Walkthrough Example

Cortex-M3 startup & exception handling

Learning objectives

At the end of this module, you will be able to:

- Explain how Tarmac trace can be used in conjunction with disassembly for debugging
- Describe how the Vector Table is used during startup
- Describe the overall flow of CMSIS startup code for Cortex-M3 processors
- Describe how to look for exceptions using Tarmac trace

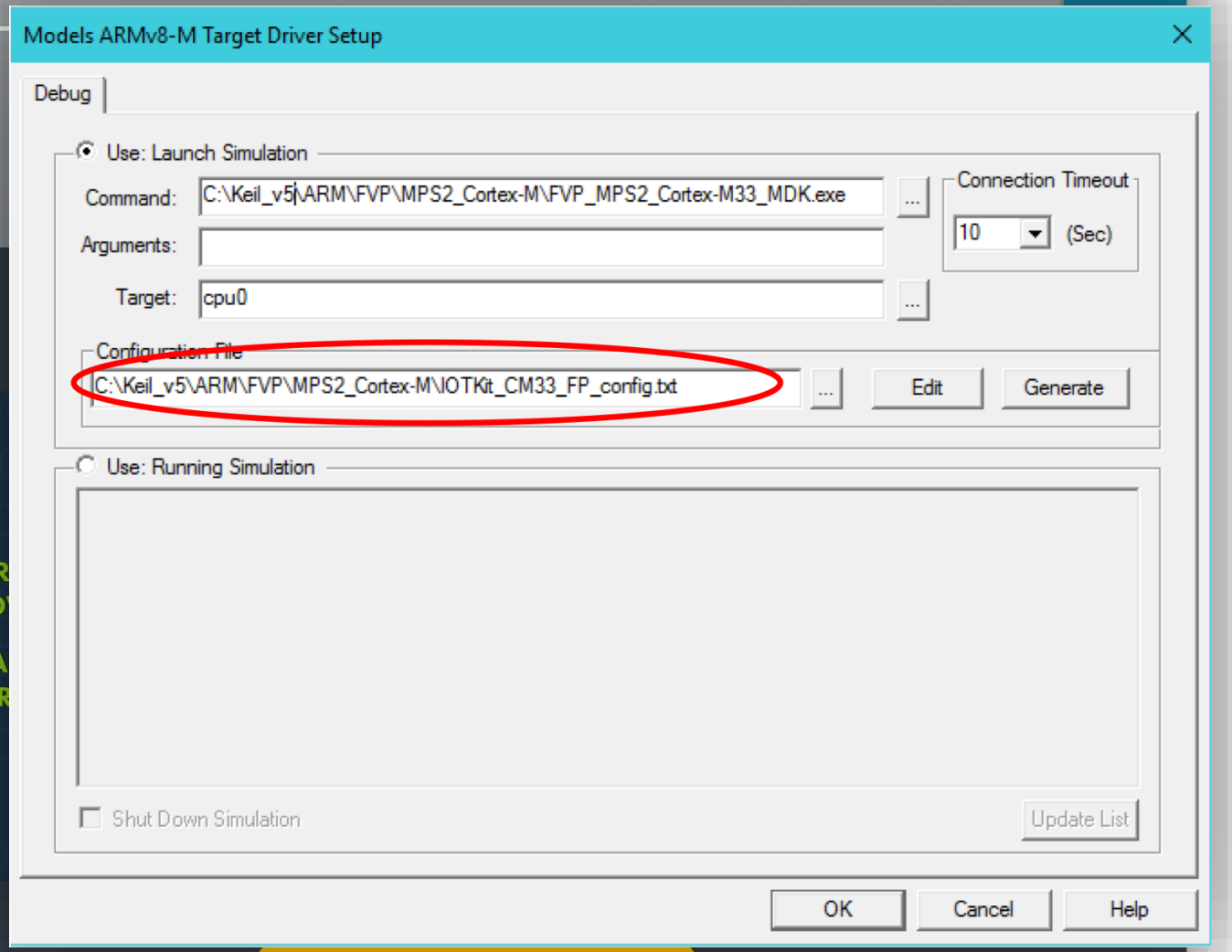
Tarmac trace

- Arm-specific trace format
- Can be collected from Fast Models / FVPs (Fixed Virtual Platforms)
- Can be collected from RTL (Register Transfer Level) simulations
- Shows instructions executed, register values transferred, memory accesses, and exceptions taken
- Documented in the Arm Fast Models documentation: https://developer.arm.com/docs/100964/1161/plug-ins-for-fast-models/tarmactrace

arm

DEVELOPMENT TOOLS AND SOFTWARE

FAST MODELS



Evaluate Fast Models

Tarmac trace example

```
8004 clk IT (8004) 0000061c 6801 T thread : LDR       r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

Tarmac trace example

```
8004 clk IT (8004) 0000061c 6801 T thread : LDR      r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

Tarmac trace example

```
8004 clk IT (8004) 0000061c 6801 T thread : LDR      r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

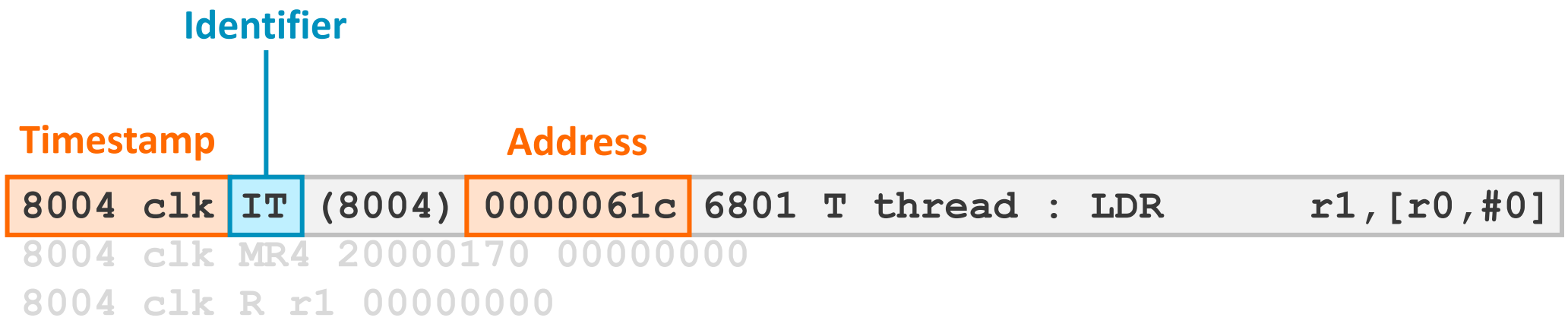
Tarmac trace example

Timestamp

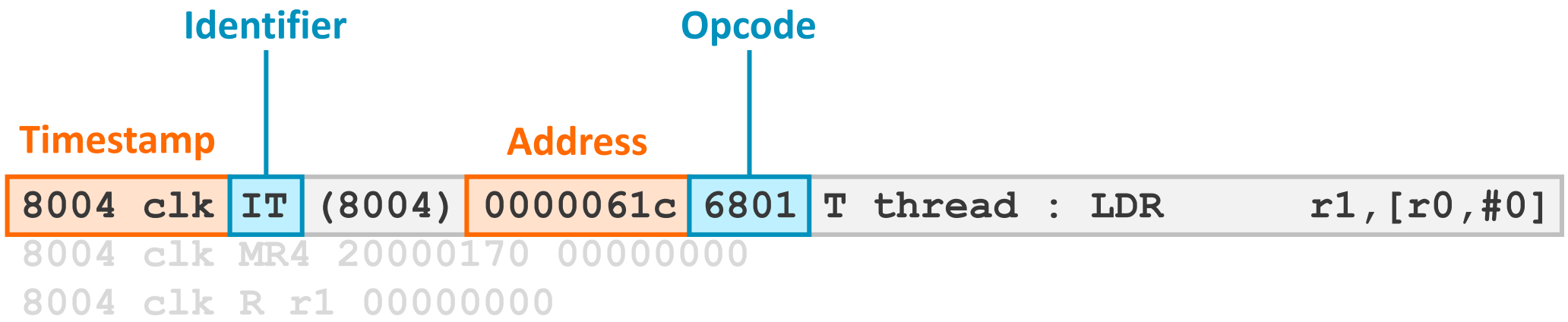
```
8004 clk IT (8004) 0000061c 6801 T thread : LDR      r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

Tarmac trace example

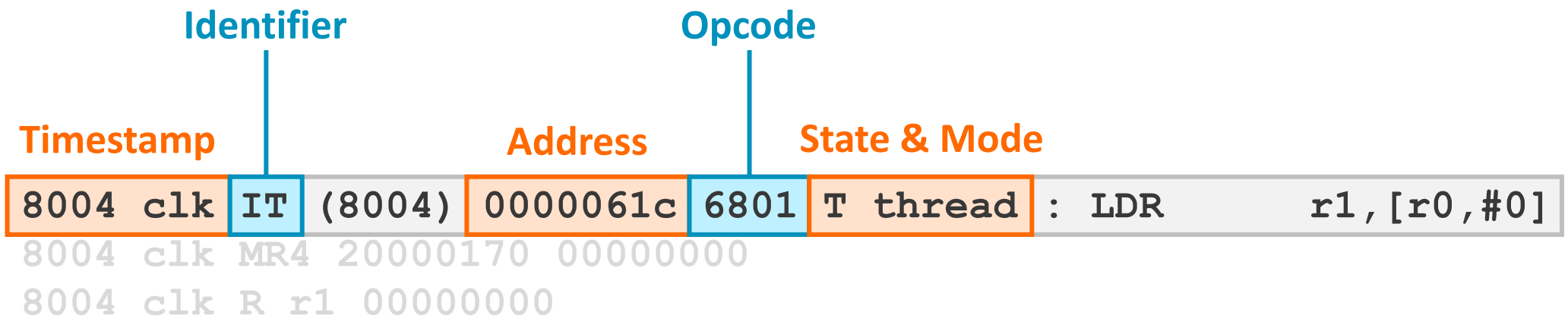
Tarmac trace example



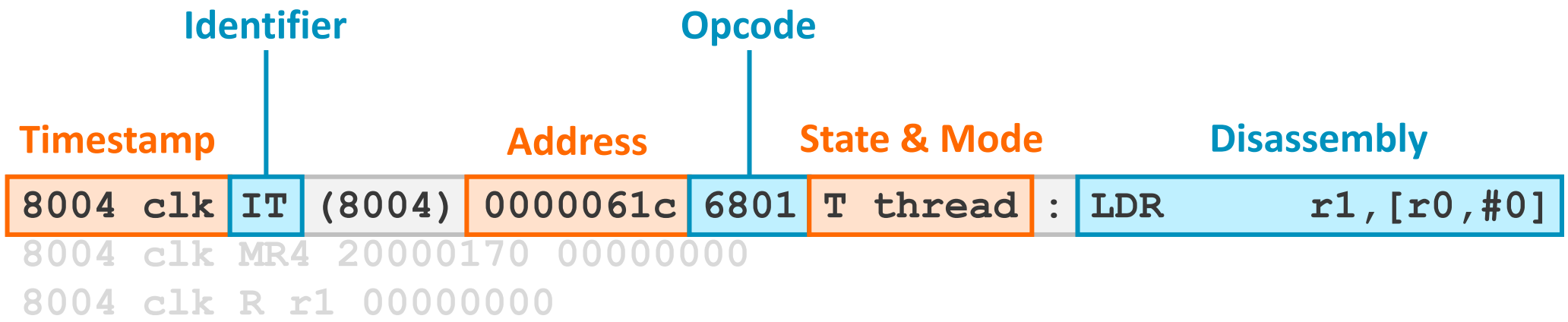
Tarmac trace example



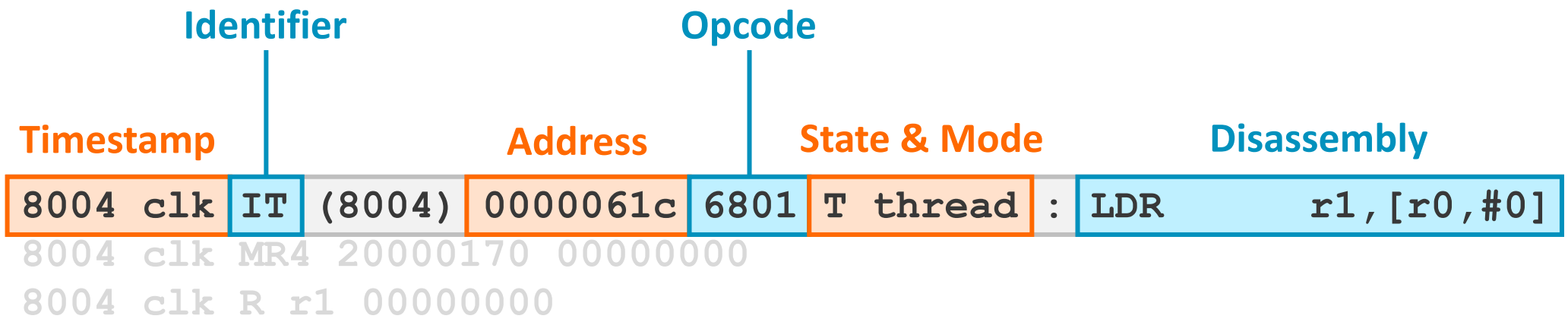
Tarmac trace example



Tarmac trace example



Tarmac trace example



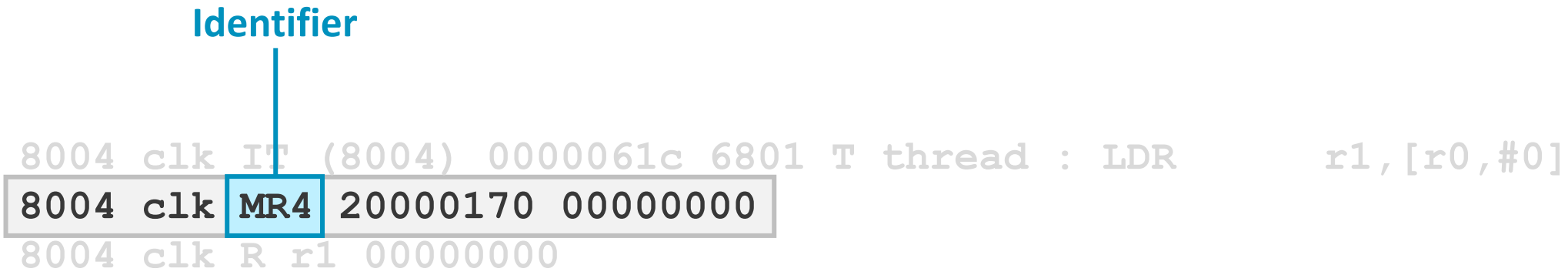
The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

Tarmac trace example

```
8004 clk IT (8004) 0000061c 6801 T thread : LDR      r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

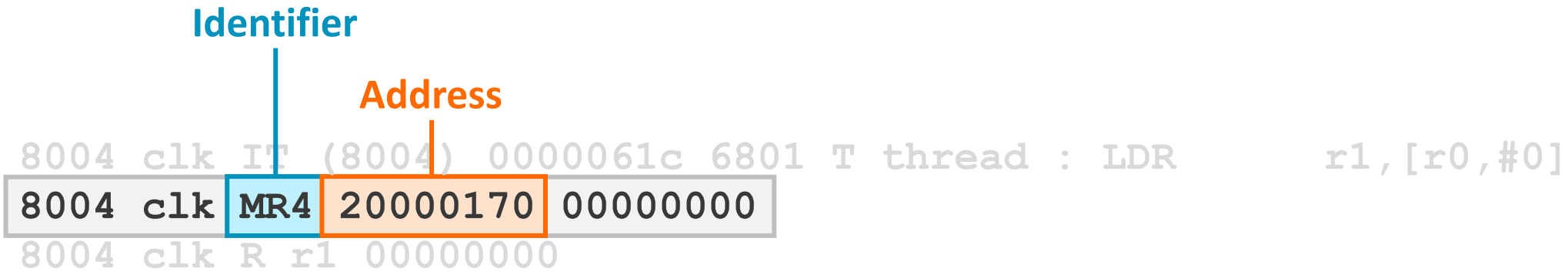
The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

Tarmac trace example



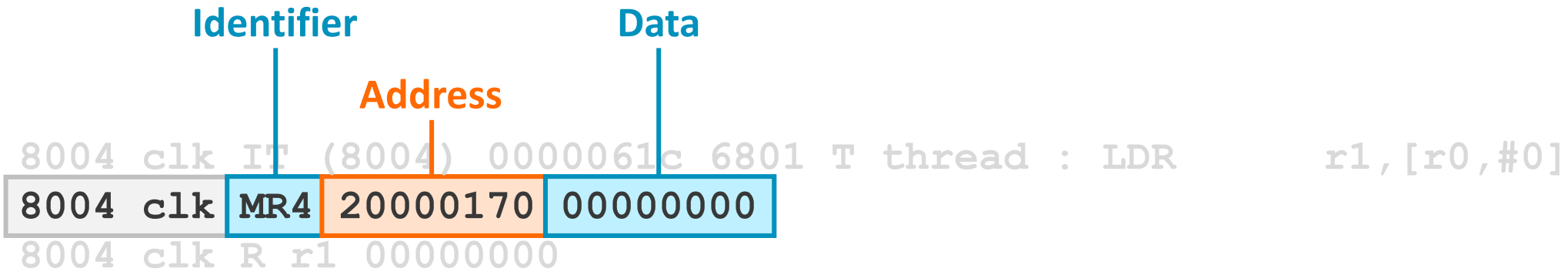
The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

Tarmac trace example



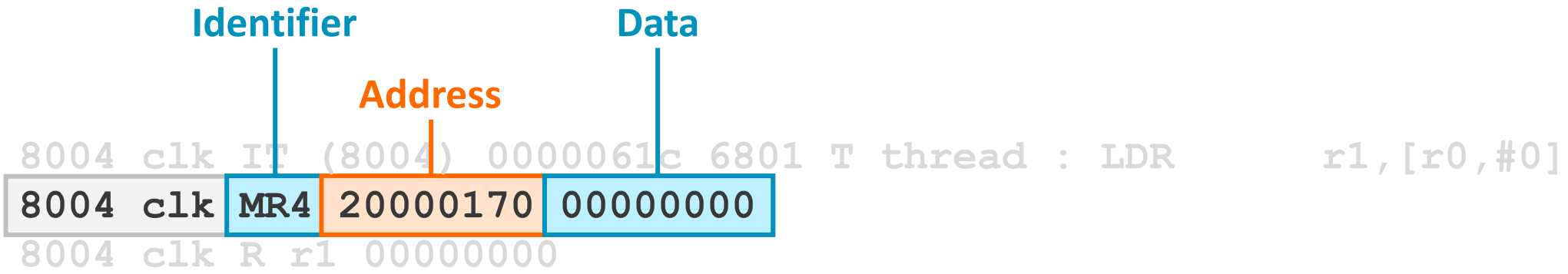
The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

Tarmac trace example



The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

Tarmac trace example



The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address `0x20000170`, and the value read was `0x0`.

Tarmac trace example

```
8004 clk IT (8004) 0000061c 6801 T thread : LDR      r1,[r0,#0]
8004 clk MR4 20000170 00000000
8004 clk R r1 00000000
```

The instruction at address **0x0000061c** was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address **0x20000170**, and the value read was **0x0**.

Tarmac trace example

The instruction at address **`0x0000061c`** was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address **`0x20000170`**, and the value read was **`0x0`**.

Tarmac trace example

The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address `0x20000170`, and the value read was `0x0`.

Tarmac trace example

The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address `0x20000170`, and the value read was `0x0`.

Tarmac trace example

The instruction at address `0x0000061c` was executed in T32 state and Thread mode at timestamp 8004.

It read 4 bytes from address `0x20000170`, and the value read was `0x0`.

The value `0x0` was written to register R1.

Question

Why is the timestamp the same for all of these lines?

```
7986 clk IT (7986) 000005fa b570 T thread : PUSH     {r4-r6,lr}
7986 clk MW4 20001120 20000064
7986 clk MW4 20001124 00412a02
7986 clk MW4 20001128 00000002
7986 clk MW4 2000112c 000008a5
7986 clk R r13_main 20001120
7986 clk R MSP 20001120
```

Startup example

- Uses startup code from CMSIS
- Sets up an interrupt handler in C
- Enables and triggers the interrupt using CMSIS functions

```
Hello, world!
Enabled device-specific timer 0
Hello from TIM0_IRQHandler()!
```

Step 0: Reset

Reset (0)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Reset (1)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Q. Where does the Reset Handler address come from?

Reset (2)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Q. Where does the Reset Handler address come from?

A. It is specified in the vector table.

Reset (3)

startup_ARMCM3.s

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

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
```

Reset (4)

startup_ARMCM3.s

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

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
```

Disassembly

```
RESET
    __Vectors
        0x00000000:    20001170    p..     DCD    536875376
        0x00000004:    0000016d    m...    DCD    365
```

Reset (5)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Q. Where does the Reset Handler address come from?

A. It is specified in the vector table.

Q. Why is the value `0x16d` in the vector table, but `0x16c` in Tarmac trace?

```
RESET
    __Vectors
        0x00000000:    20001170    p..      DCD    536875376
        0x00000004:    0000016d    m...     DCD    365
```

Reset (6)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Q. Where does the Reset Handler address come from?

A. It is specified in the vector table.

Q. Why is the value `0x16d` in the vector table, but `0x16c` in Tarmac trace?

A. Value must have bottom bit set for T32 state in table, but the actual address doesn't.

Reset (7)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Reset (8)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

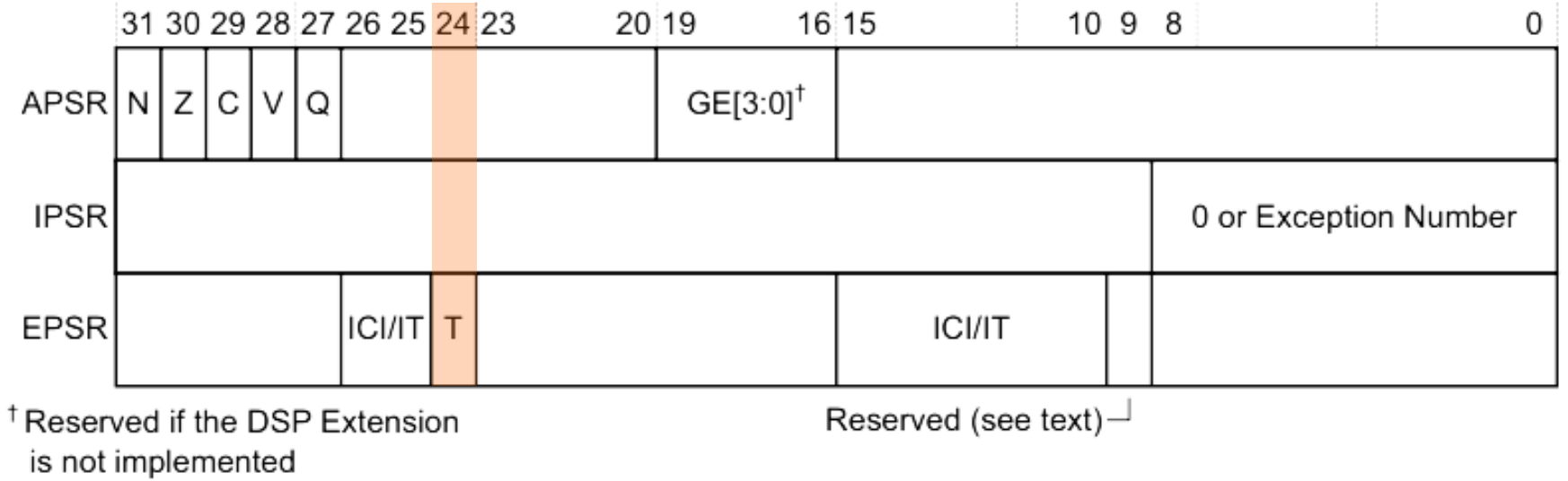


Figure B1-1 The PSR register layout

Reset (9)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

Reset (10)

```
0 clk E 0000016c 00000001 CoreEvent_RESET
0 clk R r13_main 20001170
0 clk R cpsr 01000000
0 clk R MSP 20001170
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
```

1. Reset exception occurs.
2. Stack pointer values set up.
3. CPSR value set to indicate the processor is in T32 state.
4. Start executing first instruction in the Reset Handler

Reset Handler (0)

startup_ARMCM3.s

```
Reset_Handler   PROC
                EXPORT  Reset_Handler                   [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

Reset Handler (1)

startup_ARMCM3.s

```
Reset_Handler   PROC
                EXPORT  Reset_Handler              [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

foo.c

```
/* Called by the CMSIS startup file.
   Perform any system required out-of-reset setup here.
 */
void SystemInit(void)
{
    return;
}
```

Reset Handler (2)

startup_ARMCM3.s

```
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

```
1 clk IT (1) 0000016c 4809 T thread : LDR      r0,{pc}+0x28 ; 0x194
1 clk MR4 00000194 00000c79
1 clk R r0 00000c79
2 clk IT (2) 0000016e 4780 T thread : BLX      r0
2 clk R r14 00000171
2 clk R cpsr 01000000
3 clk IT (3) 00000c78 4770 T thread : BX       lr
```

Reset Handler (3)

startup_ARMCM3.s

```
Reset_Handler   PROC
                EXPORT  Reset_Handler              [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

Q. What is the purpose of **`__main()`**? `_start()` with GCC

Reset Handler (4)

startup_ARMCM3.s

```
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

Q. What is the purpose of `__main()`? `_start()` with GCC

A. Entry point for the Arm C library, that performs memory initialization, and then executes application code.

arm

Step 1: __main()

What does `__main()` do? (0)

```
__main
        0x000000c0:    f000f802    ....    BL        __scatterload ; 0xc8
        0x000000c4:    f000f842    ..B.    BL        __rt_entry ; 0x14c
```

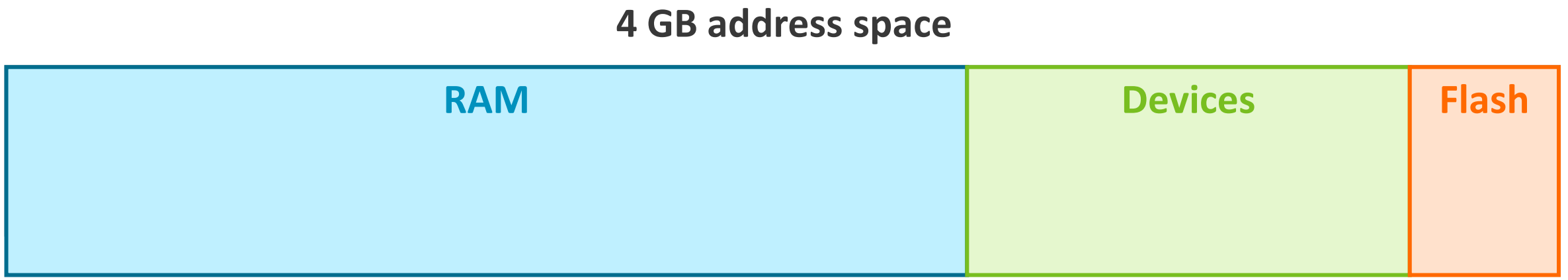
- Scatter-loading
- C library initialization
- Start executing application code

What does `__main()` do? (1)

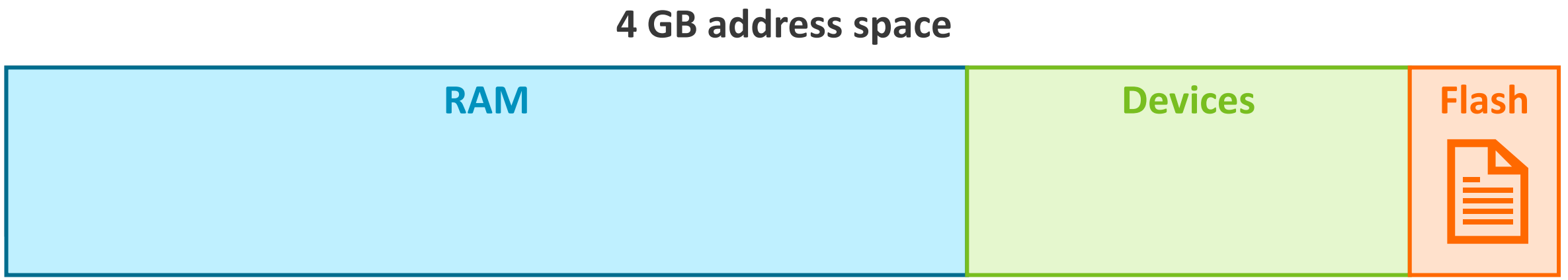
```
__main
        0x000000c0:    f000f802    ....    BL        __scatterload ; 0xc8
        0x000000c4:    f000f842    ..B.    BL        __rt_entry ; 0x14c
```

- Scatter-loading
- C library initialization
- Start executing application code

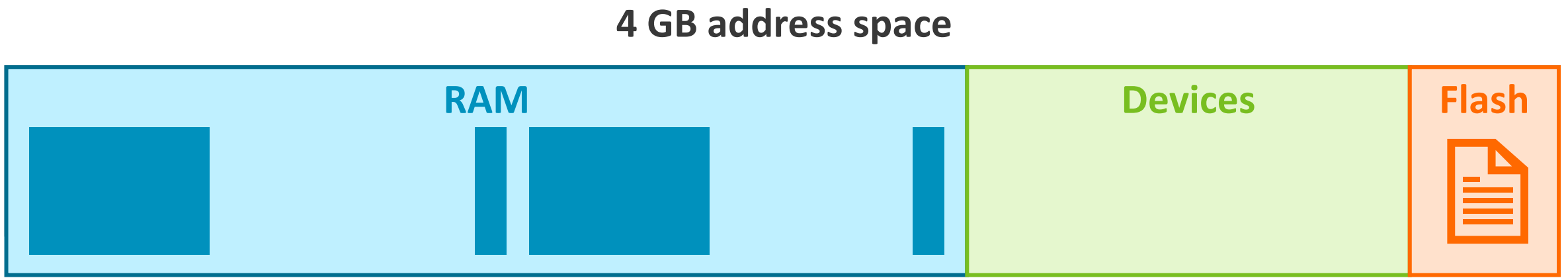
What is scatter-loading? (0)



What is scatter-loading? (1)



What is scatter-loading? (2)



What is scatter-loading? (3)

- Different types of data must be copied:
  - Un-initialized / zero-initialized variables
  - Initialized variables
  - Code itself
- For example, from Tarmac trace you can see zero-initialization being done:

```
49 clk IT (49) 00000124 c178 T thread : STMCS    r1!,{r3-r6}
49 clk MW4 20000010 00000000
49 clk MW4 20000014 00000000
49 clk MW4 20000018 00000000
49 clk MW4 2000001c 00000000
```

This sequence repeats 278 times!

What is scatter-loading? (4)

```
49 clk IT (49) 00000124 c178 T thread : STMCS    r1!,{r3-r6}
```

```
__scatterload_zeroinit
        0x00000118:    2300          .#      MOVS     r3,#0
        0x0000011a:    2400          .$      MOVS     r4,#0
        0x0000011c:    2500          .%      MOVS     r5,#0
        0x0000011e:    2600          .&      MOVS     r6,#0
        0x00000120:    3a10          .:      SUBS     r2,r2,#0x10
        0x00000122:    bf28          (.      IT       CS
        0x00000124:    c178          x.      STMCS    r1!,{r3-r6}
        0x00000126:    d8fb          ..      BHI      0x120 ; __scatterload_zeroinit + 8
        0x00000128:    0752          R.      LSLS     r2,r2,#29
        0x0000012a:    bf28          (.      IT       CS
        0x0000012c:    c130          0.      STMCS    r1!,{r4,r5}
        0x0000012e:    bf48          H.      IT       MI
        0x00000130:    600b          .`      STRMI    r3,[r1,#0]
        0x00000132:    4770          pG      BX       lr
```

What does `__main()` do? (2)

```
__main
        0x000000c0:    f000f802    ....    BL        __scatterload ; 0xc8
        0x000000c4:    f000f842    ..B.    BL        __rt_entry ; 0x14c
```

- Scatter-loading
- C library initialization
- Start executing application code

What does `__main()` do? (3)

```
__main
        0x000000c0:    f000f802    ....    BL        __scatterload ; 0xc8
        0x000000c4:    f000f842    ..B.    BL        __rt_entry ; 0x14c
```

- Scatter-loading
- C library initialization
- Start executing application code

```
__rt_entry
    0x0000014c:    f000fa01    ....    BL        __user_setup_stackheap ; 0x552
    0x00000150:    4611        .F      MOV       r1,r2
    0x00000152:    f7ffffef    ....    BL        __rt_lib_init ; 0x134
    0x00000156:    f000fda5    ....    BL        main ; 0xca4
    0x0000015a:    f000fbad    ....    BL        exit ; 0x8b8
```

Step 2: main()

Application code in `main()` (0)

```
int main(void)
{
    printf("Hello, world!\n");

    /* Initialize TIM0_IRQn */
    NVIC_SetPriority(TIM0_IRQn, 1);
    NVIC_EnableIRQ(TIM0_IRQn);
    printf("Enabled device-specific timer 0\n");

    /* Set TIM0_IRQn to pending */
    NVIC_SetPendingIRQ(TIM0_IRQn);

    return 0;
}
```

Application code in `main()` (1)

```
printf("Hello, world!\n");
```

```
1823 clk IT (1823) 00000532 beab T thread : BKPT      #0xab
```

- Fast Models support semihosting features for functions such as `printf()`

Application code in `main()` (2)

```
/* Initialize TIM0_IRQn */
NVIC_SetPriority(TIM0_IRQn, 1);
```

void NVIC_SetPriority (IRQn_Type IRQn,
uint32_t priority
)

Sets the priority for the interrupt specified by *IRQn.IRQn* can can specify any device specific interrupt, or processor exception. The *priority* specifies the interrupt priority value, whereby lower values indicate a higher priority. The default priority is 0 for every interrupt. This is the highest possible priority.

The priority cannot be set for every core interrupt. HardFault and NMI have a fixed (negative) priority that is higher than any configurable exception or interrupt.

Parameters

[in] **IRQn** Interrupt Number

[in] **priority** Priority to set

On Cortex-M3, TIM0_IRQn is Interrupt #2

Application code in `main()` (3)

```
/* Initialize TIM0_IRQn */
NVIC_SetPriority(TIM0_IRQn, 1);
```

```
2966 clk IT (2966) 00000cc0 7001 T thread : STRB     r1,[r0,#0]
2966 clk MW1 e000e402 20
2966 clk R NVIC_IPR0 00200000
```

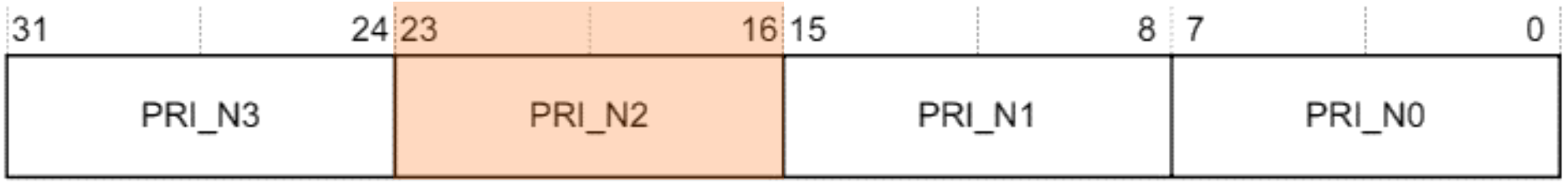
Table B3-8 NVIC register summary

Address	Name	Type	Reset	Description
0xE000E400- 0xE000E5EC	NVIC_IPR0- NVIC_IPR123	RW	0x00000000	*Interrupt Priority Registers, NVIC_IPR0-NVIC_IPR123* on page B3-686

Application code in `main()` (4)

```
/* Initialize TIM0_IRQn */
NVIC_SetPriority(TIM0_IRQn, 1);
```

```
2966 clk IT (2966) 00000cc0 7001 T thread : STRB     r1,[r0,#0]
2966 clk MW1 e000e402 20
2966 clk R NVIC_IPR0 00200000
```

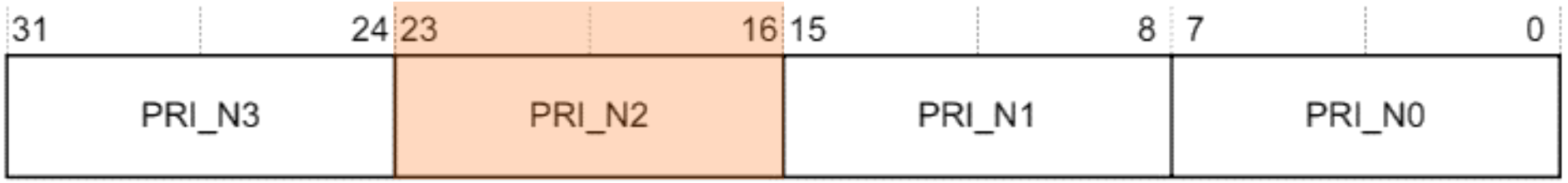


Q. How is interrupt priority of “1” equal to 0x20?

Application code in `main()` (5)

```
/* Initialize TIM0_IRQn */
NVIC_SetPriority(TIM0_IRQn, 1);
```

```
2966 clk IT (2966) 00000cc0 7001 T thread : STRB     r1,[r0,#0]
2966 clk MW1 e000e402 20
2966 clk R NVIC_IPR0 00200000
```



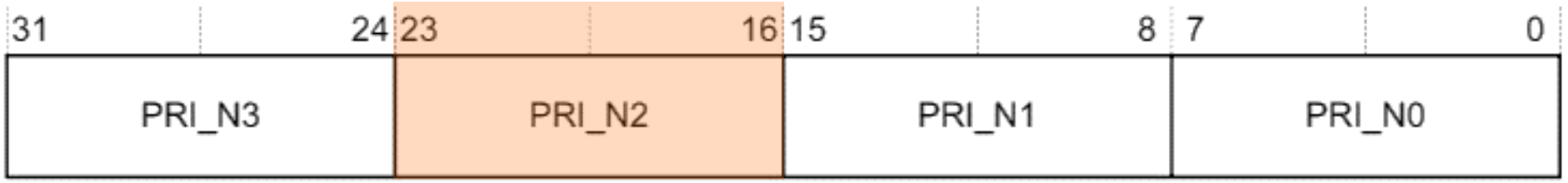
Q. How is interrupt priority of “1” equal to 0x20?

A. ~~The architecture can be quirky.~~ This implementation has only 3 bits for priority.

Application code in `main()` (6)

```
/* Initialize TIM0_IRQn */
NVIC_SetPriority(TIM0_IRQn, 1);
```

```
2966 clk IT (2966) 00000cc0 7001 T thread : STRB     r1,[r0,#0]
2966 clk MW1 e000e402 20
2966 clk R NVIC_IPR0 00200000
```



Q. How is interrupt priority of “1” equal to 0x20?

A. ~~The architecture can be quirky.~~ This implementation has only 3 bits for priority.

Step 3: Exception handling

Triggering an exception from software (1)

```
int main(void)
{
    printf("Hello, world!\n");

    /* Initialize TIM0_IRQn */
    NVIC_SetPriority(TIM0_IRQn, 1);
    NVIC_EnableIRQ(TIM0_IRQn);
    printf("Enabled device-specific timer 0\n");

    /* Set TIM0_IRQn to pending */
    NVIC_SetPendingIRQ(TIM0_IRQn);

    return 0;
}
```

- Set “pending” bit for the interrupt to 1.
- Processor takes exception and branches to exception handler.

Triggering an exception from software (2)

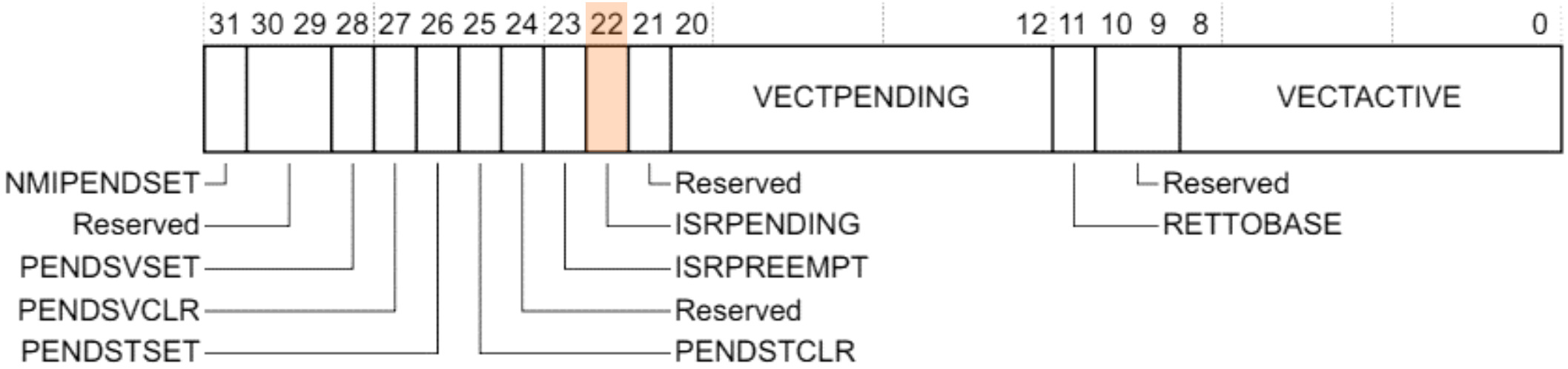
```
5484 clk IT (5484) 00000ccc f8c45100 T thread : STR      r5,[r4,#0x100]
5484 clk MW4 e000e200 00000004
5484 clk R NVIC_ISPR0 00000004
5484 clk R NVIC_ICPR0 00000004
5484 clk R ICSR 00412000
```

Table B3-8 NVIC register summary

Address	Name	Type	Reset	Description
0xE000E200- 0xE000E23C	NVIC_ISPR0- NVIC_ISPR15	RW	0x00000000	*Interrupt Set-Pending Registers, NVIC_ISPR0-NVIC_ISPR15* on page B3-685
0xE000E280- 0xE000E2BC	NVIC_ICPR0- NVIC_ICPR15	RW	0x00000000	*Interrupt Clear-Pending Registers, NVIC_ICPR0-NVIC_ICPR15* on page B3-685

Triggering an exception from software (3)

```
5484 clk IT (5484) 00000ccc f8c45100 T thread : STR      r5,[r4,#0x100]
5484 clk MW4 e000e200 00000004
5484 clk R NVIC_ISPR0 00000004
5484 clk R NVIC_ICPR0 00000004
5484 clk R ICSR 00412000
```



Handling an exception from software (1)

```
5484 clk MW4 2000115c 41000000
5484 clk MW4 20001158 00000cd0
5484 clk MW4 20001154 000006dd
5484 clk MW4 20001150 00000000
5484 clk MW4 2000114c 00412a02
5484 clk MW4 20001148 00000020
5484 clk MW4 20001144 0000002e
5484 clk MW4 20001140 00000000
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2
5484 clk R r14 fffffff9
5484 clk R r13_main 20001140
5484 clk R cpsr 41000012
5484 clk R MSP 20001140
5484 clk R CONTROL 00000000
5484 clk R NVIC_ISPR0 00000000
5484 clk R NVIC_ICPR0 00000000
5484 clk R NVIC_IABR0 00000004
5484 clk R ICSR 00000812
```

Handling an exception from software (2)

```
5484 clk MW4 2000115c 41000000
5484 clk MW4 20001158 00000cd0
5484 clk MW4 20001154 000006dd
5484 clk MW4 20001150 00000000
5484 clk MW4 2000114c 00412a02
5484 clk MW4 20001148 00000020
5484 clk MW4 20001144 0000002e
5484 clk MW4 20001140 00000000
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2
5484 clk R r14 fffffff9
5484 clk R r13_main 20001140
5484 clk R cpsr 41000012
5484 clk R MSP 20001140
5484 clk R CONTROL 00000000
5484 clk R NVIC_ISPR0 00000000
5484 clk R NVIC_ICPR0 00000000
5484 clk R NVIC_IABR0 00000004
5484 clk R ICSR 00000812
```

The processor saves the required registers to the stack automatically.

Handling an exception from software (2)

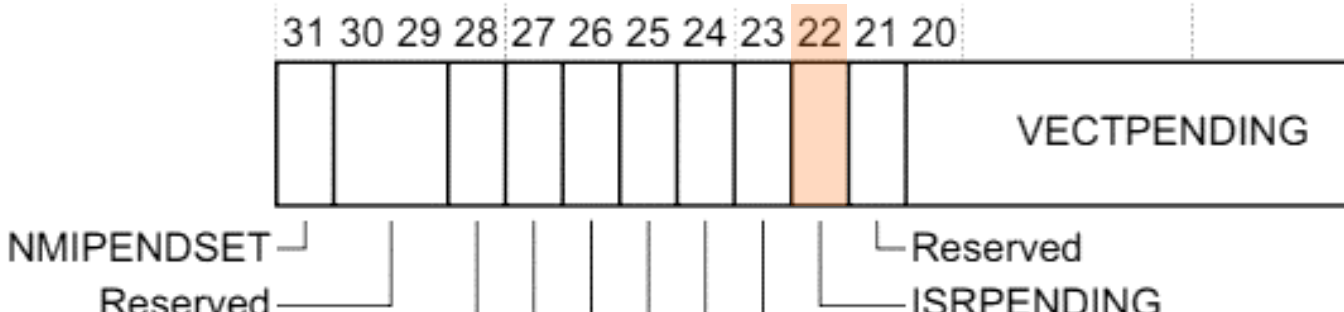
```
5484 clk MW4 2000115c 41000000
5484 clk MW4 20001158 00000cd0
5484 clk MW4 20001154 000006dd
5484 clk MW4 20001150 00000000
5484 clk MW4 2000114c 00412a02
5484 clk MW4 20001148 00000020
5484 clk MW4 20001144 0000002e
5484 clk MW4 20001140 00000000
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2
5484 clk R r14 fffffff9
5484 clk R r13_main 20001140
5484 clk R cpsr 41000012
5484 clk R MSP 20001140
5484 clk R CONTROL 00000000
5484 clk R NVIC_ISPR0 00000000
5484 clk R NVIC_ICPR0 00000000
5484 clk R NVIC_IABR0 00000004
5484 clk R ICSR 00000812
```

External interrupt #2 shown in Tarmac trace

Handling an exception from software (2)

```
5484 clk MW4 2000115c 41000000
5484 clk MW4 20001158 00000cd0
5484 clk MW4 20001154 000006dd
5484 clk MW4 20001150 00000000
5484 clk MW4 2000114c 00412a02
5484 clk MW4 20001148 00000020
5484 clk MW4 20001144 0000002e
5484 clk MW4 20001140 00000000
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2
5484 clk R r14 fffffff9
5484 clk R r13_main 20001140
5484 clk R cpsr 41000012
5484 clk R MSP 20001140
5484 clk R CONTROL 00000000
5484 clk R NVIC_ISPR0 00000000
5484 clk R NVIC_ICPR0 00000000
5484 clk R NVIC_IABR0 00000004
5484 clk R ICSR 00000812
```

ISRPENDING bit of ICSR cleared



Handling an exception from software (3)

```
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2

…

5485 clk IT (5485) 00000c7c a001 T handler : ADR      r0,{pc}+8 ; 0xc84

…

7845 clk INFO_EXCEPTION_REASON: PHASE=DEACTIVATE REASONS=UNSPECIFIED PC=0x00000518
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
7845 clk R cpsr 41000000
7846 clk IT (7846) 00000cd0 bdb0 T thread : POP      {r4,r5,r7,pc}
```

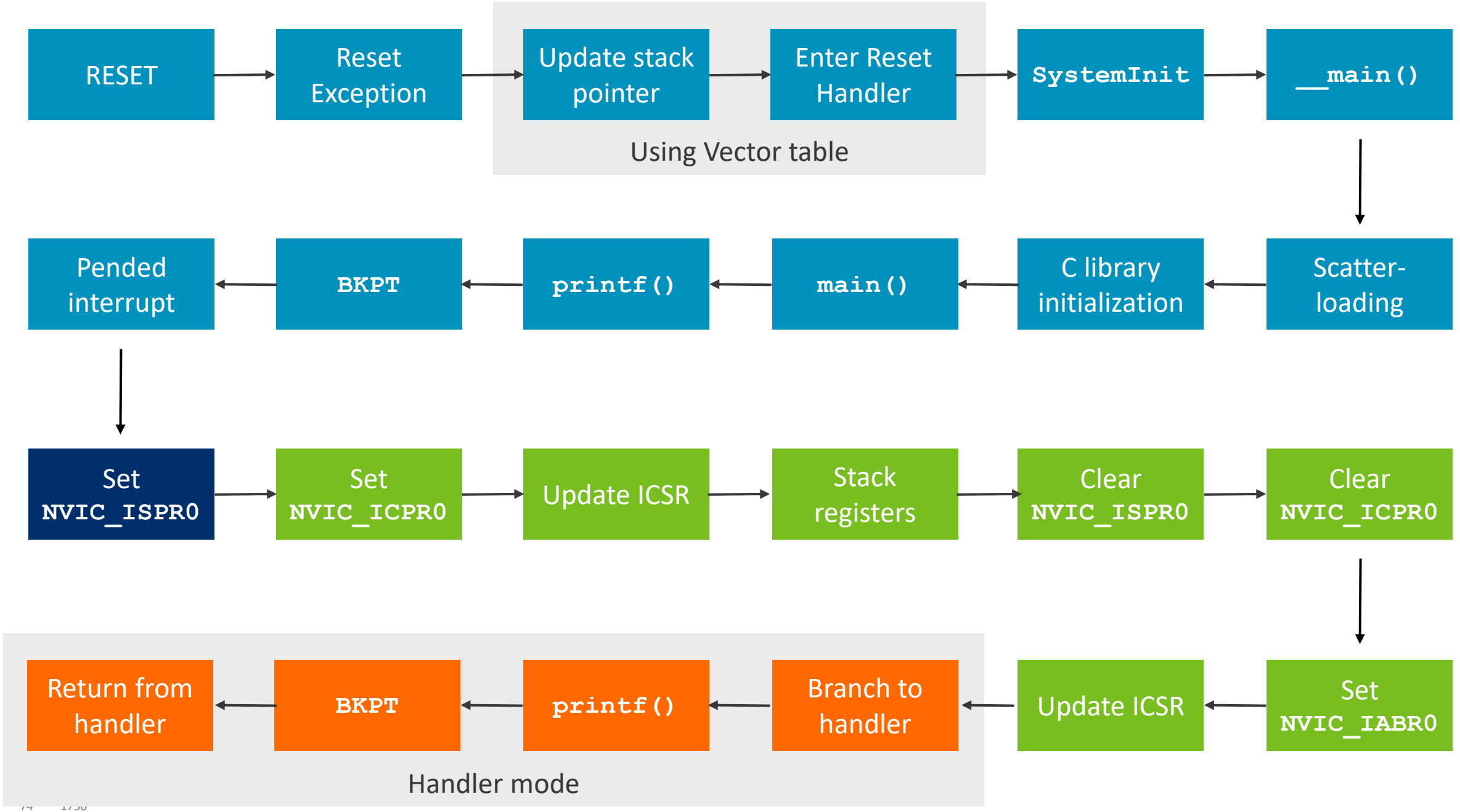
Handling an exception from software (3)

```
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2

…

5485 clk IT (5485) 00000c7c a001 T handler : ADR      r0,{pc}+8 ; 0xc84

…

7845 clk INFO_EXCEPTION_REASON: PHASE=DEACTIVATE REASONS=UNSPECIFIED PC=0x00000518
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
7845 clk R cpsr 41000000
7846 clk IT (7846) 00000cd0 bdb0 T thread : POP      {r4,r5,r7,pc}
```

Handling an exception from software (3)

```
5484 clk INFO_EXCEPTION_REASON: PHASE=ACTIVATE REASONS=UNSPECIFIED PC=0x00000cd0
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
5484 clk E 00000cd0 00000012 CoreEvent_EXT_INT2

…

5485 clk IT (5485) 00000c7c a001 T handler : ADR      r0,{pc}+8 ; 0xc84

…

7845 clk INFO_EXCEPTION_REASON: PHASE=DEACTIVATE REASONS=UNSPECIFIED PC=0x00000518
         VECTOR=EXT_INT2 FaultCause=NO_FSR_BIT
7845 clk R cpsr 41000000
7846 clk IT (7846) 00000cd0 bdb0 T thread : POP      {r4,r5,r7,pc}
```

Let’s Recap


RESET
Reset Exception
Update stack pointer
Enter Reset Handler
Using Vector table
SystemInit
__main()
Scatter-loading
C library initialization
main()
printf()
BKPT
Pended interrupt
Set NVIC_ISPR0
Set NVIC_ICPR0
Update ICSR
Stack registers
Clear NVIC_ISPR0
Clear NVIC_ICPR0
Set NVIC_IABR0
Update ICSR
Branch to handler
printf()
BKPT
Return from handler
Handler mode

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה