Detecting sporadic errors using EM trace

AN317, Autumn 2018, V 1.0

feedback@keil.com

Abstract

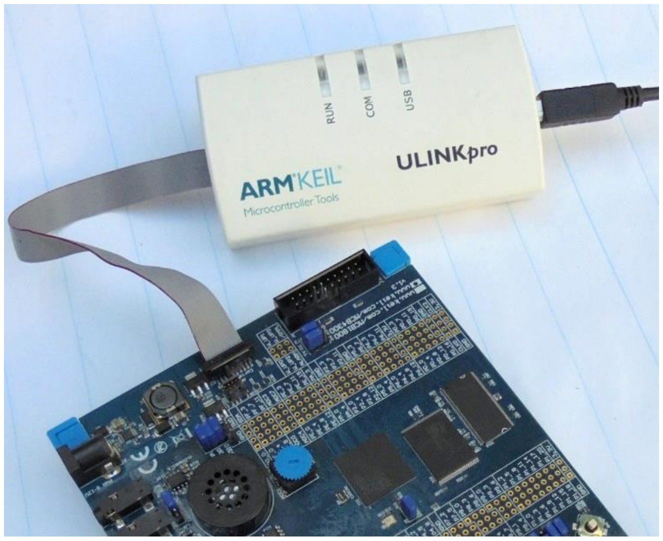
This application note explains how ULINK*pro* is used to identify sporadic errors that where caused by misconfigured hardware. It shows how the advanced debug features of the Arm Cortex-M architecture and the trace adapter helped to find the root cause of the problem that was not related to the application software (in this case a network stack) but caused by a wrong timing configuration of a Flash accelerator.

Contents

Introduction

While developing a network application on an NXP LPC43xx (Arm Cortex-M4 based), the system crashes sporadically. The application runs at 180 MHz from internal flash and works as expected for a while, but then ends up in a HardFault. Typically, the error occurs after a few minutes, but the time varies and sometimes it does not occur for quite a while.

This application note shows how we found a dependency on network activity and router behavior, which triggered a certain part of the code to execute.



Debugging the application

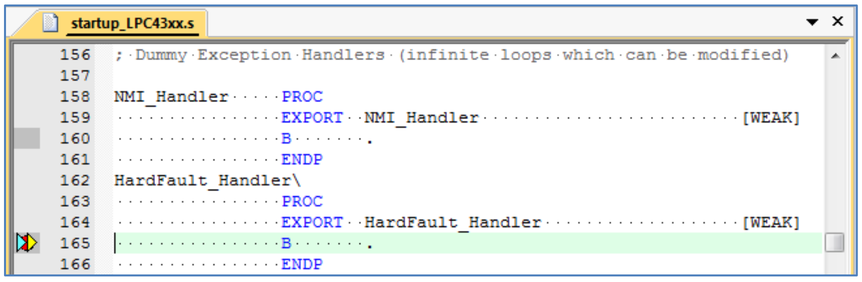
In the µVision debugger, the **Editor** and **Disassembly** windows display the code in the order as it was written. The **Trace Data** window displays the code as it was executed. There are a few exceptions to this behavior that are illustrated in this example.

Determining the cause of the HardFault

Network activity triggered by the router, caused a specific part of code to be executed. This code, the ARP resolver, was located at a critical boundary in the address space of the application. If that part of the code was executed, a HardFault occurred.

A good general rule to remember: when the program execution is halted by a fault, the last instruction displayed is the cause of the fault (or very close to it).

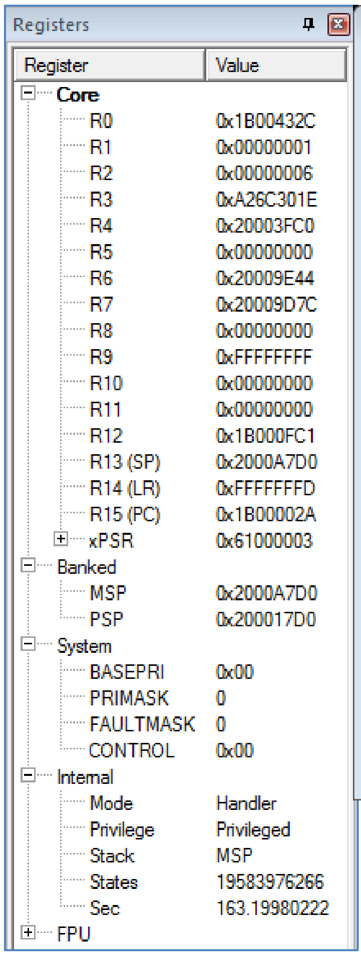
1. Connect a ULINK*pro* to the target.
2. Set a breakpoint on the HardFault exception handler entry as shown below.


startup_LPC43xx.s

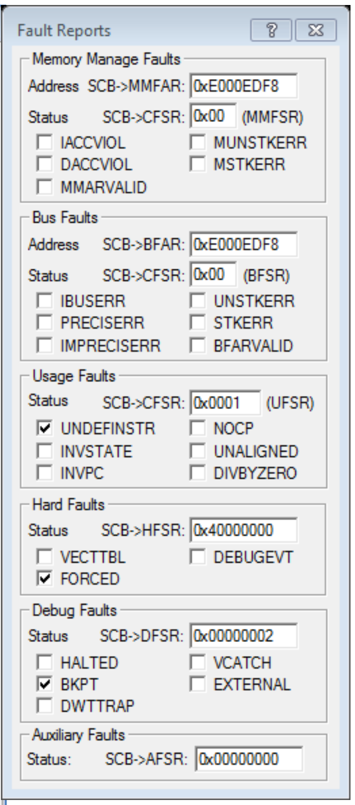
```
; Dummy Exception Handlers (infinite loops which can be modified)

NMI_Handler     PROC
                EXPORT  NMI_Handler                [WEAK]
                B       .
                ENDP
HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler          [WEAK]
                B       .
                ENDP
```

This stops the program execution when the hard fault happens. Otherwise, the trace will contain millions of the Branch (B) instruction to itself. Stopping on the fault will indicate where the error has occurred.

3. Run the program.
4. After the breakpoint is hit, examine the **Fault Reports** window. It indicates that an undefined instruction caused the fault (`UNDEFINSTR` flag).
5. Examine the **Registers** window to determine the stack frame used. Shown here, `LR = 0xFFFF_FFFD` which is the EXC_RETURN value that indicates that PSP was used for stacking.
6. PSP equals `0x2000_17D0`.


Registers

Register	Value
Core	
R0	0x1B00432C
R1	0x00000001
R2	0x00000006
R3	0xA26C301E
R4	0x20003FC0
R5	0x00000000
R6	0x20009E44
R7	0x20009D7C
R8	0x00000000
R9	0xFFFFFFFF
R10	0x00000000
R11	0x00000000
R12	0x1B000FC1
R13 (SP)	0x2000A7D0
R14 (LR)	0xFFFFFFFD
R15 (PC)	0x1B00002A
xPSR	0x61000003
Banked	
MSP	0x2000A7D0
PSP	0x200017D0
System	
BASEPRI	0x00
PRIMASK	0
FAULTMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP
States	19583976266
Sec	163.19980222
FPU	


Fault Reports

Memory Manage Faults
Address SCB->MMFAR: 0xE000EDF8
Status SCB->CFSR: 0x00 (MMFSR)
☐ IACCVIOL ☐ MUNSTKERR
☐ DACCVIOL ☐ MSTKERR
☐ MMARVALID

Bus Faults
Address SCB->BFAR: 0xE000EDF8
Status SCB->CFSR: 0x00 (BFSR)
☐ IBUSERR ☐ UNSTKERR
☐ PRECISERR ☐ STKERR
☐ IMPRECISERR ☐ BFARVALID

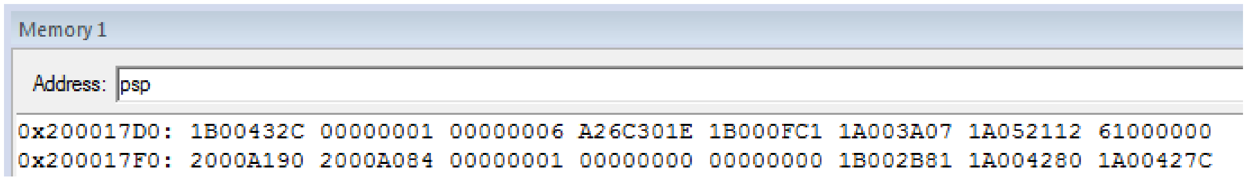
Usage Faults
Status SCB->CFSR: 0x0001 (UFSR)
☑ UNDEFINSTR ☐ NOCP
☐ INVSTATE ☐ UNALIGNED
☐ INVPC ☐ DIVBYZERO

Hard Faults
Status SCB->HFSR: 0x40000000
☐ VECTTBL ☐ DEBUGEVT
☑ FORCED

Debug Faults
Status SCB->DFSR: 0x00000002
☐ HALTED ☐ VCATCH
☑ BKPT ☐ EXTERNAL
☐ DWTTRAP

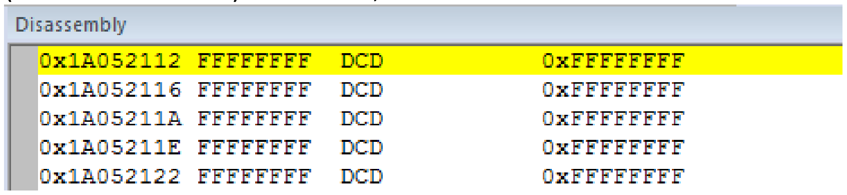
Auxiliary Faults
Status: SCB->AFSR: 0x00000000

7. Enter this address into a **Memory** window to view the stack frame.
8. Stacked registers on exception entry are examined in the **Memory 1** window shown below.
9. The stacked registers starting at 0x200_17D0 are:
   `R0, R1, R2, R3, R12, LR, ReturnAddress, xPSR.ReturnAddress`



```
0x200017D0: 1B00432C 00000001 00000006 A26C301E 1B000FC1 1A003A07 1A052112 61000000
0x200017F0: 2000A190 2000A084 00000001 00000000 00000000 1B002B81 1A004280 1A00427C
```

10. Return Address `0x1A05_2112` indicates the instruction causing the fault.
11. Examine the instruction at address `0x1A05_2112` in the Disassembly window. This is the instruction that caused the fault. It is located in the internal Flash that is not used and contains `0xFFFF_FFFF` (erased Flash value). Therefore, a hard fault due to an undefined instruction occurs.



```
0x1A052112 FFFFFFFF   DCD          0xFFFFFFFF
0x1A052116 FFFFFFFF   DCD          0xFFFFFFFF
0x1A05211A FFFFFFFF   DCD          0xFFFFFFFF
0x1A05211E FFFFFFFF   DCD          0xFFFFFFFF
0x1A052122 FFFFFFFF   DCD          0xFFFFFFFF
```

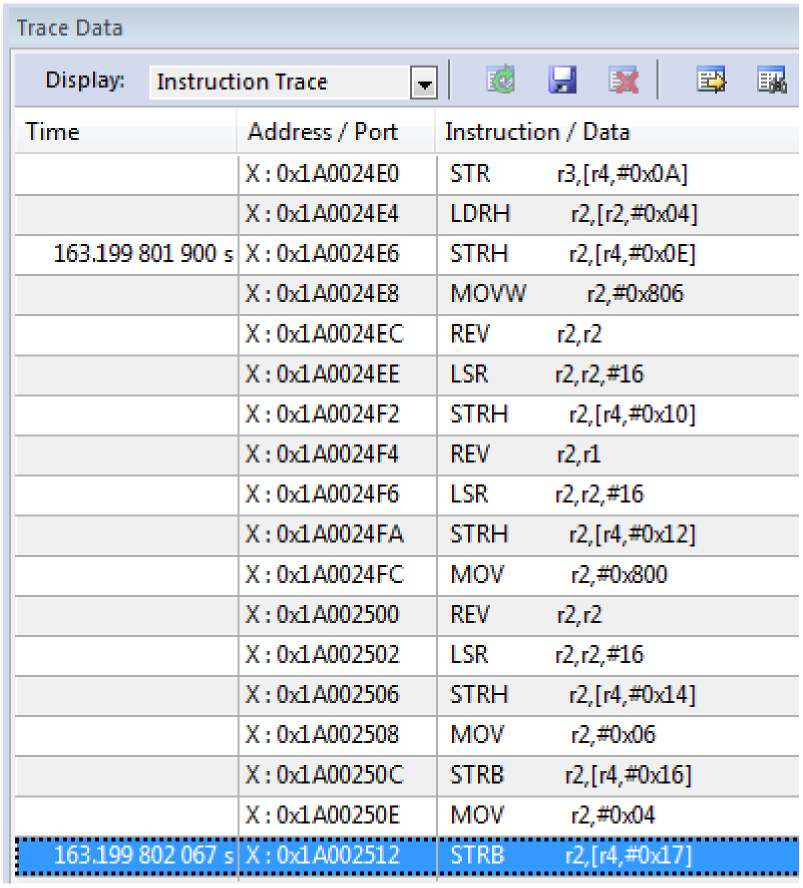
It is unclear how the program counter (PC) reached that location. Examining the state of the processor, the application, and its components at the time when the fault occurred shows no issues. It does not provide any clue why the problem occurred. Modifying the code (even the smallest change) leads to the problem not occurring anymore (Heisenbug). To continue with the investigation, the non-invasive ETM trace using the same image where the fault occurs seems to be the only way to get more information on what is happening.

Using ETM trace

The Embedded Trace Macrocell (ETM) provides high bandwidth instruction trace via four dedicated trace pins, accessible on the 20-pin Cortex Debug + ETM connector (refer to keil.com/coresight). This enhanced trace capability records a program's execution instruction-by-instruction which can be used for:

- Debugging historical sequences leading up to events of interest
- Software profiling and algorithm optimization
- Code coverage analysis

ULINK*pro* captures the full instruction trace coming from the ETM using the 20-pin connector. Here, the following trace was captured:

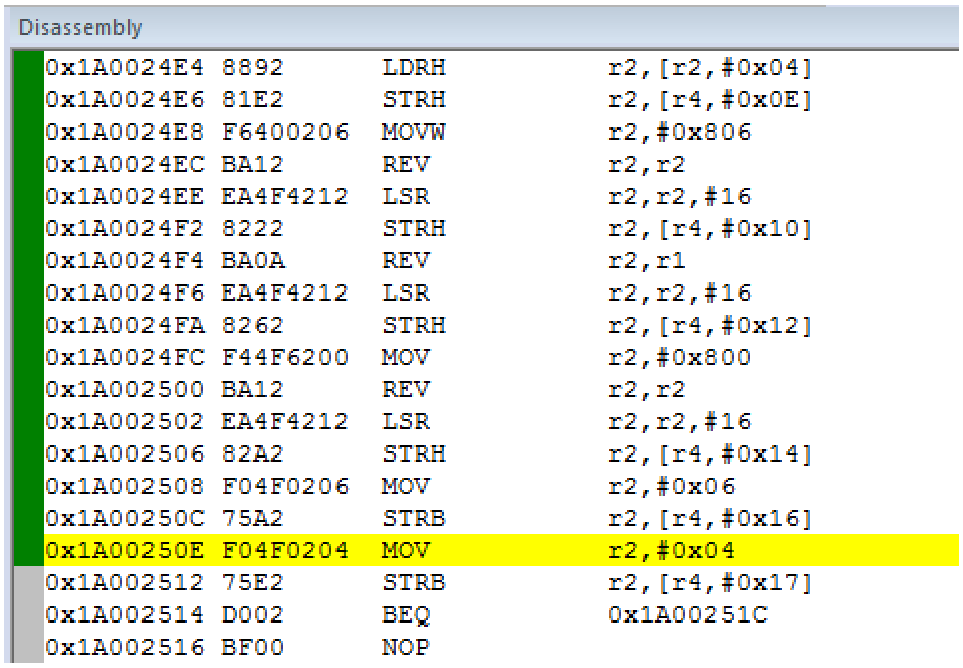


Time	Address / Port	Instruction / Data
	X : 0x1A0024E0	STR r3,[r4,#0x0A]
	X : 0x1A0024E4	LDRH r2,[r2,#0x04]
163.199 801 900 s	X : 0x1A0024E6	STRH r2,[r4,#0x0E]
	X : 0x1A0024E8	MOVW r2,#0x806
	X : 0x1A0024EC	REV r2,r2
	X : 0x1A0024EE	LSR r2,r2,#16
	X : 0x1A0024F2	STRH r2,[r4,#0x10]
	X : 0x1A0024F4	REV r2,r1
	X : 0x1A0024F6	LSR r2,r2,#16
	X : 0x1A0024FA	STRH r2,[r4,#0x12]
	X : 0x1A0024FC	MOV r2,#0x800
	X : 0x1A002500	REV r2,r2
	X : 0x1A002502	LSR r2,r2,#16
	X : 0x1A002506	STRH r2,[r4,#0x14]
	X : 0x1A002508	MOV r2,#0x06
	X : 0x1A00250C	STRB r2,[r4,#0x16]
	X : 0x1A00250E	MOV r2,#0x04
163.199 802 067 s	X : 0x1A002512	STRB r2,[r4,#0x17]

The `STRB` instruction located at `0x1A00_2512` at the bottom of the **Trace Data** window is the last execution executed according to the ETM buffer.

Interpreting the information obtained from the Trace Data window

Examining the last instructions of the Trace Data window in the **Disassembly** window shows:



Disassembly

```
0x1A0024E4 8892      LDRH     r2,[r2,#0x04]
0x1A0024E6 81E2      STRH     r2,[r4,#0x0E]
0x1A0024E8 F6400206  MOVW     r2,#0x806
0x1A0024EC BA12      REV      r2,r2
0x1A0024EE EA4F4212  LSR      r2,r2,#16
0x1A0024F2 8222      STRH     r2,[r4,#0x10]
0x1A0024F4 BA0A      REV      r2,r1
0x1A0024F6 EA4F4212  LSR      r2,r2,#16
0x1A0024FA 8262      STRH     r2,[r4,#0x12]
0x1A0024FC F44F6200  MOV      r2,#0x800
0x1A002500 BA12      REV      r2,r2
0x1A002502 EA4F4212  LSR      r2,r2,#16
0x1A002506 82A2      STRH     r2,[r4,#0x14]
0x1A002508 F04F0206  MOV      r2,#0x06
0x1A00250C 75A2      STRB     r2,[r4,#0x16]
0x1A00250E F04F0204  MOV      r2,#0x04
0x1A002512 75E2      STRB     r2,[r4,#0x17]
0x1A002514 D002      BEQ      0x1A00251C
0x1A002516 BF00      NOP
```

The last instruction that seems to be executed successfully, is `MOV R2,#0x04` at address `0x1A00_250E`. The green bar shows executed instructions and is part of code coverage. Code coverage indicates if an instruction was executed or not.

The next instruction `STRB` at address `0x1A00_2512` seems to trigger the fault. But why would the STRB cause an undefined instruction at a different address then reported in faults? Furthermore, the `MOV R2,#0x04` instruction is not really executed since the register R2 contains a different value (`R2 = 0x06`).

It seems that the `MOV` instruction was not fetched correctly from Flash and some other instruction was decoded. Most likely a branch to the address where the fault is reported.

Also, the 32-bit MOV instruction crosses a 128-bit boundary, which is another indicator that a new Flash line needs to be fetched. It is speculated that the first 16-bits of the instruction were fetched correctly (value `0xF04F`) and the second 16-bits incorrectly as `0xBE00` (and not `0x0206`).

This would effectively change the `MOV R2,#0x04` instruction (opcode 0xF04F0204) to `B 0x1A052112` opcode (0xF04FBE00) and explain why the allegedly executed `MOV` did not take place but rather a branch to an undefined instruction was executed.

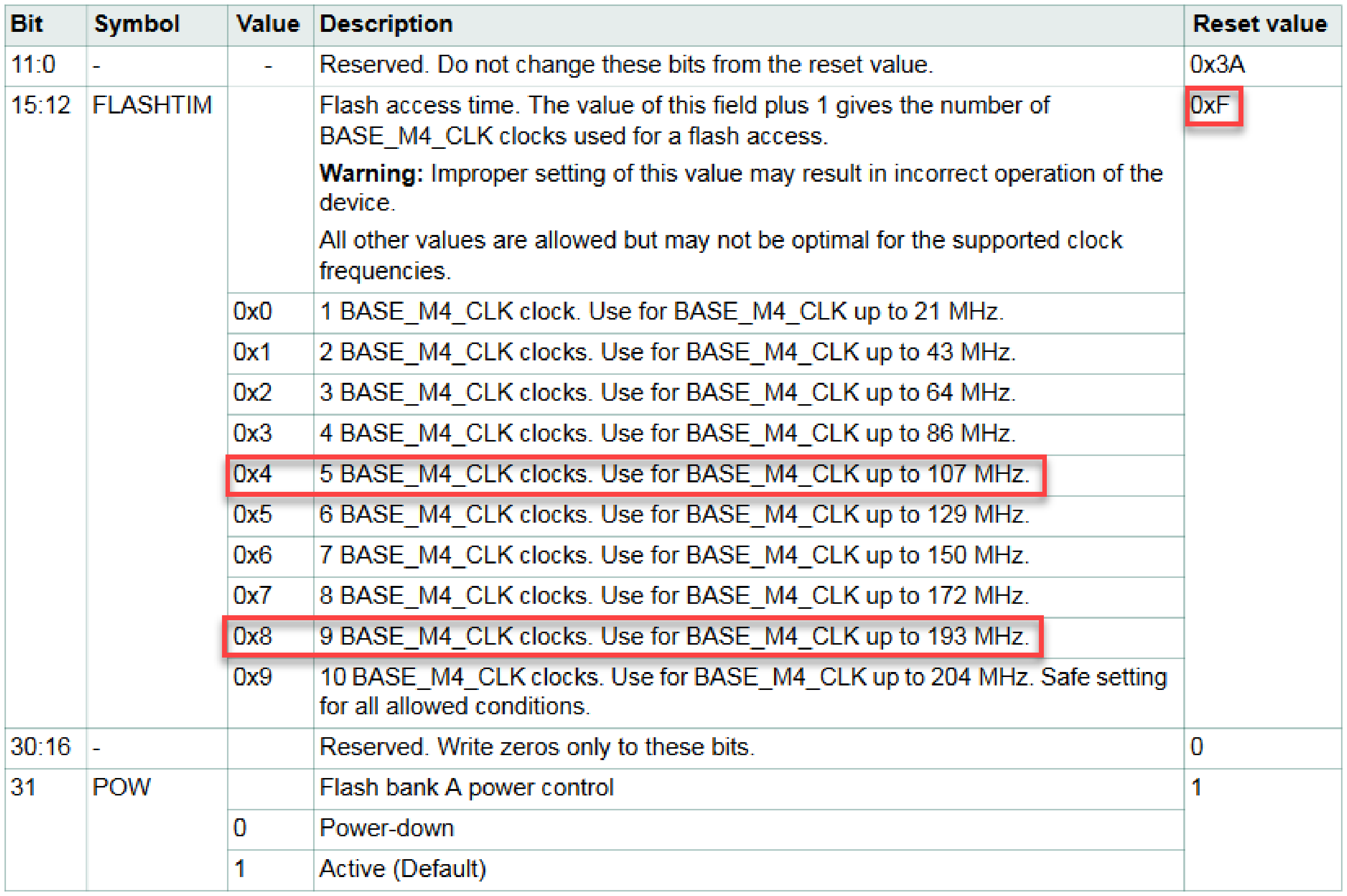
The findings point to a device or configuration problem with fetching opcodes from Flash.

NOTE

The CoreSight ETM module in Arm Cortex-M based processors provides information on program flow change such as branches and interrupts, addresses and timestamps. It is assumed the PC otherwise increments sequentially. The µVision debugger reconstructs the trace frames in the **Trace Data** window using information from the ELF/DWARF executable file (AXF). It does not display the actual opcode.

Checking the LPC43xx user's manual

The latest LPC43xx user's manual (UM10503, Rev2.4 – 22 August 2018) provides the following information:



Bit	Symbol	Value	Description	Reset value
11:0	-	-	Reserved. Do not change these bits from the reset value.	0x3A
15:12	FLASHTIM		Flash access time. The value of this field plus 1 gives the number of BASE_M4_CLK clocks used for a flash access. **Warning:** Improper setting of this value may result in incorrect operation of the device. All other values are allowed but may not be optimal for the supported clock frequencies.	0xF
		0x0	1 BASE_M4_CLK clock. Use for BASE_M4_CLK up to 21 MHz.	
		0x1	2 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 43 MHz.	
		0x2	3 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 64 MHz.	
		0x3	4 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 86 MHz.	
		0x4	5 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 107 MHz.	
		0x5	6 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 129 MHz.	
		0x6	7 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 150 MHz.	
		0x7	8 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 172 MHz.	
		0x8	9 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 193 MHz.	
		0x9	10 BASE_M4_CLK clocks. Use for BASE_M4_CLK up to 204 MHz. Safe setting for all allowed conditions.	
30:16	-		Reserved. Write zeros only to these bits.	0
31	POW		Flash bank A power control	1
		0	Power-down	
		1	Active (Default)	

Table 101 and 102. Flash Accelerator Configuration

FLASHTIM: Flash access time defaults to 0xF (16 clocks) after reset - which works safely for all CPU frequencies.

The application does not setup FLASHTIM, so the default setting should be fine. But looking at FLASHTIM after reset, reveals that is has the value of 4 (and not 0xF as stated in the manual). This would mean that the safe maximum CPU clock frequency is only 107 MHz. However, the application runs at 180 MHz from Flash which could lead to the failures as observed. Changing the FLASHTIM to 8 (CPU clock up to 193 MHz) removes the problem and confirms that the incorrect Flash Accelerator configuration was the reason for the failure observed.

It turns out that the bootloader configures FLASHTIM = 4 (max safe CPU frequency up to 107 MHz) and the documentation is wrong. This was also confirmed by NXP: community.nxp.com/thread/430097

Conclusion

A small mistake in the manual and unfortunate built-in bootloader behavior lead to random program failure. It seems that the Flash is quite failure tolerant and errors are rare. This leads to a failure that is hard to find. ULINK*pro* with instruction trace proved to be a valuable tool to tackle such problems:

1. Code coverage (provided by ETM) helps to find code that was never executed. Often, this code is from branches that are always or never taken. Code that has not been executed, has not been sufficiently tested (coverage problem). For more information, refer to www2.keil.com/mdk5/debug/coverage.
2. The problems in this example could easily be caused by a bug in the user program.
3. Remember that instruction trace points very quickly to where the problem is.
4. Problems can be created by code located far from where the error happens. ETM will turn your focus to the spot where the error occurred. You can then determine why it failed.
5. If your problem disappears and/or new ones appear when you add instrumentation to your code, such as printf's, you might have discovered a Heisenbug (or you do see an effect of your debug probe). Try ETM trace as it is non-intrusive and requires no such instrumentation.
6. Embedded software projects can become very complicated. Nowadays, they not only depend on the user's own application code, but also on third party code, and code that is inherent to the device, such as bootloaders. ETM trace can simplify finding bugs that are otherwise almost impossible to find.