Application Note

Analyzing HardFaults on Cortex-M CPU

Document: AN00016
Revision: 10
Date: January 23, 2017

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: January 23, 2017

Revision	Date	By	Description
10	170123	MM	HardFaultHandler updated.
9	160931	MC	Minor improvements.
8	160818	MC	HardFaultHandler modified to provide analysis information in debug builds only.
7	160518	MC	HardFaultHandler updated.
6	160318	MC	Improved formatting.
5	160210	MC	Minor improvements.
4	160125	MC	HardFaultHandler updated.
3	151008	MC	Minor corrections.
2	151002	MC	HardFaultHandler updated.
1	150817	TS	HardFaultHandler updated.
0	140618	TS	First version

Abstract

This application note describes the Cortex-M fault exception handling from a programmer's view and explains how to determine the cause of a hard fault.

Introduction

Fault exceptions in the Cortex-M processor trap illegal memory accesses and illegal program behavior. The following conditions are detected by fault exceptions:

- Bus Fault: detects memory access errors on instruction fetch, data read/write, interrupt vector fetch, and register stacking (save/restore) on interrupt (entry/exit).
- Memory Management Fault: detects memory access violations to regions that are defined in the Memory Management Unit (MPU); for example code execution from a memory region with read/write access only.
- Usage Fault: detects execution of undefined instructions, unaligned memory access for load/store multiple. When enabled, divide-by-zero and other unaligned memory accesses are also detected.
- Hard Fault: is caused by Bus Fault, Memory Management Fault, or Usage Fault if their handler cannot be executed.

After reset, not all fault exceptions are enabled, and with every fault the Hard Fault execption handler is executed.

Cortex-M vector table

The vector table defines all exception and interrupt vectors of the device. The vectors define the entry of an exception of interrupt handler routine. The following listing shows a typical exception table. The hard fault vector is shown in bold:

```
; Cortex M generic exception table

__vector_table
        DCD     sfe(CSTACK)
        DCD     Reset_Handler
        DCD     NMI_Handler
        DCD     HardFault_Handler
        DCD     MemManage_Handler
        DCD     BusFault_Handler
        DCD     UsageFault_Handler
```

The Hard Fault exception is always enabled and has a fixed priority (higher than other interrupts and exceptions, but lower than NMI). The Hard Fault exception is therefore executed in cases where a fault exception is disabled or when a fault occurs during the execution of a fault exception handler.
All other fault exceptions (Memory Management Fault, Bus Fault, and Usage Fault) have a programmable priority. After reset, these exceptions are disabled, and may be enabled in the system or application software using the registers in the System Control Block.

How to use the Hard Fault handler

If your application is based on an embOS start project, there is nothing further to do. Each embOS start project comes with a Hard Fault handler implementation. If your application is not based on embOS, however, as each exception handler is build with “weak” linkage in the startup code, it is very easy to create your own Hard Fault handler. Simply define a function with the name “HardFault_Handler”, or add the following handler to your application code, and ensure that the name "HardFault_Handler" is used in the vector table.

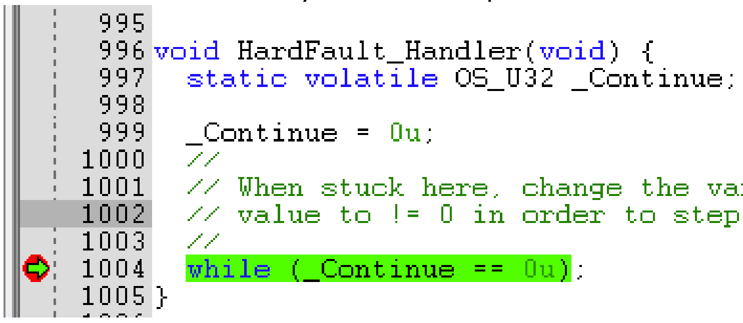
```
static volatile unsigned int _Continue;

void HardFault_Handler(void) {

  _Continue = 0u;
  //
  // When stuck here, change the variable value to != 0 in order to step out
  //
  while (_Continue == 0u);
}
```

Determining where the exception occurred

This example shows how to determine where the execption occurred. The example generates a hard fault exception upon accessing nonexistent memory at address `0xCCCCCCCC`.

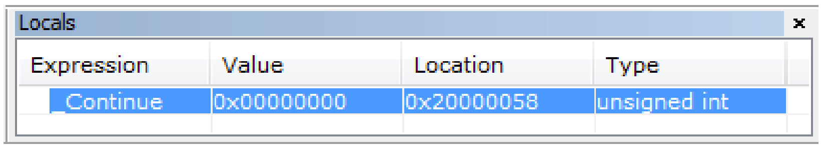
```
volatile unsigned int* p;
unsigned int n;
p = (unsigned int*)0xCCCCCCCC;
n = *p;
```

If you run this example with your debugger, it will run into the while loop in the Hard Fault handler. You may set a breakpoint there:



```

void HardFault_Handler(void) {
  static volatile OS_U32 _Continue;

  _Continue = 0u;
  //
  // When stuck here, change the variable
  // value to != 0 in order to step out
  //
  while (_Continue == 0u);
}
```

Change the variable `_Continue` to any other value than zero in a local window or watch window:



Locals

Expression	Value	Location	Type
_Continue	0x00000000	0x20000058	unsigned int

Afterwards you can step to the end of the Hard Fault handler. If you step out of the Hard Fault handler, you will reach the first instruction after the instruction which caused the hard fault.

Advanced stack analyzing

With Cortex-M it is possible to get additional information about the cause of the fault execption.

What happens when a Hard Fault occurs

When a hard fault exception occurs, the CPU saves the following register on the current stack, which can be either main stack or process stack:

	<previous>	<-- SP points here before exception
SP + 0x1C	**xPSR**	
SP + 0x18	**PC**	
SP + 0x14	**LR**	
SP + 0x10	**R12**	
SP + 0x0C	**R3**	
SP + 0x08	**R2**	
SP + 0x04	**R1**	
SP + 0x00	**R0**	<-- SP points here after exception

Table 2.1: Execution context stored on stack

Determining which stack was used before

When entering an exception handler, the LR register is updated to a special value called `EXC_RETURN` with the upper 28 bits all set to 1. This value, when loaded into the PC at the end of the exception handler execution, will cause the CPU to perform an exception return sequence. Bit 2 of the LR register determines the used stack before entering the exception.

Bit 2 of LR register	Used stack
0	Main stack
1	Process stack

Table 2.2: LR Bit 2 meaning

NVIC register

The Cortex-M NVIC includes several register that help to investigate the cause of the hard fault.

Register	Description
SYSHND_CTRL	System Handler Control and State Register
CFSR	Configurable Fault Status Register, combines MFSR, BFSR and UFSR
MFSR	Memory Management Fault Status Register
BFSR	Bus Fault Status Register
UFSR	Usage Fault Status Register
HFSR	Hard Fault Status Register
DFSR	Debug Fault Status Register
BFAR	Bus Fault Manage Address Register
AFSR	Auxiliary Fault Status Register

Table 2.3: NVIC register

Imprecise Bus Fault

The sample given above generates an imprecise bus fault. For the Cortex-M, an imprecise bus fault (as indicated by bit 10 in the CFSR register) means that a write to an invalid address was attempted.
If you look at the program counter, the faulty write is usually present in the three or more instructions leading up to the program counter address. Because of the Cortex-

M write buffer system, the program counter might have advanced slightly before the actual bus write took place, hence you need to look back slightly to find the erroneous write.

Advanced Hard Fault Handler

The advanced Hard Fault Handler constists of two seperate parts, one written in assembly and one written in "C".

The following code is written in assembly. Please note that this file needs to be pre-processed, which may require additional options to be passed to the assembler.

```
;/*********************************************************************
;*               (c) SEGGER Microcontroller GmbH & Co. KG             *
;*                        The Embedded Experts                        *
;*                           www.segger.com                           *
;**********************************************************************
;
;----------------------------------------------------------------------
;File    : HardFaultHandler.S
;Purpose : HardFault exception handler for IAR, Keil and GNU assembler.
;--------  END-OF-HEADER  ---------------------------------------------
;*/

#ifndef __IAR_SYSTEMS_ASM__
  #ifndef __CC_ARM
    #ifndef __GNUC__
      #error "Unsupported assembler!"
    #endif
  #endif
#endif

;/*********************************************************************
;*
;*       Forward declarations of segments used
;*
;**********************************************************************
;*/

#ifdef __IAR_SYSTEMS_ASM__
        SECTION CODE:CODE:NOROOT(2)
        SECTION CSTACK:DATA:NOROOT(3)
#elif defined __CC_ARM
        AREA    OSKERNEL, CODE, READONLY, ALIGN=2
        PRESERVE8
#endif

;/*********************************************************************
;*
;*       Publics
;*
;**********************************************************************
;*/

#ifdef __IAR_SYSTEMS_ASM__
        SECTION .text:CODE:NOROOT(2)
        PUBLIC  HardFault_Handler
#elif defined __CC_ARM
        EXPORT  HardFault_Handler
#elif defined __GNUC__
        .global HardFault_Handler
        .type   HardFault_Handler, function
#endif

;/*********************************************************************
;*
;*       Externals, code
;*
;**********************************************************************
;*/

#ifdef __IAR_SYSTEMS_ASM__
        EXTERN  HardFaultHandler
#elif defined __CC_ARM
        IMPORT  HardFaultHandler
#elif defined __GNUC__
        .extern HardFaultHandler
#endif

;/*********************************************************************
;*
;*       CODE segment
;*
;**********************************************************************
```

```
;*/

#ifdef __GNUC__
        .syntax unified
        .thumb
        .balign 4
        .text
#else
        THUMB
#endif

;/*********************************************************************
;*
;*       Global functions
;*
;*********************************************************************
;*/

;/*********************************************************************
;*
;*      HardFault_Handler()
;*
;*  Function description
;*    Evaluates the used stack (MSP, PSP) and passes the appropiate
;*    stack pointer to the HardFaultHandler "C"-routine.
;*
;*  Notes
;*    (1) Ensure that HardFault_Handler is part of the exception table
;*/
#ifdef __GNUC__
HardFault_Handler:
#else
HardFault_Handler
#endif
#if (defined (__IAR_SYSTEMS_ASM__) && (__ARM6M__) && (__CORE__ == __ARM6M__)) || \
    (defined (__CC_ARM) && (__TARGET_ARCH_6S_M))                               || \
    (defined (__GNUC__) && (__ARM_ARCH_6M__))
        ;// This version is for Cortex M0
        movs    R0, #4
        mov     R1, LR
        tst     R0, R1              ;// Check EXC_RETURN in Link register bit 2.
        bne     Uses_PSP
        mrs     R0, MSP             ;// Stacking was using MSP.
        b       Pass_StackPtr
#ifdef __GNUC__
Uses_PSP:
#else
Uses_PSP
#endif
        mrs     R0, PSP             ;// Stacking was using PSP.
#ifdef __GNUC__
Pass_StackPtr:
#else
Pass_StackPtr
#endif
#ifdef __CC_ARM
        ALIGN
#endif
        ldr     R2,=HardFaultHandler
        bx      R2                  ;// Stack pointer passed through R0.
#else
        ;// This version is for Cortex M3, Cortex M4 and Cortex M4F
        tst     LR, #4              ;// Check EXC_RETURN in Link register bit 2.
        ite     EQ
        mrseq   R0, MSP             ;// Stacking was using MSP.
        mrsne   R0, PSP             ;// Stacking was using PSP.
        b       HardFaultHandler    ;// Stack pointer passed through R0.
#endif

#ifdef __GNUC__
        .end
#else
        END
#endif

;/****** End Of File *************************************************/
```

The following code is written in "C":

```c
/*********************************************************************
*                (c) SEGGER Microcontroller GmbH & Co. KG            *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

----------------------------------------------------------------------
File    : SEGGER_HardFaultHandler.c
Purpose : Generic SEGGER HardFault handler for Cortex-M, enables user-
          friendly analysis of hard faults in debug configurations.
--------  END-OF-HEADER  ---------------------------------------------
*/

/*********************************************************************
*
*       Defines
*
**********************************************************************
*/
// System Handler Control and State Register
#define SYSHND_CTRL (*(volatile unsigned int*)  (0xE000ED24u))
// Memory Management Fault Status Register
#define NVIC_MFSR   (*(volatile unsigned char*) (0xE000ED28u))
// Bus Fault Status Register
#define NVIC_BFSR   (*(volatile unsigned char*) (0xE000ED29u))
// Usage Fault Status Register
#define NVIC_UFSR   (*(volatile unsigned short*)(0xE000ED2Au))
// Hard Fault Status Register
#define NVIC_HFSR   (*(volatile unsigned int*)  (0xE000ED2Cu))
// Debug Fault Status Register
#define NVIC_DFSR   (*(volatile unsigned int*)  (0xE000ED30u))
// Bus Fault Manage Address Register
#define NVIC_BFAR   (*(volatile unsigned int*)  (0xE000ED38u))
// Auxiliary Fault Status Register
#define NVIC_AFSR   (*(volatile unsigned int*)  (0xE000ED3Cu))

#ifndef   DEBUG       // Should be overwritten by project settings
  #define DEBUG  (0)  // in debug builds
#endif

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

#if DEBUG
static volatile unsigned int _Continue; // Set this variable to 1 to run further

static struct {
  struct {
    volatile unsigned int r0;            // Register R0
    volatile unsigned int r1;            // Register R1
    volatile unsigned int r2;            // Register R2
    volatile unsigned int r3;            // Register R3
    volatile unsigned int r12;           // Register R12
    volatile unsigned int lr;            // Link register
    volatile unsigned int pc;            // Program counter
    union {
      volatile unsigned int byte;
      struct {
        unsigned int IPSR : 8;           // Interrupt Program Status register (IPSR)
        unsigned int EPSR : 19;          // Execution Program Status register (EPSR)
        unsigned int APSR : 5;          // Application Program Status register (APSR)
      } bits;
    } psr;                               // Program status register.
  } SavedRegs;

   union {
    volatile unsigned int byte;
    struct {
      unsigned int MEMFAULTACT    : 1;   // Read as 1 if memory management fault
                                         // is active
      unsigned int BUSFAULTACT    : 1;  // Read as 1 if bus fault exception is active
      unsigned int UnusedBits1     : 1;
      unsigned int USGFAULTACT     : 1;    // Read as 1 if usage fault exception
                                           // is active
      unsigned int UnusedBits2     : 3;
      unsigned int SVCALLACT       : 1;    // Read as 1 if SVC exception is active
      unsigned int MONITORACT      : 1;    // Read as 1 if debug monitor exception
                                           // is active
      unsigned int UnusedBits3     : 1;
      unsigned int PENDSVACT       : 1;    // Read as 1 if PendSV exception is active
      unsigned int SYSTICKACT      : 1;    // Read as 1 if SYSTICK exception is active
      unsigned int USGFAULTPENDED : 1;     // Usage fault pended; usage fault started
                                           // but was replaced by a higher-priority
                                           // exception
      unsigned int MEMFAULTPENDED : 1;     // Memory management fault pended; memory
                                           // management fault started but was
                                           // replaced by a higher-priority exception
      unsigned int BUSFAULTPENDED : 1;     // Bus fault pended; bus fault handler was
                                           // started but was replaced by a
                                           // higher-priority exception
      unsigned int SVCALLPENDED    : 1;    // SVC pended; SVC was started but was
                                           // replaced by a higher-priority exception
      unsigned int MEMFAULTENA     : 1;    // Memory management fault handler enable
      unsigned int BUSFAULTENA     : 1;    // Bus fault handler enable
      unsigned int USGFAULTENA     : 1;    // Usage fault handler enable
    } bits;

  } syshndctrl;                            // System Handler Control and State
                                           // Register (0xE000ED24)
  union {
    volatile unsigned char byte;
    struct {
      unsigned char IACCVIOL    : 1;       // Instruction access violation
      unsigned char DACCVIOL    : 1;       // Data access violation
      unsigned char UnusedBits  : 1;
      unsigned char MUNSTKERR   : 1;       // Unstacking error
      unsigned char MSTKERR     : 1;       // Stacking error
      unsigned char UnusedBits2 : 2;
      unsigned char MMARVALID   : 1;       // Indicates the MMAR is valid
    } bits;
  } mfsr;                                  // Memory Management Fault Status
                                           // Register (0xE000ED28)

  union {
    volatile unsigned int byte;
    struct {
      unsigned int IBUSERR    : 1;         // Instruction access violation
      unsigned int PRECISERR  : 1;         // Precise data access violation
      unsigned int IMPREISERR : 1;         // Imprecise data access violation
      unsigned int UNSTKERR   : 1;         // Unstacking error
      unsigned int STKERR     : 1;         // Stacking error
      unsigned int UnusedBits : 2;
      unsigned int BFARVALID  : 1;         // Indicates BFAR is valid
    } bits;
  } bfsr;                                  // Bus Fault Status Register (0xE000ED29)
  volatile unsigned int bfar;              // Bus Fault Manage Address Register
                                           // (0xE000ED38)

  union {
    volatile unsigned short byte;
    struct {
      unsigned short UNDEFINSTR : 1;       // Attempts to execute an undefined
                                           // instruction
      unsigned short INVSTATE   : 1;       // Attempts to switch to an invalid state
                                           // (e.g., ARM)
      unsigned short INVPC      : 1;       // Attempts to do an exception with a bad
                                           // value in the EXC_RETURN number
      unsigned short NOCP       : 1;       // Attempts to execute a coprocessor
                                           // instruction
      unsigned short UnusedBits : 4;
      unsigned short UNALIGNED  : 1;       // Indicates that an unaligned access fault
                                           // has taken place
      unsigned short DIVBYZERO  : 1;       // Indicates a divide by zero has taken
                                           // place (can be set only if DIV_0_TRP
                                           // is set)
    } bits;
  } ufsr;                                  // Usage Fault Status Register (0xE000ED2A)

  union {
    volatile unsigned int byte;
    struct {
      unsigned int UnusedBits   : 1;
```

```
        unsigned int VECTBL      : 1;     // Indicates hard fault is caused by failed
                                          // vector fetch
        unsigned int UnusedBits2 : 28;
        unsigned int FORCED      : 1;     // Indicates hard fault is taken because of
                                          // bus fault/memory management fault/usage
                                          // fault
        unsigned int DEBUGEVT    : 1;     // Indicates hard fault is triggered by
                                          // debug event
      } bits;
    } hfsr;                               // Hard Fault Status Register (0xE000ED2C)

    union {
      volatile unsigned int byte;
      struct {
        unsigned int HALTED   : 1;        // Halt requested in NVIC
        unsigned int BKPT     : 1;        // BKPT instruction executed
        unsigned int DWTTRAP  : 1;        // DWT match occurred
        unsigned int VCATCH   : 1;        // Vector fetch occurred
        unsigned int EXTERNAL : 1;        // EDBGRQ signal asserted
      } bits;
    } dfsr;                               // Debug Fault Status Register (0xE000ED30)

    volatile unsigned int afsr;           // Auxiliary Fault Status Register
                                          // (0xE000ED3C) Vendor controlled (optional)
  } HardFaultRegs;
  #endif

  /*********************************************************************
  *
  *       Global functions
  *
  **********************************************************************
  */

  /*********************************************************************
  *
  *       HardFaultHandler()
  *
  *  Function description
  *    Generic hardfault handler
  */
  void HardFaultHandler(unsigned int* pStack);
  void HardFaultHandler(unsigned int* pStack) {
    //
    // In case we received a hard fault because of a breakpoint instruction, we return.
    // This may happen when using semihosting for printf outputs and no debugger
    // is connected, i.e. when running a "Debug" configuration in release mode.
    //
  if (NVIC_HFSR & (1uL << 31)) {
    NVIC_HFSR |=  (1uL << 31);   // Reset Hard Fault status
    *(pStack + 6u) += 2u;        // PC is located on stack at SP + 24 bytes;
                                 // increment PC by 2 to skip break instruction.
    return;                      // Return to interrupted application
  }
  #if DEBUG
    //
    // Read NVIC registers
    //
    HardFaultRegs.syshndctrl.byte = SYSHND_CTRL;  // System Handler Control and
                                                  // State Register
    HardFaultRegs.mfsr.byte       = NVIC_MFSR;    // Memory Fault Status Register
    HardFaultRegs.bfsr.byte       = NVIC_BFSR;    // Bus Fault Status Register
    HardFaultRegs.bfar            = NVIC_BFAR;    // Bus Fault Manage Address Register
    HardFaultRegs.ufsr.byte       = NVIC_UFSR;    // Usage Fault Status Register
    HardFaultRegs.hfsr.byte       = NVIC_HFSR;    // Hard Fault Status Register
    HardFaultRegs.dfsr.byte       = NVIC_DFSR;    // Debug Fault Status Register
    HardFaultRegs.afsr            = NVIC_AFSR;    // Auxiliary Fault Status Register
    //
    // Halt execution
    // If NVIC registers indicate readable memory, change the variable value
    // to != 0 to continue execution.
    //
    _Continue = 0u;
    while (_Continue == 0u);
  //
  // Read saved registers from the stack
  //
  HardFaultRegs.SavedRegs.r0       = pStack[0];  // Register R0
  HardFaultRegs.SavedRegs.r1       = pStack[1];  // Register R1
  HardFaultRegs.SavedRegs.r2       = pStack[2];  // Register R2
  HardFaultRegs.SavedRegs.r3       = pStack[3];  // Register R3
  HardFaultRegs.SavedRegs.r12      = pStack[4];  // Register R12
  HardFaultRegs.SavedRegs.lr       = pStack[5];  // Link register LR
  HardFaultRegs.SavedRegs.pc       = pStack[6];  // Program counter PC
  HardFaultRegs.SavedRegs.psr.byte = pStack[7];  // Program status word PSR
  //
  // Halt execution
  // To step out of the HardFaultHandler, change the variable value to != 0.
  //
  _Continue = 0u;
  while (_Continue == 0u) {
  }
#else
  //
  // If this module is included in a release configuration,
  // simply stay in the HardFault handler
  //
  (void)pStack;
  do {
  } while (1);
#endif
}

/*************************** End of file ****************************/
```