ARM Cortex-M4 Exceptions

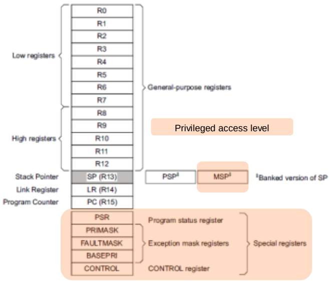
Prof. Hugo Vieira Neto

2020/1

Objective

- Study the main concepts of the exception model in the ARM Cortex-M4:
  - Thread mode e handler mode
  - Stacking, unstacking and lazy stacking
  - Tail-chaining, late arriving and POP preemption

Registers



MSP = main SP
O.S. kernel
Exceptions

PSP = process SP
Applications
(threads)

CONTROL
Selects SP
Selects PL

PRIMASK/BASEPRI
Interrupt
masking

Stack Pointer

- One may work only with MSP
- The current SP is accessed as R13 or SP
- The SP is always aligned on 32 bits (i.e. addresses which are multiples of 4)
- Instructions:
  - PUSH (stacks)
  - POP (unstacks)
- Full descending stack

Link Register

- Accessed as R14 or LR
- Stores the return address of a subroutine
- Must be saved before calling another subroutine

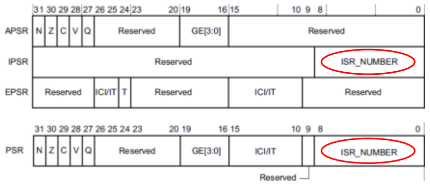
Program Counter

- Accessed as R15 or PC
- The PC contains the address of the next instruction to be executed (reason: pipeline)
- The address of an instruction is always even (bit0 = 0)
- PC bit0 is used to indicate Thumb mode
  - In branches, bit 0 of the PC must always be 1, otherwise an exception will be raised.

Special Registers

- Program Status Registers (xPSR):
  - Application Program Status Register (APSR) – RW
  - Interrupt Program Status Register (IPSR) – RO
  - Execution Program Status Register (EPSR) – RO
- Access through specific instructions:
  - MRS R0, APSR
  - MSR APSR, R0

Program Status Registers



Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception number	Indicates which exception the processor is handling

Interrupt Program Status Register

- "ISR_NUMBER" is different from IRQ number!

Bits	Name	Function
[31:6]	-	Reserved
[5:0]	Exception number	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4-10 = Reserved 11 = SVCall 12, 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0 . . .

Add 16

Special Registers

- Interrupt Mask Registers:
  - PRIMASK: used to enable / disable all interrupts and exceptions (except NMI and HardFault)
  - FAULTMASK: used to enable / disable fault exceptions
  - BASEPRI: defines interrupt masking from a priority base threshold
- Instrutions MRS, MSR, CPSIE e CPSID (change processor state + interrupt enable/disable)

Exception Masking

- BASEPRI = 0, PRIMASK = 0, FAULTMASK = 0
  - Priority level 256
  - All exceptions (priority <256) are serviced
- BASEPRI = X > 0, PRIMASK = 0, FAULTMASK = 0
  - Priority level X
  - Only exceptions with priority < X are serviced
- PRIMASK = 1, FAULTMASK = 0
  - Priority level 0
  - Only Reset, NMI and Faults (priority < 0) are serviced
- PRIMASK = 1, FAULTMASK = 1
  - Priority level -1
  - Only Reset and NMI (priority < -1) are serviced

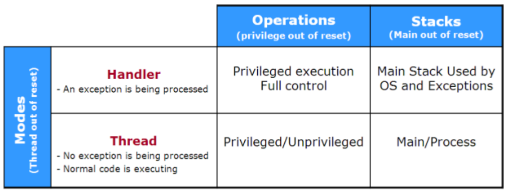
Special Registers

- CONTROL
  - Bit 1: defines the stack pointer (MSP or PSP)
  - Bit 0: defines the access level (Privileged or User)
- Bit 0 has write-only access in privileged level – once in user level, the only way to return to privileged level is via an exception.
- Access by MRS and MSR instructions
  - Use the Instruction Synchronization Barrier (ISB) instruction after the MSR instruction for immediate use of the new stack pointer

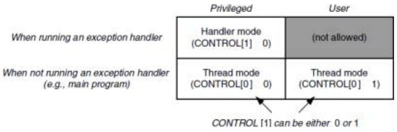
Modes, Privileges and Stacks

- Thread Mode / Handler Mode
  - Thread Mode: normal application execution
  - Handler Mode: exceptions or interrupts
- Privileged / non-privileged execution
  - Thread Mode: can be privileged or not
  - Handler Mode: always privileged
- Main Stack / Process Stack
  - Both stacks have their own pointer
  - Exceptions always use MSP in handler mode
  - Applications (thread mode) use the MSP or the PSP

Modes, Privileges and Stacks



		Operations (privilege out of reset)	Stacks (Main out of reset)
Modes (Thread out of reset)	Handler - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

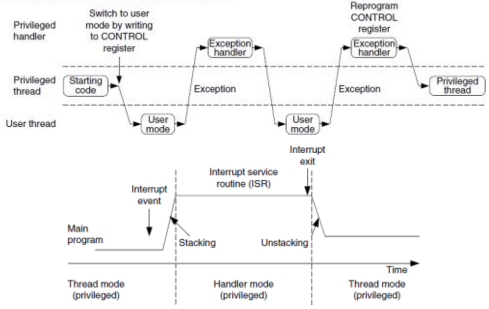


	Privileged	User
When running an exception handler	Handler mode (CONTROL[1] 0)	(not allowed)
When not running an exception handler (e.g., main program)	Thread mode (CONTROL[0] 0)	Thread mode (CONTROL[0] 1)

CONTROL [1] can be either 0 or 1

Modes and Privileges

Operation Modes and Privilege Levels in Cortex-M3.



Privileges

- In privileged level, the code has access to all resources.
- In user level, the code cannot :
  - Execute instructions such as CPSIE and CPSID, which allow changing FAULTMASK and PRIMASK
  - Access most System Control Block (SCB) registers

Exercise 1.1

- From the “EK-TM4C1294XL_IAR” workspace, select the “simple_io_main_sp” project.
- Inspect the “startup_TM4C1294.s” file:
  - What does the `Reset_Handler` function do?
- With the debugger stopped at the start of the main function, check:
  - Which processor registers have individualized bits? What are the meanings of these bits?
  - What is the state (mode) of the processor? Which registers were used to obtain this information?

Exercise 1.2

- Set up a breakpoint at the first declaration of the SysTick_Handler function and run the program until it stops:
  - What is the value of the LR register?
  - What is the state (mode) of the processor?
- Change the NMIPENDSET bit of the ICSR register (System Control Block) and run the program again:
  - What happens?
  - What is the value of the LR register?
  - What is the state (mode) of the processor?

Exercise 1.3

- Select the project “simple_io_process_sp”.
- Inspect the “startup_TM4C1294.s” file:
  - What does the `Reset_Handler` function do additionally?
- Redo the procedures in Exercise 1.2:
  - What is the value of the CONTROL register?
  - Was there any change in the values of the LR register?

Exceptions and Interrupts

- Any request to change the normal flow of a program:
  - Exception: synchronous, e.g. error detection
  - Interrupt: asynchronous, e.g. peripheral event occurrence

Cortex-M – General View

- Nested Vector Interrupt Controller (NVIC)
  - Support for multiple interrupt sources
  - Efficient handling of nested interrupts
  - Flexible architecture (highly configurable)
  - Intrinsic RTOS support

Cortex-M – General View

- Low latency interrupt architecture
- Some instructions with multiple execution cycles can be interrupted
- Hardware controlled exception entry/exit

Cortex-M – General View

- Hardware controlled interrupt entry/exit
  - Automatic context saving and restoring
  - Handling of late arrivals of higher priority interrupts
  - Handling of pending interrupts without full context restoring / saving (tail-chaining)

Exception States

- Inactive: neither pending nor active
- Pending: the exception was thrown, but has not yet been serviced
- Active: exception servicing has started, but has not yet been completed
  - Servicing of one exception can interrupt another one being serviced – in this case both exceptions are in the active state
- Active and Pending: the exception is being serviced and there is a pending exception from the same source

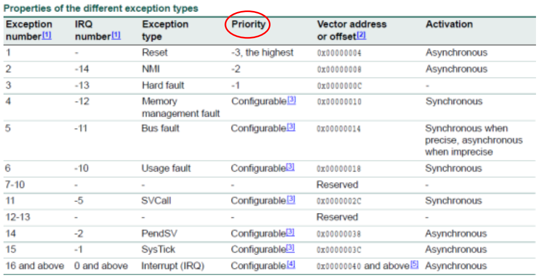
Exception Types

- Reset – priority -3 (thread mode)
- NMI – priority -2
- HardFault – priority -1
- Faults (MemManage, Bus, Usage)
- SVC – caused by the SVC instruction
- PendSV – service pending (e.g. context switching after interrupt)
- SysTick – periodically caused by the SysTick timer
- IRQn – peripheral interrupt request

Fault Exception Types

- MemManageFault: memory access faults detected by the MPU – if disabled, they escalate to HardFault
- BusFault: other types of memory bus faults than MemManage
- UsageFault: faults not related to the memory bus (e.g. undefined instruction or invalid state)

Exception Priorities



Properties of the different exception types

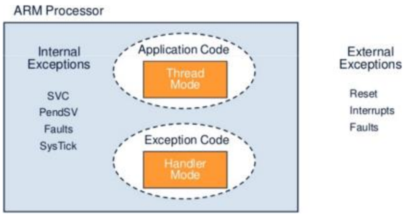
Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable[3]	0x00000010	Synchronous
5	-11	Bus fault	Configurable[3]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	Usage fault	Configurable[3]	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable[3]	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable[3]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable[4]	0x00000040 and above[5]	Asynchronous

Priority Definitions

- Core exception priorities are defined in the System Control Block (SCB)
  - SHPRn (System Handler Priority Registers)
- Peripheral interrupt priorities are defined in the Nested Vectored Interrupt Controller(NVIC)
  - IPRn (Interrupt Priority Registers)

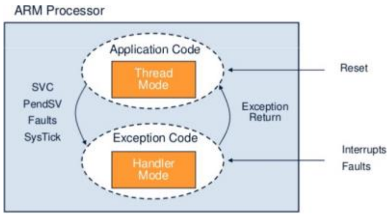
Causing Events

- Exceptions can be caused by events which are internal or external to the processor

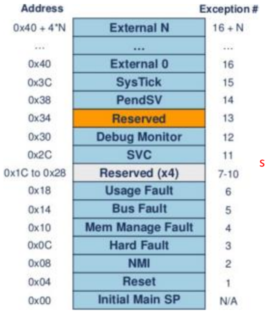


Processor State

- Thread mode after Reset
- Handler mode after any other kind of exception



Exception Vector Table



Address		Exception #
0x40 + 4*N	External N	16 + N
...	...	...
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

See table in file startup_TM4C1294.s!

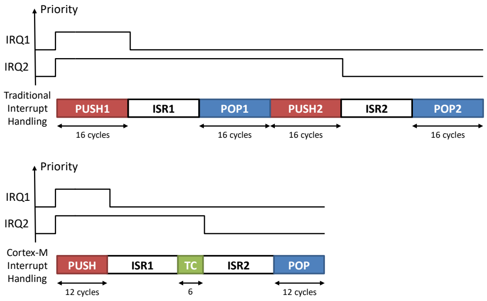
Important Definitions

- Preemption: a higher priority exception can cause preemption of an executing interrupt service routine (handler)
- Return: occurs at the end of the execution of an interrupt service routine (handler), in case
  - There is no pending exception with sufficient priority to be served (tail chaining)
  - The routine that ended is not a late arrival exception

Important Definitions

- Tail-chaining: if at the end of the execution of a handler there is a pending exception able to be handled, then the sequence of context restoration (unstacking) followed by new context saving (stacking) is not performed.

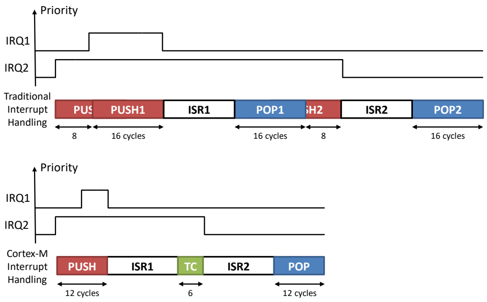
Tail-chaining Example



Important Definitions

- Late arriving: a mechanism that speeds up preemption if a higher priority exception occurs during context saving (stacking) – in this case the highest priority exception is serviced first and then, by tail-chaining, the lowest priority exception.

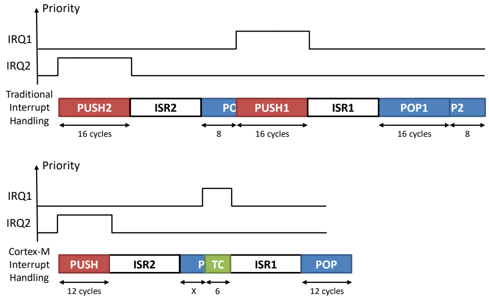
Late Arriving Example



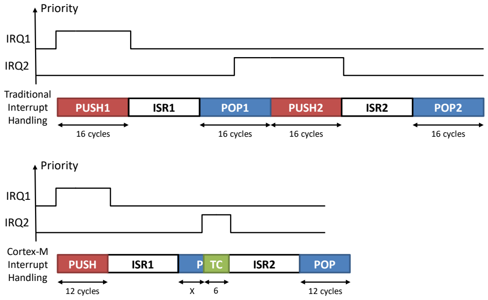
Important Definitions

- POP preemption: a mechanism that speeds up servicing of an exception that occurs during context restoration (unstacking) – in this case stacking is aborted and tail-chaining follows to service the new exception.

POP Preemption Example (1)



POP Preemption Example (2)



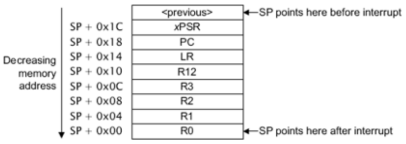
Exception Servicing

- Occurs when there is an exception in the pending state with sufficient priority* and
  - The processor is in thread mode, or
  - The pending exception has a higher priority than an exception already being serviced (preemption in handler mode)

*Sufficient priority means higher priority than the priority base threshold in BASEPRI

Exception Servicing

- Automatic context saving (stacking) occurs in the exception entry, as well as automatic context restoration (unstacking) in the exit
- Cortex-M4 (without FPU):

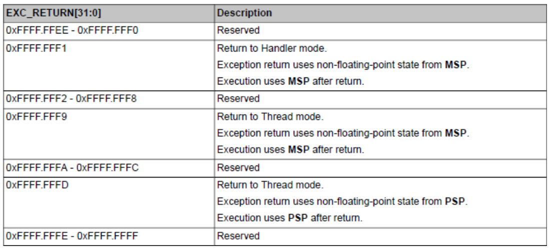

Decreasing memory address
SP points here before interrupt
SP points here after interrupt

	<previous>
SP + 0x1C	xPSR
SP + 0x18	PC
SP + 0x14	LR
SP + 0x10	R12
SP + 0x0C	R3
SP + 0x08	R2
SP + 0x04	R1
SP + 0x00	R0

Exception Servicing

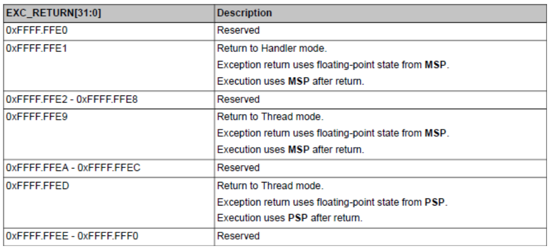
- LR := EXC_RETURN (next slides)
- The processor obtains the handler address from the vector table
- After stacking, execution branches to the handler and the exception reaches the active state
- If another exception of higher priority occurs during stacking (late-arriving condition), then it will be serviced beforehand

EXC_RETURN – Cortex-M4



EXC_RETURN[31:0]	Description
0xFFFF.FFEE - 0xFFFF.FFF0	Reserved
0xFFFF.FFF1	Return to Handler mode. Exception return uses non-floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFF2 - 0xFFFF.FFF8	Reserved
0xFFFF.FFF9	Return to Thread mode. Exception return uses non-floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFFA - 0xFFFF.FFFC	Reserved
0xFFFF.FFFD	Return to Thread mode. Exception return uses non-floating-point state from **PSP**. Execution uses **PSP** after return.
0xFFFF.FFFE - 0xFFFF.FFFF	Reserved

EXC_RETURN – Cortex-M4F



EXC_RETURN[31:0]	Description
0xFFFF.FFE0	Reserved
0xFFFF.FFE1	Return to Handler mode. Exception return uses floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFE2 - 0xFFFF.FFE8	Reserved
0xFFFF.FFE9	Return to Thread mode. Exception return uses floating-point state from **MSP**. Execution uses **MSP** after return.
0xFFFF.FFEA - 0xFFFF.FFEC	Reserved
0xFFFF.FFED	Return to Thread mode. Exception return uses floating-point state from **PSP**. Execution uses **PSP** after return.
0xFFFF.FFEE - 0xFFFF.FFF0	Reserved

Exception Return

- Occurs when an instruction writes one of the EXC_RETURN values to the PC
- Instructions used for exception return:
  - LDR PC, ...
  - LDM/POP including PC
  - BX LR (preferred)

Exercise 2.1

- From the “EK-TM4C1294XL_IAR” workspace, select the “simple_io_main_sp” project.
- Set up a breakpoint in the first instruction of the SysTick_Handler function (Disassembly window) and run the program until it stops:
  - What is contents of the top of the stack (first 8 double words)?
  - Compare the contents of the top of the stack (first 8 double words) to the contents of the core registers. What is the conclusion?

Exercise 2.2

- Redo Exercise **1.2** by changing the main function of the “simple_io_main_sp” project to use the FPU.
  - Was there any change in the values of the LR register?
  - Which stack pointer is being used? Check the debugger's Registers window (“CPU Registers” bank).
- Note: the FPU will be used if there is any operation with float type variables and `__FPU_USED` is defined as a macro (Options → C / C ++ Compiler → Preprocessor).

Exercise 2.3

- Redo Exercise 1.3 by changing the main function of the “simple_io_process_sp” project to use the FPU.
  - Was there any change in the values of the LR register?
  - Which stack pointer is being used? Check the debugger's Registers window (“CPU Registers” bank).
  - What is the difference between the “Current CPU Registers” bank and the “CPU Registers” bank of the debugger?

Extraclass Activity

- Reading:
  - Application Note: “Cortex-M4(F) Lazy Stacking and Context Switching” (Setions 1 and 2)
  - Book: “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors” (Section 13.3)
  - Website: https://www.keil.com/pack/doc/CMSIS/General/html/index.html#CM_Pack_Content (Introduction)