Case Study: Debugging Traps on TriCore™ AURIX™

2025-09-15 - Comments (0) - Debug
Note

Different processor architectures and application domains use the terms *“trap”* and *“interrupts”* with varying meanings. On some architectures, a *“trap”* refers to a system call, while in other contexts it may describe a fault condition.

This article adopts the Infineon-specific terminology for *“traps.”*

Background Information: Traps in TriCore™ AURIX™

Traps in the TriCore™ AURIX™ architecture are exceptional events that can be caused by conditions such as:

- Instruction Exception
- Illegal Memory accesses, e.g.:
    - Result from attempts to access non mapped memory regions or peripherals not yet initialized
    - Bus transactions that lead to ECC faults
- Non-Maskable Interrupt (NMI)

Traps are always active; they cannot be masked or disabled by software. The trap vector handler is stored in code memory, and the **BTV (Base Trap Vector)** register specifies the base address of the trap vector table. The contents of this register can be inspected in the `Register.view` window.

The TriCore architecture defines **eight general trap classes**, each with its own dedicated handler. The trap class determines the offset of the corresponding trap handler in program memory, relative to the base address specified in the BTV register.



Each trap is assigned a unique **Trap Identification Number (TIN)**. When a trap occurs, the TIN is automatically stored in register **D15**, allowing the Trap Service Routine (TSR) to identify the trap and take appropriate action in the application software.

Trap Types:

- **Source Classification:**
    - **Hardware traps**: Generated in response to exception conditions detected by the hardware (e.g., illegal instruction or memory protection traps).
    - **Software traps**: Intentionally generated by executing a system call or an assertion instruction.
- **Timing Classification:**
    - **Synchronous traps**: Occur during the execution (or attempted execution) of a specific instruction. The causing instruction is known precisely, and the trap is serviced immediately before execution continues.
    - **Asynchronous traps**: Triggered by hardware conditions detected externally and signaled back to the core. The exact instruction that caused the condition may not be identifiable since the CPU stops at a random location and the displayed instruction is thus not related to the trap.

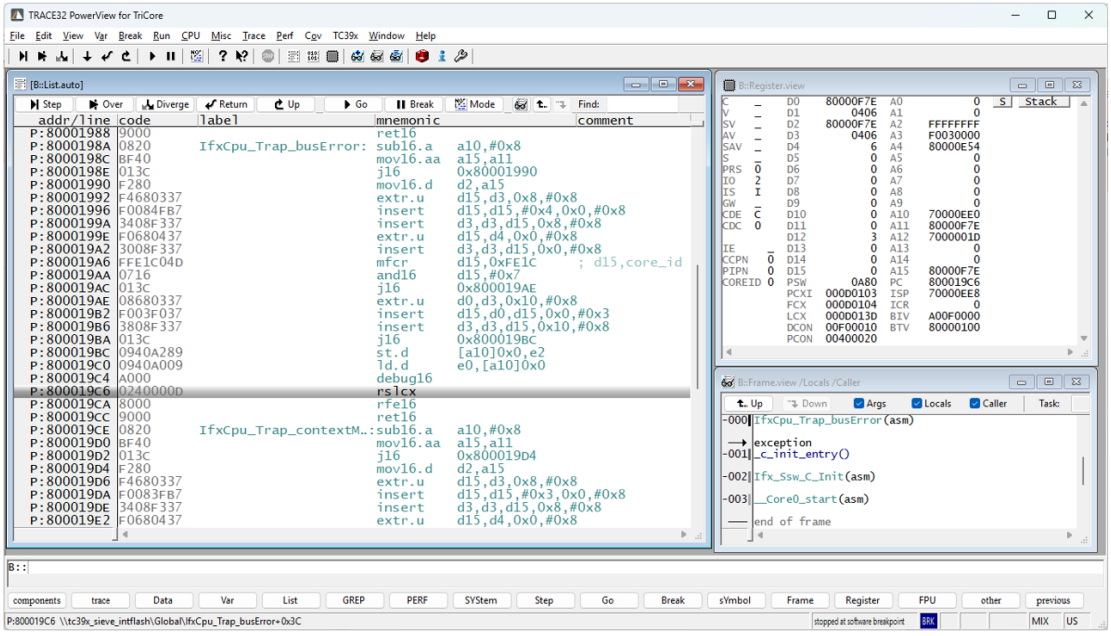
For more information, refer to the *Core Architecture Manual*.

Case Study - Trap Debugging

In this case study, we demonstrate how to debug a trap using **TRACE32 PowerView** on a **TriBoard equipped with a TC397XE**.

Initial Observations

We start with a trap condition where the symbolic information shows that the application is stopped at a **Bus Error Trap**.

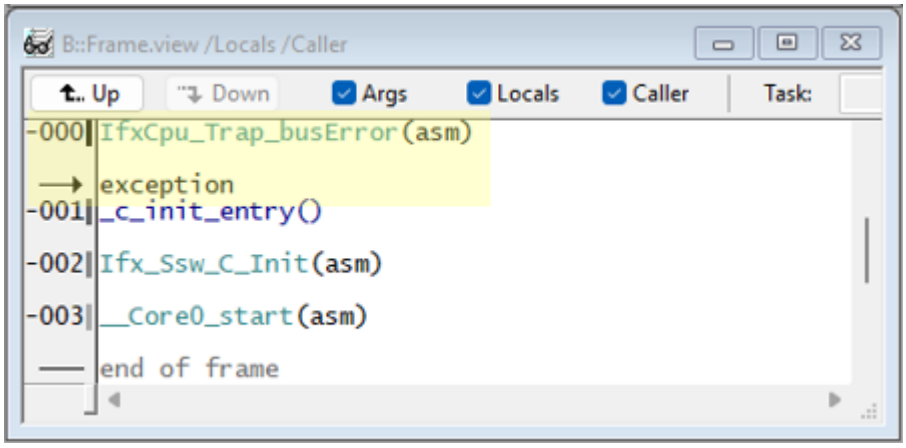


The TRACE32 PowerView status bar indicates that the target is *"stopped at a software breakpoint."* The **`List` window** confirms that the program counter points to an instruction immediately after `debug16`.

Note

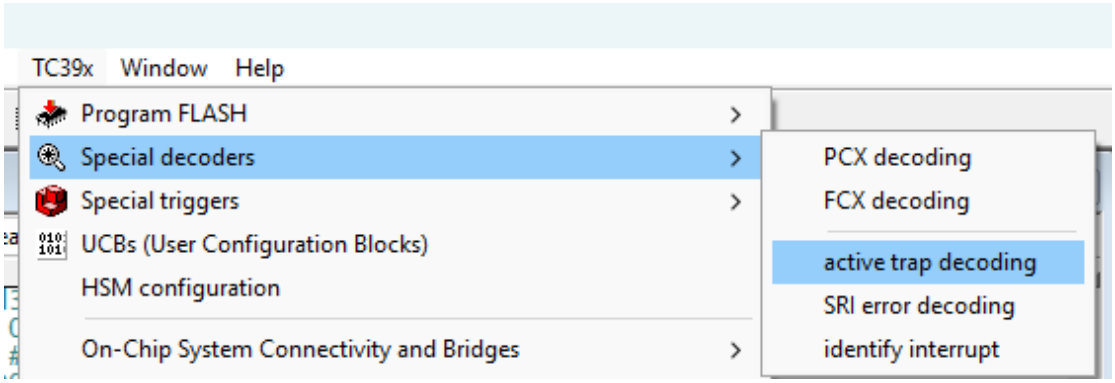
Debug instructions are typically inserted into error-handling routines. When a debugger is connected, they halt the core for inspection. Without a debugger, they behave as NOP instructions.

Examining the **`Frame.view`** window reveals an exception followed by a call to a trap-handling function.



Using TRACE32 Trap Decoding

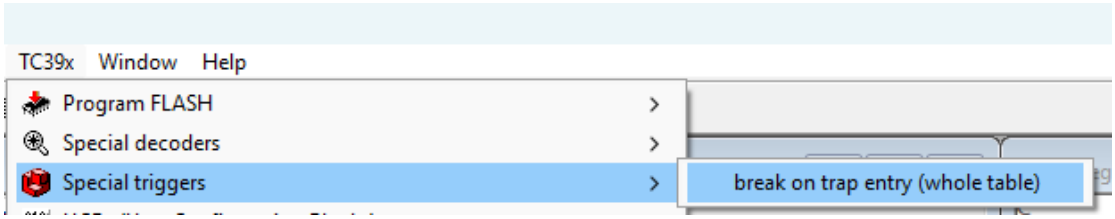
TRACE32 PowerView provides a menu for identifying the reason for a trap:
TC39x > Special decoders > active trap decoding.



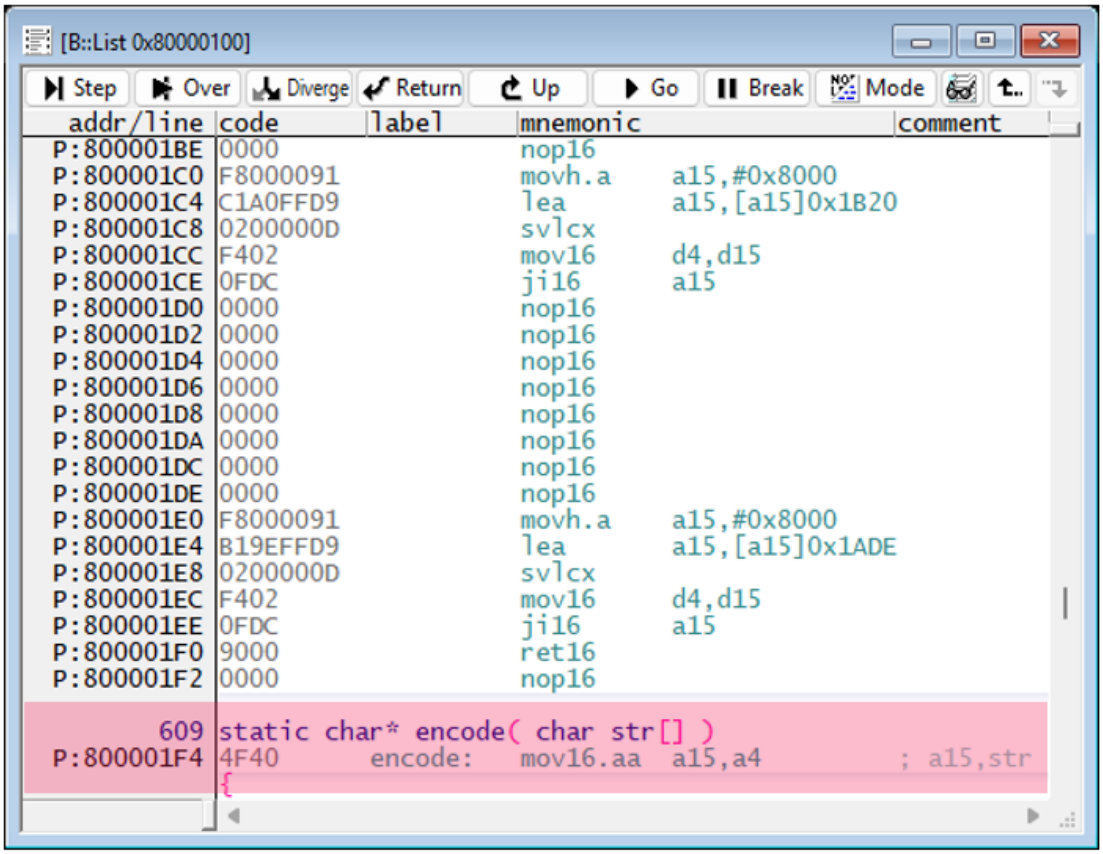
If the core is halted *inside* the Trap Service Routine (not at the trap vector itself), the AREA window may display *"No exception detected."* This is expected because the core has already moved past the trap vector.

To decode the trap reason, the target must be halted **at the trap vector itself**. TRACE32 PowerView provides a menu for setting a program breakpoint across the trap vector range:

TC39x > Special triggers > break on trap entry (whole table).

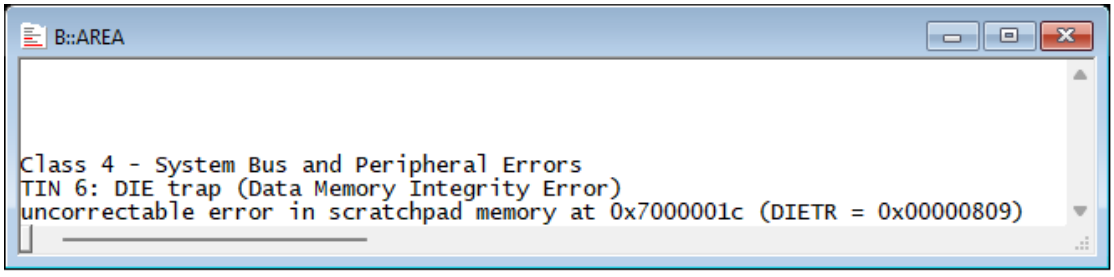


Care must be taken, as compilers may insert regular application code (e.g., the `encode` function) into unused bytes of the trap vector.



In this example, the program breakpoint must be adapted accordingly:
`Break.Set IfxCpu_Trap_vectorTable0++0xF3 /Program /Onchip`

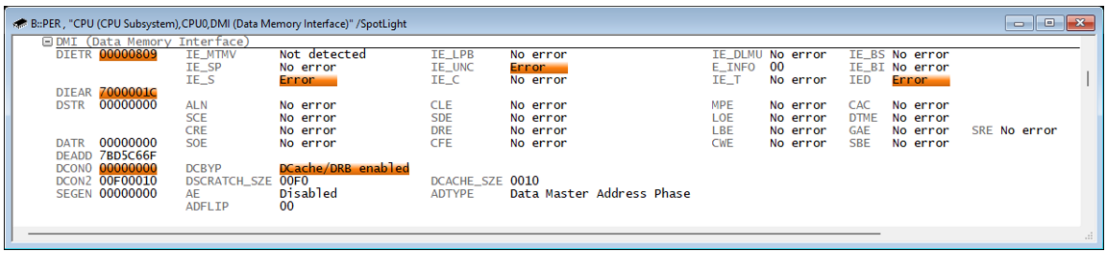
When the core is halted correctly within the trap vector range, the "**active trap decoding**" menu will show details such as the **trap class, TIN,** and other trap-specific information.



Accessing Additional Trap Details

While the *Core Architecture Manual* describes general trap mechanisms, implementation details are documented in the *Family User's Manual*. For example TC3xx User's Manual, states that more detailed information about DIE traps are to be extracted from the **DIEAR** (Data Integrity Error Address Register) and **DIETR** (Data Integrity Error Trap Register).

Using the peripheral view we can get more insights about the error e.g. the address of the memory access causing the trap!

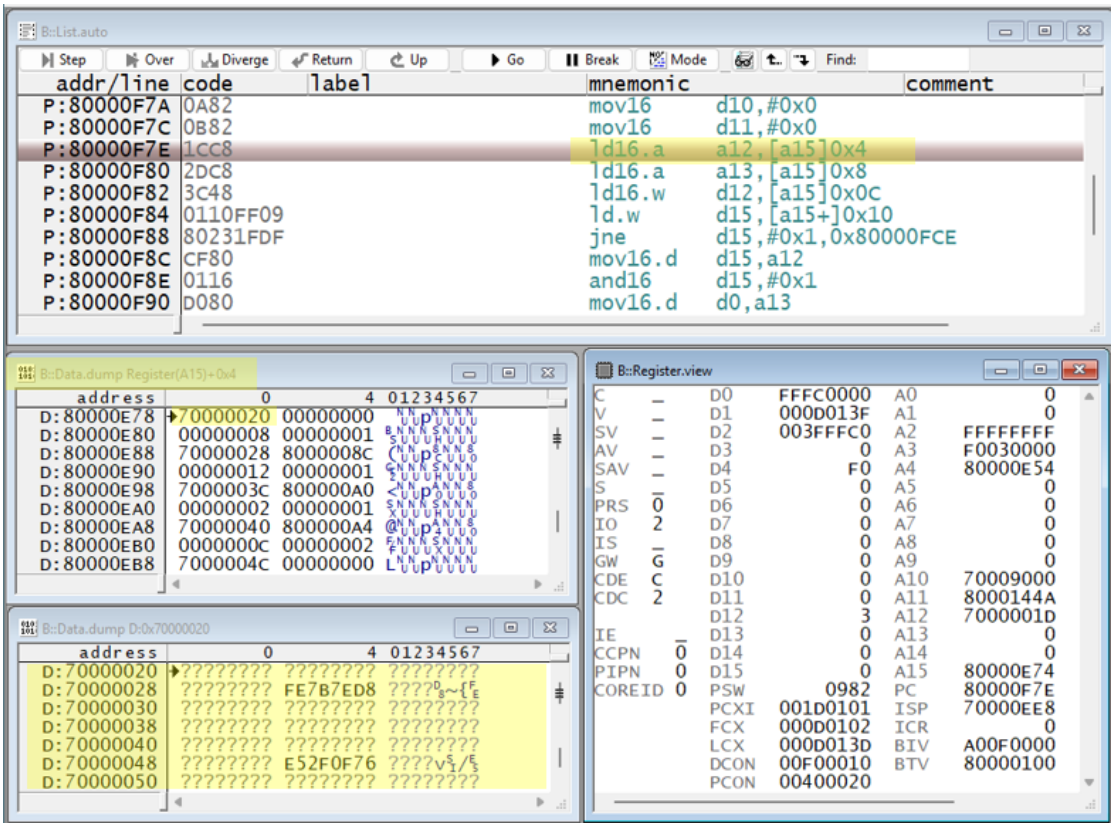


Using the Peripheral view in TRACE32, we can observe:

- DIETR : CPUx Data Integrity Error Trap Register
    - IED: Data integrity error condition detected
    - IE_S: Integrity Error - Scratchpad Memory
    - Dual Bit Error Detected
- DIEAR: Data Integrity Error Address Register
    - The access triggering the trap is actually **0x7000001C**

For **synchronous** traps, the **Stack Frame window** can also provide valuable insights such as the exact instruction that triggered the trap.
However, since a DIE trap is **asynchronous**, the direct link to the instruction that triggered it is lost. The trap may occur several instructions after the offending instruction executed. Using `Frame.Up` in such cases may lead to misleading conclusions.

In the following screenshot, the stack unwinding shows that the DIE trap occurred while the CPU was trying to read from the address **0x70000020**. The Data Integrity Error Address Register indicates that the error was triggered by a prior read access to a different address (**0x7000001C).**

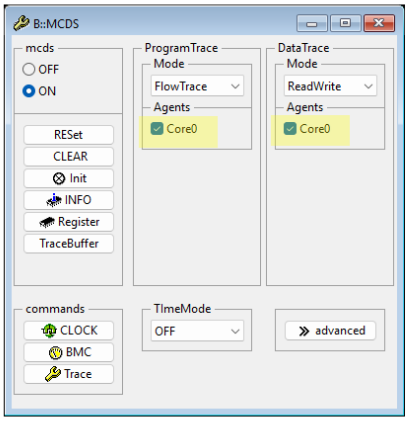


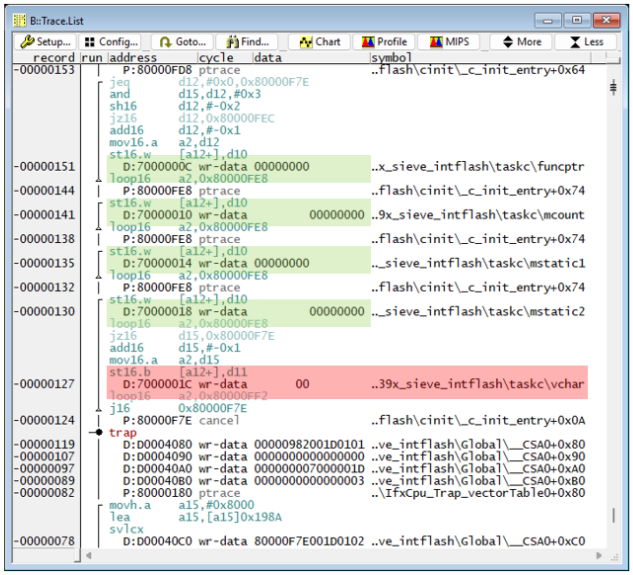
Using Trace

Another powerful debugging technique is to use **MCDS trace**.

Since this trap was caused by a memory access, tracing both program flow and data accesses gives a clearer picture.

In the **Trace.List** window, a **trap marker** appears shortly after a byte write access to the DSPR (Data Scratch-Pad RAM) of TriCore0.





Root Cause & Solution

The **DSPR (Data Scratchpad RAM)** is **ECC protected** and must be initialized before any read operation. Initialization can be performed either by software, or automatically by hardware (via `UCB_DFLASH.PROCONRAM`).

For **half-word or larger write operations**, ECC bits are pre-calculated and written alongside the data. However, for **byte write operations**, the transaction is internally transformed into a half-word read–modify–write sequence in the DMI module. This caused the detection of **uncorrectable memory integrity errors**.

Solution:
The issue is resolved by enabling **RAM initialization** through the Startup Software (SSW – the boot ROM) in `UCB_DFLASH`.

Note

If you are interested in the demo files used in this case study, please contact Lauterbach Support.