Training JTAG Interface

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

Training JTAG Interface

Introduction

For most embedded CPU architecture implementations, the JTAG port is used by the debugger to interface the chip for debugging one or more cores. The normal user will probably not need to know details of the JTAG implementation unless there is a need to debug several daisy-chained JTAG TAP controllers or to access special test functions or configurations via JTAG that are not implemented in the debugger software.

This training manual explains the basics of JTAG in case of a single TAP controller or several daisy-chained TAP controllers and how to perform a custom access to the JTAG port by using the TRACE32 software.

Related Documents

This training does not focus on any specific architecture, so the best document to find any architecture- and CPU specific information is the corresponding **Processor Architecture Manual**. Not only ARM users will find the **"ARM JTAG Interface Specifications"** in , page 1 (arm_app_jtag.pdf)**"ARM JTAG Interface Specifications"** (arm_app_jtag.pdf) interesting since it contains information applicable to any device and general information on the TRACE32 debug cable internals.

Processor Architecture Manual	**Processor Architecture Manuals**
ARM JTAG Interface Specifications	**"ARM JTAG Interface Specifications"** (arm_app_jtag.pdf)
Remote API Manual	**"API for Remote Control and JTAG Access"** (api_remote.pdf)
Command Reference for Letter J	**"General Commands Reference Guide J"** (general_ref_j.pdf)

Debugging a JTAG Session

For debugging any JTAG communication the use of a logic analyzer such as the TRACE32 PowerProbe is recommended:
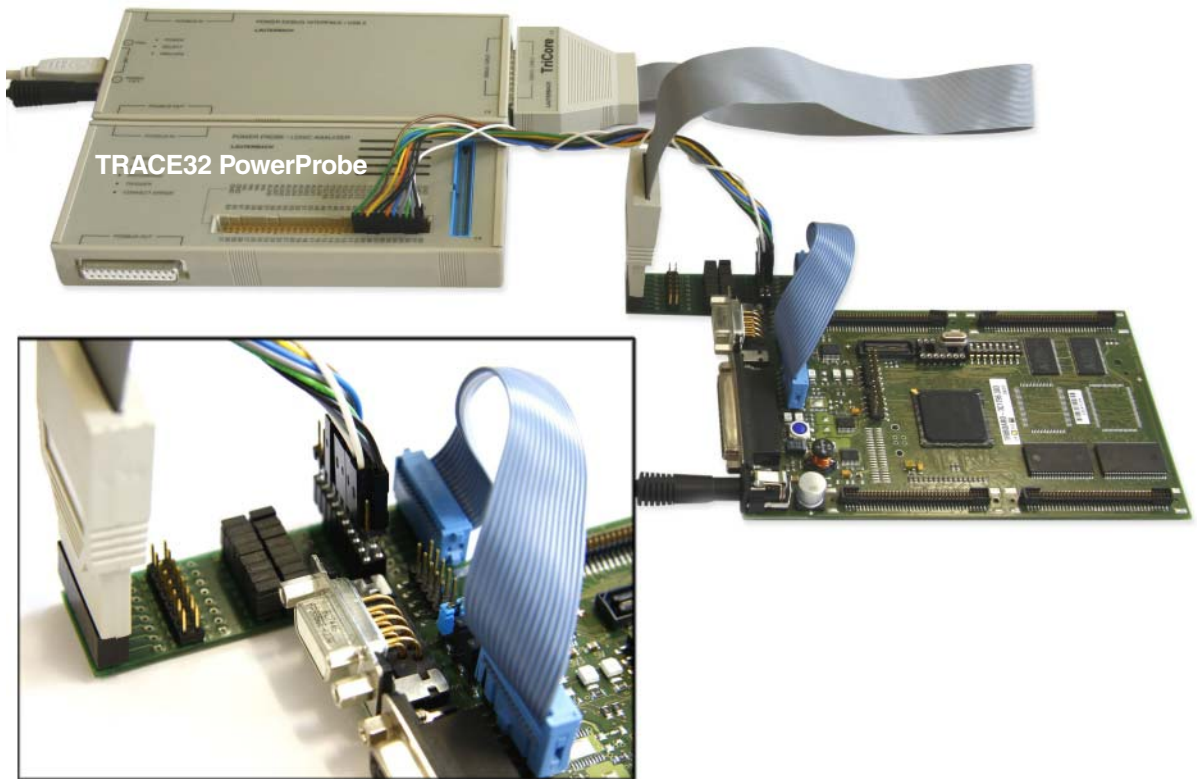
Connection details

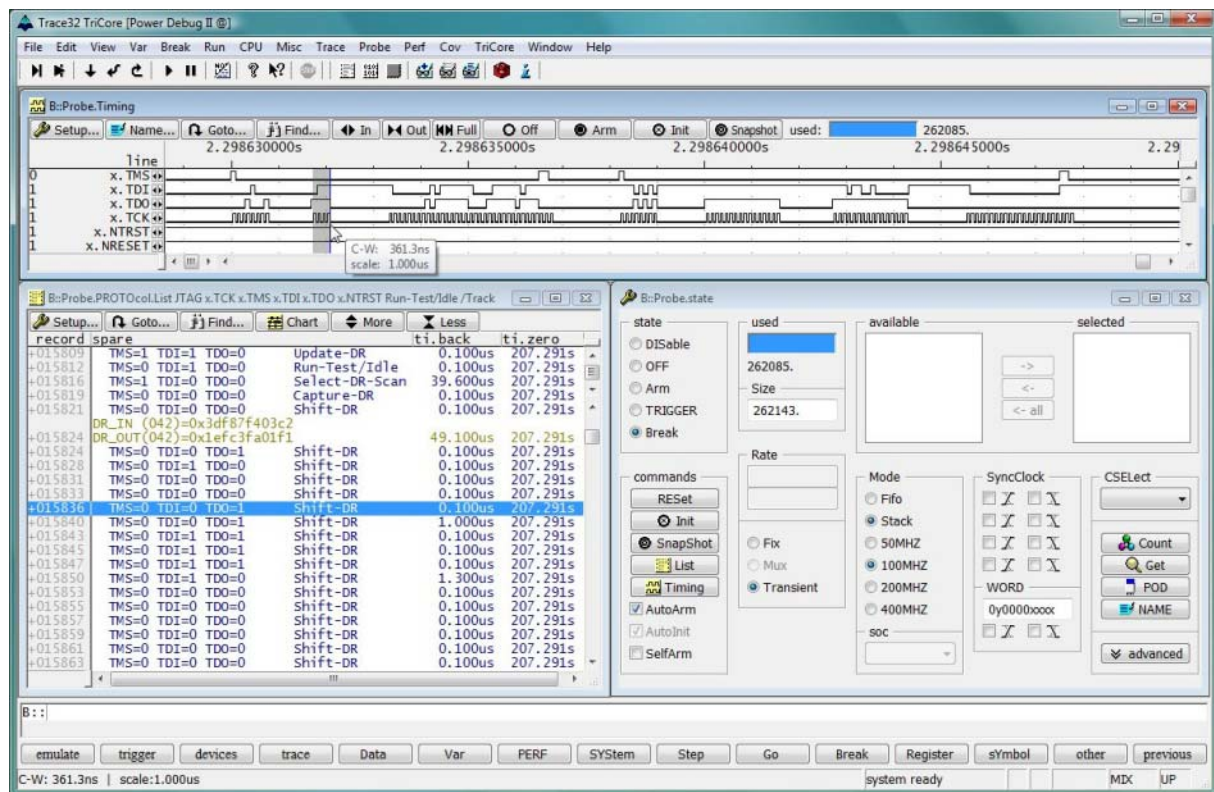By using this tool it is not only possible to record the signals but also decode the JTAG protocol for a better interpretation. The JTAG decoder is built-in but also available as source-code allowing to extend the analysis by a higher-level decoder for custom use:

JTAG Basics

JTAG is the name used for the IEEE 1149.1 standard entitled *Standard Test Access Port and Boundary-Scan Architecture* for test access ports (TAP) used for testing printed circuit boards (PCB) using boundary scan.

JTAG is the acronym for *Joint Test Action Group*, the name of the group of people that developed the IEEE 1149.1 standard.

The functionality usually offered by JTAG is *Debug Access* and *Boundary Scan*:

- **Debug Access** is used by debugger tools to access the internals of a chip making its resources and functionality available and modifiable, e.g. registers, memories and the system state.

- **Boundary Scan** is used by hardware test tools to test the physical connection of a device, e.g. on a PCB. Although this is usually not the task of a debugger tool the TRACE32 debugger offers mechanisms to access the JTAG TAP in a generic way, e.g. to perform boundary scan using a PRACTICE script or a custom application.

Although the TAP (Test Access Port) access itself is generic for all architectures, the functionality implemented behind JTAG is different for each device.

The following chapter explains all necessities for accessing a JTAG TAP. For a complete description of JTAG see the IEEE 1149.1 standard.

Main Concept

JTAG is defined as a serial communication protocol and a state machine accessible via a TAP. The DTAB (Debug and Test Access Block) is implemented on the target chip as a "passive" device that never sends data without request.

The DTAB mainly consists of the following:
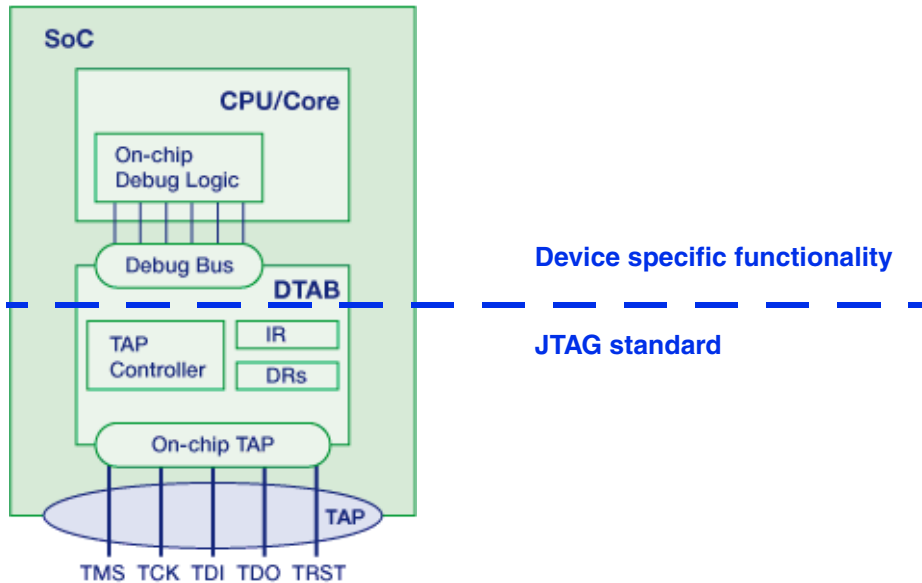
- The **TAP (Test Access Port)** with its physical connections (signals) to the external world.

- The **TAP Controller** (a 16-state state machine).

- One **IR (Instruction Register)** and several **DRs (Data Registers)**.

- The **Debug Bus** for communication with the on-chip debug logic.

TAP (Test Access Port)

The TAP defines the interface between the DTAB and the debug tool. The *JTAG Port* is the physical connector on the PCB where the debug cable is plugged.

The IEEE standard defines the following TAP signals, used for the serial communication and driving the TAP controller (JTAG state machine):

TDI	Test Data In	serial data from debugger to target
TDO	Test Data Out	serial data from target to debugger
TCK	Test Clock	
TMS	Test Mode Select	controls the TAP controller state transitions
TRST	Test Reset	optional, resets the TAP controller

The **TMS** and **TDI** line are sampled by the DTAP on each rising edge on the **TCK** line. The **TDO** line changes its value after a falling edge on the **TCK** line.

Instruction and Data Registers

The functionality of the DTAB is accessible via different instructions stuffed into the **Instruction Register**. By loading an instruction, the corresponding **Data Register** is selected for access, providing and/or accepting data according to the selected instruction.

Only a few instructions are defined by the IEEE standard, and only a few of them are mandatory, e.g. the

• BYPASS instruction (mandatory) for use in daisy-chained multi-core configurations

• IDCODE instruction (not mandatory) for identifying a device

Unused instruction codes can be freely implemented by the device manufacturer.

The **width** of the **IR** is not specified by the JTAG standard but needs to be the same for all instructions of a specific device. Since the DR is selected according to the loaded instruction the **DR width** is variable.

The following schematic shows the connection of the input signals with the DTAB and the **selection of the Data Register** depending on the content of the **Instruction Register**:
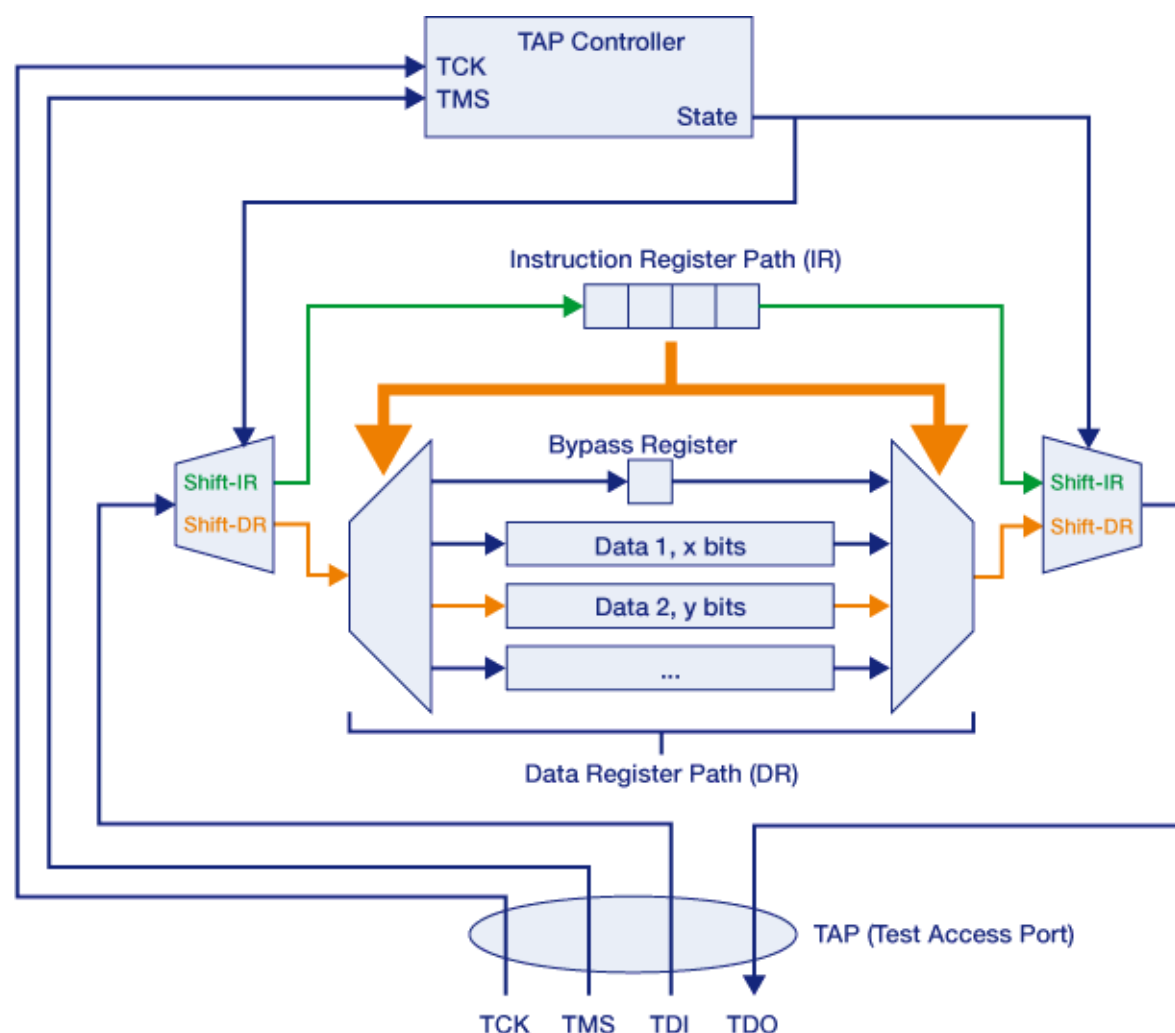
The IEEE standard defines a 16-state state machine called the TAP controller to control several actions:
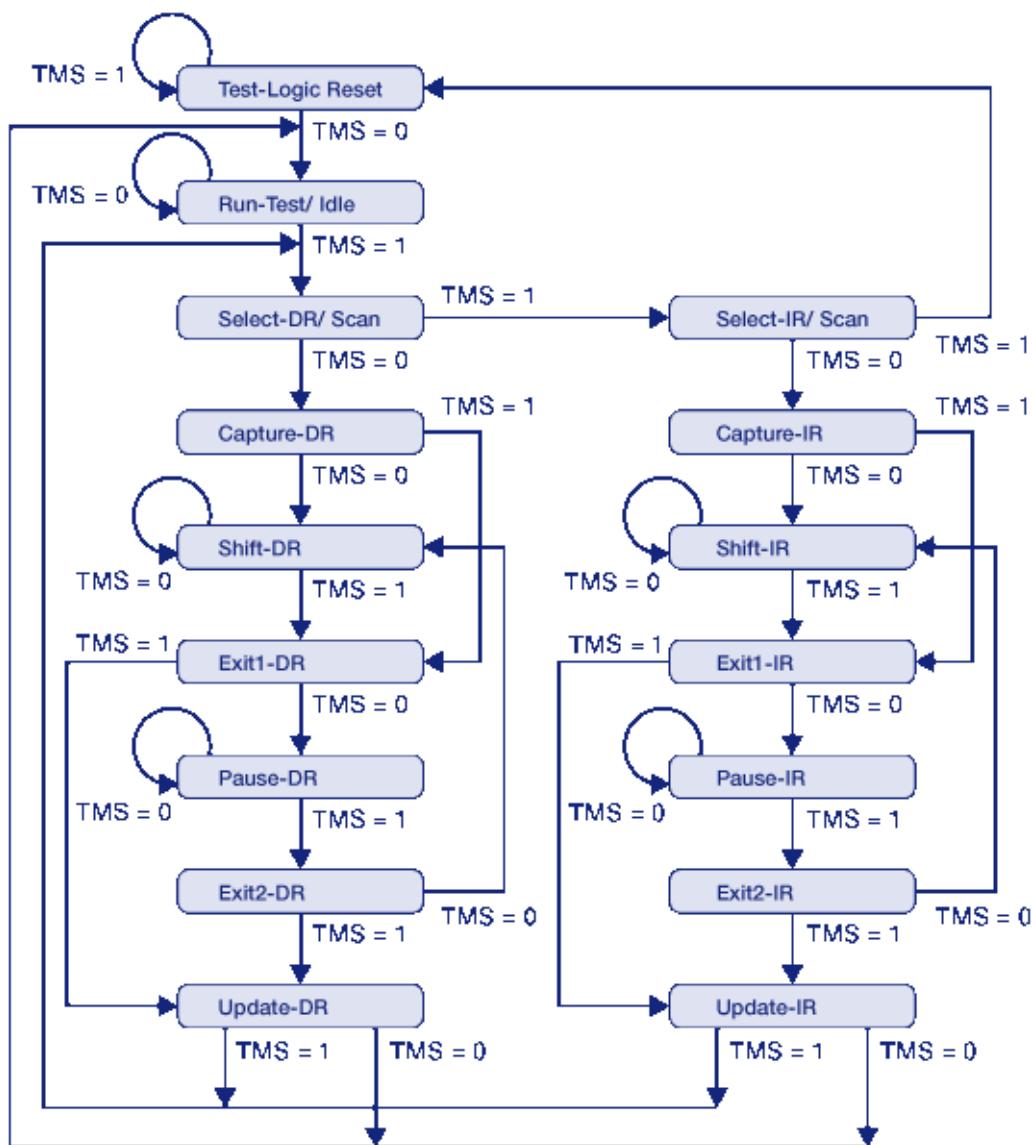
Each state of the TAP controller can be reached by a sequence of bits transmitted via the **TMS** line depending on the current state.

Normally a DR or IR shift access starts from the pause parking position, changes to one of the shift states where data is transmitted and ends up in the pause parking position again.

Pause parking position: state the TAP controller holds while waiting for the next shift operation.

The following states of the TAP controller are of importance:

• **Test Logic Reset** sets the Instruction Register to its reset value (IDCODE or BYPASS). This state can be reached from any other state by shifting five times "1" on **TMS**.

  Some CPUs do also reset the DTAB and/or the on-chip debug logic when this state is entered. As a result an active debug connection might get lost, if this state is driven by a PRACTICE script or a custom application.

• **Run-Test/ Idle** and **Select DR-Scan** are used by most debuggers as pause parking position.

• In the **Shift-IR** state the debug tool shifts an instruction into the Instruction Register. The instruction is activated once the TAP controller reaches the **Update-IR** state.

• In the **Shift-DR** state the debug tool shifts data to/from the Data Register selected by the currently loaded instruction.

Debug Bus

The debug functionality is usually not implemented in the DTAB but realized as a separate IP block. Thus the implementation of the Debug Bus and the on-chip debug logic (*Debug System*) is device specific.

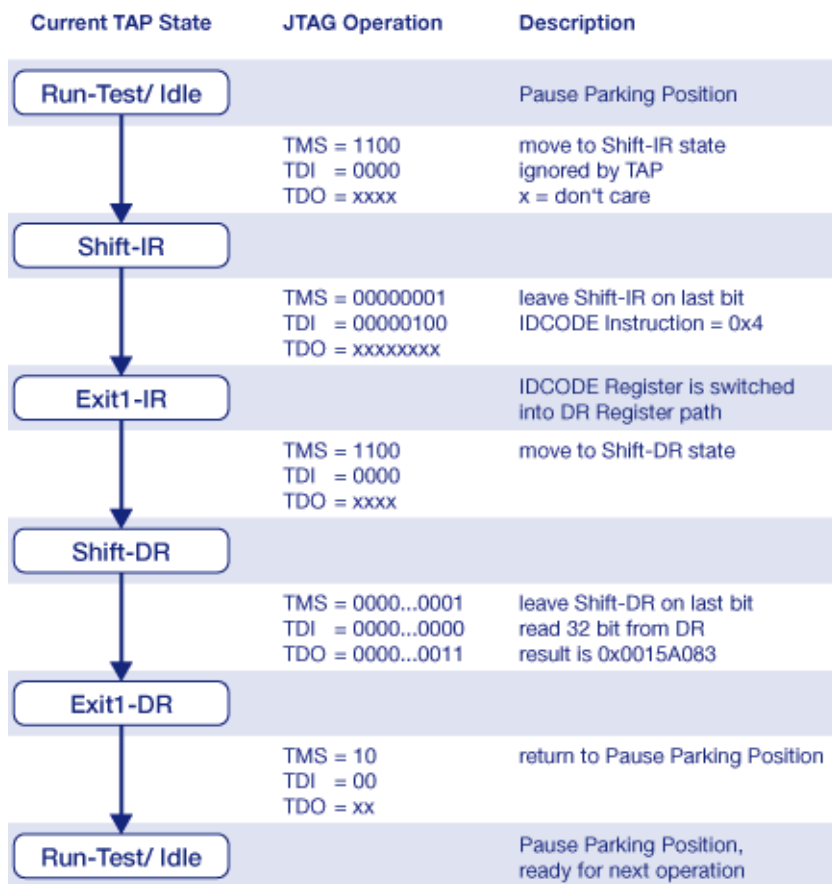On some CPUs the access to the Debug Bus is enabled by a dedicated JTAG instruction. Communication is then completely handled via a dedicated Data Register. The DTAB just enables accesses to the on-chip debug logic by using the DR path.

Other on-chip components may be accessible with their own JTAG instruction, e.g. ETM/ ETB, MCDS or an auxiliary processor. However this is implementation specific.

Communication with the DTAB

The debug tool communicates with the DTAB by reading and/or writing IRs and DRs.

The debug tool first drives the TAP Controller to the Shift-IR state to write the appropriate instruction to IR. Then it drives to the Shift-DR state where the DR can be read or written. Once the Update-DR state is reached, the processing of DR is started, e.g. the data contained in DR is forwarded to the on-chip debug system.

The following example shows how the chip ID code is read on a TriCore processor
(**IR**: 8 bits, **IDCODE DR**: 32 bits)

Current TAP State	JTAG Operation	Description
Run-Test/ Idle		Pause Parking Position
	TMS = 1100 TDI = 0000 TDO = xxxx	move to Shift-IR state ignored by TAP x = don't care
Shift-IR		
	TMS = 00000001 TDI = 00000100 TDO = xxxxxxxx	leave Shift-IR on last bit IDCODE Instruction = 0x4
Exit1-IR		IDCODE Register is switched into DR Register path
	TMS = 1100 TDI = 0000 TDO = xxxx	move to Shift-DR state
Shift-DR		
	TMS = 0000...0001 TDI = 0000...0000 TDO = 0000...0011	leave Shift-DR on last bit read 32 bit from DR result is 0x0015A083
Exit1-DR		
	TMS = 10 TDI = 00 TDO = xx	return to Pause Parking Position
Run-Test/ Idle		Pause Parking Position, ready for next operation

Reading or writing the IR or DR is performed bitwise from LSB to MSB. With every bit shifted into the TAP controller via **TDI**, the contents of DR is rightshifted one bit, providing the LSB on **TDO**.

Send Data 0xAC		IR/DR									Receive Data 0xD1

Send Data 0xAC   **IR/DR**   **Receive Data 0xD1**

10101100 → TDI | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | → TDO

1010110 → TDI | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | → TDO → 1

101011 → TDI | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | → TDO → 0 1

10101 → TDI | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | → TDO → 0 0 1

⋮

TDI | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | → TDO → 1 1 0 1 0 0 0 1
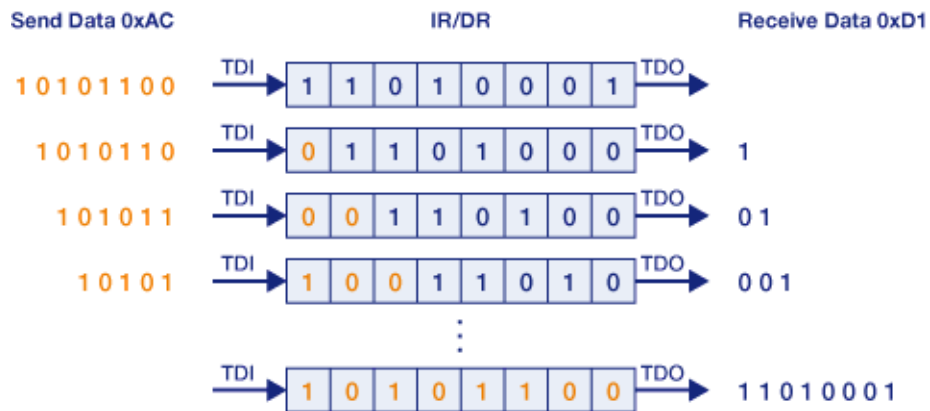
JTAG Implementation

For embedded microprocessor designs with one chip and one core there is usually only one DTAB with one TAP and so only one JTAG port. This is the **single-TAP scenario**.

Cases become more complex when there is more than one CPU core in the design. Depending on how the on-chip debug resources are implemented, there are basically two possibilities:

- There is only one DTAB and the on-chip debug logic of all cores is accessible by loading different instructions into the Instruction Register. This is a special case of the single-TAP Scenario, so physically there is no difference to a scenario with only one core.

- Each core in the embedded system has its own DTAB, accessed by its own TAP. This is the **multi-TAP scenario**, available in different flavours.

Of course any combination of these two basic concepts and their different varieties is possible. Anyway an understanding of an isolated single- and multi-TAP scenario is sufficient to create and handle complex solutions. The design recommendation is to keep it as simple as possible.

Single TAP Controller

The single TAP Scenario does not need special consideration, the signals of the chip's TAP are connected directly to the on-board JTAG connector where the debug cable is plugged.
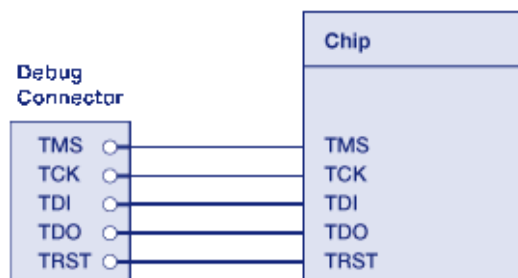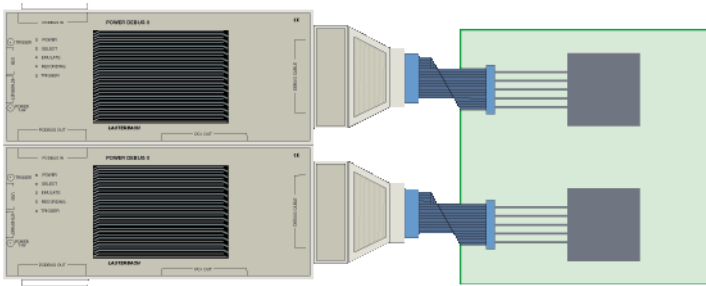
Multiple TAP Controllers

In a multi-TAP scenario each of the cores has its own DTAB. The assumption is that each DTAB has it's own JTAG TAP.

Parallel Solution

The simplest solution is having a dedicated JTAG port for each DTAB to connect a separate debugger, so each TAP can be accessed independently the same way as in the single-TAP solution:


This is the solution with the least complexity. Each debugger can control its core without caring of the others, using the full bandwidth for communication. No special multi-core setup is required, depending on the CPU each debugger runs out-of-the-box. Even designs containing special cores or architectures can be supported without any kind of adaption.

In this scenario each DTAB has its dedicated JTAG port requiring a physical connector, PCB routing and signal handling, e.g. for line compensation (cross-talk). For embedded designs with several CPU cores, each core to be debugged simultaneously requires a dedicated debugger. With more DTABs involved the costs for this solution quickly increase.

Depending on the involved core architectures, advanced features such as synchronous multi-core start are difficult to implement or even become impossible.

JTAG offers the possibility to chain different TAPs in a serial configuration. The output (**TDO**) of the previous TAP is connected to the input (**TDI**) of the next TAP:
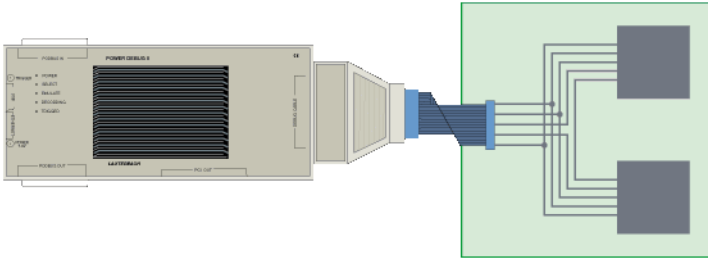
The wiring is as follows:
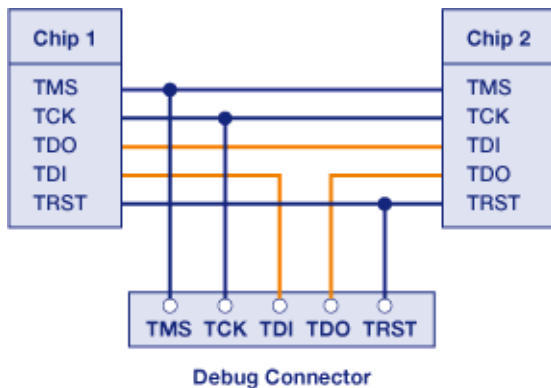
Note that only the **TDI** and **TDO** signals are chained. All other signals such as **TMS**, **TCK** and the resets are parallel and have a simultaneous effect on all participating TAPs, DTABs and cores:

• The TAP controllers of all DTABs are synchronized and in the same state.

• A Test Logic Reset state and a System Reset affects all connected DTABs and cores.

A separate TRACE32 instance is required for each core to be debugged. Even if all instances are connected to the same POWER DEBUG MODULE only one TRACE32 instance can access it's core while the other instances remain idle. Using the term **Debugger** as alias for a TRACE32 instance leads to the following consequences:

• There is a defined **master debugger**, which controls the resets. The **slave debuggers** (all others) never trigger any reset. A reset will terminate all established debug connections.

• All debuggers have a common **neutral parking position** defined where they hand over and receive communication from another debugger.

• Each debugger must know the position of it´s core in the TAP chain to correctly access it.

The chaining can either be done on board level by connecting the TAPs of each DTAB as in the example above or on chip level resulting in one JTAG port for all connected TAP controllers as shown below:

Debug Connector
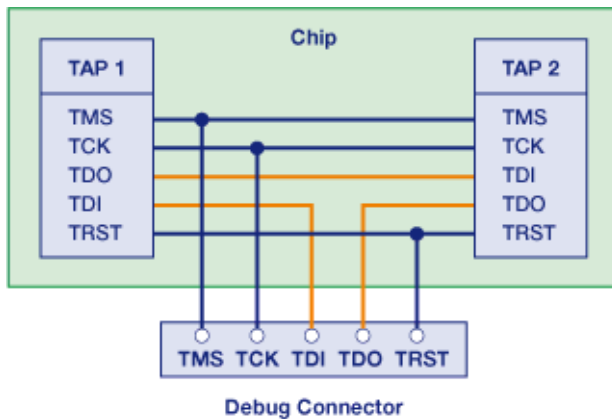
From the debugger's point of view there is no difference where the daisy-chaining is done, on-board or on-chip. Even mixed on-board/on-chip solutions are possible although the signal routing requires careful consideration.

Note that any daisy-chained scenario requires a POWER DEBUG MODULE. Older DEBUG MODULEs (without the *Power* prefix) do not support this kind of configuration.

The TAP controllers of all DTABs are synchronized. So when transmitting data in Shift-IR and Shift-DR states this means that the Instruction- and Data Registers are chained, too. Communication with all DTABs simultaneously is not possible since the connected debuggers all have different tasks to do and so have an individual communication with their core. Instead only one debugger at a time communicates with one single DTAB, by loading the Instruction Registers of all other DTABs with the BYPASS instruction ("ignore" mode).

Loading the BYPASS instruction into the Instruction Register will switch the Bypass Register into the Data Register path:

- The JTAG standard defines the **BYPASS instruction** to consist of "1"s only. In case the width of the Instruction Register (*IR-width*) is 4 bit, the JTAG instruction is 0xf, in case IR-width is 7 bit, the BYPASS instruction is 0x7f.

- The **Bypass Register** must be a 1-bit register and is preloaded with "0" in the Capture-DR state.

So depending on the position of the DTAB to be addressed, the Instruction-/Data-Registers of the other DTABs have to be filled with BYPASS instructions, respectively "bypass" bits.

On the following pages an example chip is used, which provides a chain of three DTABs (one TAP each):
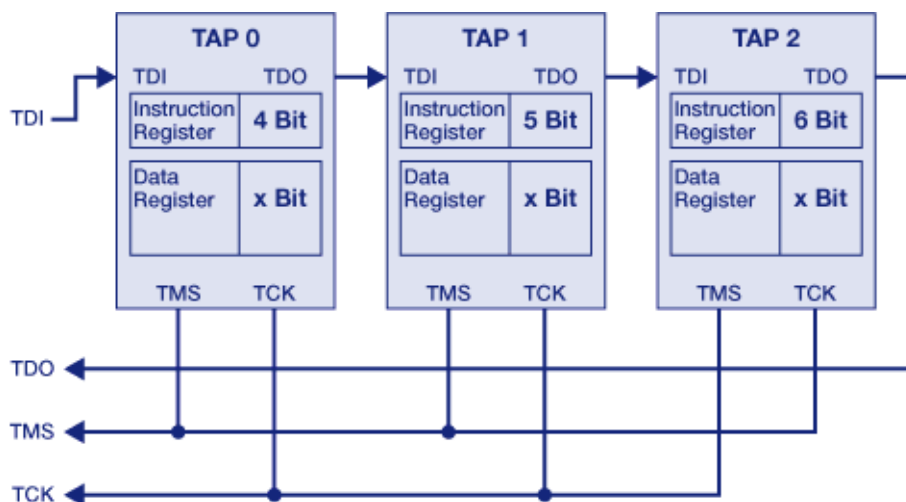
The picture below shows the Instruction/Data Registers if TAP1 is accessed.:
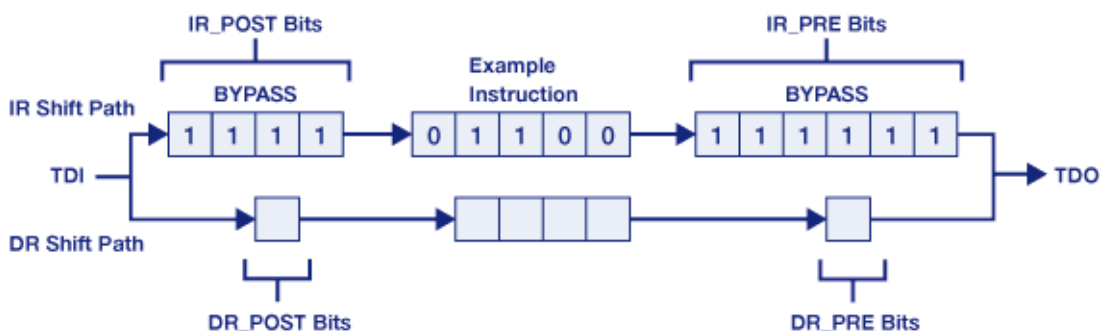
Remember that shifting is performed from LSB to MSB, so in case of an IR shift, the BYPASS instruction(s) of the TAP(s) *after* the addressed TAP are *shifted in first*. Accordingly these bits are called **IR_PRE** bits. The BYPASS instructions of the TAP(s) *before* are *shifted in last* so these bits are called **IR_POST**. The same applies to the DR path with **DR_PRE** and **DR_POST**.

On embedded designs with several TAPs the signals can be chained on board level, so only one common debug connector is required. This saves space and costs. In case the TAPs are already connected on-chip this also saves pins.

In case a single debugger hardware handles access to all cores under debug, advanced debug features such as the synchronous multi-core start can be implemented more performant.

The tool must be able to address its core within the TAP chain (multi-TAP capable), and it must be able to share the JTAG port with other tools. Because all IRs need to be accessed for shifting instructions, the complete TAP chain can only be as fast as the slowest TAP thus reducing performance of faster TAPs. This is especially the case in designs with a return clock (**RTCK**), e.g. DTABs from ARM.

The connection of the JTAG signals as well as the on-board or on-chip routing must be handled with care to avoid electrical interference, e.g. by interference of reflections at the end of branch lines.

For handling a multi-TAP system, TRACE32 needs to know only few details about the chain:

• **IR_POST** is the sum of the IR-width of all TAPs between the TDI pin of the debug connector and the addressed TAP.

• **IR_PRE** is the sum of the IR-width of all TAPs between the addressed TAP and the TDO pin of the debug connector.

• **DR_POST** usually is the sum of the bypass registers of all TAPs between the TDI pin of the debug connector and the addressed TAP.

• **DR_PRE** usually is the sum of the bypass registers of all TAPs between the addressed TAP and the TDO pin of the debug connector.

• **TAP State** is the **neutral parking position** where a TRACE32 instance hands over control of the JTAG chain to another TRACE32 instance or another debug tool. The following code are available for the neutral parking position:

0	Exit2-DR	**8**	Exit2-IR
1	Exit1-DR	**9**	Exit1-IR
2	Shift-DR	**10**	Shift-IR
3	Pause-DR	**11**	Pause-IR
4	Select-IR/ Scan	**12**	Run-Test/ Idle
5	Update-DR	**13**	Update-IR
6	Capture-DR	**14**	Capture-IR
7	Select-DR/ Scan	**15**	Test-Logic Reset

• **Master-** and **Slave Instances**: Exactly one TRACE32 instance has to control the resets (Test Reset and System Reset), all other instances are slaves.

This necessary information can either be obtained from the CPU manual or read out by using the TRACE32 command **SYStem.DETECT DaisyChain**.

The command **SYStem.DETECT DaisyChain** scans the chain and computes the multi-core settings IR_POST, IR_PRE, DR_POST and DR_PRE. The results are printed to the AREA window. Note that this only works for TAPs and chips that are fully compilant to the JTAG standard, others may result in incomplete or wrong information.

In a multi-TAP configuration it is not always possible to determine the IR width of a TAP, this is the case if the ID-code isn´t known by TRACE32 or if an ID-code is ambiguous.

Assuming a separate TRACE32 instance is used to access each of the three cores (TAPs of DTABs) in the above example and the instance for the first core is the master, the setup would be as follows:

TRACE32 instance for core 0 (TAP 0):
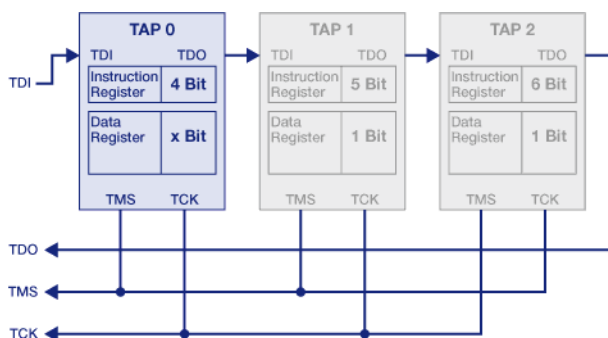

```
SYStem.CONFIG IRPOST   0.          ; no previous Instruction Registers

SYStem.CONFIG IRPRE   11.          ; consecutive Instruction Register
                                    ; bits: 5 + 6 = 11

SYStem.CONFIG DRPOST   0.          ; no previous Bypass Registers

SYStem.CONFIG DRPRE    2.          ; consecutive Bypass Register bits:
                                    ; 1 + 1 = 2

SYStem.CONFIG TAPState 12.         ; neutral parking position is
                                    ; Run-Test/Idle

SYStem.CONFIG SLAVE OFF            ; this instance controls the resets
```
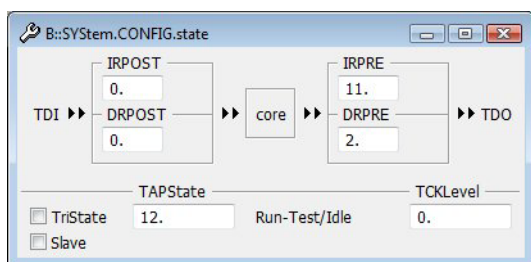
TRACE32 instance for core 1 (TAP 1):
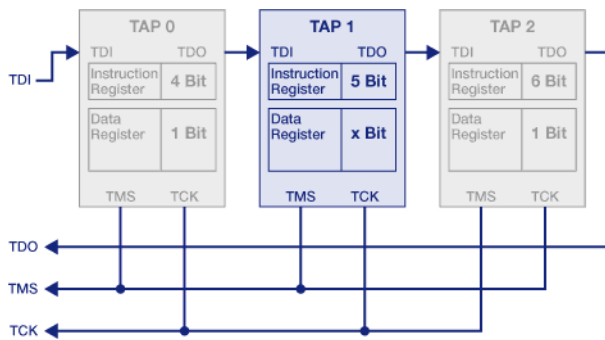

```
SYStem.CONFIG IRPOST 4.          ; previous Instruction Register bits: 4

SYStem.CONFIG IRPRE  6.          ; consecutive Instruction Register
                                 ; bits: 6

SYStem.CONFIG DRPOST 1.          ; previous Data Register bits: 1

SYStem.CONFIG DRPRE  1.          ; consecutive Data Register bits: 1

SYStem.CONFIG TAPState 12.       ; neutral parking position is
                                 ; Run-Test/Idle

SYStem.CONFIG SLAVE ON           ; this instance has no reset control
```
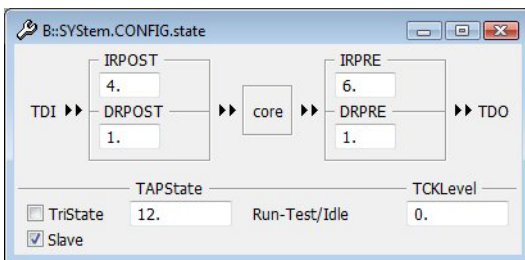
TRACE32 instance for core 2 (TAP 2):


```
SYStem.CONFIG IRPOST 9.          ; previous Instruction Register bits:
                                 ; 4 + 5 = 9

SYStem.CONFIG IRPRE  0.          ; no consecutive Instruction Registers

SYStem.CONFIG DRPOST 2.          ; previous Data Register bits: 1 + 1 = 2

SYStem.CONFIG DRPRE  0.          ; no consecutive Data Registers

SYStem.CONFIG TAPState 12.       ; neutral parking position is
                                 ; Run-Test/Idle

SYStem.CONFIG SLAVE ON           ; this instance has no reset control
```
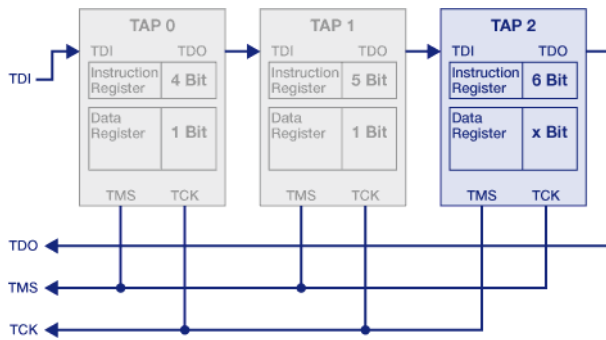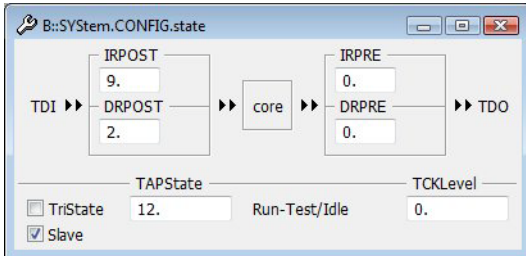
As already mentioned one disadvantage is that a daisy-chained TAP configuration can not be operated with a higher JTAG clock than the maximum possible of the slowest TAP. Depending on the involved debug architecture this can slow down the performance more or less drastically. To overcome this issue some systems offer the possibility to add or remove TAPs to or from the chain on demand.

One possible solution is to add a **board management logic**, e.g. implemented by an **FPGA**. Different TAP chains can be selected by DIP switches or jumper settings. Advantage of this solution is that the debugger is not involved in changing the chain: The user selects the desired configuration by hand and configures the debugger(s) accordingly. Due to the additionally required on-board logic this solution can only be offered at an early stage of the development process. Also it does not offer much flexibility in case an issue under debug suddenly requires changing the chain.

To add more flexibility to the system the **board management logic** could be implemented as a **master DTAB**, offering appropriate instructions and methods for changing the TAP chain layout.

A changed daisy-chain also requires an immediate change of the debugger's multi-core settings to address the selected TAP correctly. Since these switching mechanisms are highly dependent on a dedicated on-board circuit the debugger can neither know about all possible configurations nor provide any generic configuration options. Instead TRACE32 offers the possibility of a raw access to the debugger's JTAG interface for addressing the master DTAB, changing the JTAG chain and modifying the multi-core settings by using the **SYStem.CONFIG** commands. By using the PRACTICE script language this can be automatized. Additionally the TRACE32 user interface can be extended by menu- or tool-items for an on-demand configuration.

Of course the board management logic can be implemented on-chip to provide configurability of the daisy-chain at all times during the development process, reducing the necessary on-board logic.

For further information on raw access to the JTAG port see chapter **"Custom JTAG Access"** on page 26.

Both methods require that during debugging the scan chain configuration will not change again. Due to dynamic power saving states of single cores and the wish to debug them thru these states, it might be necessary to handle the scan chain configuration dynamically. In that case the switching mechanism need to be supported by debuggers firmware. The debugger itself therefor provides the command **SYStem.CONFIG MULTITAP** to select the right mechanism.

Earlier versions of TRACE32 required a separate (POWER) DEBUG MODULE for each core under debug, even if their DTABs TAPs were daisy-chained. Although this solution is still applicable today, it is recommended to switch to a solution with one POWER DEBUG MODULE only.

In case this is not possible, e.g. if only DEBUG MODULES (without the "Power" prefix) are available, each TRACE32 instance connects to its own DEBUG MODULE. All debug cables are connected to the same on-board debug connector via a special adapter. Additionally **SYStem.CONFIG TriState** has to be enabled for each TRACE32 instance. All other **SYStem.CONFIG** commands are set up as usual.

JTAG Design Recommendations
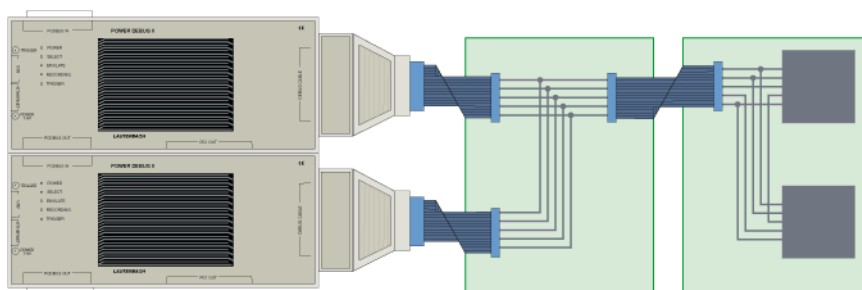
The test - and system reset signals **nTRST** and **RESET** must never be connected to each other since the debugger needs to handle them independently. Otherwise a hot attach (**SYStem.Mode Attach**) to an already running application will be impossible without resetting the CPU core or the whole system. Also other scenarios like *halt after reset* (**SYStem.Mode Up**) will fail or at least exhibit strange behavior.

There is no debugger attached when the customer finally uses the embedded application, so the reset signals have to have a defined state, allowing the debugger to override these default signals when attached. So when no debugger is connected,

- **nTRST** needs to be active in order to disable the on-chip debug resource for preventing any unwanted interference. This can be achieved by a weak pull-down resistor. The debugger fully drives the **nTRST** signal to override this default.

- **nRESET** needs either to be constantly inactive (weak pull-up) or is driven by a reset controller. The TRACE32 debug cables usually implement an open drain driver to pull-down the signal in order to reset the system. To release the system from reset, the debug cable switches to high-impedance state (tristate) to avoid disturbing the reset controller's operation. Otherwise external reset events could be prevented.

Depending on the architecture and the device, the other JTAG signals may require special wiring, as well as the non-JTAG signals, e.g. **DBGREQ**, ... For more information, see the

- Documentation of the CPU manufacturer

- Schematics of reference board designs

- **TRACE32** > **Help** menu > **Processor Architecture Manual** for the specific architecture

Regarding the PCB layout all signals should be as short as possible, branch lines should be avoided. Consider termination of branch lines and do not use capacities or inductivities for signal compensation, especially when using several debug cables from different generations.

Route the signals in a way that the influence of possible cross-talk is minimized. Extending the debug cable is not recommended. If this can not be avoided, a lower JTAG clock may be required reducing up- and download performance.

In general connect all ground lines for shielding purposes.

Custom JTAG Access

For implementing the debug features the TRACE32 debugger only makes use of a few JTAG functions. Other functions can be used by accessing the JTAG TAP controller directly, possible use cases are

- Boundary scan

- Device and design verification

- Switching to special CPU or chip modes

- Configuring the JTAG chain (see chapter **"Changing the JTAG Chain"** on page 23).

- Communication with special devices accessible via a JTAG TAP, e.g. for board setup

TRACE32 offers two interfaces for performing a custom JTAG access:

- Remote API

- **JTAG** commands.

Overview

Available Signals

The following signals can be accessed:

JTAG signals	TMS, TCK, TDI, TDO, nTRST
System signals	nRESET, VTREF
Debugger related signals	nENOUT

VTREF is read-only and indicates whether the target has power or not.

nENOUT enables or disables (tristates) the output drivers of the debug cable.

nRESET and **VTREF** are also available in a latched version to check whether there was a system reset or a power fail since the last check.

Access Levels

Depending on the use case, there are different access levels:

- **Direct Access** for reading or altering the state of a dedicated signal. This is mainly useful for signals that are not used for communication, e.g. **nRESET, VTREF,** ... .

  Another use case is selecting a certain chip mode which becomes active when a signal has a certain value at a defined event, e.g. **TMS** is high when **nRESET** becomes inactive (note that although this is common this violates the JTAG standard).

- **Raw Access** for sending and receiving bit patterns on several signals (lines) simultaneously. This allows a comfortable walk through the TAP controller's state machine, simultaneously sending and receiving data on **TDI** and **TDO**. The **TCK** signal is automatically driven, the timing depends on the JTAG clock as set by the **SYStem.JtagClock** command.

- **Low-level** Access is provided for the main use cases where the Instruction and Data Registers are to be written.

Of course any combination of these levels is possible and required in many cases.

The current state of the debugger is an important issue.

In case the debugger is in a state where it is logically not connected to the target device (**SYStem.Mode Down** or **SYStem.Mode NoDebug**) basically all JTAG manipulations are allowed. Just in case the debugger is expected to connect to the target afterwards, the signals should be reset to default state as far as possible, especially **TMS**, **TCK** and **TDI**.

A more complex case is when the work of the debugger is to be interrupted for some custom access. This should be done as follows:

1.   Lock the JTAG port for the debugger. Now the TAP controller is in the pause parking position. Consult the **Processor Architecture Manual** to find out what is the pause parking position for your core. Normally this is either 12. (Run-Test/Idle) or 7. (Select-DR Scan).

2.   Perform your JTAG access.

3.   Return to the pause parking position.

4.   Unlock the JTAG port for the debugger. The debugger will resume operation immediately.

As long as the JTAG port is locked, the debugger will not access the target.

In case an active debug session is interrupted, never issue a reset or alter the on-chip debug resources unless you know exactly what you do. This may confuse the debugger or leads to a corrupted debug session.

Remote API

The **TRACE32 Remote API** is a software interface and allows access to a running TRACE32 instance from an external application. The connection is based on UDP/IP, so the application may reside on another host. Interface support is offered for different programming languages on different host systems, e.g. C/ C++, Java, Visual Basic on Windows, Linux, Solaris or HP-UX. The necessary interface files can be found in the **demo/api/** directory.

The remote API is completely documented in the **"API for Remote Control and JTAG Access"** (api_remote.pdf). Additional to some basic functions for starting and terminating a connection with the TRACE32 instance, there are special functions for performing a raw- or low-level JTAG access as documented in chapter **ICD TAP Access API Functions**.

The main advantage is the full control of the JTAG port for almost all use cases with the cost of effort in programming. To explain the usage of the TRACE32 remote API, the following examples show different use cases and methods accessing the TAP controller. The main task will be reading out the ID-code for device identification. Most examples are based on the TriCore architecture.

Basics

The shown examples are based on the following prerequisites:

- A TRACE32 instance is running on the local host, listening for incoming remote API connections on UDP Port 20000. For this the TRACE32 configuration file (**config.t32** as default) contains these lines:

```
; TRACE32 API access
RCL=NETASSIST
PORT=20000
PACKLEN=1024
```

- The POWER DEBUG MODULE is connected to a powered target board.

- All remote API interface calls are expected to return successfully to make the examples more readable. In a real application, error checking is strongly recommended.

The TAP access functions of the TRACE32 remote API automatically lock the JTAG port for the API. Each function is passed a parameter (a handle of the type `T32_TAPACCESS_HANDLE`), which defines whether the JTAG port shall be released (unlocked) after the access:

```
T32_TAPACCESS_HOLD                    /* do not release JTAG port */

T32_TAPACCESS_RELEASE                 /* release JTAG port */
```

When unlocking, an active debugger (in **SYStem.Mode Up)** will immediately resume operation on the JTAG port. As a general recommendation, try to bundle as much operations as possible but do not block the JTAG port permanently unless the debugger is inactive.

The JTAG Port can be released with `T32_TAPAccessRelease()` at any time without condition.

The following C-code demonstrates how a custom application uses the remote API to connect to the TRACE32 instance:

```
#include "t32.h"

void main(void)
{
    /* connect to TRACE32 */
    T32_Config("NODE=", "localhost");   /* host with TRACE32 running */
    T32_Config("PORT=", "20000");       /* port where TRACE32 listens */
    T32_Init();                         /* connect to TRACE32 */
    T32_Attach(T32_DEV_ICD);             /* mandatory due to historical*/
                                         /* reasons */

    /* connected, now performing JTAG operations */
    /* execute command sequence */
    T32_TAPAccessDirect(T32_TAPACCESS_HOLD,…);
    ...

    /* terminate connection */
    T32_TAPAccessRelease();    /* release JTAG port */
    T32_Exit();
}
```

`T32_Config()` is used to specify on which host and port the TRACE32 instance is listening for incoming connections.

`T32_Init()` followed by `T32_Attach()` connect to TRACE32 .

Before `T32_Exit()` is used to terminate the connection to TRACE32 `T32_TAPAccessRelease()` unlocks the JTAG port. Note that there is no automatic return to the pause parking position!

For documentation on compilation details please refer to the remote API documentation.

Now that the connection handling is explained, the ID-code is read out using different methods.

Direct Access

The following example assumes that the debugger is inactive (in **SYStem.Mode Down** or **NoDebug**).

Timed Access to Different Signals

The task is to release a potential active system reset and to perform a TAP reset via asserting **nTRST**:

```c
#include "t32.h"

typedef unsigned long u_int32;

void main(void)
{
    /* direct TAP access */
    byte nCommand[8];    /* commands for direct TAP access */

    /* connect to TRACE32 */
    ...

    /* define access sequence:
       - enable the debug cable if it was tristated,
       - release system reset and reset TAP controller
       - then wait before disabling TAP controller reset
     */
    nCommand[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_LOW;
    nCommand[1] = T32_TAPACCESS_nRESET | T32_TAPACCESS_SET_HIGH;
    nCommand[2] = T32_TAPACCESS_nTRST  | T32_TAPACCESS_SET_LOW;
    nCommand[3] = T32_TAPACCESS_SLEEP_MS;
    nCommand[4] = 50;
    nCommand[5] = T32_TAPACCESS_nTRST  | T32_TAPACCESS_SET_HIGH;
    nCommand[6] = T32_TAPACCESS_SLEEP_MS;
    nCommand[7] = 50;

    /* execute command sequence */
    T32_TAPAccessDirect(T32_TAPACCESS_HOLD, 8, nCommand, 0);

    /* perform other JTAG operation */
    ...

    /* terminate connection */
    ...
}
```

By using the byte-array `nCommand[8]`, a sequence of line- and wait operations can be defined. So it is possible to implement a timing-specific control of each signal independently. Delays can be defined in ms and µs.

```
; line operation (write)
T32_TAPACCESS_<signal_name>| T32_TAPACCESS_SET_<signal_level>

; wait operation
T32_TAPACCESS_SLEEP_<time_unit>
<number_of_time_units>
```

The command sequence is finally executed with the `T32_TAPAccessDirect()` function. The first parameter is the handle, the second one defines the number of command bytes to execute. The third parameter is the command array, and the last parameter optionally returns the results of the corresponding commands as a byte array. If no result is required a NULL pointer can be used instead.

The following C-code shows how to read from a line:

```c
#include "t32.h"

typedef unsigned long u_int32;

void main(void)
{
    /* direct TAP Access */
    byte nCommand[2];    /* commands for direct TAP access */
    byte nResult[2];     /* results from direct TAP acccess */
    int  bPower, bReset;

    /* connect to TRACE32 */
    ...

    /* read lines */
    nCommand[0] = T32_TAPACCESS_VTREF;      /* check target power */
    nCommand[1] = T32_TAPACCESS_nRESET;     /* check system reset */
    T32_TAPAccessDirect(T32_TAPACCESS_HOLD, 2, nCommand, nResult);

    bPower = !!nResult[0];
    bReset =  !nResult[1];    /* active low */

   /* perform other JTAG operation */
   ...

    /* terminate connection */
    ...
}
```

```
; line operation (read)
T32_TAPACCESS_<signal_name>
```

The task is reading out the ID-code available from the DR right after a TAP reset:

```c
#include "t32.h"

typedef unsigned long u_int32;

void main(void)
{
    /* shift data */
    byte    nTMSBits[4];     /* bits for controlling the state machine */
    byte    nTDOBits[4];     /* bits for data from target device */
    u_int32 nIDCode = 0;     /* ID-code */

    /* connect to TRACE32 */
    ...

    /* release system reset and reset TAP controller */
    ...

    /* TAP Controller may be in unknown state,
       reset via TMS and move to Run-Test/ Idle */
    nTMSBits[0] = 0x7f;    /* TMS sequence: 1 1 1 1 1 1 1 0 */
    T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD, 8, nTMSBits, 0, 0,
                        SHIFTRAW_OPTION_NONE);

    /* move TAP Controller to Shift-DR state */
    nTMSBits[0] = 0x1;     /* TMS sequence: 1 0 0 */
    T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD, 3, nTMSBits, 0, 0,
                        SHIFTRAW_OPTION_NONE);

    /* read ID-code and leave Shift-DR state with last bit */
    T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD, 32, 0, 0, nTDOBits,
                        SHIFTRAW_OPTION_LASTTMS_ONE);

    /* return to Run-Test/Idle and return control to the debugger */
    nTMSBits[0] = 0x1;     /* TMS sequence: 1 0 */
    T32_TAPAccessShiftRaw(T32_TAPACCESS_RELEASE, 2, nTMSBits, 0, 0,
                        SHIFTRAW_OPTION_NONE);

    /* assemble ID-code */
    nIDCode = ((u_int32)nTDOBits[3]<<24) | ((u_int32)nTDOBits[2]<<16) |
              ((u_int32)nTDOBits[1]<<8)  |  (u_int32)nTDOBits[0];

    /* terminate connection */
    ...
}
```

Beside the lock handle the `T32_TAPAccessShiftRaw()` function requires the number of bits to shift. The shift data is passed by two byte arrays, one for **TMS** and one for **TDI**.

If only one or no shift arrays is passed, the `SHIFTRAW_OPITION_*` defines the behavior.

`SHIFTRAW_OPTION_NONE`	Don´t care
`SHIFTRAW_OPTION_LASTTMS_ONE`	If no TMS shift array is specified, shift at least "`1`" at the last shift, to change to the next TAP controller state

A third byte array is used for the **TDO** data. Bits are shifted starting with the LSB of the first array element. In case no input or output data is required the corresponding arrays can be replaced by NULL pointers.

```
int T32_TAPAccessShiftRaw(T32_TAPACCESS_HANDLE connection,
                          int numberofbits, byte * pTMSBits, byte * pTDIBits,
                          byte * pTDOBits, int options);
```

Low-level Access

For the main task of a custom JTAG port access, accessing IR and DR, the remote API provides functions for a one-command access from the pause parking position. The above tasks can also be implemented this way:

```c
#include "t32.h"

typedef unsigned long u_int32;

void main(void)
{
    /* shift data */
    byte    nTMSBits[4];    /* bits for controlling the state machine */
    byte    nTDOBits[4];    /* bits for data from target device */
    u_int32 nIDCode = 0;    /* ID-code */

    /* connect to TRACE32 */
    ...

    /* release system reset and reset TAP controller */
    ...

    /* TAP controller may be in unknown state,
       reset via TMS and move to pause parking position (Run-Test/Idle) */
    nTMSBits[0]  = 0x1f;    /* TMS sequence: 1 1 1 1 1 0 */
    T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD, 6, nTMSBits, 0, 0,
                          SHIFTRAW_OPTION_NONE);

    /* move TAP controller to Shift-DR state, read the 32 bit ID Code and
       returnto Run-Test/Idle state (pause parking position) and return
       control to the debugger */
    T32_TAPAccessShiftDR(T32_TAPACCESS_RELEASE, 32, 0, nTDOBits);

    /* assemble JTAG ID Code */
    nIDCode = ((u_int32)nTDOBits[3]<<24) | ((u_int32)nTDOBits[2]<<16) |
              ((u_int32)nTDOBits[1]<<8)  |  (u_int32)nTDOBits[0];

    /* terminate connection */
    ...
}
```

Beside the lock handle the T32_TAPAccessShiftIR() and T32_TAPAccessShiftIR() functions require the number of bits to shift. The shift data is passed by a byte arrays for **TDI** while **TMS** is driven automatically depending on the pause parking position. A second byte array is used for **TDO** data. Bits are shifted starting with the LSB of the first array element. In case no input or output data is required the corresponding arrays can be replaced by NULL pointers.

An active debugger always moves the TAP controller to the pause parking position when locking. For an inactive debugger the current position in the state machine is undefined.

The `T32_TAPAccessShiftIR()` and `T32_TAPAccessShiftIR()` functions can also handle daisy-chained TAP Controllers. The configuration is defined by the `T32_TAPAccessSetInfo()` function. The settings of the **SYStem.CONFIG** commands are not used.

Low-level Access Interrupting the Debugger

Things get more complicated when the current work of the debugger is to be interrupted for some low-level JTAG access.

The following example interrupts the debugger for reading out the ID-code. Since a debug session is already active, it is not possible to perform any TAP reset (neither via **nTRST** nor via **TMS**) any more.

```
#include "t32.h"

typedef unsigned long u_int32;

void main(void)
{
    /* shift data */
    byte    nTDIBits[4];    /* bits for data to target device */
    byte    nTDOBits[4];    /* bits for data from target device */
    u_int32 nIDCode = 0;    /* ID-code */

    /* connect to TRACE32 */
    ...


    /* shift IDCODE instruction */
    nTDIBits[0] = 0x04;
    T32_TAPAccessShiftIR(T32_TAPACCESS_HOLD, 8, nTDIBits, 0);

    /* read 32 bit ID Code and return control to debugger */
    T32_TAPAccessShiftDR(T32_TAPACCESS_RELEASE, 32, 0, nTDOBits);

    /* assemble JTAG ID Code */
    nIDCode = ((u_int32)nTDOBits[3]<<24) | ((u_int32)nTDOBits[2]<<16) |
              ((u_int32)nTDOBits[1]<<8)  |  (u_int32)nTDOBits[0];

    /* terminate connection */
    ...
}
```

Command Line Control

The **JTAG** commands offer the possibility of a custom JTAG access directly from the command line.

The commands can be issued either directly from the command line or from a PRACTICE script without having to write an external application. So a dedicated board-setup can be implemented in the default startup scripts.

The **JTAG** commands are documented in the **"General Commands Reference Guide J"** (general_ref_j.pdf), chapter **JTAG**.

Note that the **JTAG** commands do not offer low-level shift functions. Driving through the TAP controller's states and taking multi-TAP settings into account is up to the user.

Basics

Locking the JTAG port is simple, only the two commands **JTAG.LOCK** and **JTAG.UNLOCK** are required:

```
; Lock JTAG port
  JTAG.LOCK

; perform JTAG operation
  ...


; Release JTAG port
  JTAG.PIN TMS 0    ; Pull TMS pin to 0
  JTAG.PIN TDI 0    ; Pull TDI pin to 0
  JTAG.UNLOCK

ENDDO
```

Note that the above example also resets the **TMS** and **TDI** lines before returning control to the debugger to avoid unintended effects.

JTAG.LOCK	Disable all debugger activity on the JTAG port when the first manual command (e.g. JTAG.PIN) performs.
JTAG.PIN *<pin>* **0	1**	Set the specified signal level to *<pin>* of the JTAG connector.
JTAG.UNLOCK	Re-enable debugger activity on the JTAG port.

Direct Access

As already seen in the previous example a direct access to a dedicated signal can be easily done with the **JTAG.PIN** command. The following example assumes an inactive debugger. The example enables the debug cable, removes the system reset and performs a TAP reset via **nTRST**:

```
; Lock JTAG port and enable debug cable
  JTAG.LOCK
  JTAG.PIN ENable

; release System Reset and reset TAP Controller
  JTAG.PIN NRESET HIGH
  JTAG.PIN NTRST  LOW
  WAIT 50.MS              ; wait until reset is active
  JTAG.PIN NTRST  HIGH
  WAIT 50.MS              ; wait until reset is released

; perform JTAG operation
  ...

; Release JTAG port
  JTAG.PIN TMS 0    ; Pull TMS pin to 0
  JTAG.PIN TDI 0    ; Pull TMS pin to 0
  JTAG.UNLOCK

ENDDO
```

JTAG.PIN ENable	Enable the output drivers for all output signals of the JTAG connector.
WAIT *<time>*	Advise TRACE32 to wait the specified time.

To read the current status of a signal the TRACE32 function **JTAG.PIN(**<*pin*>**)** is used:

```
; Lock JTAG port and enable debug cable
  JTAG.LOCK
  JTAG.PIN ENable

  &power=JTAG.PIN(VTREF)
  &nreset=JTAG.PIN(NRESET)

; Release JTAG port
  JTAG.UNLOCK

  IF (&power==1)
  (
    PRINT "Target has Power"
    IF (&nreset==0)
      PRINT "Target is in Reset"
    ELSE
      PRINT "Target is not in Reset"
  )
  ELSE
    PRINT "Target has no Power"

ENDDO
```

Raw Access

The raw access is used for shifting bit patterns on single lines. The following example shows how to read out the JTAG ID code after a TAP reset:

```
; Lock JTAG port and enable debug cable
  JTAG.LOCK
  JTAG.PIN ENable

; release System Reset
; reset TAP Controller and move to "Run-Test/ Idle"
; ID-code is automatically loaded to IDCODE register is available
  ...

; read 32 bit IDCODE register:

; move from "Run-Test/ Idle" to "Shift-DR" state
  JTAG.SHIFTTMS 1 0 0

; 16 Bit "Shift-DR", WITHOUT exiting "Shift-DR"
  JTAG.SHIFTTDI 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
; get data read out via TDO
  &lowid=JTAG.SHIFT()

; 16 Bit "Shift-DR", WITH exiting to "Exit1-DR"
  JTAG.SHIFTREG 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
; get data read out via TDO
  &highid=JTAG.SHIFT()

; move from "Exit1-DR" to "Run-Test/ Idle" state
  JTAG.SHIFTTMS 1 0

; release JTAG port
  JTAG.PIN TMS 0    ; Pull TMS pin to 0
  JTAG.PIN TDI 0    ; Pull TDI pin to 0
  JTAG.UNLOCK

; assemble ID-code
  &idcode=FORMAT.HEX(4,&highid)+FORMAT.HEX(4,&lowid)
  PRINT "ID-code of device : &idcode"

ENDDO
```

The PRACTICE commands **JTAG.SHIFTTMS** is used to shift an arbitrary bit pattern on **TMS** while zero is shifted on **TDI**. The leftmost bit is shifted in first. Any number of bits can be shifted.

JTAG.SHIFTTMS *<bit>* [*<bit> …*]	Walk through the states of the TAP controller.
JTAG.SHIFTTDI *<bit>* [*<bit> …*]	Shift data if the TAP controller is in "Shift-DR" state.
JTAG.SHIFTREG*<bit>* [*<bit> …*]	Shift data if the TAP controller is in "Shift-DR" state. Leave "Shift-DR" state at the end of shift.

JTAG.SHIFTTDI and **JTAG.SHIFTREG** are used to shift data on **TDI** with the leftmost bit shifted first. The difference in both commands is the pattern shifted simultaneously on **TMS**: Using **JTAG.SHIFTTDI** results in a zero bit TMS sequence and so stays in Shift-IR or Shift-DR state. Using **JTAG.SHIFTREG** shifts a one on the last **TMS** bit for leaving to Exit1-ID or Exit1-DR. Any number of bits can be shifted.

With every shift operation the data on **TDO** is read out. To obtain the data of the last **TMS** or **TDI** shift, the function **JTAG.SHIFT()** is used.

JTAG.SHIFT()	Return TDO output
FORMAT.HEX(*<width>,<value>***)**	Return a string of the specified *<width>*, that shows *<value>* as a hex. number

Many Instruction- and Data Registers have a length of a multiple of 8, 16 or 32 bits. In this case the **JTAG.SHIFT*** commands can be used in a different way:

```
; Lock JTAG port and enable debug cable
  JTAG.LOCK
  JTAG.PIN ENable

; release System Reset
; reset TAP Controller and move to "Run-Test/ Idle"
  ...

; read JTAG ID Code:

; move TAP controller from "Run-Test/ Idle" to "Shift-DR" state
  JTAG.SHIFTTMS 1 0 0

; 32 Bit "Shift-DR", WITH exiting to "EXIT1-DR"
  JTAG.SHIFTREG %Long 0x0
; get data read out via TDO
  &idcode=JTAG.SHIFT()

; move TAP controller from "Exit1-DR" to "Run-Test/ Idle" state
  JTAG.SHIFTTMS 1 0

; Release JTAG port
  JTAG.PIN TMS 0    ; pull TMS pin to 0
  JTAG.PIN TDI 0    ; pull TDI pin to 0
  JTAG.UNLOCK

  PRINT "ID-code of device : &idcode"

ENDDO
```

In this notation, the length of the shift is specified by the well-known PRACTICE operators, e.g. **%Byte**, **%Word** or **%Long**. The bits to shift are passed as a hexadecimal value, LSB is shifted first.

JTAG.SHIFTREG %LONG *<value>*	Shift 32-bit data if the TAP controller is in "Shift-DR" state. Leave "Shift-DR" state at the end of shift.

JTAG Commands via the Remote API

Of course it is possible to use the described commands via the **TRACE32 Remote API** by using the **T32_Cmd()** or **T32_CmdWin()** interface functions.