

CPU_Trapping_Recognition_1

for KIT_AURIX_TC334_LK

TRAP error recognition and reaction

AURIX™ TC3xx Microcontroller Training
V1.0.0



[Please read the Important Notice and Warnings at the end of this document](#)

Scope of work

This example shows how to identify the root cause of a trap.

The tutorial describes what types of traps are supported by the AURIX™ microcontroller, their root causes and how to identify them. AURIX™ architecture supports different types of traps. Three different traps can be provoked with this example and the tutorial guides the user through the needed steps to observe the root cause of each trap.

Introduction

- › A trap occurs as a result of an event such as a Non-Maskable Interrupt (NMI), an instruction exception, a memory management exception or an illegal access. Traps are always active; they cannot be disabled by software

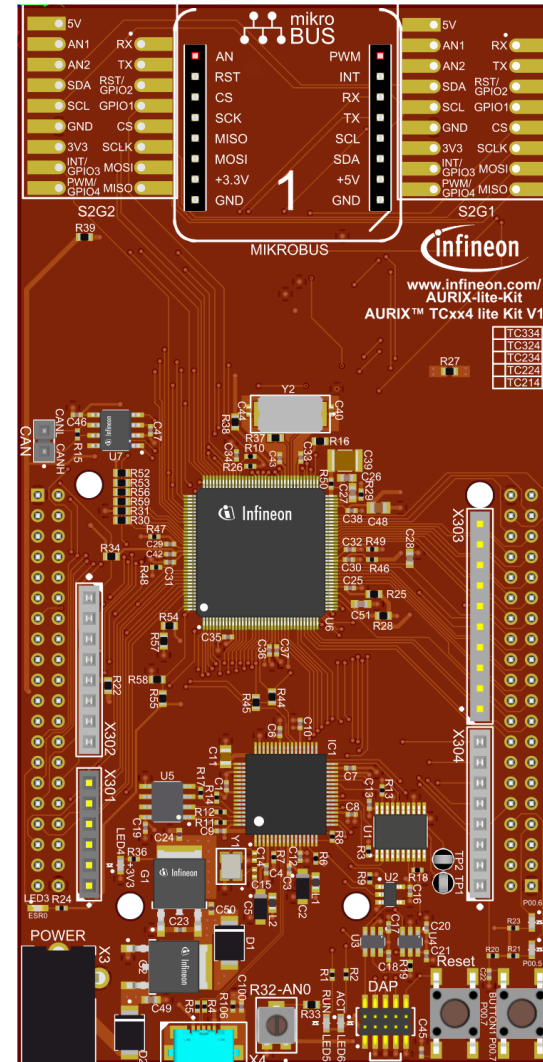
- › The TriCore™ architecture specifies eight general classes for traps. Each trap class has its own trap handler. Within each class, specific traps are distinguished by a Trap Identification Number (TIN)

- › Traps can be further classified as synchronous or asynchronous, and as hardware or software generated

- › Three different combinations of trap types are supported:
 - Synchronous and hardware generated
 - Asynchronous and hardware generated
 - Synchronous and software generated

Hardware setup

This code example has been developed for the board KIT_A2G_TC334_LITE.



Implementation

Supported traps

The following table provides an overview about all supported traps and their types:

Table 8 Supported Traps

TIN	Name	Synch. / Asynch.	HW / SW	Definition
Class 0 – MMU				
0	VAF	Synch.	HW	Virtual Address Fill.
1	VAP	Synch.	HW	Virtual Address Protection.
Class 1 – Internal Protection Traps				
1	PRIV	Synch.	HW	Privileged Instruction.
2	MPR	Synch.	HW	Memory Protection Read.
3	MPW	Synch.	HW	Memory Protection Write.
4	MPX	Synch.	HW	Memory Protection Execution.
5	MPP	Synch.	HW	Memory Protection Peripheral Access.
6	MPN	Synch.	HW	Memory Protection Null Address.
7	GRWP	Synch.	HW	Global Register Write Protection.
Class 2 – Instruction Errors				
1	IOPC	Synch.	HW	Illegal Opcode.
2	UOPC	Synch.	HW	Unimplemented Opcode.
3	OPD	Synch.	HW	Invalid Operand specification.
4	ALN	Synch.	HW	Data Address Alignment.
5	MEM	Synch.	HW	Invalid Local Memory Address.
Class 3 – Context Management				
1	FCD	Synch.	HW	Free Context List Depletion (FCX = LCX).
2	CDO	Synch.	HW	Call Depth Overflow.
3	CDU	Synch.	HW	Call Depth Underflow.

Table 8 Supported Traps (cont'd)

TIN	Name	Synch. / Asynch.	HW / SW	Definition
4	FCU	Synch.	HW	Free Context List Underflow (FCX = 0).
5	CSU	Synch.	HW	Call Stack Underflow (PCX = 0).
6	CTYP	Synch.	HW	Context Type (PCXI.UL wrong).
7	NEST	Synch.	HW	Nesting Error: RFE with non-zero call depth.
Class 4 – System Bus and Peripheral Errors				
1	PSE	Synch.	HW	Program Fetch Synchronous Error.
2	DSE	Synch.	HW	Data Access Synchronous Error.
3	DAE	Asynch.	HW	Data Access Asynchronous Error.
4	CAE	Asynch.	HW	Coprocessor Trap Asynchronous Error.
5	PIE	Synch.	HW	Program Memory Integrity Error.
6	DIE	Asynch.	HW	Data Memory Integrity Error.
7	TAE	Asynch.	HW	Temporal Asynchronous Error
Class 5 – Assertion Traps				
1	OVF	Synch.	SW	Arithmetic Overflow.
2	SOVF	Synch.	SW	Sticky Arithmetic Overflow.
Class 6 – System Call¹⁾				
	SYS	Synch.	SW	System Call.
Class 7 – Non-Maskable Interrupt				
0	NMI	Asynch.	HW	Non-Maskable Interrupt.

Please refer to the TriCore™ TC1.6.2 core architecture manual and the AURIX™ TC3xx User's Manual for detailed information about each trap.

Implementation

Trap types

- › **Synchronous traps:**
 - Synchronous traps are associated with the execution or attempted execution of specific instructions or with attempts to access a virtual address that requires the intervention of the memory-management system
 - The trap is triggered and serviced immediately
- › **Asynchronous traps:**
 - Since asynchronous traps are associated with hardware conditions, they are similar to interrupts
 - They are routed via the trap vector
 - Some asynchronous traps are triggered indirectly from instructions, that have been previously executed, but the direct association with the instructions causing the trap is lost
- › **Hardware traps:**
 - Hardware traps are generated in response to exception conditions detected by the hardware
 - In most, but not all cases, the exception conditions are associated with the attempted execution of a particular instruction
- › **Software traps:**
 - Software traps are generated as an intentional result of executing a system call or an assertion instruction

Implementation

Trap handling

- › When a trap occurs, a trap identifier is generated by hardware. The trap identifier has two components that can be used to determine more information about the trap and why it was caused (refer to slide [Supported traps](#)):
 - The Trap Class Number (TCN)
 - The Trap Identification Number (TIN)

- › In most cases, the debugger will stop the code execution within one of the trap handlers (implemented in the iLLD header *IfxCpu_Trap.c*)

- › An instance of the structure *IfxCpu_Trap* is declared within each trap handler. When a trap occurs, the instance provides four information fields about the trap:
 - **tCpu**: Which CPU caused the trap
 - **tClass**: TCN, Class of the trap (refer to slide [Supported traps](#))
 - **tId**: TIN, Id of the trap (refer to slide [Supported traps](#))
 - **tAddr**: Return Address (RA) (refer to the [next slide](#))

Implementation

Return Address

- › The Return Address (RA) might help to locate the specific line of code which caused the trap
- › The return address, which is stored in the instance of the ***IfxCpu_Trap*** structure, is read from the return address register A[11]
- › Depending on the **trap type**, the return address is different:
 - For most of the **synchronous** traps, the return address is the 32-bit Program Counter (PC) of the instruction that caused the trap. (The PC holds the address of the instruction which is currently running when the core is halted.)
 - On a **System Call (SYS)** trap, triggered by the SYSCALL instruction, the return address points to the instruction immediately following SYSCALL
 - A **Free Context List Depletion (FCD)** trap is generated after a context save operation that causes the free context list becoming “almost empty”.
The responsible for the FCD trap can be a hardware interrupt or a trap handler. The operation responsible for the context save normally is completed before the FCD trap is executed. Because of this, the return address of the FCD trap is the first instruction of the trap/interrupt/called routine or the instruction following a Save Lower Context (SVLCX) or Begin Interrupt Service Routine (BISR) instruction
 - For an **asynchronous** trap, the return address is the address of the instruction that would have been executed next, if the asynchronous trap had not been triggered

Implementation

Additional debug information

- › The bit field `ERROR_ADDRESS` of the **Data Error Address Register (DEADD)** contains the trap address information for the data memory. The content of the DEADD register is valid if the **Data Synchronous Trap Register (DSTR)** or the **Data Asynchronous Trap Register (DATR)** register are non-zero (depending on the trap type). The bit fields in the DSTR and the DATR registers can provide additional information about the trap (refer to the AURIX™ TC3xx User's Manual)
 - These information are valid in case traps such as:
 - **Data Address Alignment (ALN)**
 - **Data Access Synchronous Error (DSE)**
 - **Data Access Asynchronous Error (DAE)**
 - **Invalid Local Memory Address (MEM)**
 - **Memory Protection Write (MPW)**
 - **Memory Protection Read (MPR)**
 - **Memory Protection Peripheral Access (MPP)**
 - **Memory Protection Null Address (MPN)**

Implementation

Additional debug information

- › The **Program Memory Interface Synchronous Trap Register (PSTR)** contains synchronous trap information for the program memory system. The register is updated with trap information for **Program Fetch Synchronous Error traps (PSE)**
- › The **Program (or Data) Integrity Error Address Register (PIEAR / DIEAR)** and the **Program (or Data) Integrity Error Trap Register (PIETR / DIETR)** can be interrogated to determine the source of the **Program (or Data) Memory Integrity Error (PIE / DIE)** more precisely

Implementation

Trap provocation

- › Three different combinations of trap types can be provoked in this example:
 - Synchronous Hardware trap
 - Asynchronous Hardware trap
 - Synchronous Software trap

- › The trap provocation is implemented in the function *run_trap_provocation()* and can be started by setting one of the three *g_provokeXYTrap* (**X = Synchronous / Asynchronous; Y = Hardware / Software**) variables

- › The implemented code for the first two traps is based on the MTU_MBIST_1 and SMU_IR_Alarm_1 examples. For further information on the code, please refer to the specific tutorials

- › The third trap is provoked by using two instructions: *__mtcr()* (Move To Core Register) and *trapv* (assembly code). For further information on these instructions, please refer to the TriCore™ TC1.6.2 core architecture manual - Instruction set manual

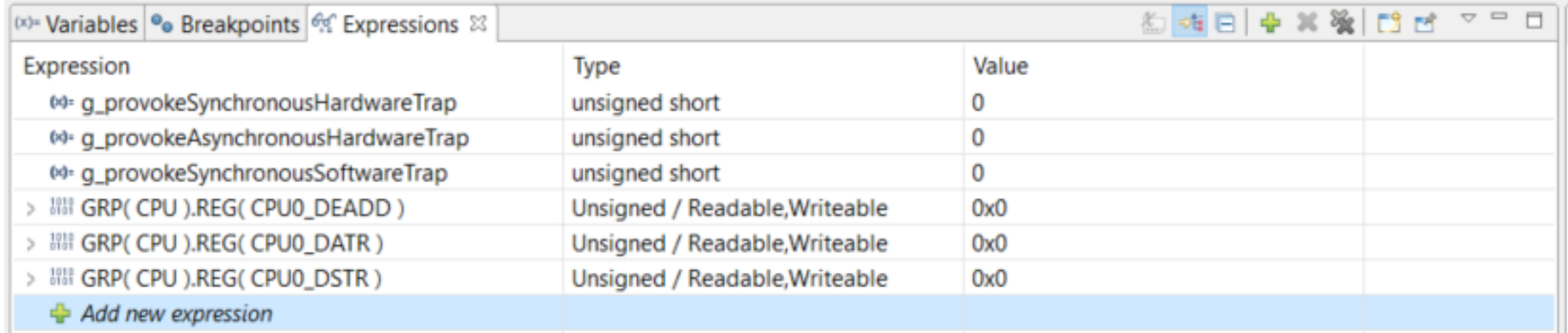
Note: *__mtcr()* is an intrinsic function of the Tasking compiler, which moves contents of a data register to the addressed Core Special Function Register (CSFR). *__mtcr()* performs a Move to Core Register (**MTCR**) TriCore™ instruction and is followed by an **ISYNC** instruction.

- › For a better understanding of the trap behavior, the required code instructions used to avoid the cause of each trap, are implemented and can be activated by setting the **AVOID_PROVOCATION** macro to true

Run and Test

After code compilation and flashing the device, perform the following steps:

- › Add the three variables “***g_provokeSynchronousHardwareTrap***”, “***g_provokeAsynchronousHardwareTrap***” and “***g_provokeSynchronousSoftwareTrap***” in the Expressions window of the debugger
- › Add the three registers DEADD, DATR and DSTR in the Expressions window of the debugger



Expression	Type	Value
<code>g_provokeSynchronousHardwareTrap</code>	unsigned short	0
<code>g_provokeAsynchronousHardwareTrap</code>	unsigned short	0
<code>g_provokeSynchronousSoftwareTrap</code>	unsigned short	0
> <code>GRP(CPU).REG(CPU0_DEADD)</code>	Unsigned / Readable,Writeable	0x0
> <code>GRP(CPU).REG(CPU0_DATR)</code>	Unsigned / Readable,Writeable	0x0
> <code>GRP(CPU).REG(CPU0_DSTR)</code>	Unsigned / Readable,Writeable	0x0
+ Add new expression		

Run and Test

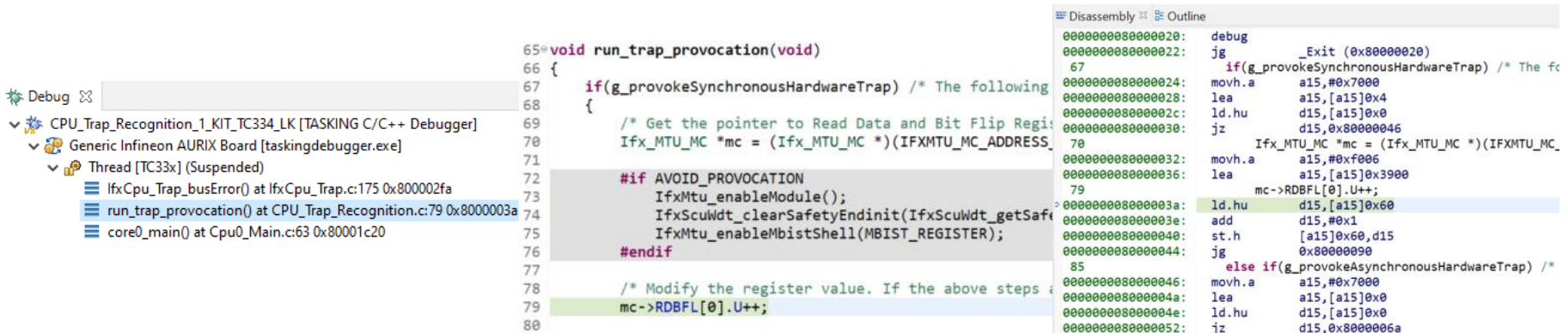
1.1 Synchronous hardware trap

- › Provoke the synchronous hardware trap by setting the value of “***g_provokeSynchronousHardwareTrap***” in the “Expressions” window to “1”
- › Press the “Resume” button to start the program
- › Observe the following information:
 - The debugger stopped in the ***IfxCpu_Trap_busError()*** function (***IfxCpu_Trap.c***)
 - The “Variables” window of the debugger displays the “***trapWatch***” structure and the value of its parameters
 - The trap is provoked by CPU0, it is a trap of class 4, the trap id is 2 and the Return Address (RA) is 0x8000003a (2147483706₁₀)
 - It is a Data Access Synchronous Error ([Trap table](#), class 4 and tin 2)

Run and Test

1.2 Synchronous hardware trap

- › Observe the following information:
 - The call stack in the “Debug” window displays the function which was called before the trap occurred (in this case the function ***run_trap_provocation()***, the address displayed behind this function equals the Return Address (RA))
 - By clicking on this function, the debugger jumps to the specific code line in the ***CPU_Trapping_Recognition.c*** file and to the corresponding assembly line in the “Disassembly” window. The address of the assembly line equals the Return Address (RA)



The screenshot displays a debugger interface with three main windows:

- Debug Window (Left):** Shows the call stack for the thread [TC33x] (Suspended). The current function is `run_trap_provocation()` at `CPU_Trapping_Recognition.c:79 0x8000003a`.
- Source Code Window (Middle):** Shows the C code for `void run_trap_provocation(void)`. The function starts with an `if` statement checking `g_provokeSynchronousHardwareTrap`. Inside the `if` block, it gets a pointer to the Read Data and Bit Flip Register (`Ifx_MTU_MC *mc`) and then calls `mc->RDBFL[0].U++;` at line 79.
- Disassembly Window (Right):** Shows the assembly code corresponding to the source code. The instruction `mc->RDBFL[0].U++;` is highlighted in green, corresponding to the assembly instruction `ld.hu d15,[a15]0x60` at address `000000008000003a`.

Run and Test

1.3 Synchronous hardware trap

- › Observe the following additional information:
 - The **LBE** bit field in the **DSTR** register is set (Load Bus Error - Data load from bus causing error, refer to AURIX™ TC3xx User's Manual)
 - The **DEADD** register displays the address 0xf0063960, which is the address of the modified register that caused the trap
 - By running a file search (Search -> File) for the address, the search finds the specific **RDBFLO** register which equals the modified **MBIST** DMA register

Expression	Type	Value
g_provokeSynchronousHardwareTrap	unsigned short	1
g_provokeAsynchronousHardwareTrap	unsigned short	0
g_provokeSynchronousSoftwareTrap	unsigned short	0
GRP(CPU).REG(CPU0_DEADD)	Unsigned / Readable,Writeable	0xf0063960
GRP(CPU).REG(CPU0_DATR)	Unsigned / Readable,Writeable	0x0
GRP(CPU).REG(CPU0_DSTR)	Unsigned / Readable,Writeable	0x4
+ Add new expression		

```

CPU_Trap_Recognition_1_KIT_TC334_LK
├── Libraries
│   ├── Infra
│   │   ├── Sfr
│   │   │   ├── TC33A
│   │   │   │   ├── _Reg
│   │   │   │   │   └── IfxMtu_reg.h
│   │   │   │   │       └── 10.867: #define MTU_MC41_RDBFLO /*lint --e(923, 9078)*/ (*(volatile Ifx_MTU_MC_RDBFL*)0xf0063960u)

```

Run and Test

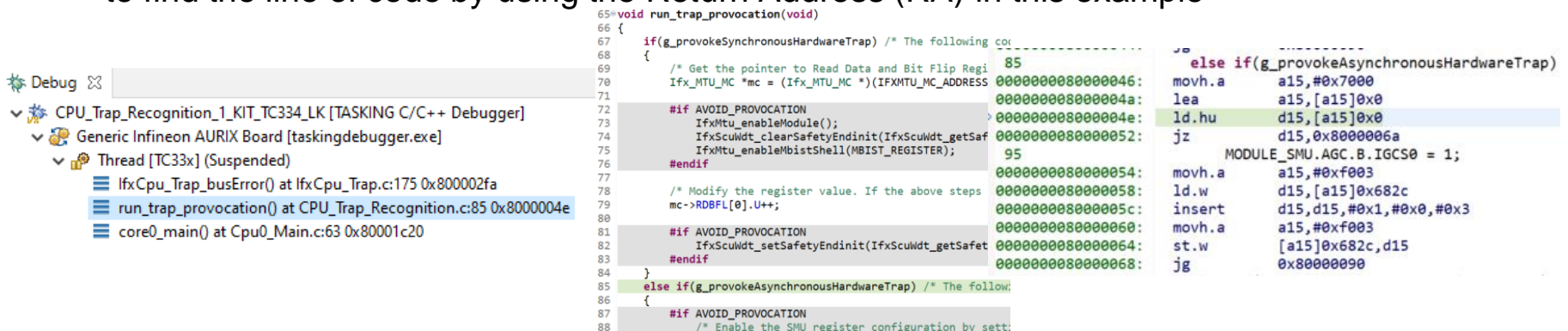
2.1 Asynchronous hardware trap

- › Restart the program by pressing the “Restart” button in the debugger
- › Provoke the asynchronous hardware trap by setting the value of “***g_provokeAsynchronousHardwareTrap***” in the “Expressions” window to “1”
- › Press the “Resume” button to start the program
- › Observe the following information:
 - The debugger stopped in the ***IfxCpu_Trap_busError()*** function (***IfxCpu_Trap.c***)
 - The “Variables” window of the debugger displays the “***trapWatch***” structure and the values of its parameters
 - The trap is provoked by CPU0, it is a trap of class 4, the trap id is 3 and the Return Address (RA) is 0x8000004e (2147483726₁₀)
 - It is a Data Access Asynchronous Error ([Trap table](#), class 4 and tin 3)

Run and Test

2.2 Asynchronous hardware trap

- › Observe the following information:
 - The call stack in the “Debug” window displays the function which was called before the trap occurred (in this case the function ***run_trap_provocation()***, the address displayed behind this function equals the Return Address (RA))
 - By clicking on this function, the debugger jumps to the specific code line in the ***CPU_Trapping_Recognition.c*** file and to the corresponding assembly line in the “Disassembly” window. The address of the assembly line equals the return address
 - Because it is an asynchronous trap, the specific code line is **not** pointing to the line which is causing the trap. It is the code line of the instruction that would have been executed next, if the asynchronous trap had not been triggered
 - Since there is no other instruction within the function ***run_trap_provocation()***, it is impossible to find the line of code by using the Return Address (RA) in this example



```

65=void run_trap_provocation(void)
66 {
67     if(g_provokeSynchronousHardwareTrap) /* The following code
68     {
69         /* Get the pointer to Read Data and Bit Flip Register
70         Ifx_MTU_MC *mc = (Ifx_MTU_MC *) (IFXMTU_MC_ADDRESS
71         0000000080000046:
72         #if AVOID_PROVOCATION
73         IfxMtu_enableModule();
74         IfxScuWdt_clearSafetyEndinit(IfxScuWdt_getSafetyEndinit);
75         IfxMtu_enableMbstShell(MBIST_REGISTER);
76         #endif
77         000000008000004a:
78         /* Modify the register value. If the above steps
79         mc->RDBFL[0].U++;
80         000000008000004e:
81         #if AVOID_PROVOCATION
82         IfxScuWdt_setSafetyEndinit(IfxScuWdt_getSafetyEndinit);
83         #endif
84         0000000080000052:
85     }
86     else if(g_provokeAsynchronousHardwareTrap) /* The following
87     {
88         #if AVOID_PROVOCATION
89         /* Enable the SMU register configuration by setting
90         0000000080000054:
91         95
92         MODULE_SMU.AGC.B.IGCS0 = 1;
93         0000000080000058:
94         movh.a    a15,#0xf003
95         000000008000005c:
96         ld.w     d15,[a15]0x682c
97         0000000080000060:
98         insert  d15,d15,#0x1,#0x0,#0x3
99         0000000080000064:
100        movh.a    a15,#0xf003
101        0000000080000068:
102        st.w     [a15]0x682c,d15
103        000000008000006c:
104        jg      0x80000090
105    }
106    }
107 }
  
```

Run and Test

2.3 Asynchronous hardware trap

- › Due to the fact that the Return Address (RA) cannot be used, the following information might help to locate the cause of the trap:
 - The **SBE** bit field in the **DATR** register is set (Store Bus Error - Data store to bus causing error, refer to AURIX™ TC3xx User's Manual)
 - The **DEADD** register displays the address 0xf003682c, which is the address of the modified register that caused the trap
 - By running a file search (Search -> File) for the address, the search finds the specific **SMU_AGC** register which equals the modified register. The name of the modified register helps to find the code line which is causing the trap (By using another search for “AGC”)

Expression	Type	Value
g_provokeSynchronousHardwareTrap	unsigned sh...	0
g_provokeAsynchronousHardwareTrap	unsigned sh...	1
g_provokeSynchronousSoftwareTrap	unsigned sh...	0
GRP(CPU).REG(CPU0_DEADD)	Unsigned / ...	0xf003682c
GRP(CPU).REG(CPU0_DATR)	Unsigned / ...	0x8
SBE	Readable,W...	0x1
CWE	Readable,W...	0x0
CFE	Readable,W...	0x0
SOE	Readable,W...	0x0
GRP(CPU).REG(CPU0_DSTR)	Unsigned / ...	0x0
+ Add new expression		

```

CPU_Trap_Recognition_1_KIT_TC334_LK
├── Libraries
│   └── Infra
│       └── Sfr
│           └── TC33A
│               └── _Reg
│                   └── lfxSmu_reg.h
│                       91: #define SMU_AGC /*lint --e(923, 9078)*/ (*(volatile lfx_SMU_AGC*)0xf003682Cu)
CPU_Trap_Recognition_1_KIT_TC334_LK
├── Libraries
└── CPU_Trap_Recognition.c
    95: MODULE_SMU.AGC.B.IGCS0 = 1;
  
```

Run and Test

3.1 Synchronous software trap

- › Restart the program by pressing the “Restart” button in the debugger
- › Provoke the synchronous software trap by setting the value of “***g_provokeSynchronousSoftwareTrap***” in the “Expressions” window to “1”
- › Press the “Resume” button to start the program
- › Observe the following information:
 - The debugger stopped in the ***IfxCpu_Trap_assertion()*** function (***IfxCpu_Trap.c***)
 - The “Variables” window of the debugger displays the “***trapWatch***” structure and the value of its parameters
 - The trap is provoked by CPU0, it is a trap of class 5, the trap id is 1 and the Return Address (RA) is 0x8000008a (2147483786₁₀)
 - It is an Arithmetic Overflow Error ([Trap table](#), class 5 and tin 1)

Run and Test

3.2 Synchronous software trap

- › Observe the following information:
 - The call stack in the “Debug” window displays the function which was called before the trap occurred (in this case the function ***run_trap_provocation()***, the address displayed behind this function equals the Return Address (RA))
 - By clicking on this function, the debugger jumps to the specific code line in the ***CPU_Trap_Recognition.c*** file and to the corresponding assembly line in the “Disassembly” window. The address of the assembly line equals the Return Address (RA)

Debug ⓘ

- ✓ CPU_Trap_Recognition_1_KIT_TC334_LK [TASKING C/C++ Debugger]
 - ✓ Generic Infineon AURIX Board [taskingdebugger.exe]
 - ✓ Thread [TC33x] (Suspended)
 - ▣ lfxCpu_Trap_assertion() at lfxCpu_Trap.c:186 0x800002b6
 - ▣ **run_trap_provocation() at CPU_Trap_Recognition.c:121 0x8000008a**
 - ▣ core0_main() at Cpu0_Main.c:63 0x80001c20

104	<pre> else if(g_provokeSynchronousSoftwareTrap) { Ifx_CPU_PSW psw; /* Variable of the type P /* Get the content of the Program Status Word reg psw.U = __mfcr(CPU_PSW); psw.B.USB = 0x40; /* Set the overflow bit i #if AVOID_PROVOCATION psw.B.USB = 0x0; /* Reset the overflow bit #endif __mctr(CPU_PSW, psw.U); /* Write the modified reg /* TRAPV instruction (trap on overflow) call, ass * If the overflow bit is set, an Arithmetic Over */ __asm("trapv"); } else { /* Do nothing */ } } </pre>	104	<pre> else if(g_provokeSynchronousSoftwareTrap) { movh.a a15,#0x7000 lea a15,[a15]0x8 ld.hu d15,[a15]0x0 jz d15,0x80000090 psw.U = __mfcr(CPU_PSW); mfcr d0,#0xfe04 psw.B.USB = 0x40; /* Set the overflow bit mov d15,#0x40 insert d15,d0,d15,#0x18,#0x8 __mctr(CPU_PSW, psw.U); /* Write the modified r mctr #0xfe04,d15 isync __asm("trapv"); trapv psw.U = __mfcr(CPU_PSW); jg 0x80000090 } ret Ifx__imaskldmst(event, 1, __mfcr(CPU_CORE_ID), 1); mfcr d15,#0xfe1c imask d0/d1,#0x1,d15,#0x1 ldmst [a4]0x0,d0/d1 } </pre>
-----	---	-----	--

References



- › AURIX™ Development Studio is available online:
- › <https://www.infineon.com/aurixdevelopmentstudio>
- › Use the „*Import...*“ function to get access to more code examples.



- › More code examples can be found on the GIT repository:
- › https://github.com/Infineon/AURIX_code_examples



- › For additional trainings, visit our webpage:
- › <https://www.infineon.com/aurix-expert-training>



- › For questions and support, use the AURIX™ Forum:
- › <https://www.infineonforums.com/forums/13-Aurix-Forum>

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-12

Published by

Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

CPU_Trap_Recognition_1
_KIT_TC334_LK

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.