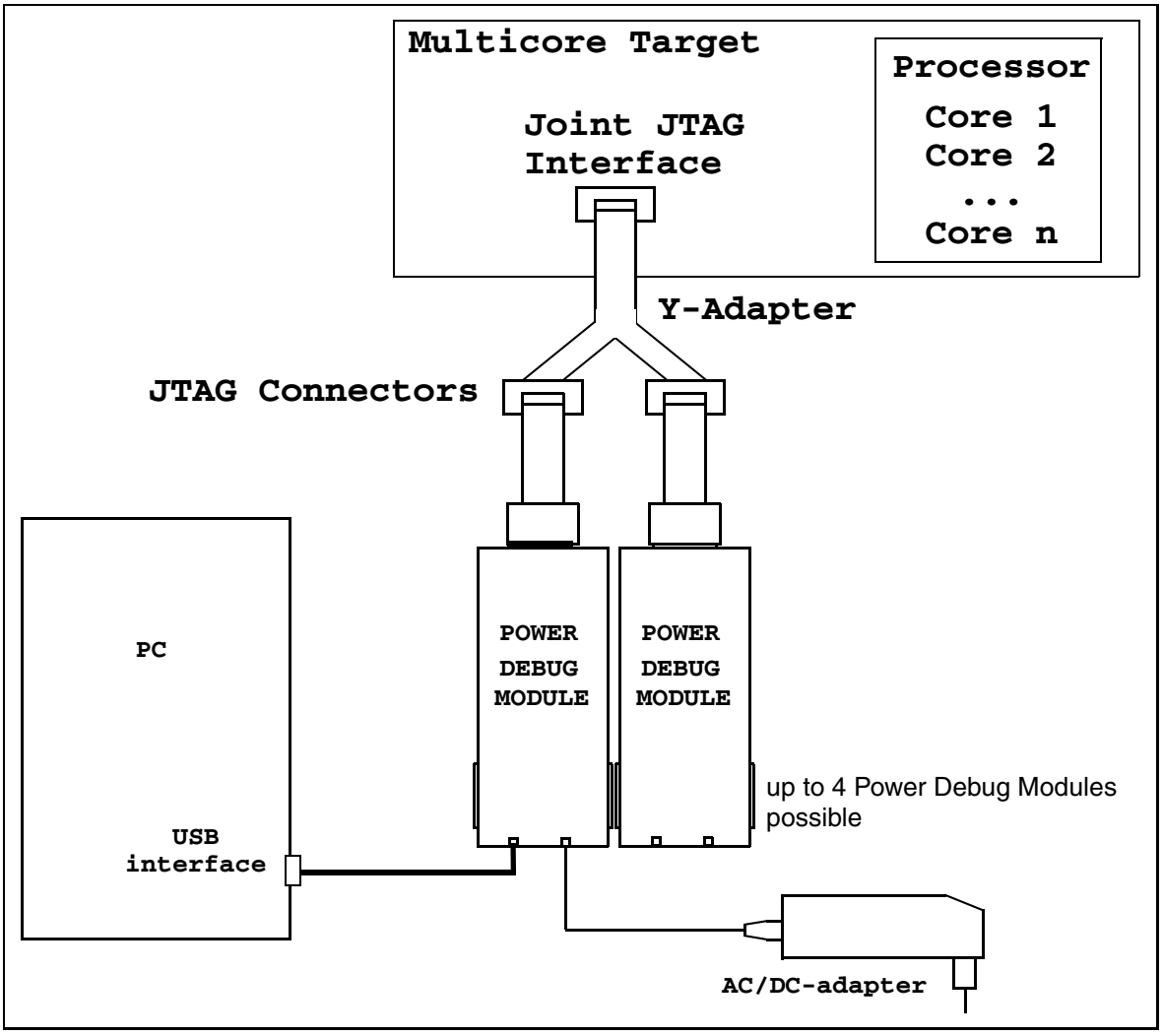


Description: The TRACE32 debugging software running on the Host-PC uses a joint JTAG interface to communicate with the cores on the multicore-IC. With Power Debug Ethernet Module the configuration looks likewise, except of interposed hub and ethernet connection at the host PC.



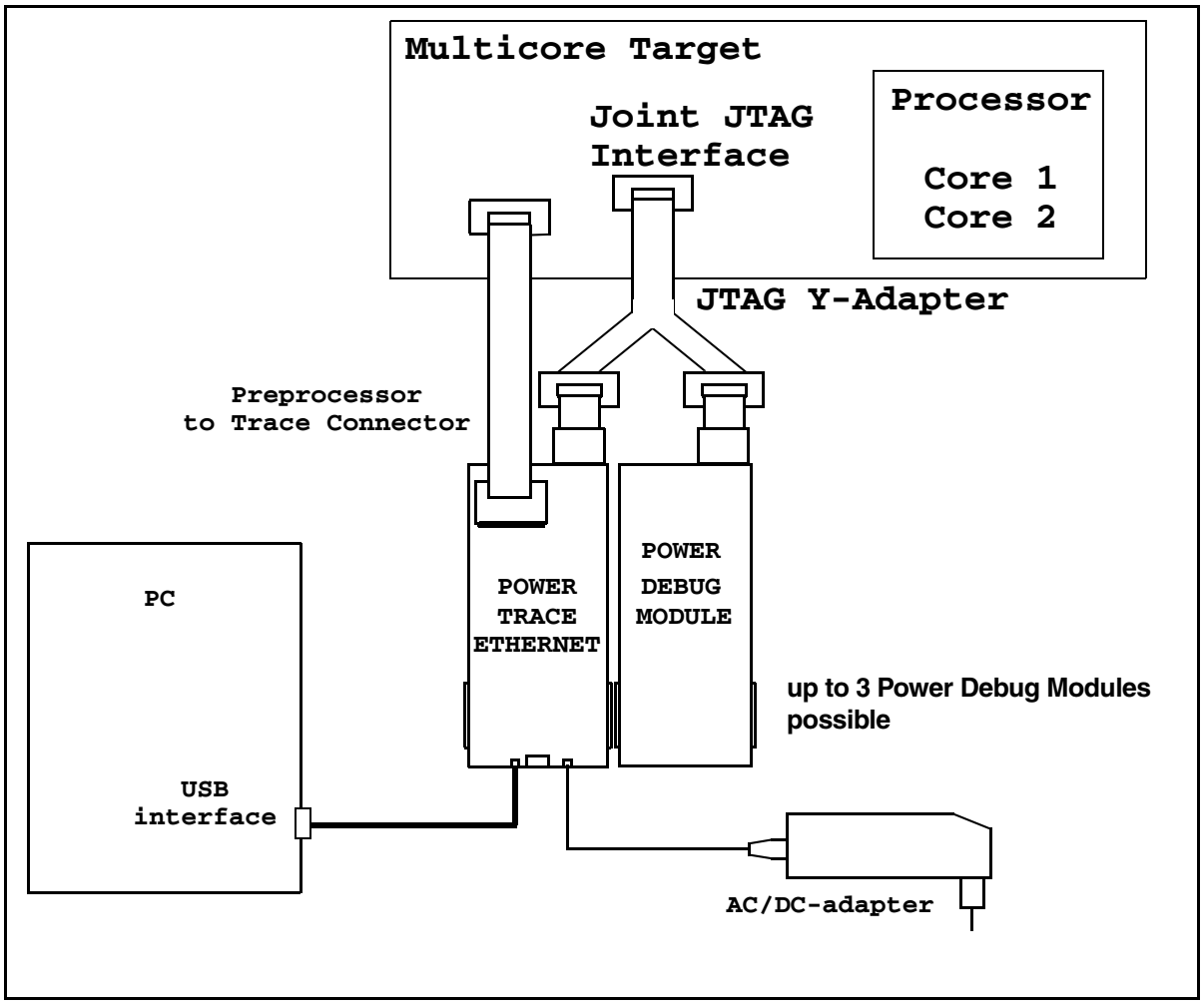
Description: The TRACE32 debugging software running on the Host-PC uses a joint JTAG interface to communicate with the cores on the multicore-IC. If several cores on the target provide trace information it is possible to select which core is accessed by the Trace Preprocessor. If there are several trace ports on the target, refer to **configuration 5**.

Using the USB connector of the Power Trace Ethernet module, the configuration looks likewise (without the Ethenet hub of course).



Description: The TRACE32 debugging software running on the host-PC uses separate debug boxes to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not needed.

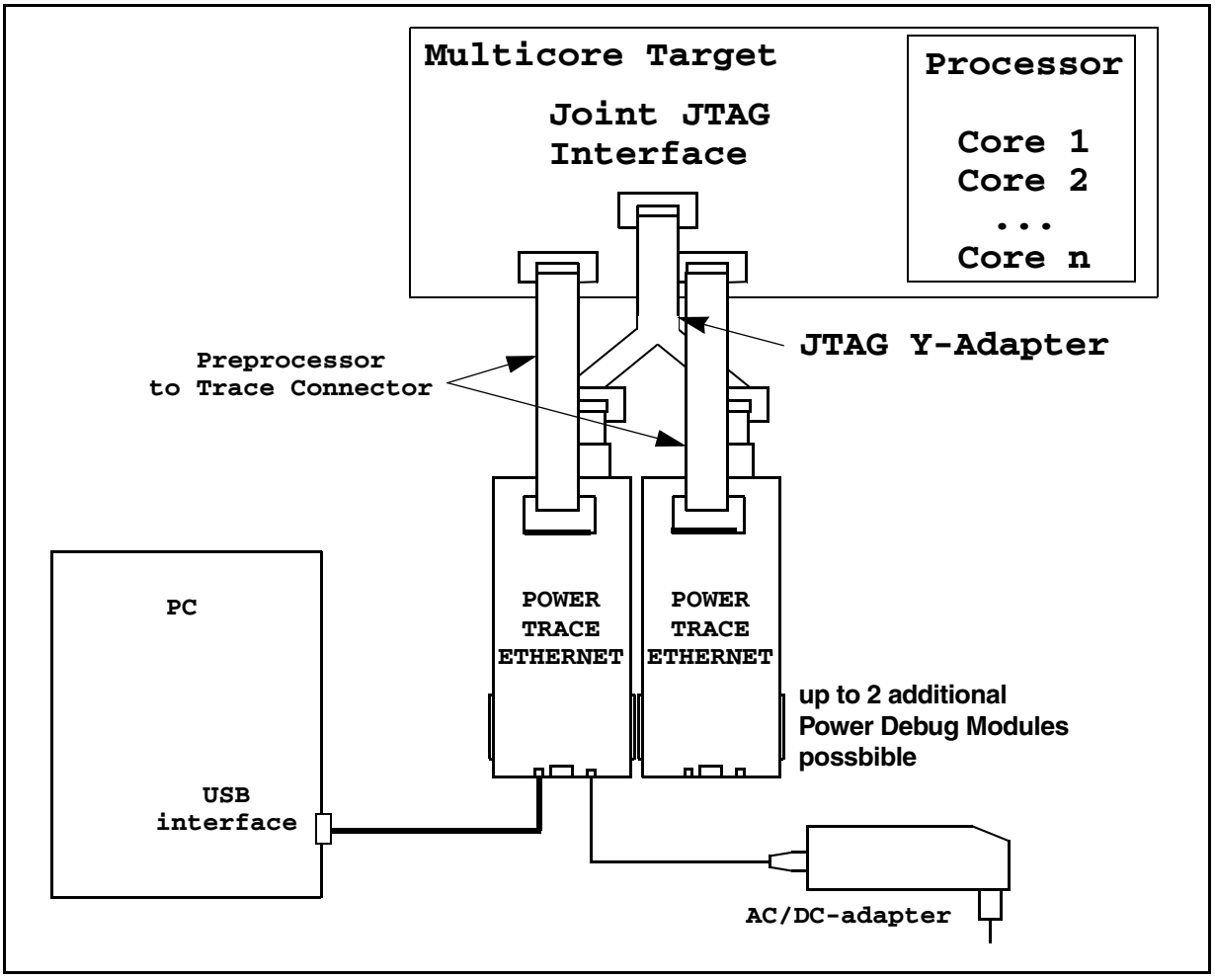
The number of Power Debug Modules needed depends on the number of cores to be debugged.



Description: The TRACE32 debugging software running on the Host-PC uses separate JTAG interfaces to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. If several cores on the target provide trace information it is possible to select which core is accessed by the Trace Preprocessor. If there are several trace ports on the target, refer to **configuration 5**.

Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not needed.

The number of Power Debug Modules needed depends on the number of cores to be debugged.



Description: The TRACE32 debugging software running on the Host-PC uses separate debug boxes to communicate with the cores on the multicore-IC. To do this, one debug cable for each core is necessary. Some target boards already provide several JTAG connectors that represent a single one actually, in that case the y-adapter is not necessary.

The number of additional Power Debug Modules needed depends on the number of cores to be debugged.

General

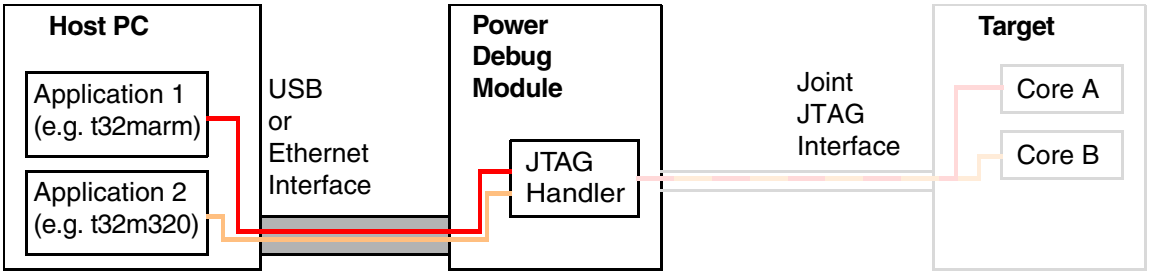
At multicore debugging, each core belongs to one instance of the TRACE32 software, i.e., there are as many TRACE32 instances resp. executables running as cores are to be debugged simutaneously.The appropriate executable to be started is titled according to the name of the core family, and can be checked in the properties of the installed debugger shortcuts.

```
t32m<corefamily>.exe                    ; on Windows

t32m<corefamily>                        ; on Linux or workstations
```

After installation, the particular predefined config files (config.t32) are for single core debugging only. Therefore it is necessary to add certain settings as described in the following. It is recommended to copy the existing config.t32 into a new file, the filename is freely choseable, and can be provided when starting the executable.

Config file settings for single device solution

At single device solution, all TRACE32 applications use the same debug box:

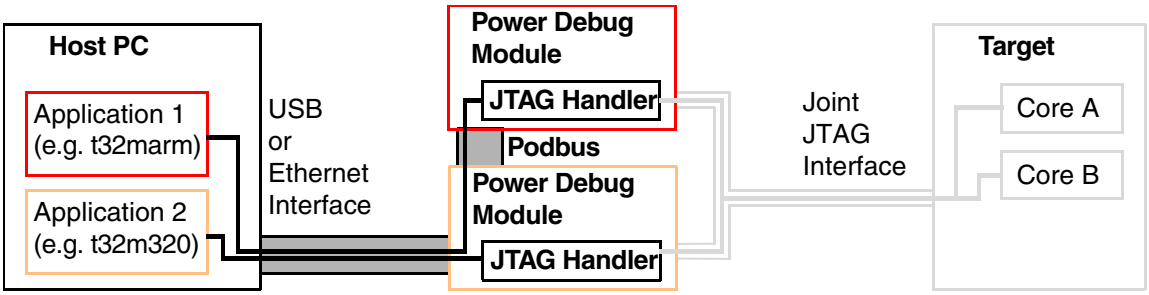


To specify this, we use **core=** setting within PBI section:

```
PBI=                                    ; within config file of first core
CORE=1
…

PBI=                                    ; within config file of 2nd core
CORE=2
…

…                                       ; possibly more config files
```

Config file settings for multi device solution

At multi device solution, each TRACE32 application uses its own debug box:



To specify this, we use the **use=** setting within PBI section:

```
PBI=                                    ; within config file of first core
USE=10

PBI=                                    ; within config file of 2nd core
USE=01

…                                       ; possibly more config files
```

In addition, it is necessary to specify the **ID=** setting within OS section (the ID is used to store PRACTICE command history and associated source code files within TMP directory, therefore it is alternatively possible to use same ID and assign separate TMP directories to each TRACE32 instance):

```
OS=                                     ; within config file of first core
ID=T32_CORE1
…

OS=                                     ; within config file of 2nd core
ID=T32_CORE2
…

…                                       ; possibly more config files
```

Config file settings for both single and multi device solutions

To use the start/stop synchronization between the different core debuggers, the InterCom port settings are necessary (**PORT=** setting within the **IC=** section, see also chapter **Start Stop Synchronisation**):

```
IC=NETASSIST                            ; within config file of first core
PORT=20001

IC=NETASSIST                            ; within config file of 2nd core
PORT=20002

…                                       ; possibly more config files
```

First TRACE32 instance:

```
; The configuration file for the ARM9 core
; ========================================

; 1. Specify the ARM9 core as the first core.

PBI=
CORE=1
USB

; 2. Specify an ID for the ARM9 debugger.

OS=
SYS=.\
TMP=C:\temp
ID=T32ARM9
HELP=H:\T32NEW\pdf
; 3. Specify a better core identification for the TRACE32 window.

SCREEN=
HEADER=TRACE32-ARM Multicore                    ; Header for the
                                                TRACE32 window
; T32 InterCom for Start/Stop synchronisation

IC=NETASSIST
PORT=20002
```

In the example above the call syntax would be (assuming the config filename is “config_core1.t32”:

```
t32marm -c config_core1.t32
```

Second TRACE32 instance:

```
; The configuration file for the TMS320C55X core
;=================================================

; 1. Specify the TMS320C55X core as the second core.

PBI=
CORE=2
USB

; 2. Specify an ID for the TMS320C55X debugger.

OS=
SYS=.\
TMP=C:\temp
HELP=H:\T32NEW\pdf
ID=T32C55x
; Each debugger creates copies of source file into the TMP directory. The
; specified ID allows each
; debugger to identify its source files.

; 3. Specify a better core identification for the TRACE32 window.

SCREEN=
HEADER=TRACE32-C55x Multicore         ; Header for the TRACE32 window
PALETTE 1 = 255 255 223               ; Yellow background color

; T32 InterCom for Start/Stop synchronisation
IC=NETASSIST
PORT=20001
```

In the example above, the call syntax would be (assuming the config filename is “config_core2.t32”:

```
t32m320 -c config_core2.t32
```

Besides, there is the possibility to use just one generic template config file for all cores. The particular settings are passed as parameter. Example for generic config file:

```
IC=NETASSIST
PORT=${1}

; Environment variables
OS=
ID=T32${1}
TMP=C:\temp
SYS=.\
HELP=h:\t32new\pdf

PBI=
${3}
${4}
${5}
${6}

; Printer settings
PRINTER=WINDOWS


; Screen fonts
SCREEN=
FONT=SMALL
HEADER=TRACE32 ${2} Multicore @${3} ${4} ${5}  ${6}
```

The template variables ${1} to ${6} are replaced textually by corresponding calling parameters. In that way, it is possible to select even the connection method (Parallel, USB or Ethernet) by parameters.

In this generic example the call syntax could be (assuming the config filename is “config_mc.t32”:

```
; Start ARM executable as 1st core via USB and InterCom port 20001
t32marm -c config_mc.t32 20001 ARM USB CORE=1

; Start TMS320 executable as 2nd core via USB and InterCom port 20002
t32m320 -c config_mc.t32 20002 C55x USB CORE=2
```

Examples for Ethernet connection:

```
; Start ARM executable as 1st core via ETH and InterCom port 20001
t32marm -c config_mc.t32 20001 ARM NET NODE=10.2.2.234 CORE=1

; Start TMS320 executable as 2nd core via ETH and InterCom port 20002
t32m320 -c config_mc.t32 20002 C55x NET NODE=10.2.2.234 CORE=2
```

Using the **OS** command of TRACE32 PRACTICE language, it its possible to start further instances of TRACE32 (see also **PowerView User's Guide"OS"** in PowerView Command Reference, page 210 (ide_ref.pdf)). This offers the possibility to start multiple core debuggers from one PRACTICE script file (*.cmm). In combination with a generic config file, this is a very smart way to set up multicore debugging.

```
; start tpu executable as 2nd core via USB and InterCom port 20002
OS t32mtpu -c config_mc.t32 20002 eTPU_A USB CORE=2
```

For a description of the parameters refer to chapter **Generic config file for each TRACE32 instance**, see above.

The very first instance, however, has to be started in common way (by batch file or shortcut), as described in chapter **Generic config file for each TRACE32 instance**, see above:

```
; start PPC executable as 1st core via USB and InterCom port 20001
t32mppc -c config_mc.t32 20001 PowerPC USB CORE=1

; start PPC executable as 1st core via ETH and InterCom port 20001
t32mppc -c config_mc.t32 20001 PowerPC NET NODE=10.2.2.234 CORE=1
```

Complete example PRACTICE script for generic starting of multiple instances on Windows, using MPC5554 (with E200 and two eTPU cores), in which it is dynamically selectable to use USB or Ethernet connection:

```
; Sample eTPU setup script

; start eTPU debugger software and check InterCom
============================
&nodename=NODENAME()

&etpu1="InterCom localhost:20002"
&etpu2="InterCom localhost:20003"

; Test is etpu software is already running or not
IF !InterCom.PING(localhost:20003)
(
     IF "&nodename"==""
     (
          print "assuming connection is USB"
          OS t32mtpu -c config_mc.t32 20002 eTPU_A USB CORE=2
          ; wait until software is started
          InterCom.WAIT localhost:20002
          OS t32mtpu -c config_mc.t32 20003 eTPU_B USB CORE=3

     )
     ELSE
     (
          PRINT "connection is ETH"
          OS t32mtpu -c config_mc.t32 20002 eTPU_A NET NODE=&nodename
          PACKLEN=1024 CORE=2
          ; wait until software is started
          InterCom.WAIT localhost:20002
          OS t32mtpu -c config_mc.t32 20003 eTPU_B NET NODE=&nodename
          PACKLEN=1024 CORE=3
     )
     ; wait some time until the software finished booting
     ; if you get and InterCom timeout error on the following framepos
     ; commands
     ; increase time
     WAIT 3.s

     ; multicore settings
============================================================
     InterCom.WAIT localhost:20002
     InterCom.WAIT localhost:20003

     &etpu1 SYStem.CONFIG.CORE 2.
     &etpu2 SYStem.CONFIG.CORE 3.
)
```

```
; arrange debugger windows
framepos          0.   0.  88.  25.
&etpu1 framepos   0.  38. 181.  26.
&etpu2 framepos  94.   0.  88.  20.

; attach debugger to cores
;=====================================================
sys.u
&etpu1 sys.m.attach
&etpu2 sys.m.attach

enddo
```

In the example above, the other TRACE32 instances are accessed from first instance by

InterCom localhost:<port_number>

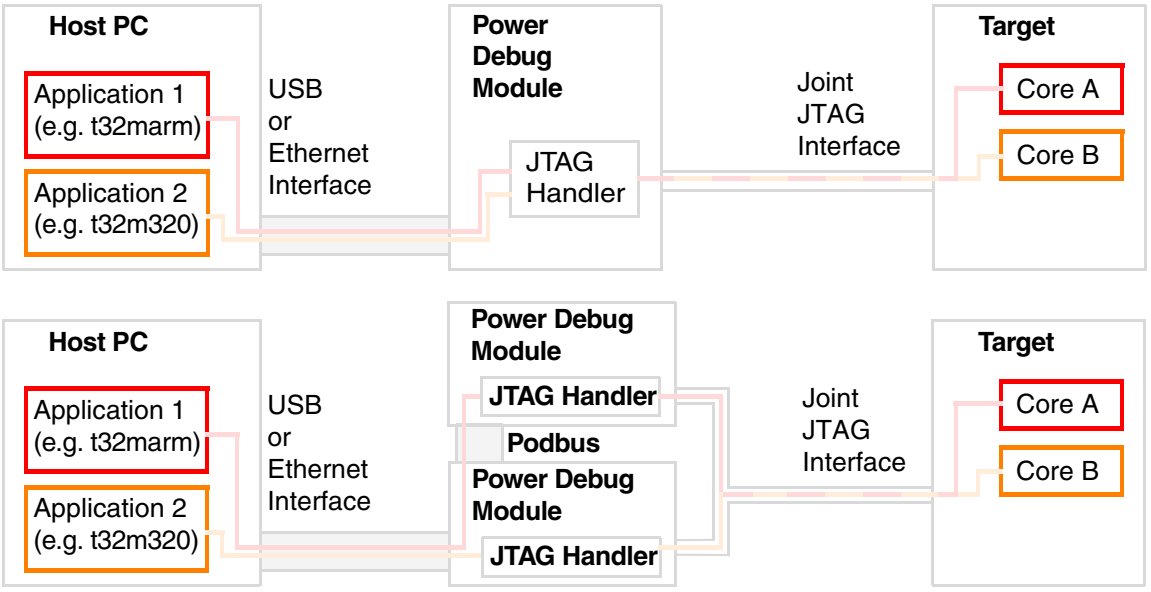
command, which is stored into local PRACTICE variables (**&etpu1** and **&etpu2**).

Automatic setup of config files

(for Windows only)

The usage of the LAUTERBACH add-on **T32Start** is described in **"T32Start"** (app_t32start.pdf).

The installation of the TRACE32 software and the config files is finished now. All TRACE32 applications access the same JTAG port, either by single device or by multi device solution. Now we have to tell the TRACE32 applications which of the cores they should debug.



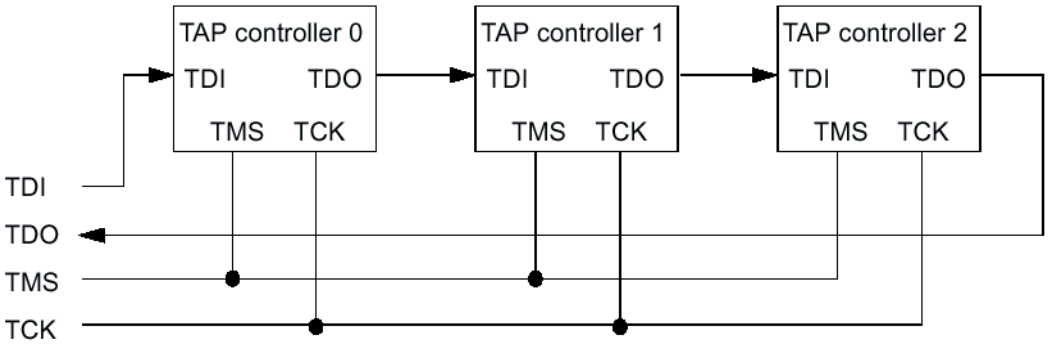
The communication to the specified core is done not until the **SYStem.Up** command, and for preparation (besides the proper processor selection) there might be some additional multicore settings to be done.

There is a number of PRACTICE sample scripts for different processors available in the **~~/demo/...** folder or directly from LAUTERBACH support (for contact possibilities, see www.lauterbach.com), which may assist you in setting up multicore debugging.

Basically, there are two different ways to address the particular core:

- daisy chaining
- multiplexing

For those processors that have multiple so-called TAP (=Test Access Port) controllers, eg. one for each core, it is necessary to provide the TRACE32 applications with daisy chain settings.



The JTAG port of the individual cores are arranged serially (daisy chained). The TAP controller controls the scanning of data in the instruction register and the data register of the JTAG architecture.

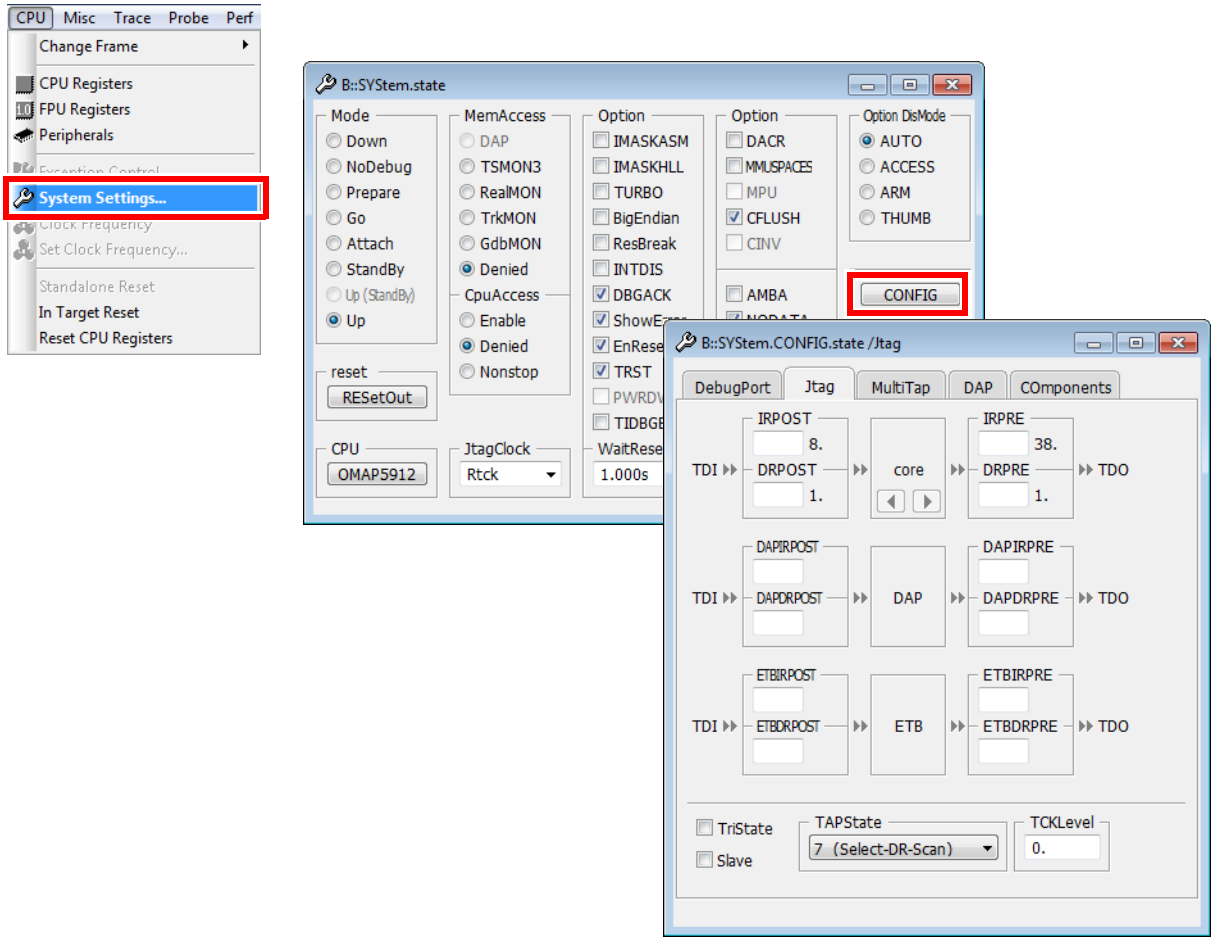
- The instruction register has a specific width for each architecture
- The width of the data register depends on the instruction used (always 1 bit if the instruction register contains the BYPASS instruction)

According to the JTAG convention there is an option to generate sequences for the scan chain that ensure that individual TAP controllers behave neutrally (BYPASS). To address a specific core the debugger must therefore be configured in such a way that, when generating the scan chain, BYPASS sequences are created for all TAP controllers before and after this core. The associated settings are either determined by selection of the processor (if fixed) and/or can be modified in the **SYStem.CONFIG.state** window.

To access the SYStem.CONFIG.state window, do one of the following:

- Choose **CPU** menu > **System Settings**, and then click the **CONFIG** button.
- At the TRACE32 command line, type:

SYStem.CONFIG.state	Shows the state of the MULTICORE setting.



IRPRE *<number>* of instruction register bits of all cores in the JTAG chain between the current core and the TDO signal.

IRPOST *<number>* of instruction register bits of all cores in the JTAG chain between TDI signal and the current core.

DRPRE *<number>* of cores resp. TAP controllers in the JTAG chain between the current core and the TDO signal (one data register bit per TAP controller which is in BYPASS mode).

DRPOST *<number>* of cores resp. TAP controllers in the JTAG chain between the TDI signal and the current core (one data register bit per TAP controller which is in BYPASS mode).

Slave All debuggers except one must have this option active. Only one debugger - the master - is allowed to control the signals nTRST and nSRST (nRESET).

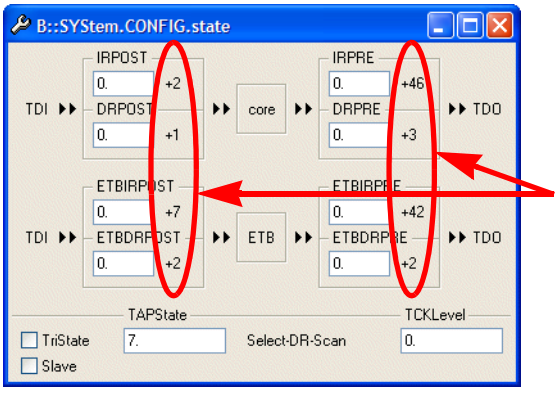
In case of multi device solution, the following additional settings have to be made:

TriState — If more than one debug box shares the same JTAG port, this option is required. The debugger switches after each JTAG access to tristate mode. Then other debuggers can access the port.

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. The default value is recommended.

TCKLevel — Level of TCK signal when all debuggers are tristated (0 in case of pull-down on target, 1 if pull-up).

In case the daisy chain settings are preconfigured by processor selection, they are displayed within **SYStem.CONFIG.state** window:



automatic daisy chain
settings by processor selection

The example above shows both core and ARM-ETB daisy chain settings, at which they both are located within same daisy chain sequentially. If there are additional TAP controllers beyond the selected multicore processor located on the target board (e.g. several daisy-chained multicore chips together on one target board), the necessary values have to be entered into edit controls.

For some processors (like STARCORE or NIOS-II) it is necessary to specify individually which core the current TRACE32 instance should debug.

```
SYStem.CPU <multicore_processor>      ; select multicore processor type
SYStem.CONFIG.CORE 1                  ; specify current TRACE32 instance
                                      ; as core 1 debugger

SYStem.CPU <multicore_processor>      ; select multicore processor type
SYStem.CONFIG.CORE 2                  ; specify current TRACE32 instance
                                      ; as core 2 debugger
```

The **SYStem.CONFIG.CORE** setting does technically NOT refer to **CORE=** setting within config file. Of course, it is recommended to use same core number.

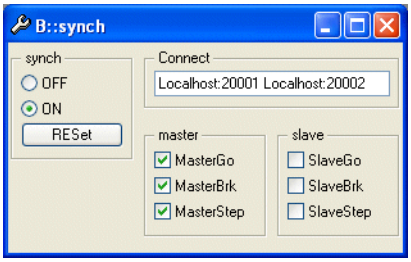
Settings

To use the Start/Stop Synchronisation, the InterCom port settings in the **IC=** section of the config file(s) are required (refer to chapter **Setup and Start of TRACE32 Software**.

The synchronisation itself is done by **SYnch** commands, according to the following steps:

1. Set up the **SYnch** command for first core as master core

```
SYnch.state    ;Display the current settings of the SYnch command
```



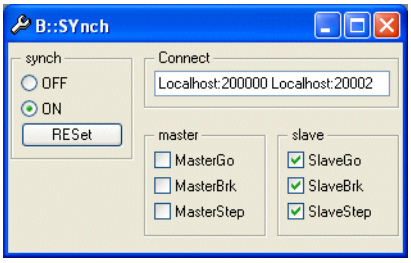

```
SYnch.Connect Localhost:20001 [<further_ports>]
```

SYnch Settings	
Connect	Establish a connection to the debugger attached to the defined communication port(s). Several debuggers resp. ports can be specified, separated by space.
MasterGo ON	If the program execution is started, the program execution for all other processors which have SlaveGo ON is also started.
MasterBrk ON	If the program execution is stopped, the program execution for all other debuggers which have SlaveBrk ON is also stopped.
MasterStep ON	If an asm single step is executed, all processors which have SlaveStep ON will also asm single step.

It is optionally possible to set both master and slave settings ON, to let this core react on other core events, too.

	The **PORT=** value within IC section of the config file of the current core has to match the value(s) of the **SYnch.Connect** command(s) of the coressponding debugger(s), while the **SYnch.Connect** value(s) of the current core has(have) to match those values from the config files of the other core debugger(s). In other words, the **PORT=** value(s) within config file(s) are server port numbers, while the **SYnch.Connect** values are client port numbers (see also chapter **Setup and Start of TRACE32 Software**).

2. Set up the SYnch command for the further cores.



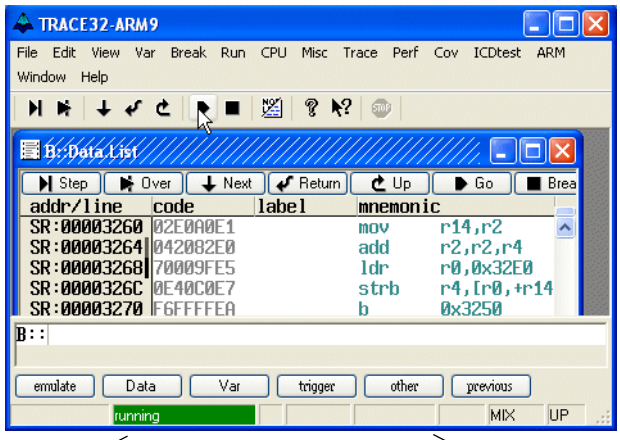

```
SYnch.Connect Localhost:20000 [<further_ports>]
```

SYnch Settings	
Connect	Establish a connection to the debugger attached to the defined communication port
SlaveGo ON	The program execution is started, if a processor with MasterGo ON starts its program execution.
SlaveBrk ON	The program execution is stopped, if a processor with MasterBrk ON stops its program execution.
SlaveStep ON	A asm single step is performed, if a processor with MasterStep On performs an asm single step.

It is optionally possible to set both master and slave settings ON, to let other cores react on current core events, too.

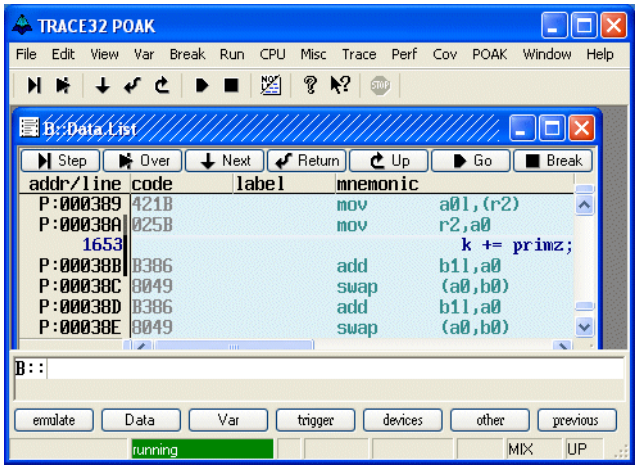
Summary (Example with one master and two slave core debuggers):

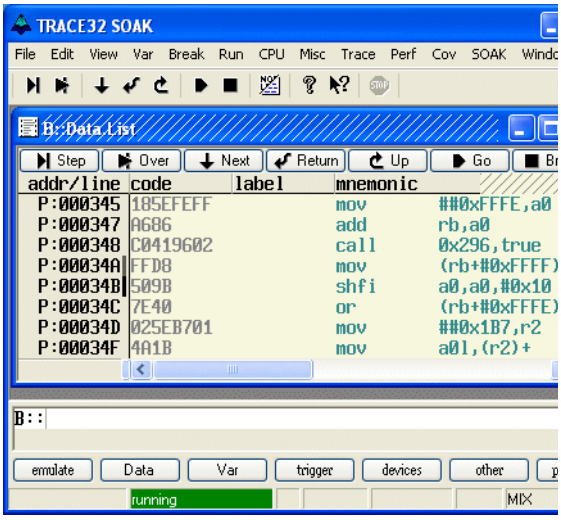
```
; Complete script has to be run on TRACE32 instance of first core

; SYNCH commands for 1st core
SYNCH.RESet
SYNCH.Connect Localhost:20001 Localhost:20002
SYNCH.MasterGo ON
SYNCH.MasterBRK ON
SYNCH.MasterStep ON

; SYNCH commands for 2nd core
InterCom.execute Localhost:20001 SYNCH.RESet
InterCom.execute Localhost:20001 SYNCH.Connect Localhost:20000
Localhost:20002
InterCom.execute Localhost:20001 SYNCH.SlaveGo ON
InterCom.execute Localhost:20001 SYNCH.SlaveBRK ON
InterCom.execute Localhost:20001 SYNCH.SlaveStep ON

; SYNCH commands for 3rd core
InterCom.execute Localhost:20002 SYNCH.RESet
InterCom.execute Localhost:20002 SYNCH.Connect Localhost:20000
Localhost:20001
InterCom.execute Localhost:20002 SYNCH.SlaveGo ON
InterCom.execute Localhost:20002 SYNCH.SlaveBRK ON
InterCom.execute Localhost:20002 SYNCH.SlaveStep ON

ENDDO
```

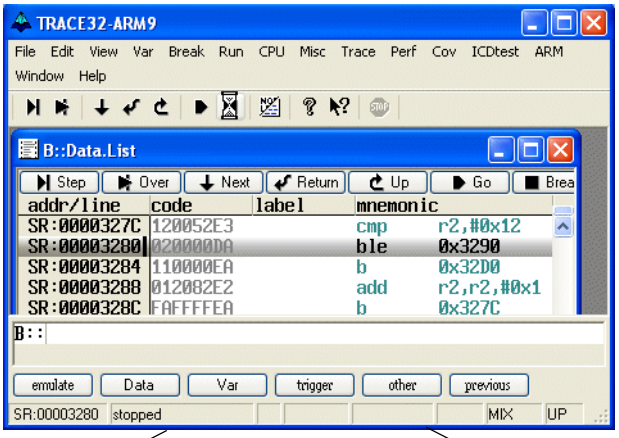


The master starts the program execution

Both slaves are started

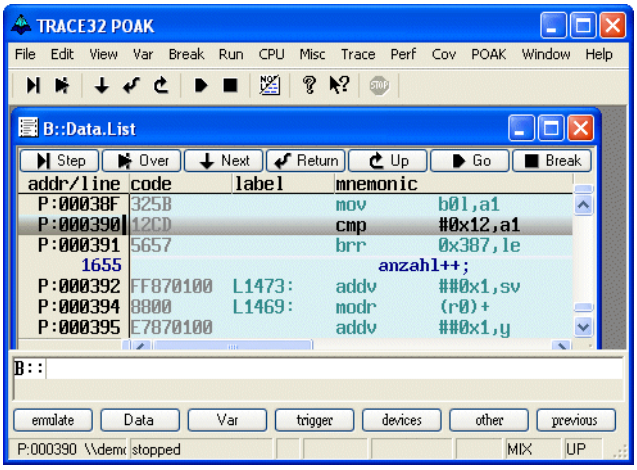


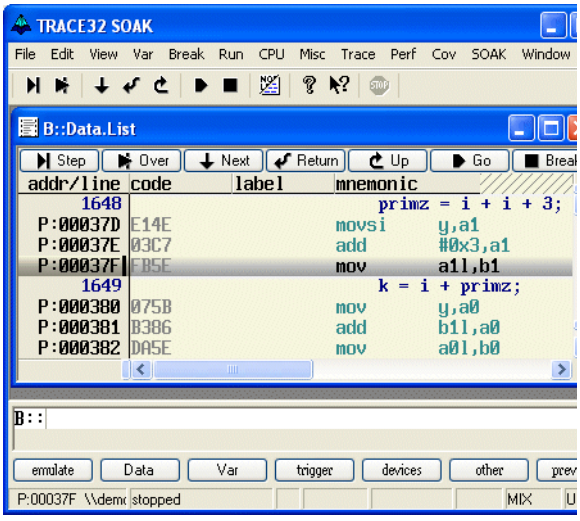




The master stops the program execution

Both slaves are stopped





There is a time delay between reaction of different cores. The reaction time of the appended slave core(s) depends on the technical realization of the synchronization. In worst case, this is done by software, i.e. the master debugger informs the slave debugger(s) via socket communication on the host about specified events. In best case, the synchronization takes place directly on the processor, and the in between solutions are PODBUS trigger signal or to connect certain PINs, if available, on the JTAG connector or the board. Which solution is offered by the particular multicore debugging environment is depending on the chip and/or board hardware. The LAUTERBACH Debuggers always use the fastest available possibility. The software synch mechanism starts all cores simultaneously by TRACE32 software, either by socked communication (slow) or internal synchronized within TRACE32 (fast). The fast method is only implemented for some processors, while the software method by socket is always available.

Special on STEP-Signal: By selecting MasterStep/SlaveStep ON, only the asm single steps are synchronized. The HLL steps are published by master debugger as a sequence of (asm)step-go-break (or just go-break), which means the slave debugger(s) do(es) not really perform its own HLL single step.

Solution:	**Software (slow)**	**Software (fast)**	**PODBUS Trigger**	**Hardware (JTAG or Board)**	**Chip Hardware**
Delay time:	**1-5 ms**	**~10 µs**	**100-300 ns**	**~10 ns**	**0-10 ns**

Multiple ICD Debuggers can be started and stopped synchronously. The time delay depends on the implementation of the debug port. On a 68332 the start time delay is about 10us and the stop delay about 5us. The coupling of the different systems is made by the **SYnch** command.

NOTE:	This command requires that the InterCom system is configured in the host. The synchronous break can be made with the **Trigger** command. The following example shows the configuration files and commands required for a system where one master controls two slaves:

Configuration file for master:

```
PBI=
USE=100
IC=NETASSIST
PORT=20000
```

Configuration file for slave 1:

```
PBI=
USE=010
IC=NETASSIST
PORT=20001
```

Configuration file for slave 2:

```
PBI=
USE=001
IC=NETASSIST
PORT=20002
```

Startup commands for master:

```
sy.connect localhost:20001 localhost:20002
sy.mastergo on
sy.on
trigger.out busa on
```

Startup commands for slaves:

```
sy.slavego on
sy.on
trigger.set busa on
```