Developing a Generic Hard Fault handler for ARM Cortex-M3/Cortex-M4

Niall Cooling

Feabhas Limited

www.feabhas.com

FEABHAS
EMBEDDING SOFTWARE COMPETENCE

ARM 2013 TechCon™
Where Intelligence Connects

Divide by zero error

```
int div(int lho, int rho)
{
   return lho/rho;
}
```

```
int main(void)
{
   int a = 10;
   int b = 0;
   int c;
   …
   c = div(a, b);
   …
```

Enabling hardware reporting of div0 errors

- To enable hardware reporting of div0 errors we need to configure the **Configuration and Control Register** (CCR).

- The CCR is part of the Cortex-M's **System Control Block** (SCB) and controls entry trapping of divide by zero and unaligned accesses among other things.

- The CCR bit assignment for div0 is:
  - **[4] DIV_0_TRP** Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0:
    - 0 = do not trap divide by 0
    - **1 = trap divide by 0**.

  - When this bit is set to 0, a divide by zero returns a quotient of 0.

```
SCB->CCR |= 0x10;
```

ARMv7-M Vector Table

Address	Vector	
0x00	Initial Main SP	Initial Stack Pointer
0x04	Reset	Initial Program Counter
0x08	NMI	Non-Maskable Interrupt
0x0c	**Hard Fault**	**Non-privilege access**
0x10	Memory Manage	MPU generated
0x14	Bus Fault	Instruction prefetch abort or data access error
0x18	Usage Fault	Invalid instructions
0x1c-0x28	*Reserved*	
0x2c	SVCall	System Service Call (SVC) – used for RTOS entry calls
0x30	Debug Monitor	
0x34	*Reserved*	
0x38	PendSV	Pended System Call – used for RTOS context switching
0x3c	SysTick	
0x40	IRQ0	First Device Specific IRQ (0-239)
….	More IRQs	

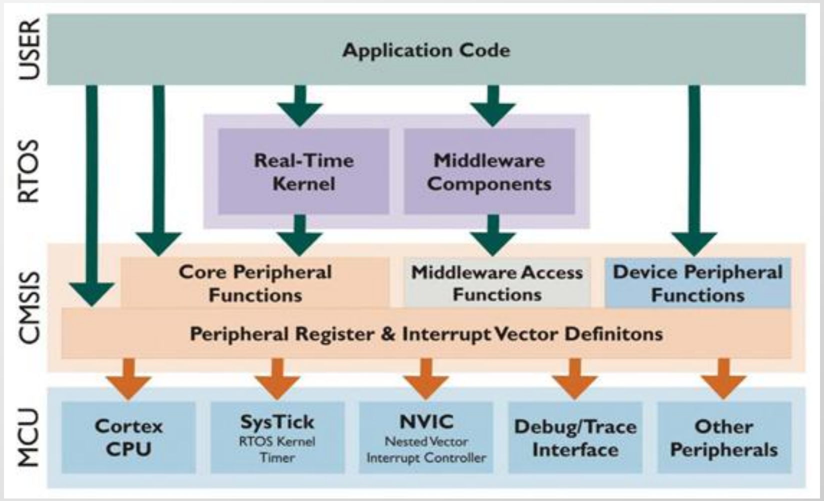
$\left.\begin{array}{l}\text{Initial Stack Pointer}\\ \text{Initial Program Counter}\\ \text{Non-Maskable Interrupt}\\ \text{Non-privilege access}\end{array}\right\}$ Minimal Set

CMSIS-Core

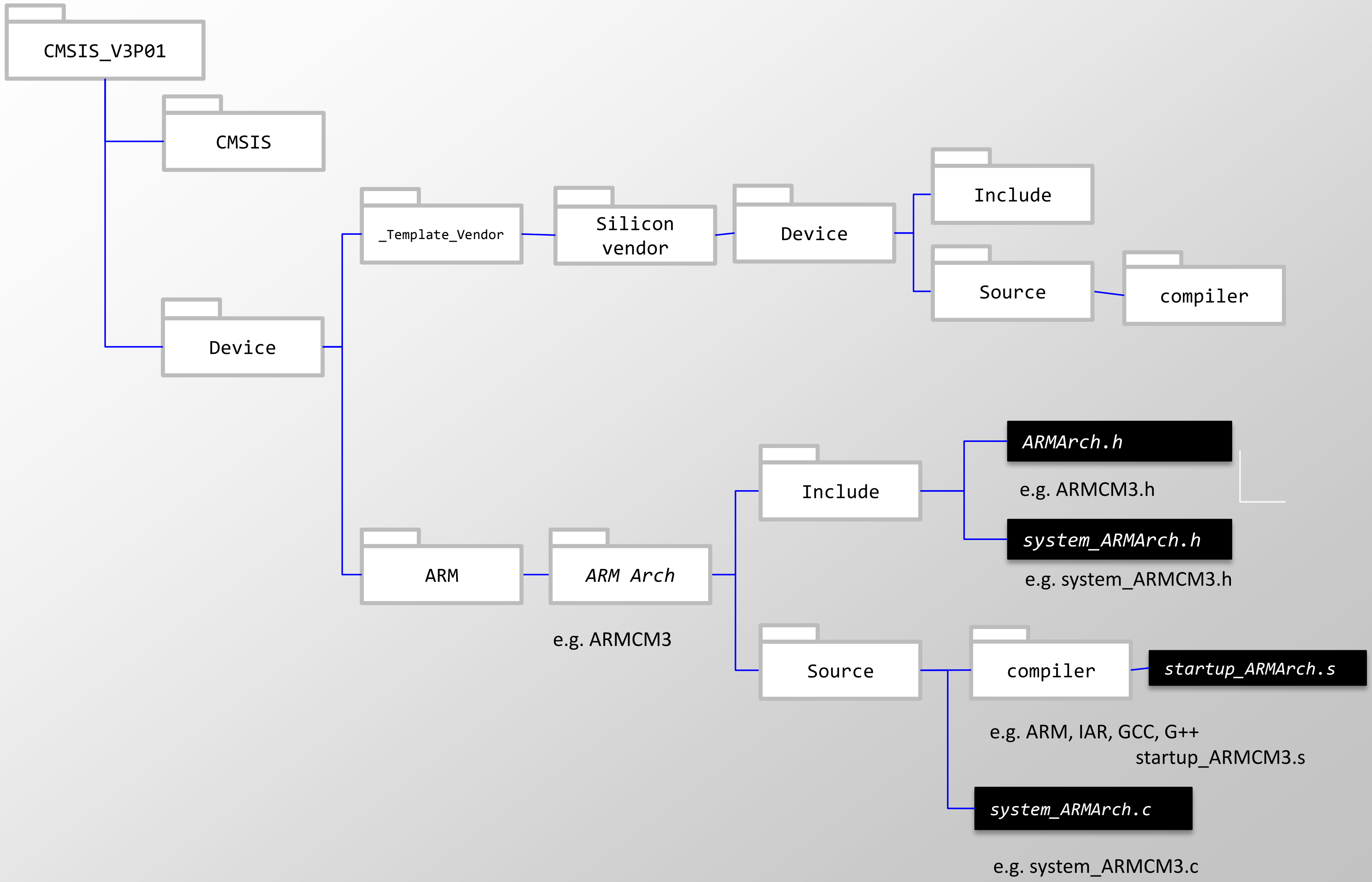
- Cortex™-M3 & Cortex™-M0

- Core Peripheral Access Layer
  - Cortex-M Core Register Access
  - Cortex-M Instruction Access
  - NVIC Access Functions
  - SysTick Configuration Function

- Instrumented Trace Macrocell (ITM)
  - Cortex ™ -M3 ITM Debug Access
    - ITM_SendChar
  - Cortex ™ -M3 additional Debug Access
    - ITM_ReceiveChar


USER
RTOS
CMSIS
MCU
Application Code
Real-Time Kernel
Middleware Components
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Register & Interrupt Vector Definitons
Cortex CPU
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals

CMSIS v3.0 - Device



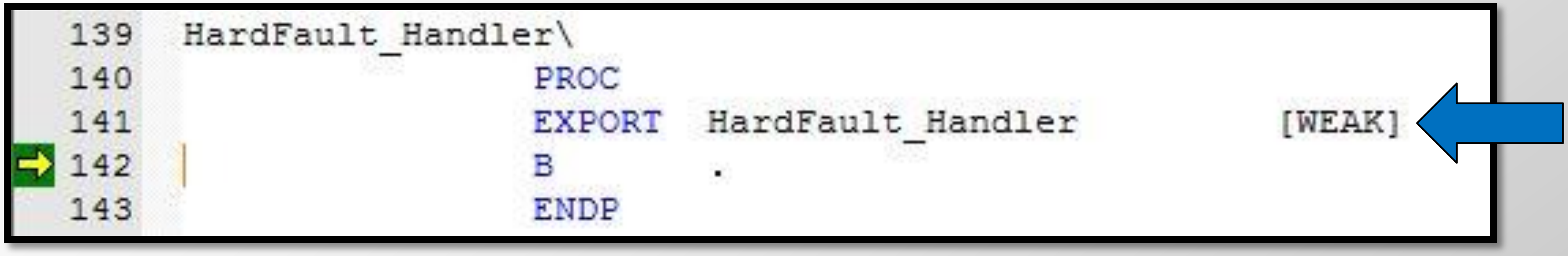
CMSIS ARM/Keil Application startup_Device.s

```
; Vector Table Mapped to Address 0 at Reset

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                …
```

```
; 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
MemManage_Handler\
```

ARM/Keil uVision Execution

- Need to halt execution yourself
- Execution will stop here:



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

- Need to replace the default HardFault_Handler with our own code
- Normally need to modify the asm file
- **Weak linkage** to the rescue!

Weak linkage

```
void simple(void) __attribute__((weak)) ;

int main (void)
{
    simple();
}

void __attribute__((weak)) simple(void)
{
    printf("In WEAK simple\n");
}
```

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

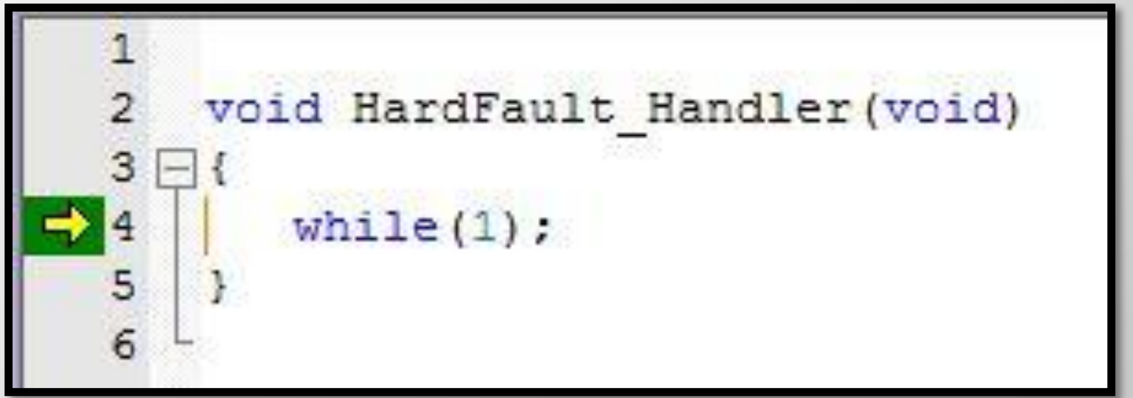
Overriding weak linkage

```
void simple(void) __attribute__((weak)) ;

int main(void)
{
    simple();
}

void __attribute__((weak)) simple(void)
{
    printf("In WEAK simple\n");
}
```

```
void simple(void)
{
    printf("overridden in: %s\n", __FILE__);
}
```

Override The Default Hard Fault_Handler

```
void HardFault_Handler(void)
{
   while(1);
}
```



```
void HardFault_Handler(void)
{
    while(1);
}
```

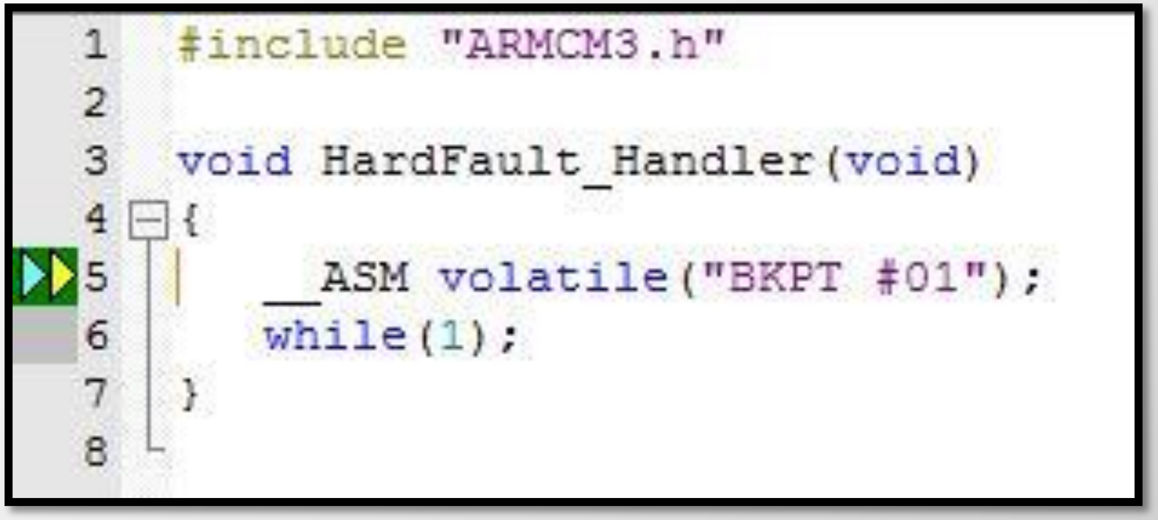
Enter debug state automatically

- Rather than having to enter breakpoints via your IDE, we want to force the processor to enter debug state automatically if a certain instruction is reached (a sort of debug based *assert*).
- Inserting the BKPT (breakpoint) ARM instruction in our code will cause the processor to enter debug state.
- The “immediate” following the opcode normally doesn't matter (*but always check*) except it shouldn't be 0xAB (which is used for semihosting).

```
#include "ARMCM3.h"

void HardFault_Handler(void)
{
   __ASM volatile("BKPT #01");
   while(1);
}
```

Run and Break

- Now when we run the execution will halt when the BKPT instruction is executed

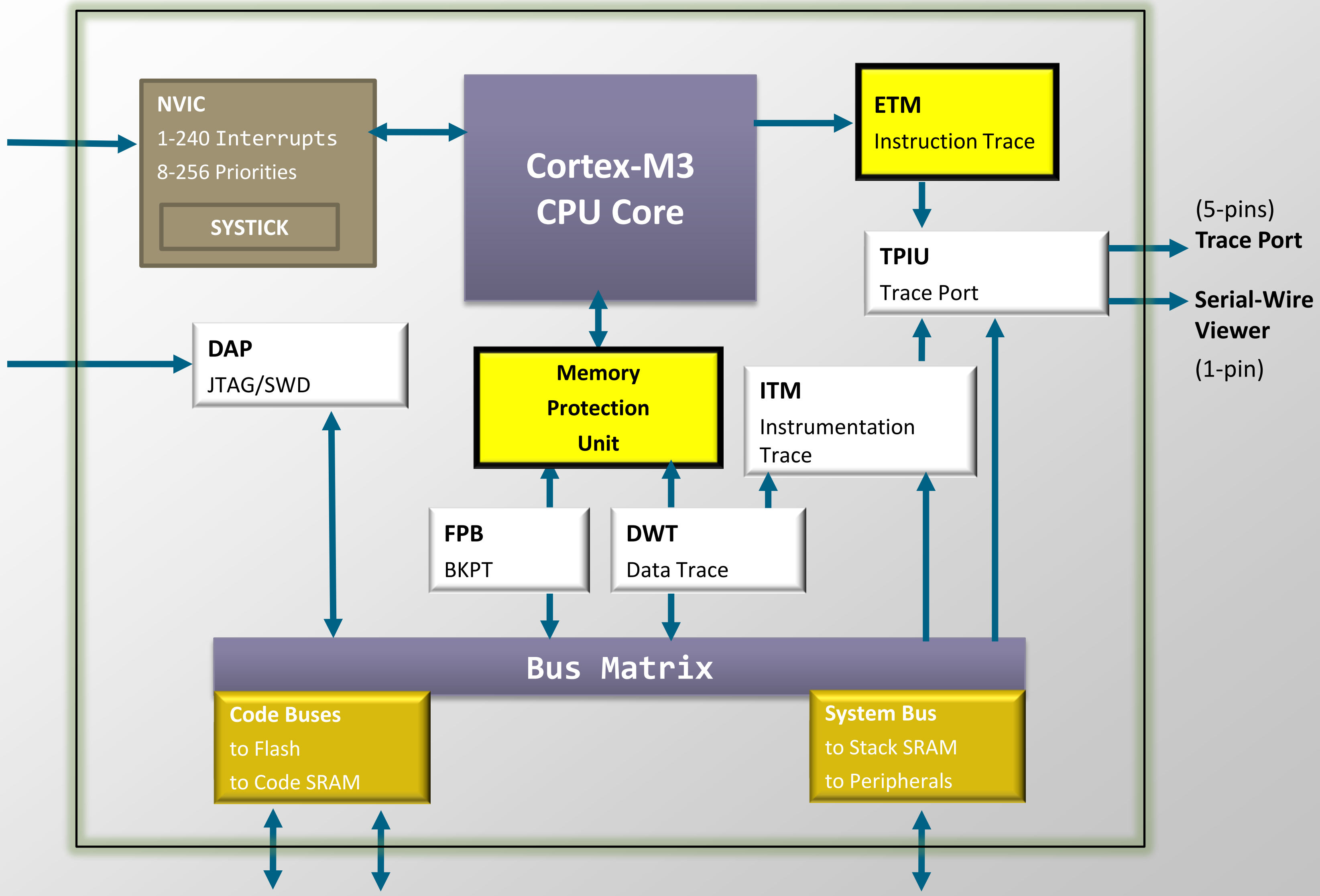


```
#include "ARMCM3.h"

void HardFault_Handler(void)
{
    __ASM volatile("BKPT #01");
    while(1);
}

```

Error Message Output

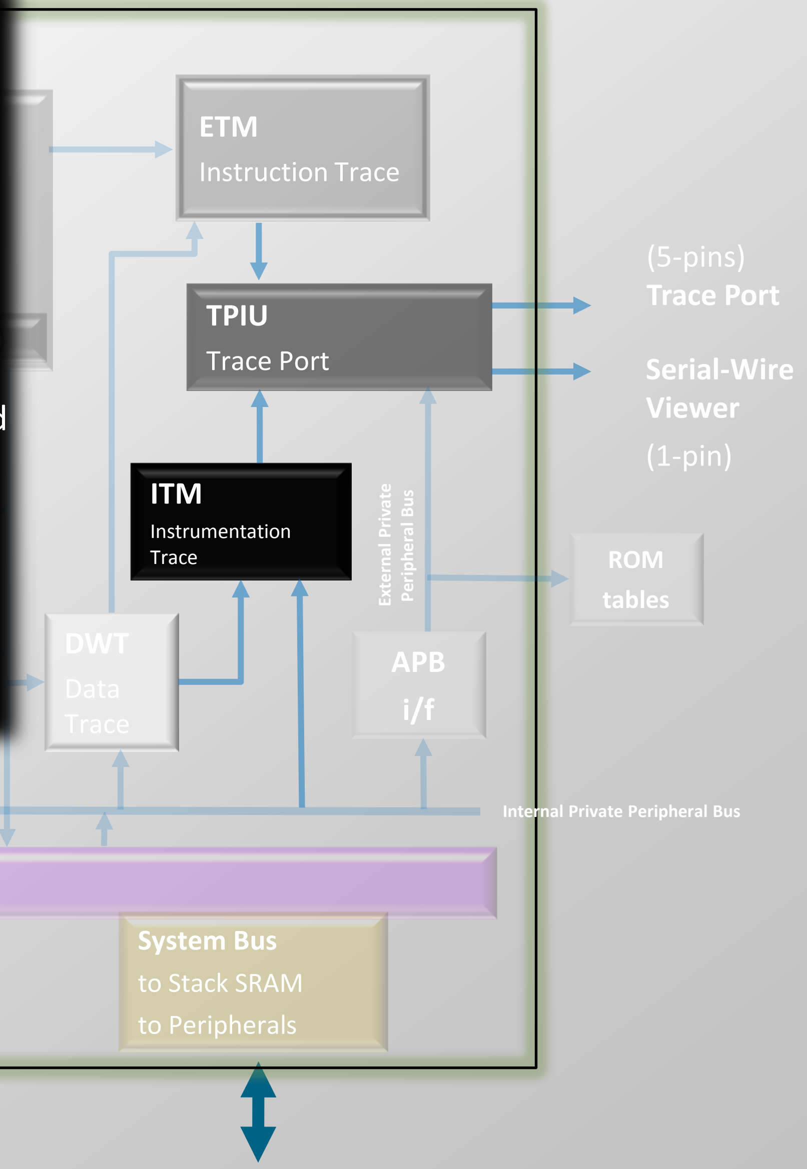
- The next step in developing the fault handler is the ability to report the fault.
- One option is, of course, to use stdio (stderr) and semihosting.
- However, as the support for semihosting can vary from compiler to compiler
- The "better" alternative is to use Instrumented Trace Macrocell (ITM) utilizing the CMSIS wrapper function ITM_SendChar

ARM Cortex-M3 Processor



Instrumentation Trace Macrocell (ITM)

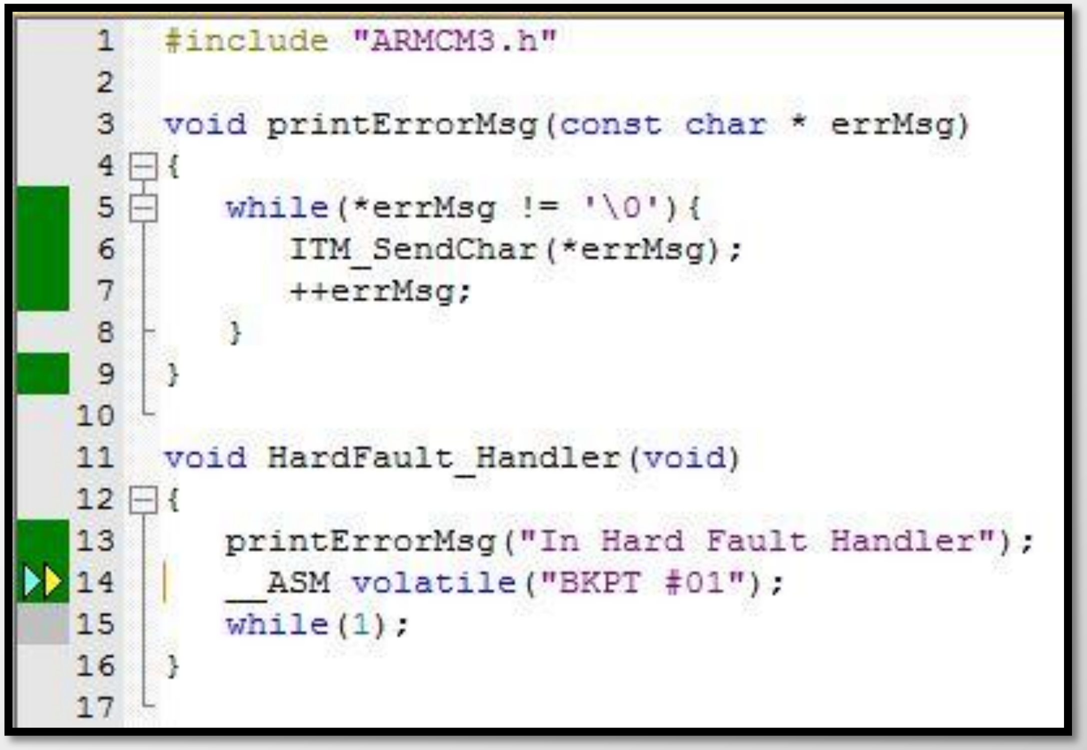
- Generates several types of Trace Packets
  - Software trace
    - Software can write directly to ITM, causing packets to be emitted
    - Similar to using printf() to debug a C program
  - Hardware trace
    - Packets are generated by the DWT and emitted by the ITM
  - Timestamps



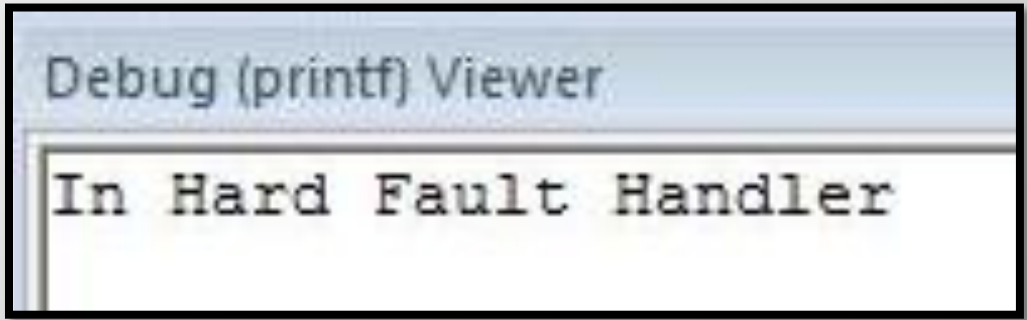
ITM Based Error Message Output

```
void printErrorMsg(const char * errMsg)
{
   while(*errMsg != '\0'){
      ITM_SendChar(*errMsg);
      ++errMsg;
   }
}
```

Error Message Output



```
#include "ARMCM3.h"

void printErrorMsg(const char * errMsg)
{
    while(*errMsg != '\0'){
        ITM_SendChar(*errMsg);
        ++errMsg;
    }
}

void HardFault_Handler(void)
{
    printErrorMsg("In Hard Fault Handler");
    __ASM volatile("BKPT #01");
    while(1);
}

```



Debug (printf) Viewer

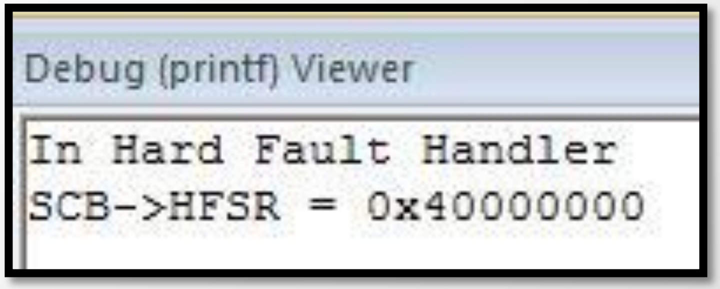
```
In Hard Fault Handler
```

Hard Fault Status Register (HFSR)

- Within the Cortex-M3's System Control Block (SCB) is the HardFault Status Register (HFSR).
- CMSIS defines symbols giving access these register (SCB->HFSR)

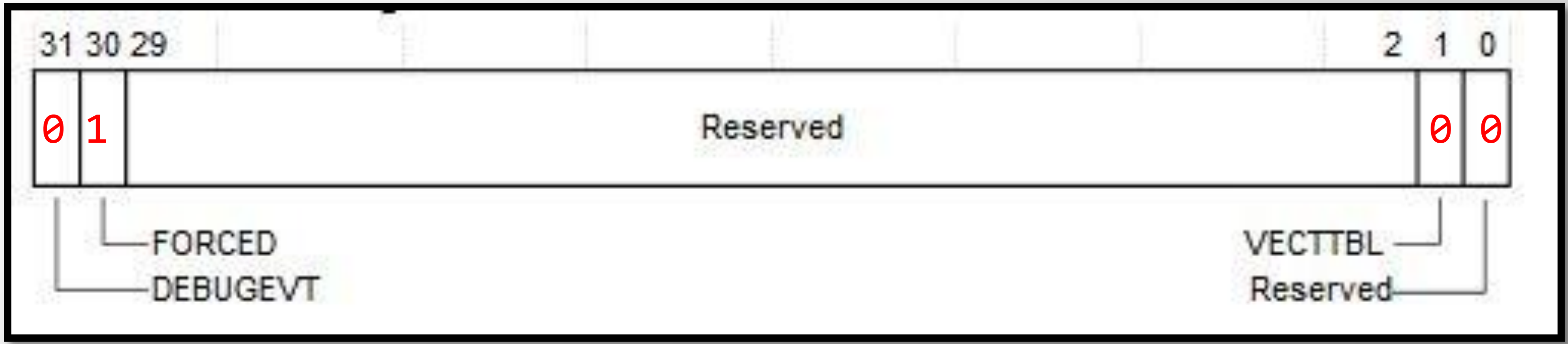
```
void HardFault_Handler(void)
{
   static char msg[80];
   printErrorMsg("In Hard Fault Handler\n");
   sprintf(msg, "SCB->HFSR = 0x%08x\n", SCB->HFSR);
   printErrorMsg(msg);
   __ASM volatile("BKPT #01");
   while(1);
}
```

Hard Fault Status Register (HFSR)



```
In Hard Fault Handler
SCB->HFSR = 0x40000000
```

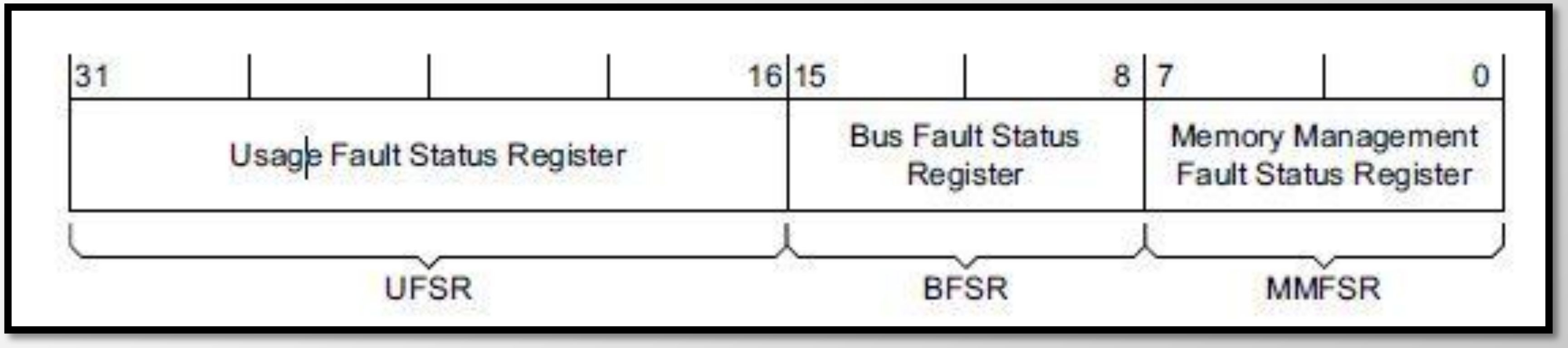
By examining the HFSR bit configuration, we can see that the FORCED bit is set.



When this bit is set to 1, the HardFault handler must read the other fault status registers to find the cause of the fault.

Configurable Fault Status Register (SCB->CFSR)

- A forced hard fault may be caused by a bus fault, a memory fault, or as in our case, a usage fault.
- For brevity, here we're only going to focus on the Usage Fault


31
16
15
8
7
0
Usage Fault Status Register
Bus Fault Status Register
Memory Management Fault Status Register
UFSR
BFSR
MMFSR

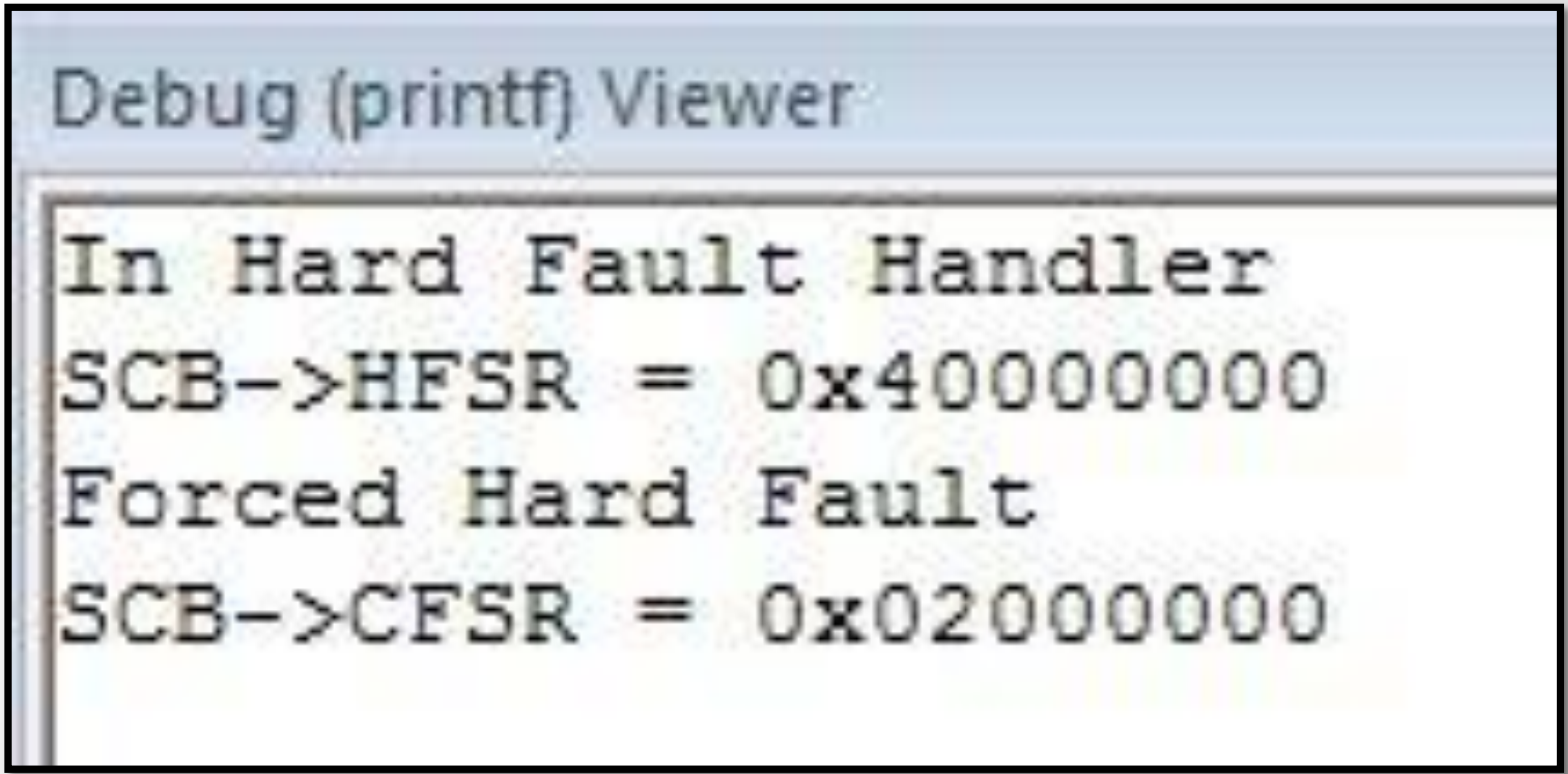
Updated Fault Handler for FORCED flag

So given what we know to date, our basic Fault Handler can be updated to:

- check if the **FORCED** bit is set (`if ((SCB->HFSR & (1 << 30)) != 0)`)
- and if so, print out the contents of the **CFSR**

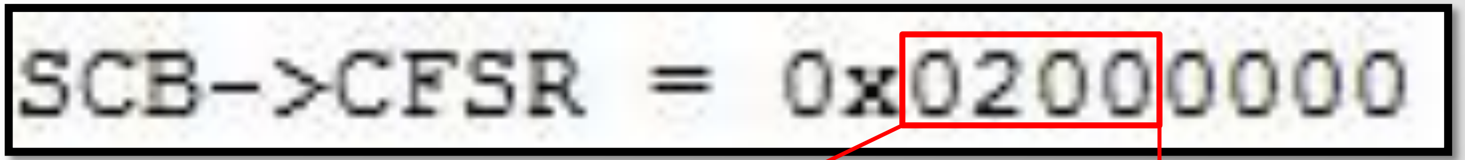
```
void HardFault_Handler(void)
{
   static char msg[80];
   printErrorMsg("In Hard Fault Handler\n");
   sprintf(msg, "SCB->HFSR = 0x%08x\n", SCB->HFSR);
   printErrorMsg(msg);
   if ((SCB->HFSR & (1 << 30)) != 0) {
      printErrorMsg("Forced Hard Fault\n");
      sprintf(msg, "SCB->CFSR = 0x%08x\n", SCB->CFSR );
      printErrorMsg(msg);
   }
   __ASM volatile("BKPT #01");
   while(1);
}
```

CFSR Output



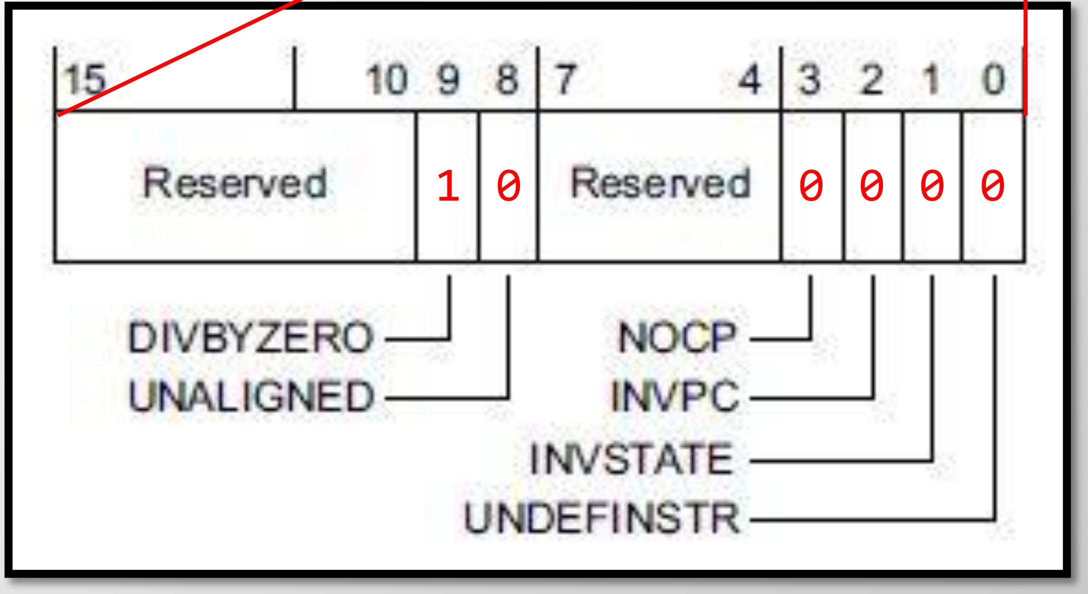
```
In Hard Fault Handler
SCB->HFSR = 0x40000000
Forced Hard Fault
SCB->CFSR = 0x02000000
```

Usage Fault Status Register (UFSR)



```
SCB->CFSR = 0x02000000
```

The bit configuration of the UFSR is shown below, and unsurprisingly the output shows that bit 9 (DIVBYZERO) is set.

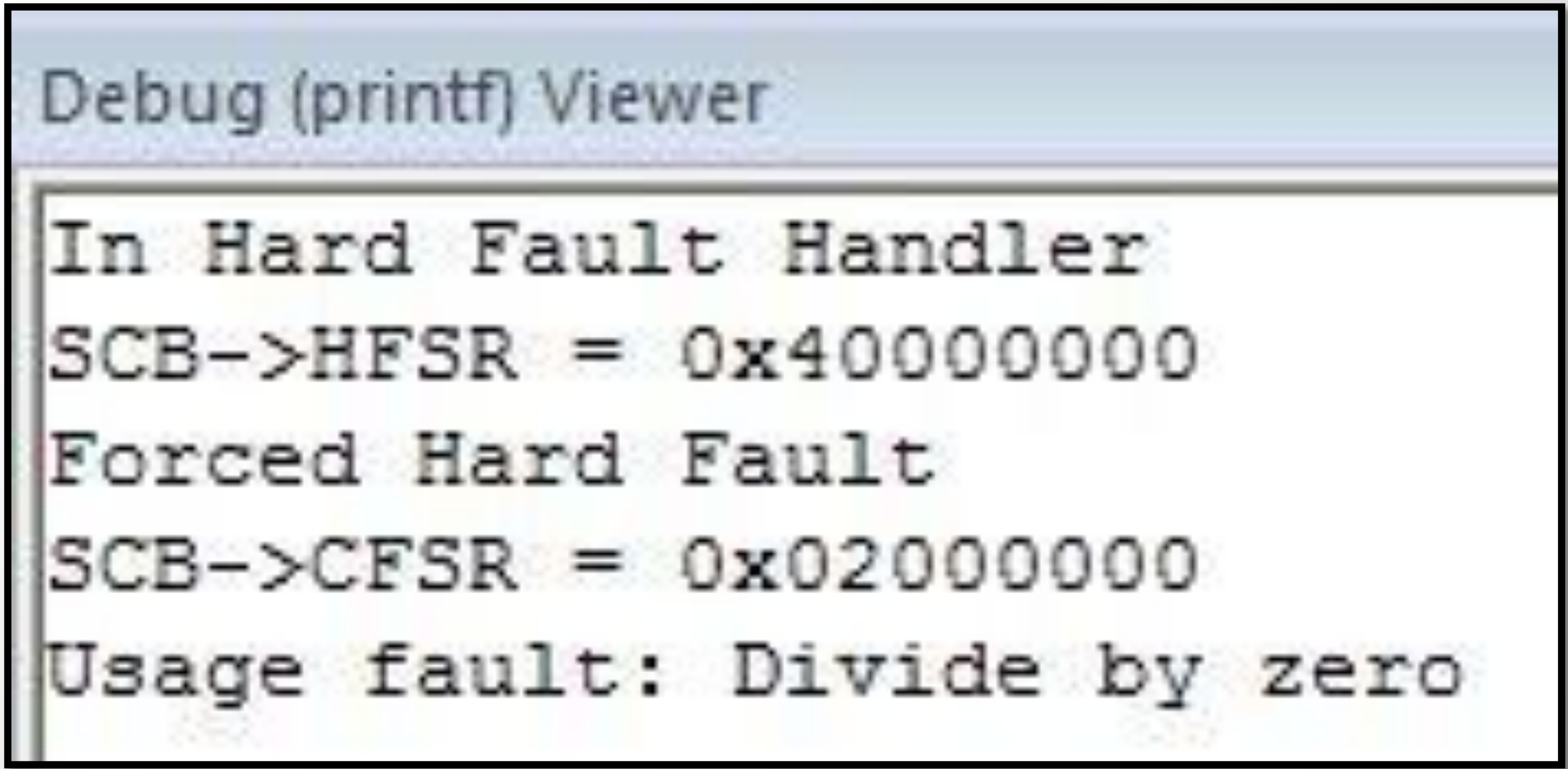


We can now extend the HardFault handler to mask the top half of the CFSR, and if not zero then further report on those flags, as in:

UFSR Fault Message

```
void HardFault_Handler(void)
{
   static char msg[80];
   printErrorMsg("In Hard Fault Handler\n");
   sprintf(msg, "SCB->HFSR = 0x%08x\n", SCB->HFSR);
   printErrorMsg(msg);
   if ((SCB->HFSR & (1 << 30)) != 0) {
      printErrorMsg("Forced Hard Fault\n");
      sprintf(msg, "SCB->CFSR = 0x%08x\n", SCB->CFSR );
      printErrorMsg(msg);
      if((SCB->CFSR & 0x030F0000) != 0) {
         printUsageErrorMsg(SCB->CFSR);
      }
   }
   __ASM volatile("BKPT #01");
   while(1);
}

void printUsageErrorMsg(uint32_t CFSRValue)
{
   printErrorMsg("Usage fault: ");
   CFSRValue >>= 16; // right shift to lsb

   if((CFSRValue & (1<<9)) != 0) {
      printErrorMsg("Divide by zero\n");
   }
}
```

Usage Fault Output



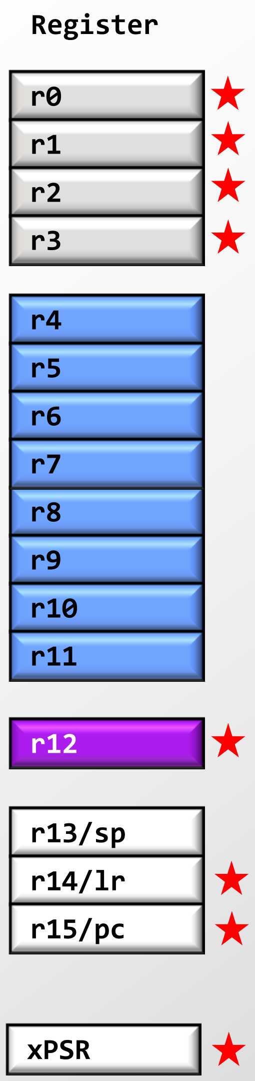
```
In Hard Fault Handler
SCB->HFSR = 0x40000000
Forced Hard Fault
SCB->CFSR = 0x02000000
Usage fault: Divide by zero
```

Register dump

- One final thing we can do as part of any fault handler is to dump out known register contents as they were at the time of the exception.

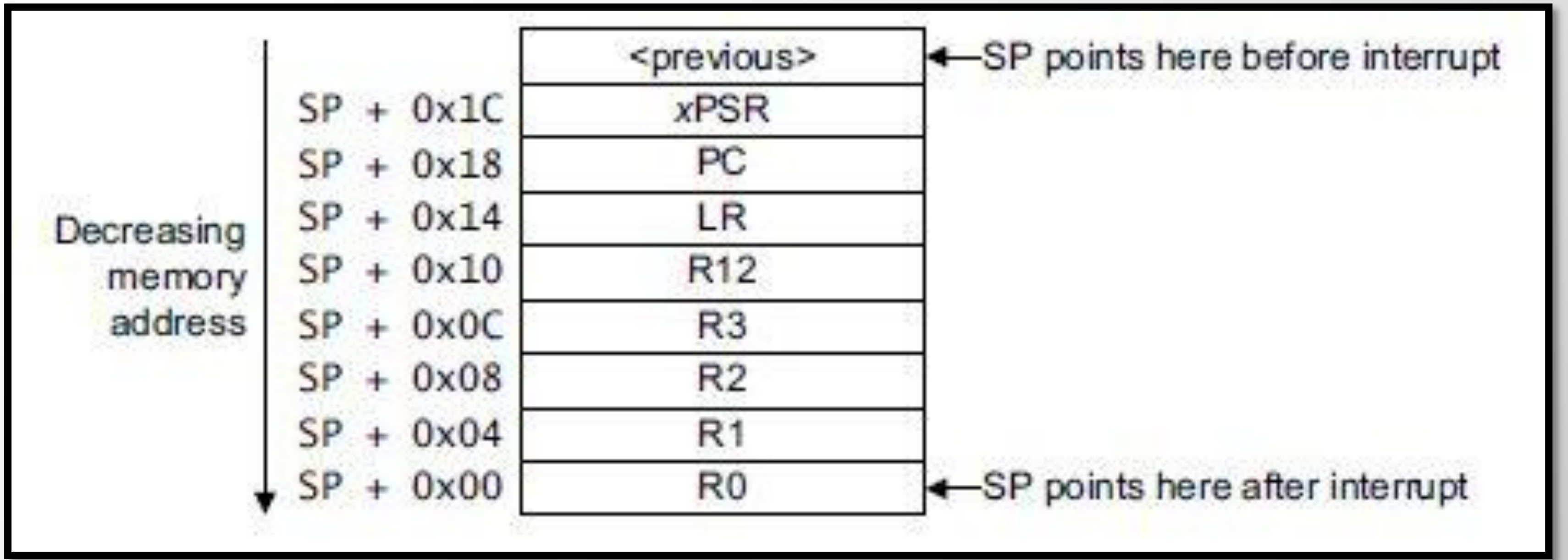
- One really useful feature of the Cortex-M architecture is that a core set of registers are automatically stacked (by the hardware) as part of the exception handling mechanism.

Interrupt Register Usage



- ISR functions do not require compiler specific directives
  - All managed in hardware
- Upon receiving an interrupt, the processor will finish current instruction
  - Some opcodes may be interrupted for better interrupt latency
- Then, processor state automatically saved to the stack over the data bus
  - {PC, R0-R3, R12, R14, xPSR}
- On Cortex-M3/M4, in parallel, the ISR address is prefetched on the instruction bus
  - ISR ready to start executing as soon as stack push is complete

Set of stacked registers



Procedure Call Standard for the ARM Architecture

- ARM have defined a set of rules for function entry/exit
- This is part of ARM's ABI and is referred to as the ARM Architecture Procedure Call Standard (AAPCS), e.g.

Register	Synonym	Role
R0	a1	Argument 1 / word result
R1	a2	Argument 2 / double-word result
R2	a3	Argument 3
R3	a4	Argument 4

  - More complex returns are passed via an address stored in a1

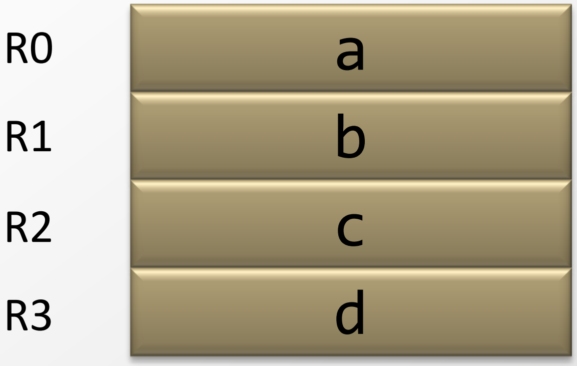
- Sub-word sized arguments will still use a whole register
  - char and short
- double-word arguments will be passed in multiple registers
  - double and long long

AAPCS Register Usage

	Register	
Arguments into function otherwise usable as scratch	r0	Result(s) from function
	r1	
	r2	
	r3	
Register variables Must be preserved	r4	
	r5	
	r6	
	r7	
	r8	
	r9	
	r10	
	r11	
Scratch register (corruptible)	r12	
Stack Pointer	r13/SP	
Link Register	r14/LR	
Program Counter	r15/PC	

Parameter Passing (4 Parameters or fewer)

```
int callerP4(void)
{
    return funcP4(1,2,3,4);
}
```



```
int funcP4(int a, int b, int c, int d)
{
    return a+b+c+d;
}
```

```
 callerP4 PROC
;;;1      int callerP4(void)
;;;2      {
;;;3          return funcP4(1,2,3,4);
000000  2304                MOVS     r3,#4
000002  2203                MOVS     r2,#3
000004  2102                MOVS     r1,#2
000006  2001                MOVS     r0,#1
000008  f7ffbffe            B.W      funcP4
;;;4      }
;;;5
                            ENDP
```

Using the AAPCS

- Using this knowledge in conjunction with AAPCS we can get access to these register values.

- First we modify our original HardFault handler by:
  - modifying it's name to "Hard_Fault_Handler" (no weak linkage)
  - adding a parameter declared as an array of 32-bit unsigned integers.

```
void Hard_Fault_Handler(uint32_t stack[])
```

- Based on AAPCS rules, we know that the parameter label (stack) will map onto register r0.

New HardFault_Handler

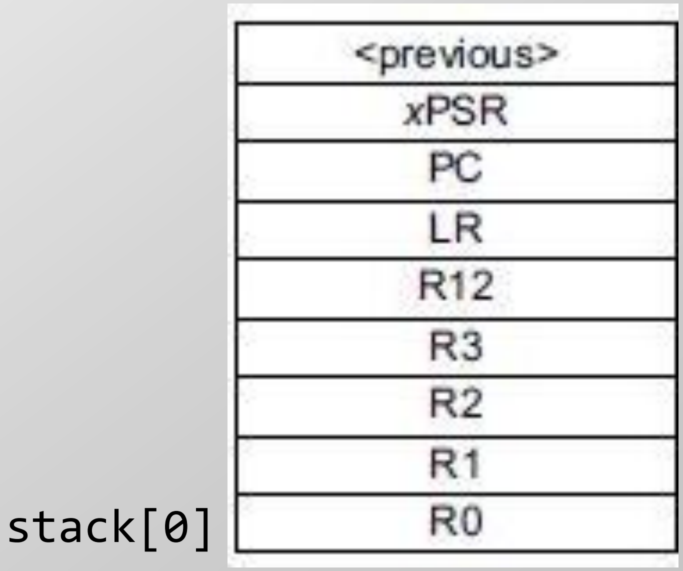
- We now implement the actual HardFault_Handler.
- This function simply copies the current Main Stack Pointer (MSP) into r0 and then branches to our renamed Hard_Fault_Handler (this is based on ARM/Keil syntax):

```
__asm void HardFault_Handler(void)
{
   MRS r0, MSP;
   B __cpp(Hard_Fault_Handler)
}
```

StackDump function

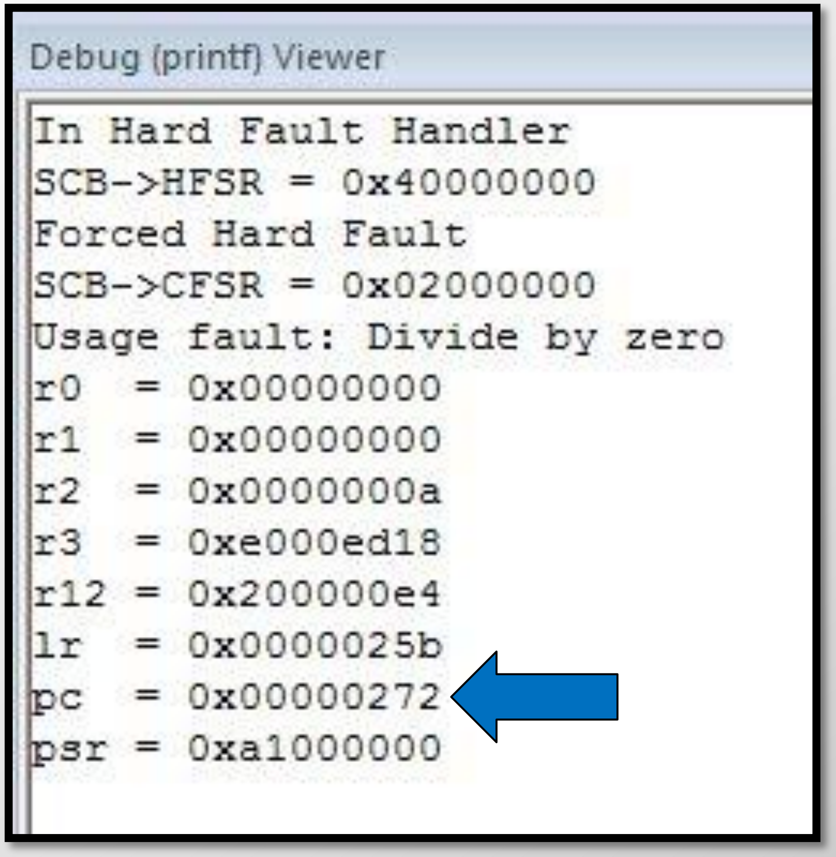
- Finally we implement a function to dump the stack values based on their relative offset

```
enum { r0, r1, r2, r3, r12, lr, pc, psr};

void stackDump(uint32_t stack[])
{
   static char msg[80];
   sprintf(msg, "r0  = 0x%08x\n", stack[r0]);
   printErrorMsg(msg);
   sprintf(msg, "r1  = 0x%08x\n", stack[r1]);
   printErrorMsg(msg);
   sprintf(msg, "r2  = 0x%08x\n", stack[r2]);
   printErrorMsg(msg);
   sprintf(msg, "r3  = 0x%08x\n", stack[r3]);
   printErrorMsg(msg);
   sprintf(msg, "r12 = 0x%08x\n", stack[r12]);
   printErrorMsg(msg);
   sprintf(msg, "lr  = 0x%08x\n", stack[lr]);
   printErrorMsg(msg);
   sprintf(msg, "pc  = 0x%08x\n", stack[pc]);
   printErrorMsg(msg);
   sprintf(msg, "psr = 0x%08x\n", stack[psr]);
   printErrorMsg(msg);
}
```



Stack Dump Output

This function can then be called from the Fault handler, passing through the stack argument. Running the program should result is the following output:



Debug (printf) Viewer

```
In Hard Fault Handler
SCB->HFSR = 0x40000000
Forced Hard Fault
SCB->CFSR = 0x02000000
Usage fault: Divide by zero
r0  = 0x00000000
r1  = 0x00000000
r2  = 0x0000000a
r3  = 0xe000ed18
r12 = 0x200000e4
lr  = 0x0000025b
pc  = 0x00000272
psr = 0xa1000000
```

Examining the output, we can see that the program counter (pc) is reported as being the value `0x00000272`, giving us the opcode generating the fault.

Tracing the PC

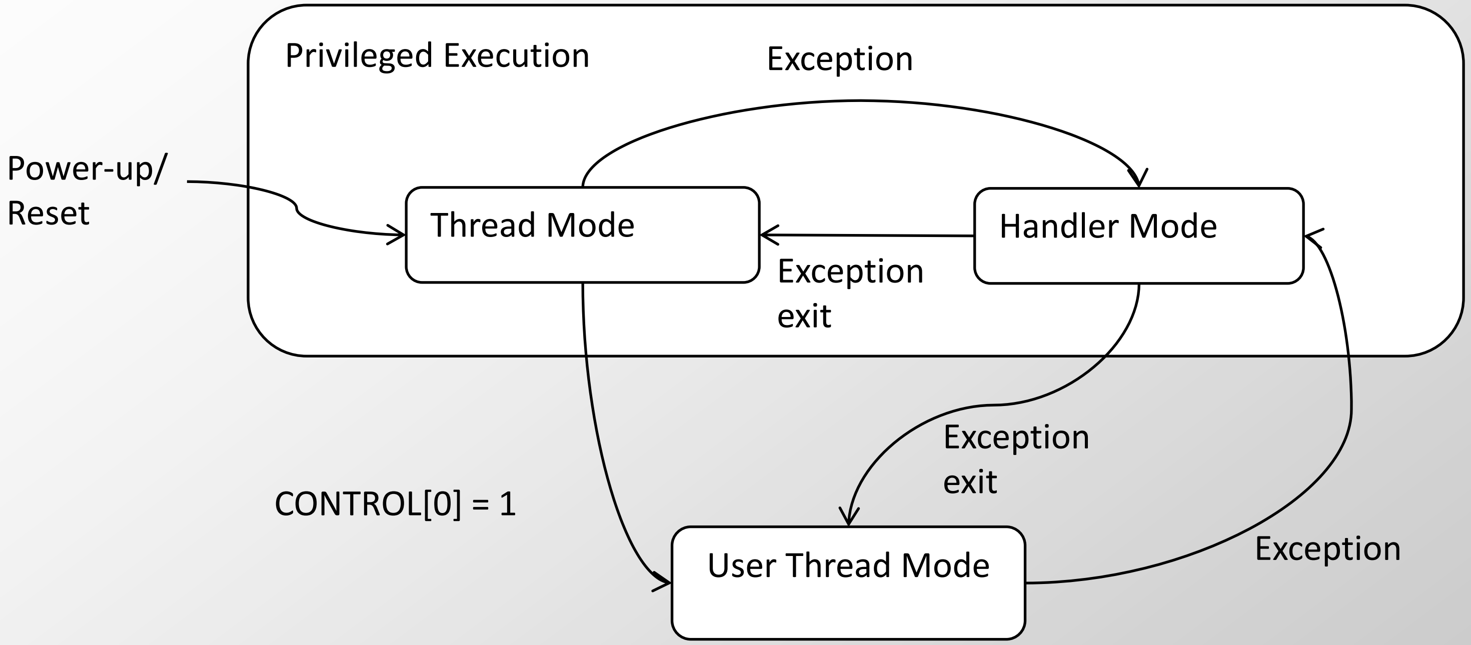
If we disassemble the image using the command:

```
fromelf -c CM3_Fault_Handler.axf –output listing.txt
```

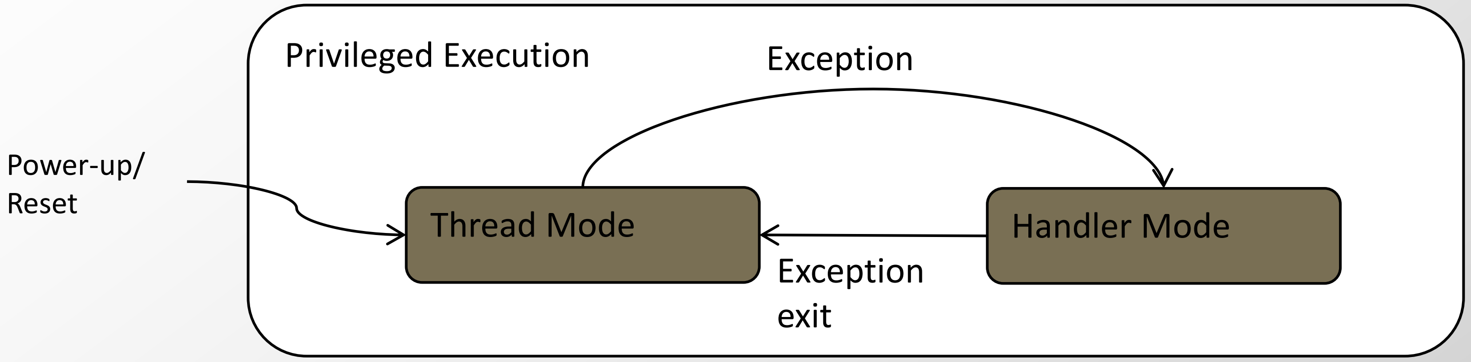
By trawling through the listing (listing.txt) we can see the SDIV instruction at offending line (note also r2 contains 10 and r1 the offending 0).

```
.text
div
0x00000270: 4602          MOV r2,r0
0x00000272: fb92f0f1      SDIV r0,r2,r1
0x00000276: 4770          BX lr
.text
```

Dealing with the Cortex-M Operational modes

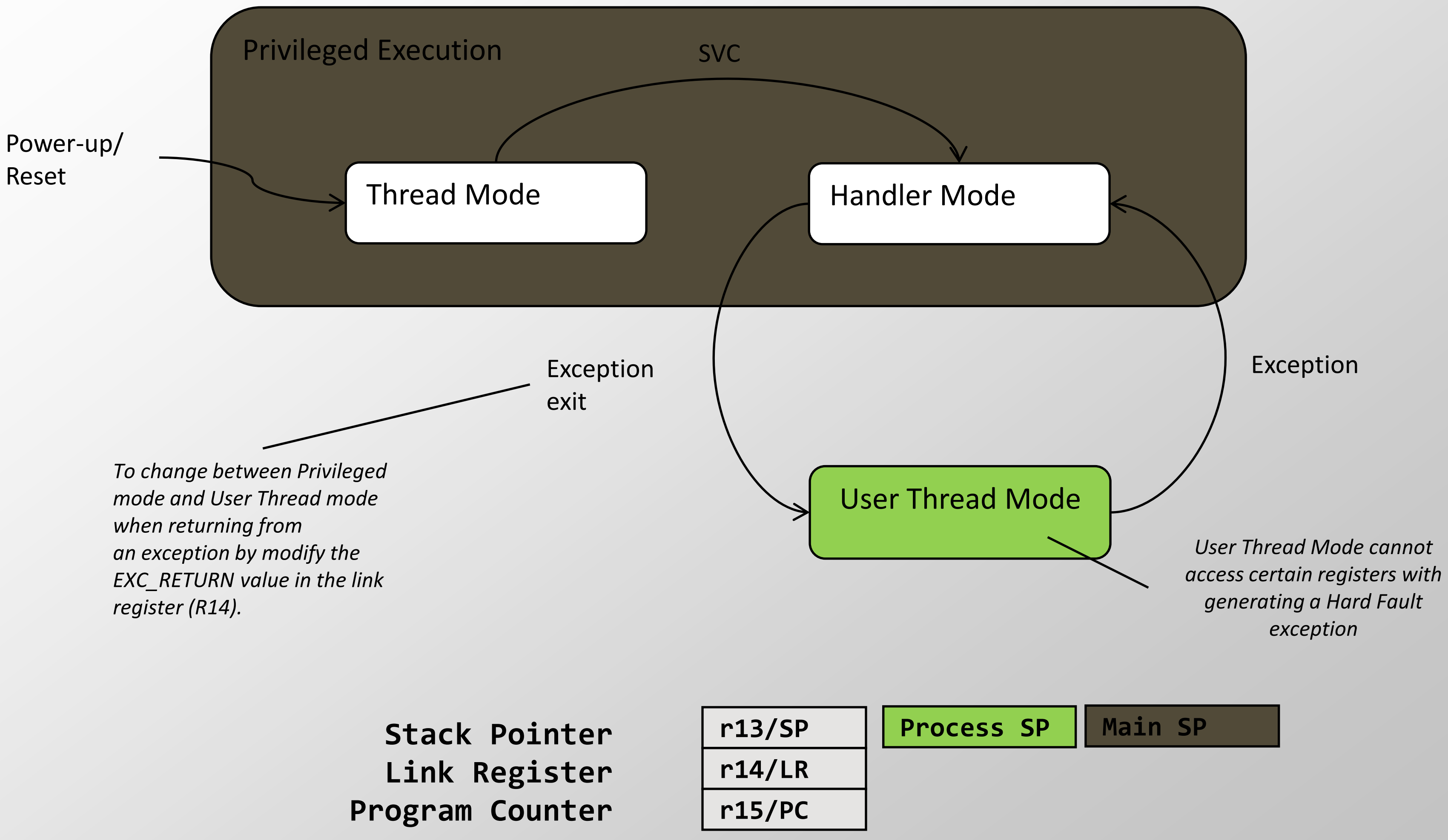


Operational modes – No RTOS



Privileged execution has full access to all processor, NVIC and MPU registers

Operational modes - RTOS



To change between Privileged mode and User Thread mode when returning from an exception by modify the EXC_RETURN value in the link register (R14).

User Thread Mode cannot access certain registers with generating a Hard Fault exception

Thread / Handler mode

- Finally, if you're going to use the privilege/non-privilege model, you'll need to modify the HardFault_Handler to detect whether the exception happened in Thread mode or Handler mode.
- This can be achieved by checking bit 3 of the HardFault_Handler's Link Register (lr) value.
- Bit 3 determines whether on return from the exception, the Main Stack Pointer (MSP) or Process Stack Pointer (PSP) is used.

```
__asm void HardFault_Handler(void)
{
   TST lr, #4     // Test for MSP or PSP
   ITE EQ
   MRSEQ r0, MSP
   MRSNE r0, PSP
   B __cpp(Hard_Fault_Handler)
}
```

Final Notes

- The initial model for fault handling can be found in Joseph Yiu's excellent book “The Definitive Guide to the ARM Cortex-M3”
- The code shown was built using the ARM/Keil MDK-ARM Version 4.60 development environment (a 32Kb size limited evaluation is available from the Keil website)
- The code, deliberately, has not been refactored to remove the hard-coded (magic) values.
- Code available on GitHub at git://github.com/feabhas/CM3_Fault_Handler

Thank You

Niall Cooling
Feabhas Limited
5 Lowesden Works
Lambourn Woodlands
Hungerford
Berks. RG17 7RY
UK
+44 1488 73050
www.feabhas.com

niall.cooling(at)feabhas.com
@feabhas
blog.feabhas.com
uk.linkedin.com/in/nscooling