LAUTERBACH
DEVELOPMENT TOOLS

XC800 Debugger

MANUAL

XC800 Debugger

Version 06-Jun-2024

Introduction

This document describes the processor specific settings and features for TRACE32-ICD for the Infineon XC800 CPU family.

Please keep in mind that only the **Processor Architecture Manual** (the document you are reading at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs supported by Lauterbach. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Brief Overview of Documents for New Users

Architecture-independent information:

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

Architecture-specific information:

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **“OS Awareness Manuals”** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

Warning

WARNING:	To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF. Recommendation for the software start: 1. Disconnect the Debug Cable from the target while the target power is off. 2. Connect the host system, the TRACE32 hardware and the Debug Cable. 3. Power ON the TRACE32 hardware. 4. Start the TRACE32 software to load the debugger firmware. 5. Connect the Debug Cable to the target. 6. Switch the target power ON. 7. Configure your debugger e.g. via a start-up script. Power down: 1. Switch off the target power. 2. Disconnect the Debug Cable from the target. 3. Close the TRACE32 software. 4. Power OFF the TRACE32 hardware.

Starting up the debugger is done as follows:

Select the device prompt for the ICD Debugger and reset the system.

```
B::
```

The device prompt `B::` is normally already selected in the TRACE32 command line. If this is not the case, enter `B::` to set the correct device prompt. The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

5. Specify the CPU specific settings.

   ```
   SYStem.CPU <cpu_type>
   ```

   The default values of all other options are set in such a way that it should be possible to work without modification. Please consider that this is probably not the best configuration for your target.

6. Set up data for electrical interface.

   ```
   SYStem.JtagClock <frequency>
   ```

   Use the subcommands of **MAP** to define inaccessible memory areas. Bus errors can be removed by executing SYStem.Up. Make sure that there isn’t any TRACE32 window open which accesses to a inaccessible memory that is not masked out, otherwise the bus error can occur again.

7. Enter debug mode.

   ```
   SYStem.Up
   ```

   This command resets the CPU and enters debug mode. After this command is executed, it is possible to access memory and registers.

8. Load your application program.

   ```
   Data.LOAD.OMF myprogram /Verify     ; OMF specifies the format,
                                       ; myprogram is the file name)
   ```

   The format of the **Data.LOAD** command depends on the file format generated by the compiler. This test discovers a problem with the electrical connection, wrong chip configurations or linker command file settings.

   A detailed description of the **Data.LOAD** command and all available options is given in the **“General Commands Reference”**.

The start-up can be automated using the programming language PRACTICE. A typical start sequence for the XC888-8FF is shown below:

```
b::                                 ; Select the ICD device prompt

WinCLEAR                            ; Clear all windows

SYStem.CPU XC888                    ; Select CPU

SYStem.Up                           ; Reset the target and enter debug mode

Data.LOAD.OMF MYPROG /VERFY         ; Load the application, verify the
                                    ; process

Go main                             ; Run and break at main()

Data.List                           ; Open source window

Register.view /SpotLight            ; Open register window

Var.Local                           ; Open window with local variables
```

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command this may have the following reasons.

- The JTAG lines are not connected correctly.
- The target has no power.
- The pull-up resistor between the JTAG[VCCS] pin and the target VCC is too large.
- The target is in reset:

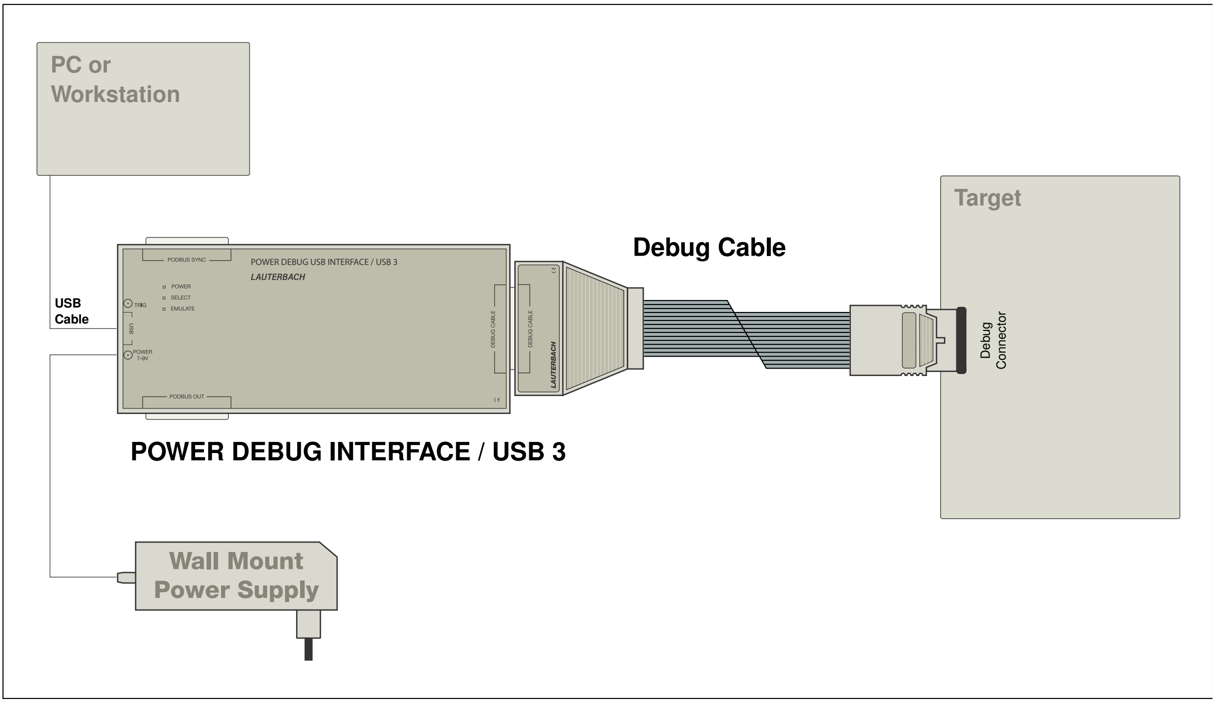
  The debugger controls the processor reset and use the RESET line to reset the CPU on every SYStem.Up. Therefore no external R-C combination or external reset controller is allowed.
- There is logic added to the JTAG state machine:

  By default the debugger supports only one processor in one JTAG chain. If the processor is only one member of a JTAG chain the debugger has to be informed about the target JTAG chain configuration. Use the SYStem.CONFIG command to specify the position of the device in the JTAG-chain.
- There are additional loads or capacities on the JTAG lines.

FAQ

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

Configuration



The processor type must be selected by the **SYStem.CPU** command before issuing any other target related commands.

XC800 Specific Implementations

Breakpoints

There are two types of breakpoints available: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are the default breakpoints for program breakpoints. A software breakpoint is implemented by patching a break code into the memory.

There is no restriction in the number of software breakpoints.

On-chip Breakpoints

The resources for the on-chip breakpoints are provided by the CPU.

The following list gives an overview of the on-chip breakpoints for the XC800:

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

	On-chip Breakpoints	**Instruction Breakpoints**	**Read/Write Breakpoints**	**Data Breakpoint**
XC800	4	up to 4 up to 1 range (2 single needed)	up to 1 single address read or address range up to 1 single address write or address range	—

CPU specific SYStem Settings and Restrictions

SYStem.state — Open system window

```
Format:          SYStem.state
```

Opens a window with settings of CPU specific system commands. Settings can also be changed here.

SYStem.CONFIG — Configure debugger according to target topology

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

<parameter>:     CORE       <core>

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

The **SYStem.CONFIG** commands have no effect in Simulator. These commands describe the physical configuration at the JTAG port and the trace port of a multi-core hardware target. Since the simulator normally just simulates the instruction set, these commands will be ignored. Refer to the relevant **Processor Architecture Manual** in case you want to know the effect of these commands on a debugger.

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger about the TAP controller position in the JTAG chain, if there is more than one core in the JTAG chain (e.g. Arm + DSP). The information is required before the debugger can be activated e.g. by a **SYStem.Up**. See **Daisy-chain Example**.
For some CPU selections (**SYStem.CPU**) the above setting might be automatically included, since the required system configuration of these CPUs is known.

TriState has to be used if several debuggers (“via separate cables”) are connected to a common JTAG port at the same time in order to ensure that always only one debugger drives the signal lines. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: nTRST must have a pull-up resistor on the target, TCK can have a pull-up or pull-down resistor, other trigger inputs need to be kept in inactive state.

Multicore debugging is not supported for the DEBUG INTERFACE (LA-7701).

CORE

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

DRPRE

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

DRPOST

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

IRPRE

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

IRPOST

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

TAPState

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

TCKLevel

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

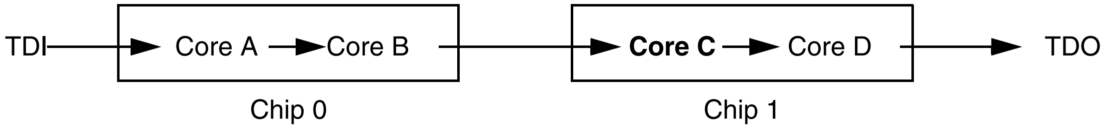
TriState

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

Slave

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

Daisy-Chain Example



Below, configuration for core C.

Instruction register length of

- Core A: 3 bit
- Core B: 5 bit
- Core D: 6 bit

```
SYStem.CONFIG.IRPRE  6.                ; IR Core D
SYStem.CONFIG.IRPOST 8.                ; IR Core A + B
SYStem.CONFIG.DRPRE  1.                ; DR Core D
SYStem.CONFIG.DRPOST 2.                ; DR Core A + B
SYStem.CONFIG.CORE 0. 1.               ; Target Core C is Core 0 in Chip 1
```

TapStates

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

Format:	**SYStem.CONFIG.CORE** *<core_index> <chip_index>* **SYStem.MultiCore.CORE** *<core_index> <chip_index>* (deprecated)
<chip_index>:	1 … i
<core_index>:	1 … k

Default *core_index*: depends on the CPU, usually 1. for generic chips

Default *chip_index*: derived from CORE= parameter of the configuration file (config.t32). The CORE parameter is defined according to the start order of the GUI in T32Start with ascending values.

To provide proper interaction between different parts of the debugger, the systems topology must be mapped to the debugger's topology model. The debugger model abstracts chips and sub cores of these chips. Every GUI must be connect to one unused core entry in the debugger topology model. Once the **SYStem.CPU** is selected, a generic chip or non-generic chip is created at the default *chip_index.*

Non-generic Chips

Non-generic chips have a fixed number of sub cores, each with a fixed CPU type.

Initially, all GUIs are configured with different *chip_index* values. Therefore, you have to assign the *core_index* and the *chip_index* for every core. Usually, the debugger does not need further information to access cores in non-generic chips, once the setup is correct.

Generic Chips

Generic chips can accommodate an arbitrary amount of sub-cores. The debugger still needs information how to connect to the individual cores e.g. by setting the JTAG chain coordinates.

Start-up Process

The debug system must not have an invalid state where a GUI is connected to a wrong core type of a non-generic chip, two GUIs are connected to the same coordinate or a GUI is not connected to a core. The initial state of the system is valid since every new GUI uses a new *chip_index* according to its CORE= parameter of the configuration file (config.t32). If the system contains fewer chips than initially assumed, the chips must be merged by calling **SYStem.CONFIG.CORE**.

SYStem.CONFIG.state

Display target configuration

Format:	**SYStem.CONFIG.state** [**/**<*tab*>]
<*tab*>:	**DebugPort** | **Jtag**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<*tab*>	Opens the **SYStem.CONFIG.state** window on the specified tab: **DebugPort**, **JTAG**.
DebugPort	Lets you configure the electrical properties of the debug connection, such as the communication protocol or the used pinout.
Jtag	Informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip.

SYStem.CPU

Select CPU

Format:	**SYStem.CPU** <*cpu*>
<*cpu*>:	**XC866** | **XC866L** | **XC886** | **XC888** | **XC886C** | **XC888C** | **XC886CM** | **XC888CM** | **XC886LM** | **XC888LM** | **XC886CLM** | **XC888CLM** | **XC878** | **XC878M** | **XC878CM** | **XC878L** | **XC878C** | **TC2X_SCR** | **TLE9832** | **TLE9834**

Selects the processor type.

SYStem.MemAccess

Select run-time memory access method

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

Default: Denied.

Enable **CPU** (deprecated)	A run-time memory access is made without CPU intervention while the program is running. This is only possible on the instruction set simulator.
StopAndGo	Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.

SYStem.Mode Establish communication with the target

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

Down	The CPU is held in reset, debug mode is not active. Default state and state after fatal errors.
NoDebug	Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tri-stated.
Go	Resets the target and enables the debugger and start the program execution. Program execution can be stopped by the break command or if any break condition occurs.
Attach	User program remains running (no reset) and the debug mode is activated. After this command the user program can be stopped with the break command or if any break condition occurs.
Up	Resets the target, sets the CPU to debug mode and stops the CPU. After the execution of this command the CPU is stopped and all register are set to defaults.
StandBy	Not supported.

SYStem.LOCK Tristate the JTAG port

Format:	**SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the JTAG port will be performed by the debugger. While locked the JTAG connector of the debugger is tristated. The intention of the **SYStem.LOCK** command is, for example, to give JTAG access to another tool.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to

SYStem.Option.LittleEndian Treat memory as little endian

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

Default: OFF.

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

When the **SYStem.Option.TRAPEN** check box is checked, the debugger sets the TRAP_EN flag in the Extended Operation (EO) register before executing the next GO command.

- The XC800 extends the 8051 instruction set with the special command

  ```
  MOVC @(DPTR++),A
  ```

  to write data (e.g. from a I2C LPC memory IC) into program RAM. As the 8051 instruction set is only 8 bit wide, and there were no unused opcodes available, the TRAP opcode 0A5h is re-used for this instruction.

- The functionality of the 0A5h opcode is determined by the bit TRAP_EN in the Extended Operations (EO) register (usually EO.4).

  - When TRAP_EN=1 (set), 0A5h means “TRAP”.
  - When TRAP_EN=0 (reset), 0A5h means “MOVC @(DPTR++),A.

- This conflicts with the operation of software breakpoints. Software breakpoints are set by replacing an instruction with a “TRAP“ instruction. When the processor stops in debug mode, the original instruction is restored. The next STEP or GO command then executes the instruction.

- Therefore a software breakpoint may be illegally interpreted as an MOVC operation if you

  - disable the TRAP_EN check box.
  - Manually reset the TRAP_EN bit (EO.4)

Format:	**SYStem.JtagClock** [<*frequency*> | **RTCK** | **ARTCK** <*frequency*> | **CTCK** <*frequency*> | **CRTCK** <*frequency*>] **SYStem.BdmClock** [<*frequency*> ...] (deprecated)
<*frequency*>:	**6 kHz … 80 MHz** **1250000. | 2500000. | 5000000. | 10000000.**

Default frequency: 10 MHz.

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines. A very high frequency will not work on all systems and will result in an erroneous data transfer. Therefore we recommend to use the default setting if possible.

<*frequency*>	• The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window. • Besides a decimal number like “100000.”' short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.

When the debugger is not working correctly (e.g. memory is flickering), decrease the JtagClock.

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

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

The on-chip breakpoints can only cover specific ranges. If a range cannot be programmed into the breakpoint, it will automatically be converted into a single address breakpoint when this option is active. This is the default. Otherwise an error message is generated.

```
TrOnchip.CONVert ON
Break.Set 0x1000--0x17ff /Write        ; sets breakpoint at range
Break.Set 0x1001--0x17ff /Write        ; 1000--17ff sets single breakpoint
…                                      ; at address 1001

TrOnchip.CONVert OFF                   ; sets breakpoint at range
Break.Set 0x1000--0x17ff /Write        ; 1000--17ff
Break.Set 0x1001--0x17ff /Write        ; gives an error message
```

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

Format: **TrOnchip.RESet**

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

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

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

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

OCDS1 Connector

Signal	Pin	Pin	Signal
TMS	1	2	VCCS
TDO	3	4	GND
N/C	5	6	GND
TDI	7	8	RESET-
TRST-	9	10	BRKOUT-
TCLK	11	12	GND
BRKIN-	13	14	N/C
RESERVED	15	16	N/C

A standard 2 x 8 pin header (pin-to-pin spacing: 0.1 inch = 2.54mm) is required on the target.

- Do not connect the “reserved” pin.
- Do connect all GND pins and all N/C pins for shielding purpose, though they are connected on the debugger.

VCCS is the processor power supply voltage. It is used to detect if target power is on and its voltage level determines the output buffer level of the debugger. That means the output voltage of the debugger signals (TMS, TDI, TCLK, TRST-, BRKIN-) depends directly on VCCS. VCCS can be 2.25 … 5.5 V. The output buffer takes about 2 mA.

- RESET- is controlled by an open drain driver. An external watchdog must be switched off if the In-Circuit Debugger is used.
- BRKIN and BRKOUT- must be configured in MCBS (Multi Core Break Switch) for before they can be used.
- VIHmin = 2.0 V, VILmax = 0.8 V for the input pins TDO, BRKOUT-.

For an example design please see the Infineon Evaluation Board schematics.

Pins	Connection	Description	Recommendations
1	TMS	Test Mode Select	None.
2	VCCS	VCC Sense	Connect to Chip I/O voltage VCC.
3	TDO	Test Data Out	If there are multiple devices on the JTAG chain, connect TDO to the TDI signal of the next device in the chain.
4, 6, 12	GND	System Ground Plan	Connect to digital ground.
7	TDI	Test Data In	No other devices in the JTAG chain are allowed between the Debug Cable and the XC800.
8	$\overline{\text{RESET}}$	Reset	Connect to /PORST and connect /PORST to VCC via a 10 K pull-up resistor. Do not connect to /TRST.
9	$\overline{\text{TRST}}$	Test Reset	Connect to /TRST if available. Do not connect to /PORST.
10	$\overline{\text{BRKOUT}}$	Break out	None.
11	TCLK	Test clock	None.
13	BRKIN	Break input	None.
5, 14, 15, 16	NC	Not Connected	Connect to Ground.

Memory Classes

The following memory classes are available:

Memory Class	Description
P	Code space (program)
X	External data space including XRAM
I	Internal RAM (Indirect Address)
D	Special Function Registers (non-mapped) + Internal RAM (Direct Address)

The low 128 bytes of the internal data memory can be accessed with the memory classes I and D. The upper 128 bytes in the memory class D represent the Special Function Registers SFR (standard). The Special Function Registers (standard, mapped and paged) can be accessed in the peripherie window.

XRAM can be read/written as program memory or external memory.