LAUTERBACH
DEVELOPMENT TOOLS

XC2000/XC16x/C166CBC Debugger

XC2000/XC16x/C166CBC Debugger

XC2000/XC16x/C166CBC Debugger

Version 06-Jun-2024

Introduction

This document describes the processor specific settings and features for the debugger ICD-166CBC and the debugger ICD-166SV2. (You can find the description of ROM Monitors for 80C166 family at **“C166 Monitor”** (monitor_c166.pdf)

ICD-166CBC supports processors based on the C166CBC and C166SV1 core, like PMB2850 (E-GOLD), PMB6850 (E-GOLD+), PMB7850 (E-GOLD+V3), SDA6000 (M2), INKA, C165UTAH, C161U, PEF20580 (DOLCE), …

ICD-166SV2 supports processors based on the C166SV2 core, like XC161CJ, XC164CS, …

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.

ICD/AICD

In the following we use the short form ICD (In-Circuit Debugger) for debug systems running on the debug box “Debug Interface” and AICD (Active In-Circuit Debugger) for debug systems running on the debug box “Power Debug Interface”, “Power Debug Ethernet” and “Power Trace”.

For installation and to make you familiar with the main features of the debugger see the manual “Quick Installation and Tutorial”.

Brief Overview of Documents for New Users

Architecture-independent information:

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

Architecture-specific information:

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

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known XC2000/XC16x/C166CBC based hardware.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

  You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software.

You can also manually navigate in the `~~/demo/c166/` subfolder of the system directory of TRACE32.

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.

Monitor Routine

The monitor routine is required **for ICD-166CBC, only**. There is no monitor routine when using ICD-166SV2.

The following resources are used by the debugger:

- Stack Memory: 8 bytes of memory on the current stack.
- Program Memory: 32 bytes of program memory for a exception routine.
- A TRAP vector (4 bytes) at 20H.

The exception routine will be loaded automatically by the debugger on a **SYStem.Up** at the address which is selectable by the **SYStem.Option.MONBASE** command. Also the jump command will be written automatically to 20H.

You must do all settings (e.g. BUSCON register) to make write access to these locations possible. If there is ROM, you must place the exception routine and trap vector in the ROM yourself and you must inform the debugger by using the **SYStem.Option.MONBASE** command about the location. Exception: see **SYStem.MODE Prepare**.

The command sequence of the exception routine is depending on the option **SYStem.Option.WATCHDOG**.

WATCHDOG = ON:

```
push r0
bset psw.0x6
loop:srvwdt
jb psw.0x6,loop
pop r0
push dpp3
mov dpp3,#0x3
atomic #4
mov dpp3:0x30fc,zeros
bclr tfr.0xc
pop dpp3
reti
```

The binary code is:

```
0xEC, 0xF0, 0x6F, 0x88, 0xA7, 0x58, 0xA7, 0xA7,
0x8A, 0x88, 0xFC, 0x60, 0xFC, 0xF0, 0xEC, 0x03,
0xE6, 0x03, 0x03, 0x00, 0xD1, 0x30, 0xF6, 0x8E,
0xFC, 0xF0, 0xCE, 0xD6, 0xFC, 0x03, 0xFB, 0x88
```

WATCHDOG = OFF:

```
push r0
bset psw.0x6
loop: diswdt
jb psw.0x6,loop
pop r0
push dpp3
mov dpp3,#0x3
atomic #4
mov dpp3:0x30fc,zeros
bclr tfr.0xc
pop dpp3
reti
```

The binary code is:

```
0xEC, 0xF0, 0x6F, 0x88, 0xA5, 0x5A, 0xA5, 0xA5,
0x8A, 0x88, 0xFC, 0x60, 0xFC, 0xF0, 0xEC, 0x03,
0xE6, 0x03, 0x03, 0x00, 0xD1, 0x30, 0xF6, 0x8E,
0xFC, 0xF0, 0xCE, 0xD6, 0xFC, 0x03, 0xFB, 0x88
```

At the location 20H a jump to this function has to be placed (if address = 1FFFC0H):

```
jmps 0x1f,0x0ffc0
```

The binary code is (if address = 1FFFC0H):

```
0xfa, 0x1f, 0xc0, 0xff
```

Quick Start

Check if there is a suitable script file for your hardware in ~~/demo/c166/etc/. Read the comments in the script file.

After finishing the preparations (see **Monitor**) starting up the debugger is done as follows:

1. Select the device prompt B: for the ICD Debugger.

   ```
   b:
   ```

   If you are working with the PODPC card device `b::` is already selected.

2. Select the CPU type to load the CPU specific settings.

   ```
   SYStem.CPU PMB2850
   ```

   If you are working with the PODPC card, the correct CPU family is selected automatically after start-up.

3. Tell the debugger where's ROM on the target.

   ```
   MAP.BOnchip 0x100000++0x0fffff
   ```

   This command is necessary for the use of on-chip breakpoints.

4. 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.

5. Load your application program.

   ```
   Data.LOAD.IEEE PROG166        ; IEEE specifies the format, PROG166 is
                                 ; the file name
   ```

   The option of the **Data.LOAD** command depends on the file format generated by the compiler. A detailed description of the **Data.LOAD** command is given in the **"General Commands Reference"**.

The start-up can be automated using the programming language PRACTICE. An example of a start-up sequence is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

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

WinCLEAR                            ; Clear all windows

MAP.BOnchip 0x100000++0x0fffff      ; Specify where's ROM

SYStem.cpu PMB2850                  ; Select the processor type

SYStem.Up                           ; Reset the target and enter debug
                                    ; mode

Data.LOAD.IEEE PROG166              ; Load the application

Register.Set PC main                ; Set the PC to function main

List.Mix                            ; Open disassembly window        *)

Register.view /SpotLight            ; Open register window           *)

Frame.view /Locals /Caller          ; Open the stack frame with
                                    ; local variables                *)

Var.Watch %Spotlight flags ast      ; Open watch window for variables *)

PER.view                            ; Open window with peripheral register
                                    ;                                *)

Break.Set sieve                     ; Set breakpoint to function sieve

Break.Set 0x1000 /Program           ; Set software breakpoint to address
                                    ; 1000 (address 1000 is in RAM)

Break.Set 0x101000 /Program         ; Set on-chip breakpoint to address
                                    ; 101000 (address 101000 is in ROM)
                                    ; See restrictions in On-chip
                                    ; Breakpoints.)
```

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

Quick Start for Tracing with MCDS On-chip Trace

It is assumed that you are tracing a XC2000ED.

1. Start and Stop Tracing

```
SYStem.CPU XC2000ED                       ; select XC2000ED CPU
```

Load your application and prepare for debug.

2. Specify Trace Source and Recording Options

Select the source what should be recorded (e.g. Program Flow and Timestamps). When enabling Timestamps, the CPU clock has to be added also.

```
MCDS.SOURCE C166 FlowTrace ON              ; enable TriCore program flow
                                           ; trace

MCDS.TimeStamp TICKS                       ; enable Ticks as timestamps

MCDS.CLOCK SYStem 80.0MHz                  ; configure CPU clock for correct
                                           ; timestamp evaluation
```

3. Start and Stop Tracing

```
Go                                        ; start tracing

Break                                     ; stop tracing
```

Note that tracing can also be stopped by a breakpoint.

4. View the Results

```
Onchip.List                               ; view recorded trace data
```

Memory Classes

The following memory classes are available:

Memory Class	Description
P	Program
D	Data

Since C166CBC has von Neumann architecture there is no difference in the use of these memory classes.

If you use the memory classes E, EP or ED the memory is accessed even if the target CPU is running. There is no difference in the use of E, EP and ED. The JTAG debugger use the Debug Peripheral Event Controller (DPEC) to access memory. This acts like a cycle stealing DMA. If a Data.Dump window is opened by using one of these memory classes, the window contents will also be refreshed while the processor is running (see also **SYStem.Option.DUALPORT**). Please note that in this case the program will not be executed at full speed.

CPU specific SYStem Commands

SYStem.CPU

Select the CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**PMB2850** | **PMB6850** | **PMB7850** | **PEF20580** | **SDA6000** | **C165UTAH** | **INKA** | …(ICD-166CBC) **XC161CJ,** …**,XC2287,** …**XE167F,** …**XC2000ED,** …(ICD-166SV2)

Default: PMB2850 (ICD-166CBC), C166SV2 (ICD-166SV2)
Selects the processor type.

SYStem.JtagClock — Define the JTAG frequency

Format:	**SYStem.JtagClock** *<rate>* **SYStem.BdmClock** (deprecated)
<rate>:	**EXT** | **1000.** … **10000000.** (**ICD**) **10000.** … **50000000.** (**AICD**)

Default 5 MHz (**ICD**), 10 MHz (**AICD**).

Selects the frequency for the JTAG clock. This influences the speed of data transmission between target and debugger.

EXT selects the clock on the pin CPUCLOCK of the JTAG connector as clock source.

Attention: The frequency of the JTAG clock must be lower than the system clock frequency!

Not all values in between the frequency range can be generated by the debugger. The debugger will select and display the possible value if it can not generate the exact value.

SYStem.MemAccess — Select run-time memory access method

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

Enable **CPU** (deprecated)	Memory access during program execution to target is enabled.
Denied (default)	Memory access during program execution to target is disabled.
StopAndGo	Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed. For more information, see below.

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*>:	**Attach**
	Down
	Go
	NoDebug
	Prepare
	StandBy
	Up

Default: Down

Selects the target operating mode.

“Debug mode is active” means the communication channel via debug port (JTAG) is established. The features of the “on-chip debug support” (OCDS) are enabled and available.

Attach	User program remains running (no reset). Debug mode is active. This mode can be entered from state “NoDebug”, if debugging should be enabled without a target reset. After this command the user program can be stopped with the break command or if any break condition occurs. XC2xxx: Attach is only possible if a pull-up resistor is on TRST pin.
Down	The CPU is in reset. Debug mode is not active. Default state and state after fatal errors.
Go	The user application is running. Debug mode is active. After this command the program can be stopped with the break command or if any break condition occurs.
NoDebug	The user application is running. Debug mode is not active. Debug port is tristate. In this mode the target should behave as if the debugger is not connected.
Prepare	ICD-166CBC: The CPU is halted. Communication to CPU is established. In this mode memory access is possible, run control (step, go, break) is not available. This command can be entered in the command line, only. A valid user program after power on is not required. See “~~/demo/c166/etc/egold/demo.cmm”. ICD-166SV2: Behaves as UP

StandBy	This mode is not supported.
Up	The CPU runs in debug monitor routine (ICD-166CBC) or is in halt mode (ICD-166SV2). Debug mode is active. In this mode the user application can be started and stopped. This is the most typical way to activate debugging.

If the mode “Go” or “Attach” or “Prepare” is selected, this mode will be entered, but the control button in the SYStem window jumps to the mode “UP”.

The “Emulate” LED on the debug module is on when the debug mode is active and the CPU is running.

XC2xxx: If the routing of the JTAG pins is changed (Register DBGPRR), only mode “Attach” and “Go” are possible.

SYStem.LOCK — Lock and tristate the debug port

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

Default: OFF.

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

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. For tab descriptions, see below.
DebugPort	Informs the debugger about the debug connector type and the communication protocol it shall use.
Jtag	Informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip.

SYStem.CONFIG Configure debugger according to target topology

Format:	**SYStem.CONFIG** *<parameter> <number_or_address>* **SYStem.MultiCore** *<parameter> <number_or_address>* (deprecated)
<parameter>:	**CORE** *<core>*
<parameter>: (JTAG):	**DRPRE** *<bits>* **DRPOST** *<bits>* **IRPRE** *<bits>* **IRPOST** *<bits>* **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**] **Slave** [**ON** | **OFF**]

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.

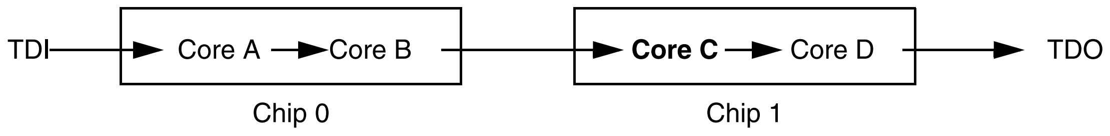
TriState has to be used if several debuggers ("via separate cables") are connected to a common JTAG port at the same time in order to ensure that always only one debugger drives the signal lines. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to 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
```

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.DAP — Define mapping for DAP pins

The **SYStem.CONFIG.DAP** commands are used to map the unused JTAG pins for additional features.

SYStem.CONFIG.DAP.BreakPIN — Define mapping of break pins

Format:	**SYStem.CONFIG.DAP.BPIN** [**PortPort** | **TdiPort** | **PortTdo** | **TdiTdo**]

Default: PortONLY.

This command maps a Break Bus to either a GPIO port pin or an unused JTAG pin. It is dependent on the selected debug port type which Break Bus can be mapped to which pin:

	Break Bus 0	**Break Bus 1**
PortPort	GPIO port pin	GPIO port pin
TdiPort	TDI pin	GPIO port pin
PortTDO	GPIO port pin	TDO pin
TdiTdo	TDI pin	TDO pin

SYStem.CONFIG.DAP.DAPENable — Enable DAP mode on PORST

Format:	**SYStem.CONFIG.DAP.DAPEN** [**TARGET** | **ON** | **OFF**]

Default: TARGET.

Defines if the DAP Interface of the CPU is enabled during a Power On Reset (PORST). This command requires that the debugger DAP Interface is enabled by **SYStem.CONFIG.DEBUGPORTTYPE** before.

For target boards where a pull-up resistor on nTRST line permanently enables the DAP Interface the TARGET setting is required.

In case the CPU DAP Interface should not be enabled although a debugger is attached to the target board, the OFF setting is recommended. When performing a **SYStem.Mode Go** or **SYStem.Mode Up**, the debugger enables the CPU DAP Interface automatically when performing the PORST. Note that a **SYStem.Mode Attach** is not possible in this case.

If the CPU DAP Interface should be enabled as long as the debugger is attached, the ON setting is required. All **SYStem.Mode** options are possible in this case, including hot attach.

See **"Application Note Debug Cable TriCore"** (app_tricore_ocds.pdf) for details.

SYStem.CONFIG.DAP.USERn — Configure and set USER pins

Format:	**SYStem.CONFIG.DAP.USER0** [**In** | **Out** | **Set** *<level>*] **SYStem.CONFIG.DAP.USER1** [**In** | **Out** | **Set** *<level>*]
<level>:	**Low** | **High**

- Default for **USER0**: In.
- Default for **USER1**: Out and Low.

Configures the `USER0` and `USER1` pins of the 10 pin DAP Debug Connector as input or output. The output level can be Low or High.

Use the functions **DAP.USER0()** and **DAP.USER1()** for reading the current status.

The availability of the USER pins depends on the Debug Cable, the selected Interface Mode and the DAP Enabling Mode. See **"Application Note Debug Cable TriCore"** (app_tricore_ocds.pdf) for details.

SYStem.CONFIG.DEBUGPORTTYPE — Set debug cable interface mode

Format:	**SYStem.CONFIG.DEBUGPORTTYPE** [**JTAG** | **DAP2** | **SPD**] **SYStem.CONFIG.Interface** [**JTAG** | **DAP2** | **SPD** (deprecated)

Default: JTAG.

This command is used to configure the Interface Mode used by the Debugger. Both CPU and Debug Cable must support this mode, see **"Application Note Debug Cable TriCore"** (app_tricore_ocds.pdf) for details.

SYStem.Option.DUALPORT Run-time memory access for all windows

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

Default: OFF.

The JTAG debugger use the Debug Peripheral Event Controller (DPEC) to access memory. This acts like a cycle stealing DMA. Therefore memory access can be done even while the CPU is running. On activating this option the opened data windows will also be refreshed while a user program is running. Please consider that in this mode the user program will not be executed at full speed.

SYStem.Option.IDLEFIX Periodically activate/deactivate JTAG connection

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

Default: OFF.

This is a bug fix for PMB7850 which is only available on obsolete ICD hardware. The permanent JTAG connection did not allow the processor to switch to idle mode which was required for flash programming.

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 the value before the step.

SYStem.Option.MonBase — Define start address of debug monitor

Format:	**SYStem.Option.MonBase** *<start_address>*

ICD-166CBC only. Default: 0x1fffc0.

This is the start address where the exception routine is or will be loaded. The size of the exception routine is at the moment 32/48 (**ICD**/**AICD**) bytes.

SYStem.Option.PERSTOP — Enable global peripheral suspend signal

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

Default: OFF.

This controls the operation mode of the peripherals (e.g. timer), when a debug event is raised. A debug event causes the peripherals to suspend, if this option is activated and the suspend enable bit in the peripheral module is set.

SYStem.Option.PERSTOPFIX — Break CPU via ONCHIP break register

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

Default: OFF.

If asynchronous Break is used and **SYStem.Option.PERSTOP** is set, then this option should be set, too. Workaround to use complete functionality of peripheral suspend (XC2xxx and XC16x)

SYStem.Option.BRKOUT — Activates BRKOUT signal

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

Default: OFF.

Activates the BRKOUT signal on the DEBUG connector. Must be connected to the CPU pin.

SYStem.Option.WATCHDOG — Disable or serve watchdog

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

Default: OFF.

This controls if the watchdog is active (on) during the debug session or not (off). See also chapter **Monitor Routine**.

SYStem.Option.TRACEENABLE — Disable traceport

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

Only PMB7890 and XGOLD110.

Default: OFF with Debugger. ON with Combiprobe

Disable/enable trace port from the PMB7890 and XGOLD110.

SYStem.Option.DebugLevel — Debug level

Format:	**SYStem.Option.DebugLevel** <*level*>
<*level*>:	**1. … 15. TRAP ATOMIC**

Only CPUs with C166SV2 core

Default: TRAP

The CPU can be interrupted. The debug level defined the level, where all lower and equal level interrupts are breaked.

SYStem.Option.BootModeIndex

BootModeIndex

Format:	**SYStem.Option.BootModeIndex** *<interface>*
<interface>:	**JTAG1** | **JTAG2** | **JTAG3** | **JTAG4** | **JTAG5** | **JTAG6** | **JTAG7** | **JTAG8** | **DAP1** | **DAP2** | **SPD**

Only XC2xxxLE and XC2xxxULE CPUs

Selects the Debug Interface which is programmed via ASC with the **SYStem.Mode Prepare**

JTAG1 Mode: TCK=P2.9 TMS=P5.4 TDI=P5.2 TDO=P10.12

JTAG2 Mode: TCK=P2.9 TMS=P5.4 TDI=P10.10 TDO=P10.12

JTAG3 Mode: TCK=P2.9 TMS=P10.11 TDI=P5.2 TDO=P10.12

JTAG4 Mode: TCK=P2.9 TMS=P10.11 TDI=P10.10 TDO=P10.12

JTAG5 Mode: TCK=P10.9 TMS=P5.4 TDI=P5.2 TDO=P10.12

JTAG6 Mode: TCK=P10.9 TMS=P5.4 TDI=P10.10 TDO=P10.12

JTAG7 Mode: TCK=P10.9 TMS=P10.11 TDI=P5.2 TDO=P10.12

JTAG8 Mode: TCK=P10.9 TMS=P10.11 TDI=P10.10 TDO=P10.12

DAP1 Mode: DAP0=P2.9 DAP1=P10.12

DAP2 Mode: DAP0=P10.9 DAP1=P10.12

SPD Mode: at P10.12

SYStem.Option.ICFLUSH   Flush instruction cache

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

Default: ON.

If enabled, the InstructionCache will be flushed before GO or Step operations. This is required to enforce consistency between cache and external program memory when the program memory was updated (e.g. for setting software breakpoints). Typically the option shall be left enabled except when debugging cache consistency problems in the target. The option is only relevant for XC22xxI, XC23xxE and XC27x8X because they have a program cache.

SYStem.Option.IDLEDEBUG   Debug in IDLE state

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

Only XGOLD110 ES2

If enabled, it is possible to break the aplication if the CPU is in IDLE state. Access to the peripheral register via real-time access is also possible.

SYStem.Option.WaitReset   Delay between PORST and JTAG shifts

Format:	**SYStem.Option.WaitReset** <*time*>
<*time*>:	**800.ms** ... **5000.ms**

Only XC2xxx and XE16x.

Default: 1750.ms.

Change the delay between the rising edge of the POST line and the first shifts from the Debugger. Only necessary to change the delay if a **SYStem.Mode Up** does not stop at the reset vector.

MCDS Onchip Trace

MCDS Onchip Trace Features

Onchip tracing is only possible with an Infineon Emulation Device (ED), offering the MCDS (MultiCore Debug Solution) for implementing trace triggers, filters and generation of trace messages (MCDS messages).

Use **Trace.METHOD Onchip** for selecting the onchip trace.

Supported Features

- Program Flow Trace
- Data Trace
- Ownership Trace
- Timestamps
- **Simple Trace Control**

See the **Onchip.Mode** commands for a general setup of the on-chip trace, and the **MCDS** commands for a detailed setup of the on-chip MCDS resources.

Trace Control

The On-chip settings can be done with the **Onchip** commands, e.g. from the **Onchip.view** window. .

Simple Trace Control

Additionally triggers and filters on data address and data value can be configured.

- Trace all
- Trace to
- Trace from
- Trace from to
- Trigger
- Enable

MCDS uses compression to efficiently use the limited amount of on-chip trace memory. TRACE32 requires a synchronization point to decode all consecutive trace messages.

Examples

A successfully loaded target application is needed for the following examples.

Example 1: Trace function sieve() only

```
MCDS.view                                 ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace ON             ; enable C166 program flow trace

Onchip.List                               ; show trace list window

Break.Set v.range(sieve) /Program         ; enable the trace as long as
/Onchip /TraceEnable                      ; function sieve() is executed
```

Enable the trace as long as the function sieve() is executed. Execution in sub-functions is not recorded.

Example 2: Trace function sieve() and sub-functions

```
MCDS.view                                 ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace ON             ; enable C166 program flow trace

Onchip.List                               ; show trace list window

Break.Set sieve /Program /Onchip          ; enable trace on entering
/TraceON                                  ; function sieve()

Break.Set y.exit(sieve) /Program          ; disable trace on leaving
/Onchip /TraceOFF                         ; function sieve()
```

Trace the complete function sieve(), including execution in any sub-function.

Example 3: Trace until

```
MCDS.view                                ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace ON            ; enable C166 program flow trace

Onchip.List                              ; show trace list window

Break.Set v.end(sieve) /Program          ; disable trace on leaving
/Onchip /TraceTrigger                    ; function sieve()
```

Stop tracing when end of function sieve() is reached, C166 keeps running. **Onchip.TDelay** can be used to stop recording after a programmable period (percentage of the trace memory). See the **Trace.Trigger** command for more information.

Example 4: Trace write accesses to a variable

```
MCDS.view                                ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace OFF           ; disable C166 program flow
                                         ; trace

MCDS.SOURCE C166 WriteAddr ON            ; enable C166 write address
MCDS.SOURCE C166 WriteData ON            ; and write data trace

Onchip.List                              ; show trace list window

Break.Set flags+0x0C /Write /Onchip      ; enable recording when C166
/TraceEnable                             ; writes to address flags+0x0C
```

Trace all write accesses to variable flags with offset 0xc, Program Flow trace is disabled to save on-chip trace memory.

Example 5: Trace specific write accesses to a variable

```
MCDS.view                                ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace ON            ; enable C166 program flow trace

MCDS.SOURCE C166 WriteAddr ON            ; enable C166 write address
MCDS.SOURCE C166 WriteData ON            ; and write data trace

Onchip.List                              ; show trace list window

Break.Set flags+0x0C /Write              ; enable recording when C166
/Data.Byte 0x01 /Onchip                  ; writes 0x01 to address flags+0x0C
/TraceEnable
```

Enable recording when TriCore writes 0x01 with an access width of 8 bits to address flags+0x0C. The code that triggered the write access is also recorded. Due to pipeline effects and internal delays the recorded code may not exactly match the write instruction.

Example 6: Trace specific write accesses to a variable

```
MCDS.view                                ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace ON            ; enable C166 program flow trace

MCDS.SOURCE C166 WriteAddr ON            ; enable C166 write address
MCDS.SOURCE C166 WriteData ON            ; and write data trace

Onchip.List                              ; show trace list window

Break.Set v.range(sieve)                 ; enable recording when C166
/MemoryWrite flags+0x0C                  ; writes 0x01 to address flags+0x0C
/Data.Byte 0x01 /Onchip                  while executing function sieve()
/TraceEnable
```

Enable recording when C166 writes 0x01 with an access width of 8 bits to address flags+0x0C while executing function sieve() (excluding sub-function). The code that triggered the write access is also recorded. Due to pipeline effects and internal delays the recorded code may not exactly match the write instruction.

BenchMarkCounter

The BenchMarkCounter are only available with a XC2000ED device.

For information about *architecture-independent* **BMC** commands, refer to **“BMC”** (general_ref_b.pdf).

For information about *architecture-specific* **BMC** commands, see command descriptions below.

BMC.CNTx.EVENT — Configure the performance monitor

Format:	**BMC.CNT0 | CNT1** … **CNT7.EVENT** <*event*>
<*option*>:	**NONE** **Delta** **Echo** **NINST** **IDLE** **STALL** **IRA** **SYNC_RQ**

NONE	Switch off the performance monitor
Delta	Counts hits of the Delta-Marker, if specified.
Echo	Counts hits of the Echo-Marker, if specified.
NINST	Counts the number of instructions.
IDLE	Counts the number of idle cycles.
STALL	Counts the number of stall cycles.
IRA	Counts the number of interrupts acknowledged.
SYNCH_RQ	The counter is incremented at the beginning of a new paragraph of the trace buffer memory.

Useful Features

This chapter gives an overview on some useful features. Please consult the documentation of the corresponding commands for more information.

Runtime Measurement

```
MCDS.view                                 ; show MCDS setup window

MCDS.SOURCE C166 FlowTrace OFF            ; disable C166 program flow trace

MCDS.TimeStamp Relative                   ; sets relative timestamp messages

MCDS.Clock SYStem 80.MHz                  ; sets system clock

Break.Set sieve /Program /Onchip          ; enable trace on entering
/TraceEnable                              ; function sieve()

Break.Set y.exit(sieve) /Program          ; enable trace on leaving
/Onchip /TraceEnable                      ; function sieve()

Break.SetFunc sieve                       ; sets marker Alpha at begin and
                                          marker Beta at end of function
                                          sieve()

Trace.STATistic.DURation                  ; analyze the time between sieve()
                                          entry and sieve() exit
```

Program BootModeIndex (only XC2xxxULE/LE)

```
SYStem.CPU XC2210U-8F                     ; select CPU

SYStem.CONFIG.DEBUGPORTTYPE DAP           ; select DAP interface

SYStem.Option.BootModeIndex DAP1          ; select the desired Interface

SYStem.JtagClock 9600.                    ; selects the ASC speed 9600 BAUD

SYStem.Mode Prepare                       ; programs the BootModeIndex
```

This chapter describes the possibility to program the BootModeIndex via the Debug Cable

This is only possible if the LA-3815 Conv. 16 Pin JTAG to DAP is used.

Breakpoints

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

Software Breakpoints on Instructions

Software breakpoints are the default breakpoints. They can only be used in RAM areas.There is no restriction in the number of software breakpoints.

On-chip Breakpoints

The following list gives an overview of the usage of the on-chip breakpoints by TRACE32-ICD:

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction breakpoints:** Number of on-chip breakpoints that can be used for program and spot breakpoints
- **Data breakpoints:** Number of on-chip breakpoints that can be used as read or write breakpoints.

On-chip Breakpoints	Instruction Breakpoints	Data Breakpoints
4	up to 4	up to 4 write up to 1 read

You can check your currently set breakpoints with the command **Break.List**

If no more on-chip breakpoints are available you will get a message on trying to set an on-chip breakpoint.

On-chip Breakpoints in FLASH/ROM

With the command **MAP.BOnchip** *<range>* it is possible to inform the debugger where you have ROM (FLASH,EPROM) on the target. If a breakpoint is set within the specified address range the debugger uses automatically the available on-chip breakpoints.

Example for Breakpoints

Assume you have a target with FLASH from `0` to `0xFFFFF` and RAM from `0x100000` to `0x11FFFF`. The command to configure TRACE32 correctly for this configuration is:

```
Map.BOnchip 0x0--0x07FFFF
```

You inform the debugger that he shall use on-chip breakpoints instead of software breakpoints in the address range 0-7FFFH (though your flash is up to address FFFFFH).

Examples for instruction breakpoints:

```
Break.Set 0x100000 /Program     ; software breakpoint, instruction

Break.Set 0x101000 /Program     ; software breakpoint, instruction

Break.Set 0xx /Program          ; software breakpoint, instruction
```

Three instruction breakpoints are set. Software breakpoints are used.

```
Break.Set 0x100 /Program        ; on-chip breakpoint, instruction

Break.Set 0x0ff00 /Program      ; on-chip breakpoint, instruction
```

Two instruction breakpoints are set. On-chip breakpoints are used, because of the MAP.BOnchip command.

```
Break.Set 0x9FFFF /P /Onchip    ; on-chip breakpoint, instruction
```

A instruction breakpoint is set. On-chip breakpoint is used, because of the /Onchip option.

```
Break.Set 0x8FFFF /Program      ; error message
```

This causes an error, because the debugger tries to set a software breakpoint at this location.

```
Break.Set 0x8FFFF++0x100 /P     ; on-chip breakpoint, instruction,
                                ; range
```

Breakpoint on an instruction range 8FFFF-900FFH will be set, even if this range is not declared by MAP.BOnchip command. The reason is that0 for range events always on-chip breakpoints will be used.

Examples for breakpoints on data:

```
Break.Set 0x100000 /Write       ; on-chip Breakpoint, data write access
```

Breakpoint on write access to 100000H.

```
  Break.Set 0x9FFFF /Read          ; on-chip Breakpoint, data read access
```

Breakpoint if read access to 9FFFFH. For breakpoints on data always on-chip breakpoint will be used.

TrOnchip Commands

TrOnchip.state Display on-chip trigger window

Format:	**TrOnchip.state**

Opens the **TrOnchip.state** window.

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.TEnable — Set filter for the trace

Format:	**TrOnchip.TEnable** <*par*> (deprecated)

Refer to the **Break.Set** command to set trace filters.

TrOnchip.TOFF — Switch the sampling to the trace to OFF

Format:	**TrOnchip.TOFF** (deprecated)

Refer to the **Break.Set** command to set trace filters.

TrOnchip.TON — Switch the sampling to the trace to “ON”

Format:	**TrOnchip.TON EXT** | **Break** (deprecated)

Refer to the **Break.Set** command to set trace filters.

TrOnchip.TTrigger — Set a trigger for the trace

Format:	**TrOnchip.TTrigger** <*par*> (deprecated)

Refer to the **Break.Set** command to set a trigger for the trace.

TrOnchip.VarCONVert — Adjust complex breakpoint in on-chip resource

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.

TrOnchip.Address — Define address selector

Format: **TrOnchip.Address Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

The address/range for an address selector can not be defined directly. Set an breakpoint of the type Alpha, Beta or Charly to the address/range.

```
Break.Set 1000 /Alpha           ; set an Alpha breakpoint to 1000
/Onchip                         ; use Alpha breakpoint as address
                                ; selector for the TrOnchip unit
```

TrOnchip.CYcle — Define access type

Format: **TrOnchip.CYcle Read** | **Write** | **eXecute**

Defines on which cycle the program execution stops.

Read	Stop the program execution on a read access.
Write	Stop the program execution on a write access.
eXecute	Stop the program execution on an instruction is executed.

TrOnchip.Data Define data selector

Format: **TrOnchip.Data** [*<range> <value> <bitmask>*]

TrOnchip.NoMatch Define match or nomatch comparison

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

Default: OFF

TrOnchip.TaskID Define task ID comparison

Format: **TrOnchip.TaskID** [*<value> <bitmask>*]

The application must write to the DTIDR (Task ID Register) at address D:0xF0D8. This register is intended to be used by advanced real time operating systems to store the task ID of the active task.

Connectors

JTAG Connector

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

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

Do not connect the “reserved” pins.

Do connect all GND pins for shielding purpose, though they are connected on the debugger. It is recommended to connect also the N/C pin to GND.

At least the signals TMS, VCCS, TDO, GND, TDI, /RESET, /TRST, TCLK are required.

CPUCLOCK is only necessary if it should be used as jtag clock source. It is not used on **AICD**. /BrkIN, /BrkOUT are used for trigger input / output feature.

VCCS is the processor power supply voltage. It is used to detect if target power is on and it is used to supply the output buffers of the debugger (it takes about 2 mA). That means the output voltage of the debugger signals (TMS, TDI, /TRST, TCLK, /BrkIN) depends directly on VCCS. VCCS can be 2.25 … 5.5 V.

/RESET is controlled by an open drain driver. (An external watchdog must be switched off if the In-Circuit Debugger is used.)

VIHmin = 2.0 V, VILmax = 0.8 V for the input pins VCCS, TDO, CPUCLOCK, /BrkOUT.

When having multiple JTAG TAP controllers in the chain, make sure that the C166 is the first device in the chain. It is not possible to have another XC2000, XC16x or TriCore in the same chain.

DAP Connector

Signal	Pin	Pin	Signal
VREF	1	2	DAP1
GND	3	4	DAP0
GND	5	6	USER0
GND	7	8	DAPEN- (TRST-)
GND	9	10	RESET- (PORST-)

Only possible with Whisker Debug Cable and LA-3815 Conv. 16 Pin JTAG to DAP

On the target board the standard connector is a 0.05 inch double row 10 pins micro terminal, which is available from many sources e.g. Samtec FTSH (SMT Mount) series. It is offered as a standard dual row header 1.27 mm x 1.27 mm with 0.4 mm square pins.

Troubleshooting

SYStem.Up Errors

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

- The target has no power.
- ICD-166CBC only: The trap program routine is not available or the **MONBASE** address is not correct or the jump to this function at 20H is missing (see **Monitor**). Another restriction is that a valid program which disables the debugger must start after power on. If one of these conditions are not fulfilled (e.g. if there is no program on the target) it can be helpful to activate the debugger by **SYStem.Mode Prepare** to do some initialization with the debuggers help. See also the PRACTICE script files (*.cmm) in ~~/demo/c166/etc/.
- ICD-166CBC only: Unfortunately it is possible that a faulty user program which starts after power on produces a situation where it is not possible for the debugger to establish communication with the processor. To avoid this situation we recommend to place a endless loop (after disabling the watchdog and settings like stack and chip selects) at the begin of the user application. Then the debugger can establish the communication and you can step through the program to come closer to the faulty program part. It is also useful to have the possibility (jumper on chip select signal) to switch off the program memory on the target. Even if you already have the situation I described above, you can then use **SYStem.Mode Prepare** to establish communication and for example re-program the program memory if it is placed in a flash device.
- External controlled /RESET line:

  The debugger controls the processor reset and use the /RESET line to reset the CPU on most of the **SYStem.Mode** commands. Therefore only not active, open-drain driver may be connected to this /RESET signal.
- There are additional loads or capacities on the JTAG lines or the JTAG cable is elongated.

FAQ

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

Technical Data

Operation Voltage

Adapter	OrderNo	Voltage Range
OCDS Debugger for C166CBC (ICD)	LA-7755	2.5 .. 5.2 V

Adapter	OrderNo	Voltage Range
OCDS Debugger for XC2000/C166S V2 (ICD)	LA-7759	2.5 .. 5.2 V