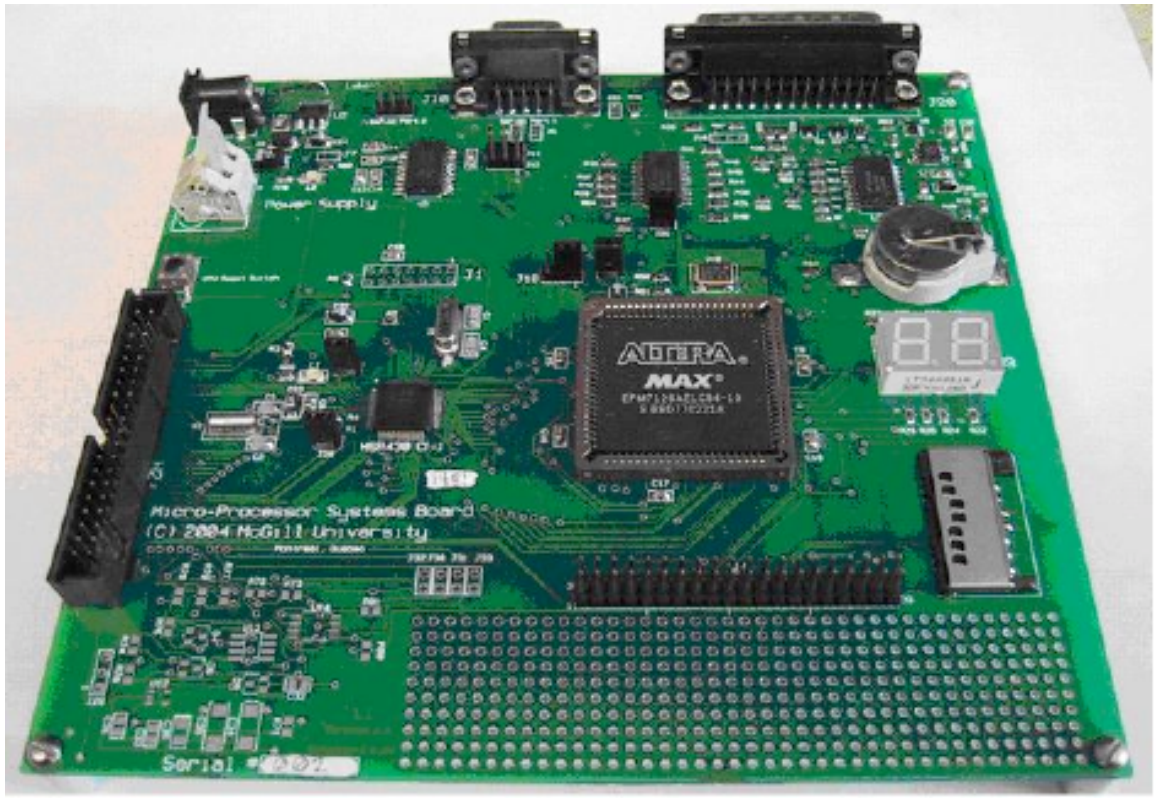
ECSE 426
Exceptions & SW – Part 1

Zeljko Zilic

zeljko.zilic@mcgill.ca



McGill

Acknowledgments: to STMicroelectronics for material on processors and the board

Outline

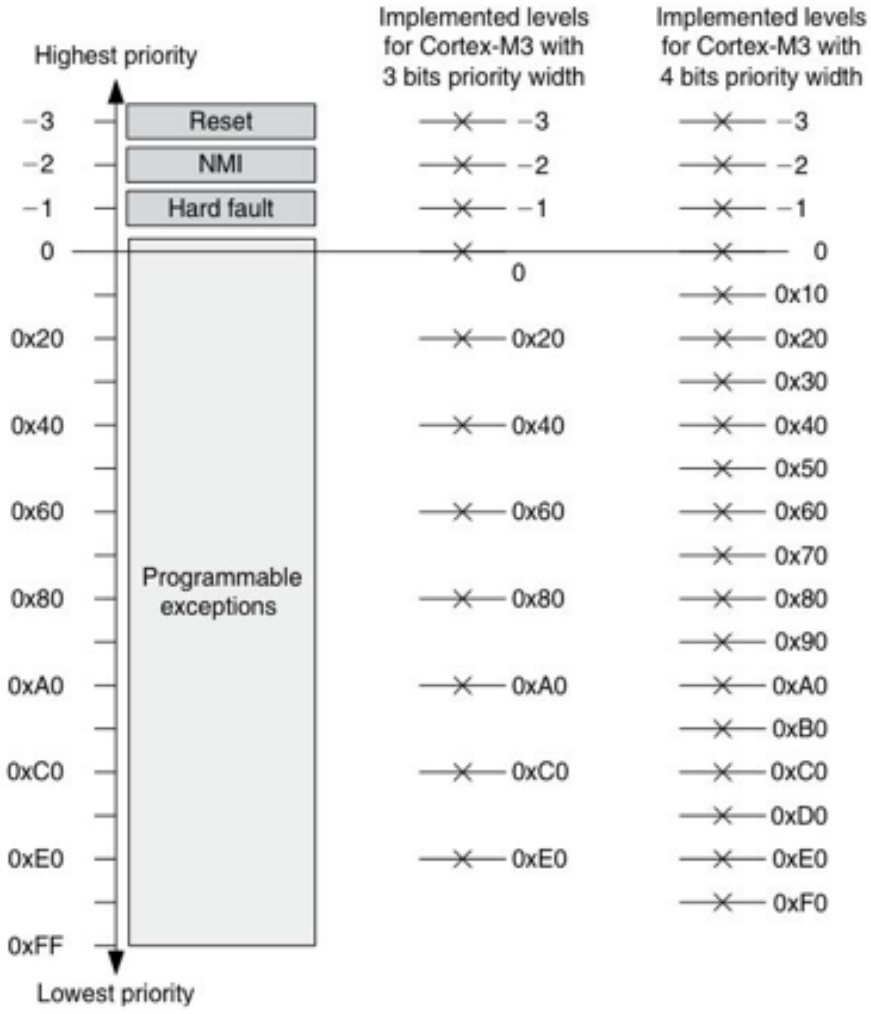
- Quiz!
- Exception Types
  - Reset, NMI, Faults
  - Interrupts and Interrupt Processing
- Lab: Part 2 of Lab 1
  - Grading and Report

Role of Exceptions

- Asynchronous programming/interaction style
  - Huge interest nowadays (callbacks, delegates, co-routines, …)
- Examples: receiving data, calls; failures
- Rather than polling (and wasting energy, cycles), let the events trigger action

Exceptions

- Asynchronous processing of various types:
  - Reset
  - Non-maskable interrupt
  - Various faults (mem, bus, usage)
  - Supervisor call
  - Debug and monitoring
  - External interrupts
- Each type can have multiple subtypes/priorities



Cortex-M Exception Types

No.	Exception Type	Priority	Type of Priority	Descriptions
1	Reset	-3 (Highest)	fixed	Reset
2	NMI	-2	fixed	Non-Maskable Interrupt
3	Hard Fault	-1	fixed	Default fault if other hander not implemented
4	MemManage Fault	0	settable	MPU violation or access to illegal locations
5	Bus Fault	1	settable	Fault if AHB interface receives error
6	Usage Fault	2	settable	Exceptions due to program errors
7-10	Reserved	N.A.	N.A.	
11	SVCall	3	settable	System Service call
12	Debug Monitor	4	settable	Break points, watch points, external debug
13	Reserved	N.A.	N.A.	
14	PendSV	5	settable	Pendable request for System Device
15	SYSTICK	6	settable	System Tick Timer
16	Interrupt #0	7	settable	External Interrupt #0
……	……..	……..	settable	……..
256	Interrupt#240	247	settable	External Interrupt #240

Vector Table

Vector Table starts at address 0

- In the code section of the memory map

Contains addresses of exception handlers

- Not Branch instructions like older ARM processors

Main stack pointer initial value in location 0

- Set up by hardware during Reset

Vector Table can be relocated

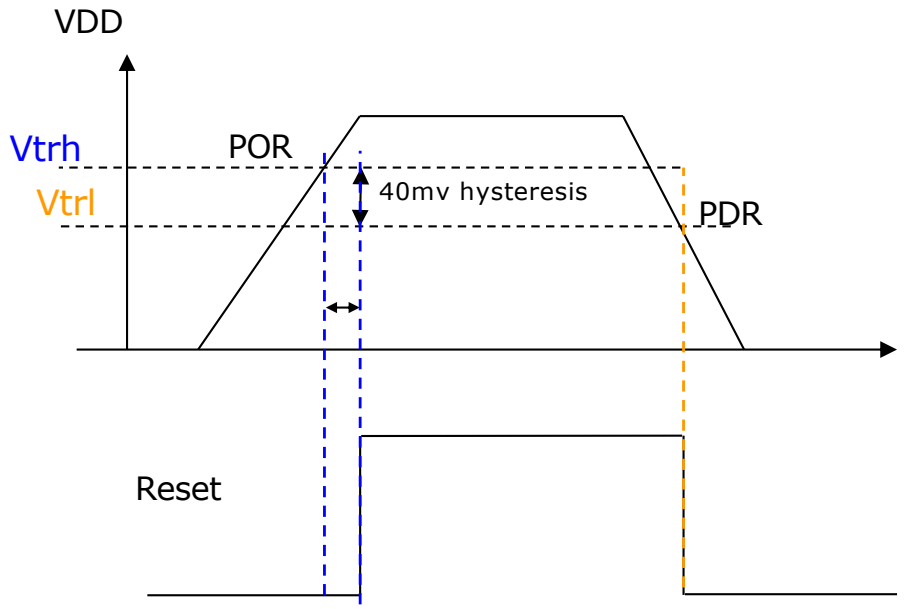
Address	Vector
0x00	Initial Main SP
0x04	Reset
0x08	NMI
0x0C	Hard Fault
0x10	Memory Manage
0x14	Bus Fault
0x18	Usage Fault
0x1C-0x28	Reserved
0x2C	SVCall
0x30	Debug Monitor
0x34	Reserved
0x38	PendSV
0x3C	Systick
40	IRQ0
…	More IRQs

Internal Power On Reset (POR)

No Need for External Reset Circuit

Integrated POR / PDR guarantees reset when Vdd is out of spec

POR and PDR have a typical hysteresis of 40mV



McGill

Internal Programmable Voltage Detector (PVD)

Provides early warning of power failure

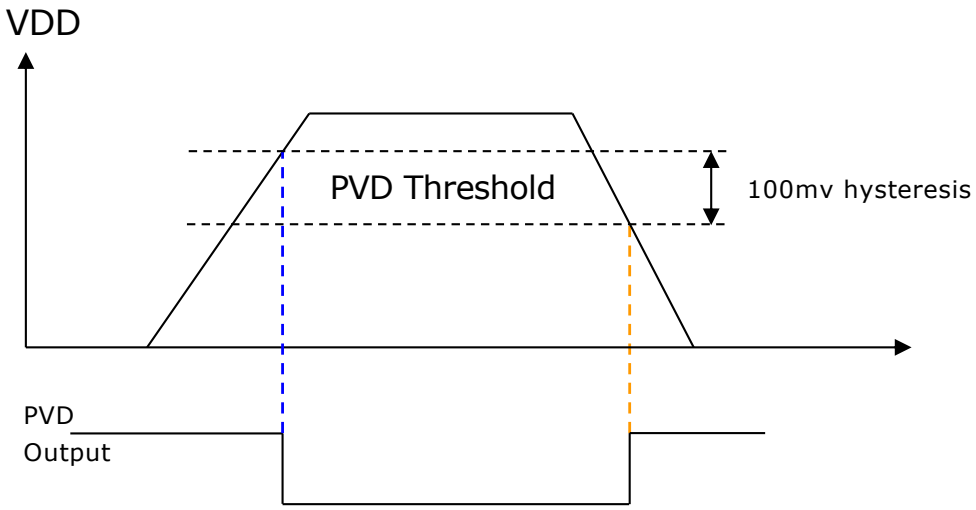
Enabled by software

Compares V_{DD} to a configurable threshold

2.2V to 2.9V, 100mV steps

Generates an interrupt

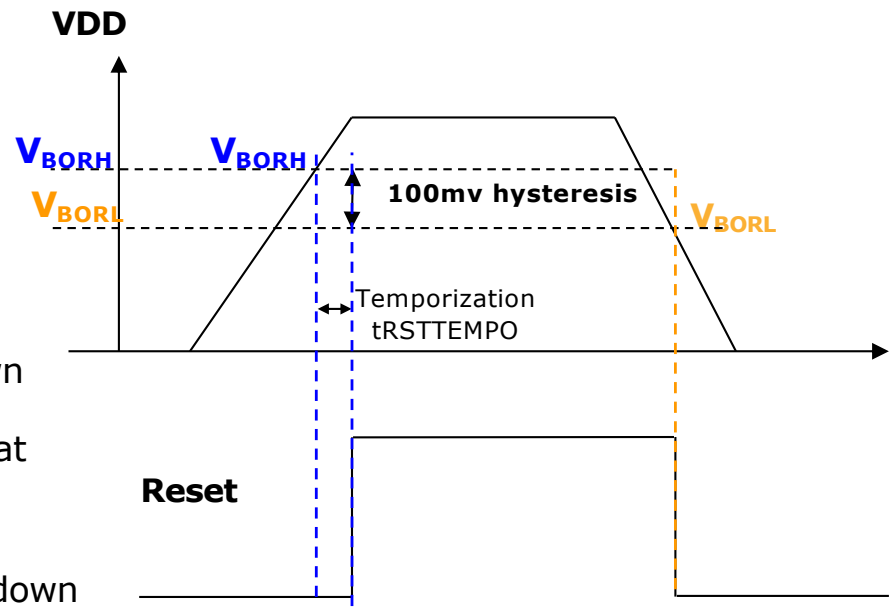
Use: puts the MCU into a safe state



McGill

Brown Out Reset (BOR)

- During power on, the Brown out reset (BOR) keeps the device under reset until the supply voltage reaches the specified V_{BOR} threshold.
  - No need for external reset circuit
- BOR have a typical hysteresis of 100mV
- BOR Levels are configurable by option bytes:
  - BOR OFF: **2.1 V** at power on and **1.62 V** at power down
  - BOR LOW **(DEFAULT)** : **2.4 V** at power on and **2.1 V** at power down
  - BOR MEDIUM: **2.7 V** at power on and **2.4 V** at power down
  - BOR HIGH: **3.6 V** at power on and **2.7 V** at power down



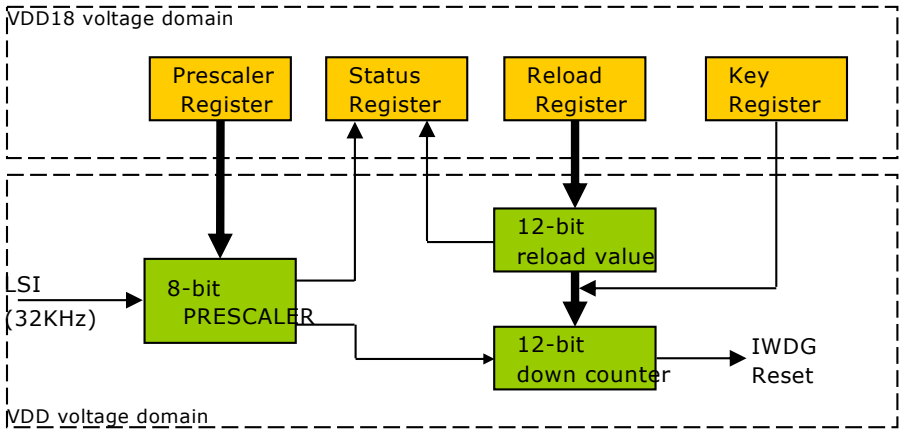
Watchdog Concept

- One of oldest (embedded) microprocessor tricks in the book
- Need for action when processor "runs away"
  - Discover, bring back to normal state
- Concept: periodically refresh a timer, unless when dead
- Two watchdog blocks in Cortex M

Independent Watchdog (IWDG)

Not Clocked by Main Processor Clock!

- Selectable HW/SW start through option byte
- Advanced security features:
  - IWDG clocked by internal low-speed RC clock
  - Stays active even if the main clock fails
  - Once enabled the IWDG can't be disabled
  - Safe Reload Sequence (key)
  - IWDG function remains functional in STOP and STANDBY modes



- To prevent IWDG reset: write 0xAAAA key value at regular intervals before the counter reaches 0

Best suited to applications that require the watchdog to run as a totally independent process outside the main application

- Reset flag to inform when a IWDG reset occurs
- Min-max timeout value @32KHz: 125µs / 32.8s

McGill

Quiz

- Which clock feeds the IWDG down counter?

  _____

- Can the IWDG be started by HW? How?

  _____

- What is the maximum IWDG timeout?

  _____

McGill

Reset: Window Watchdog (WWDG)

Configurable time-window, can be programmed to detect abnormally late or early application behavior

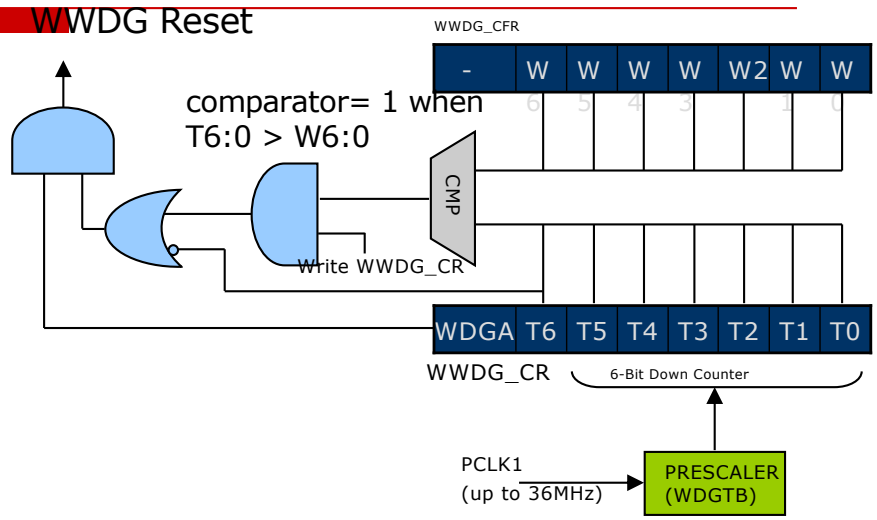
Conditional reset

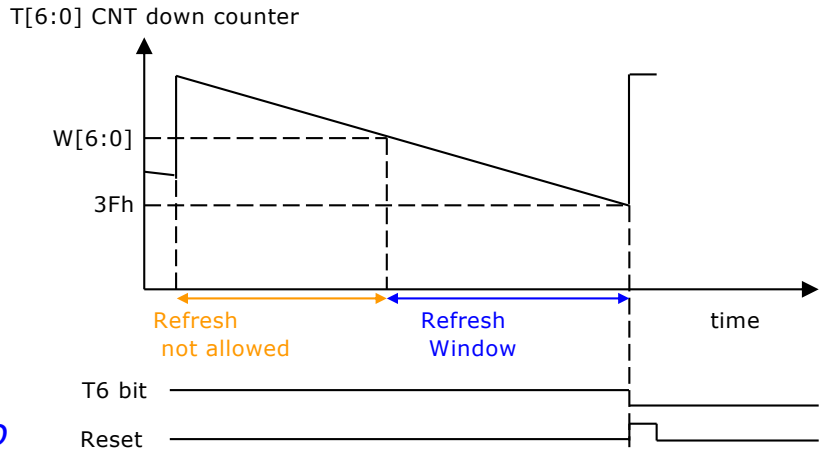
- Reset when down-counter value becomes less than 0x40
- Reset if down counter is reloaded outside the time-window

Reset flag to tell when WWDG reset occurs

Min-max timeout value @36MHz: 113µs / 58.25ms

Best suited to applications which require the watchdog to react within an accurate timing window





McGill

Quiz

- When can be generated the WWDG reset?

  _____

- What is the maximum WWDG timeout?

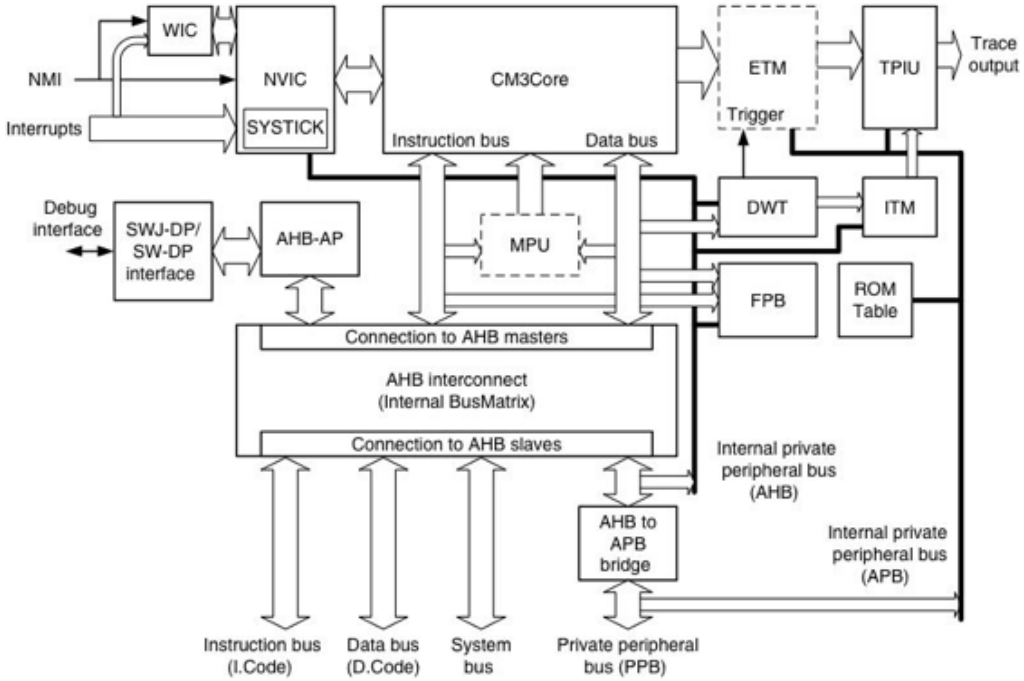
  _____

Nested Vectored Interrupt Control

NVIC - Interrupts handled in hardware

- Allows handlers written purely in C, no overhead
- Load/Store Multiple instruction (LDM/STM) is interruptible

- STM32 implements 84 exception / interrupt sources
  - 68 maskable peripheral interrupt sources
  - 16 interrupt sources from the Cortex-M3 core
    - 16 programmable priority levels in STM32



Cortex Interrupts

Entry

Processor state automatically saved to stack over data bus (SYSTEM)

{R0-R3, R12, LR, PC, xPSR}

In parallel, ISR prefetched on the instruction bus(ICODE)

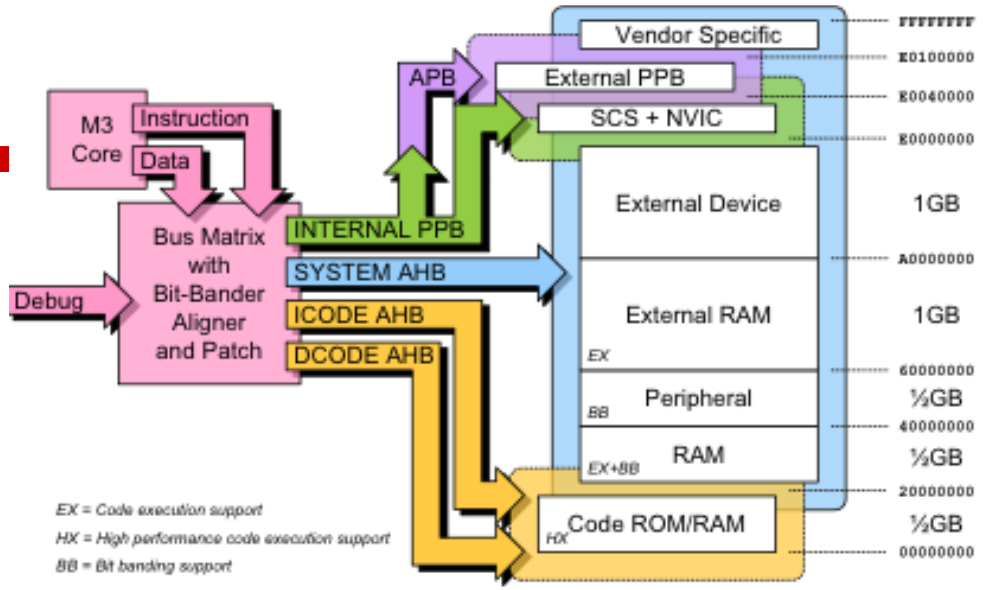
-ISR ready to start executing as soon as stack PUSH complete

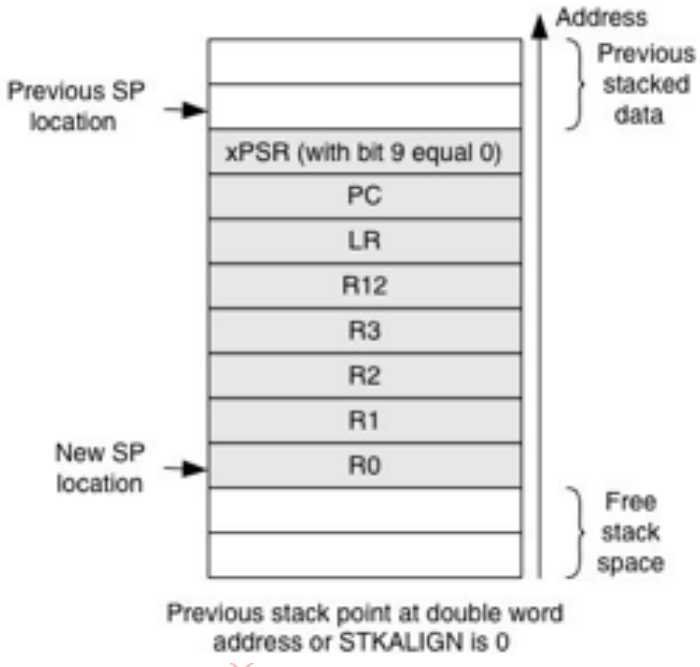
Exit

Processor state is automatically restored from the stack

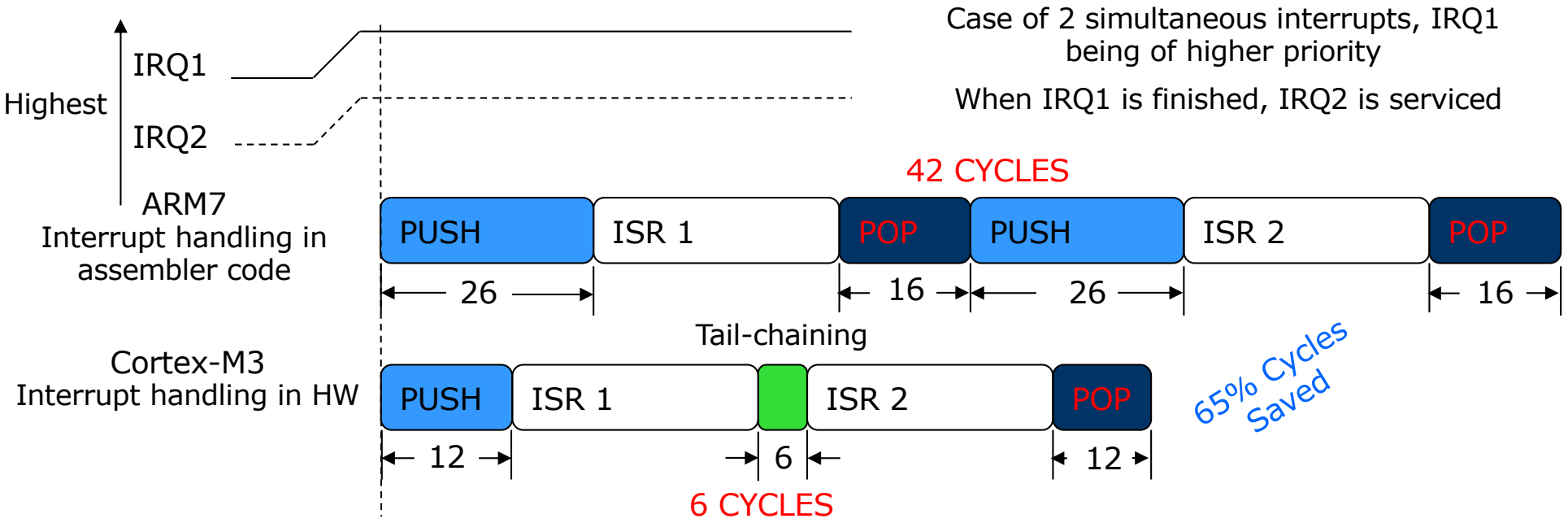
In parallel, interrupted instruction is prefetched for execution upon POP

- Stack POP can be interrupted -> new ISR immediately executed





Fast Interrupts- Tail Chaining



In Cortex-M, ISR2 has only a 6-cycle delay. ISR2 has been '*tail-chained*'

ICODE (vector) and SYSTEM (stack) used in parallel

McGill

Outline

- Demo of IWDG+GPIO setup and use
- Exception Processing
  - Handler code/Interrupt Service Routine (ISR)
  - Exceptions in support of OS: SVC/Pending SV
- *Lab1 Report*: document your solution to the lab, such that it gets appreciated
  - Showcase correctness, completeness (handling all cases), performance
- Lab 2: GPIO + DAC

Exception Handlers (ISRs) in C

- Recall: C compilers follow AAPCS (ARM Architecture Procedure Call Standard)
- C function can modify R0 to R3, R12, R14 (LSR) and PSR
  - Caller-saved registers
  - Code that calls a subroutine must save them to memory if it needs them after the function call
- If C function needs to use R4-R11, it should save these registers onto stack memory and restore them before exiting
  - Callee-saved registers
- Typically R0-R3 are input parameters, and R0 is return result (+ R1 for 64 bit result)

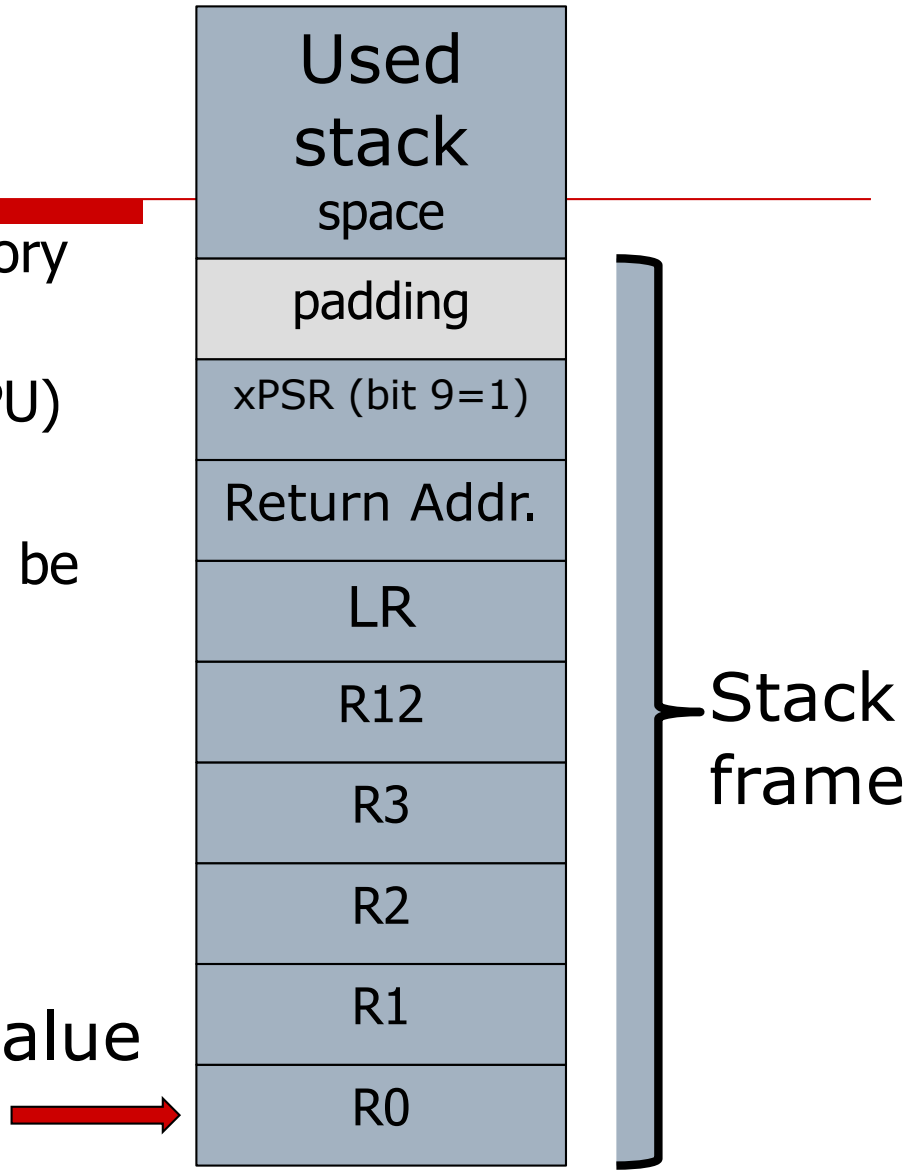
Exception Handlers in C

- Exception mechanism needs to save R0-R3, R12, LR and PSR at exception entrance automatically
- Restore them at exception exit under control of processor's hardware
- Value of return address (PC) is not stored in LR as in normal C function calls
  - Exception mechanism puts an EXC_RETURN code in LR at exception entry
  - Value of return sequence also needs to be saved
- 8 registers need to be saved (+ floating point, if used in ISR)

Stack Frame

- Block of data pushed to stack memory at exception entrance
- 8 words (no FPU) and 17 words (FPU)

- General purpose registers R0-R3 can be easily accessed
- Use stack pointer related addressing
- Pass information to software triggered interrupts or SVC handlers



EXC_RETURN

- When processor enters exception handler or interrupt service routine
  - Value of link register (LR) updated to a code called EXC_RETURN
  - Used to trigger exception return mechanism when loaded into program counter (PC)
  - Some bits provide extra information about the exception sequence
  - 4: Stack frame type = 1 (8 words) or 0 (26 words)
  - 3: Return mode = 1 (Thread) or 0 (Handler)
  - 2: Return stack = 1 (Process stack) or 0 (Main stack)

Exception sequence

- Exception occurs
  - Need to push registers into stack (form stack frame)
  - Perform vector fetch
  - Start exception handler instruction fetch

- Multiple bus interfaces
  - Reduce interrupt latency: do stacking and flash memory access in parallel
  - Stacking: data bus
  - Vector fetch + instruction fetch: instruction bus

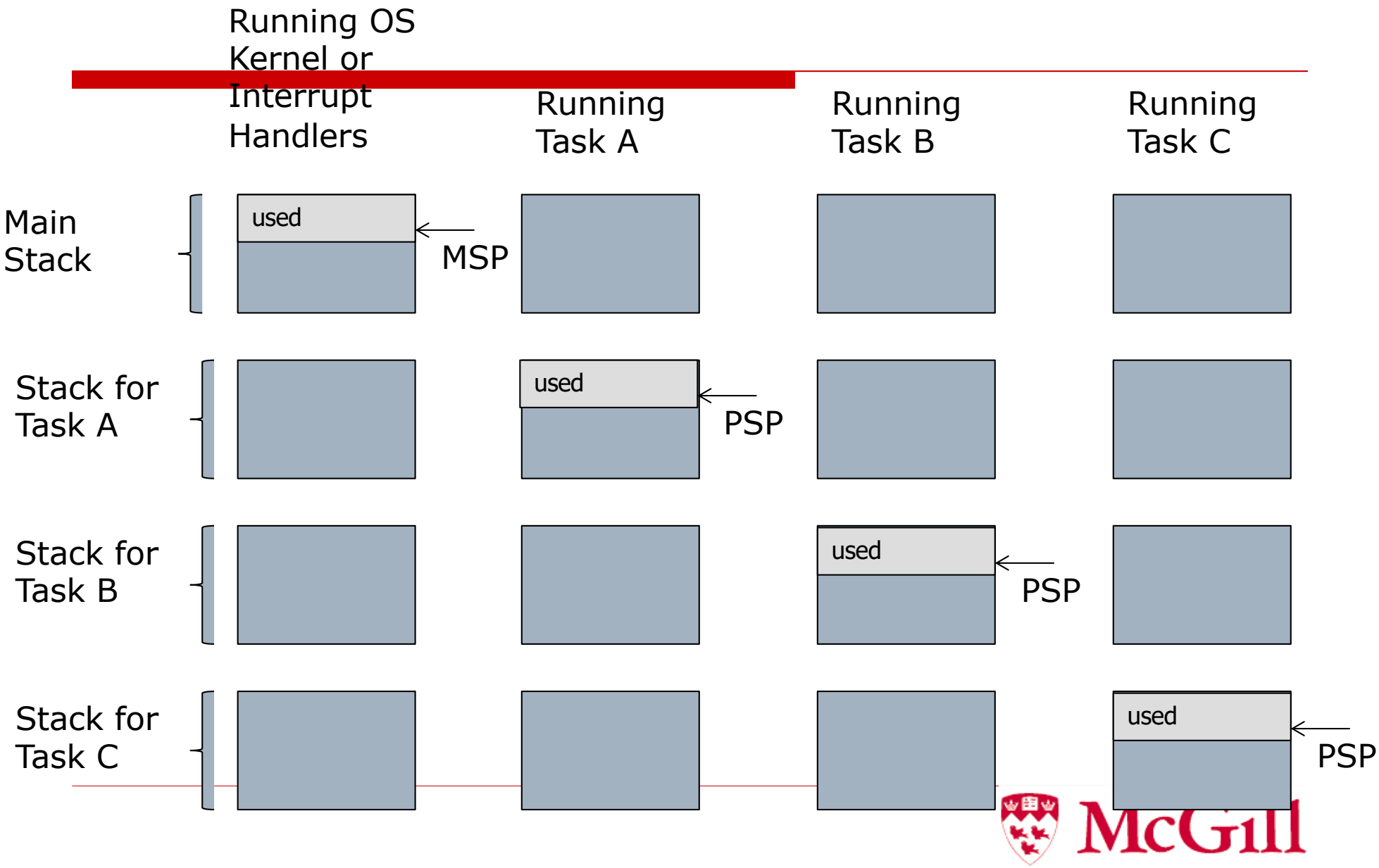
OS Support

- Shadowed stack pointer
  - Two stack pointers available
  - MSP used for OS kernel and interrupt handlers. PSP used by application tasks
- Systick timer
  - Simple timer included inside processor.
  - Embedded OS can be used on a wide range of Cortex-M parts
- Supervisor Call (SVC) and Pendable Service Call (PendSV)
  - Allow context switching and other essential embedded OS operations
- Unprivileged execution level
  - Basic security model that restricts access of some application tasks
- Exclusive access
  - Exclusive load and store instructions are useful for semaphore and mutual exclusive (MUTEX) operations in the OS

Shadowed Stack Pointer

- Main Stack Pointer (MSP)
  - Default stack pointer
  - Used in Thread mode when CONTROL bit[1] (SPSEL) is 0
  - Always used in Handler mode
- Processor Stack Pointer (PSP)
  - Used in Thread mode when CONTROL bit[1] (SPSEL) is 1
- In systems with an embedded OS or RTOS
  - Exception handlers (including OS kernel) use MSP
  - Application tasks use PSP
  - Each application task has its own stack space
  - Context switching in OS updates PSP each time context is switched

Shadowed Stack Pointer



McGill

Shadowed Stack Pointer

- If application task has a problem that leads to stack corruption, stack of OS and other tasks likely remains intact
- Stack space for each task only needs to cover maximum stack usage plus one level of stack frame
- OS can use Memory Protection Unit (MPU) to define the stack region which an application task can use
  - If task has a stack overflow problem, MPU can trigger a MemManage fault exception
  - Prevent task from overwriting memory regions outside its allocated stack space

Context Switching

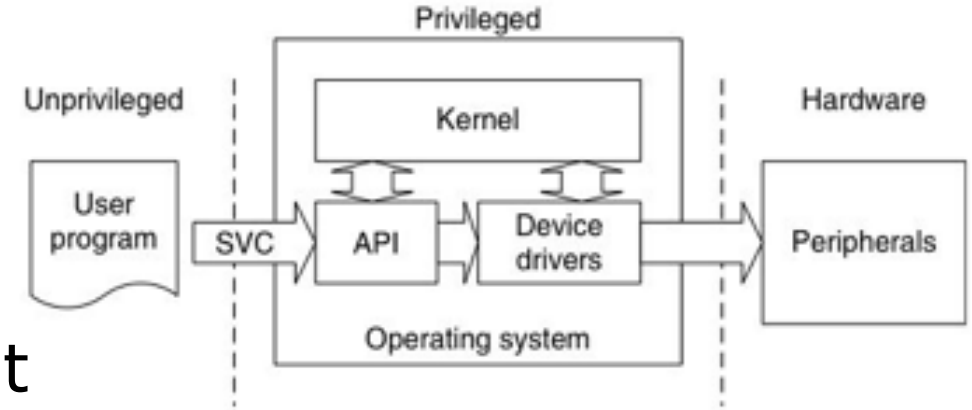
- OS needs to switch between different tasks
- Context switching usually carried out in the PendSV handler
- Can be triggered by a periodic SysTick exception
- Inside context switching operation
  - Save current status of registers in the current task
  - Save the current PSP value
  - Set the PSP value to the last SP value for the next task
  - Restore the last values for the next task
  - Use exception return to switch to the task
- Context switching is carried out in PendSV
  - Typically at lowest priority level
  - Prevents context switching from happening in the middle of an interrupt handler

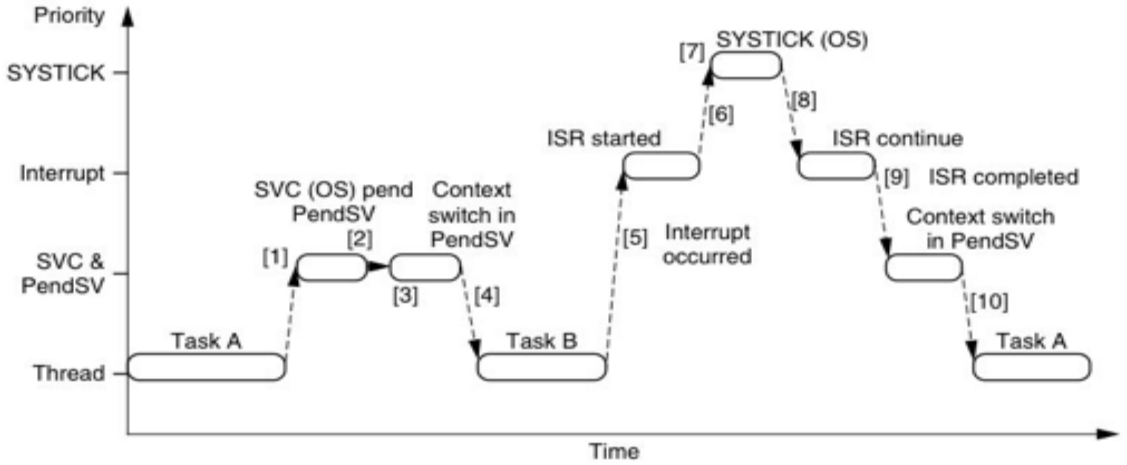
Supervisor Calls: SVC & PendSV

- SVC (Supervisor Call) and PendSV (Pendable Service Call)
  - Important for OS designs
- SVC Instruction
  - Keil/ARM: `__svc`
  - Portable; hardware abstraction
  - Can write to NVIC using a software trigger interrupt register, but several instructions might execute while the interrupt is pending
  - With __svc, the SVC handler executes immediately (except when another higher priority exception arrives)
- Can be used as API to allow application tasks to access system resources

Supervisor Calls: SVC & PendSV

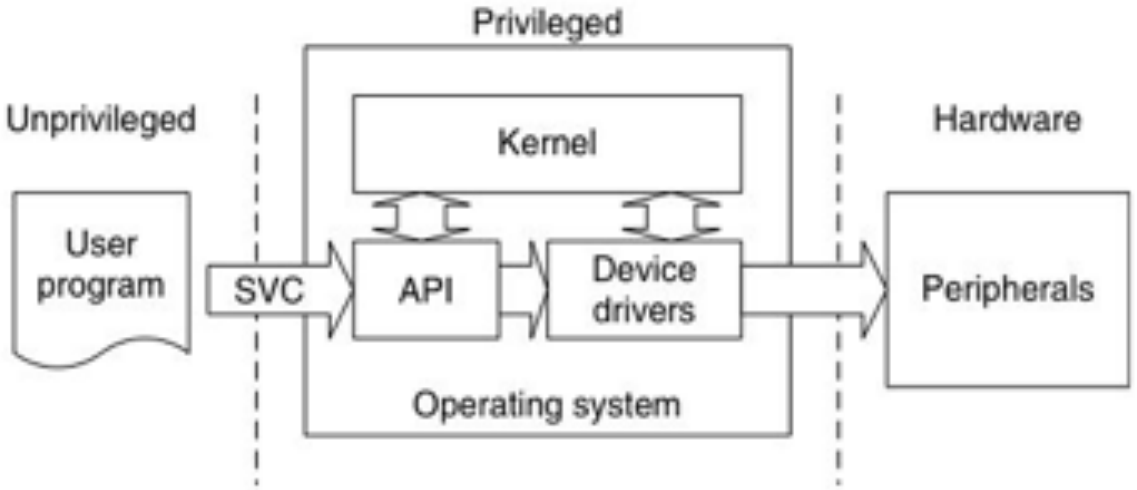
- Supervisor Call (SVC): for system function calls
  - Portable; HW abstraction
  - Can't nest! (no SVC in SVC)
- SVC Instruction
  - Keil/ARM: `__svc`
- Pendable SV: can wait/nest
- SysTick: OS clock
  - Good for RTOS
  - 24-bit down counter
  - 2 clock sources
  - Only priviledged mode





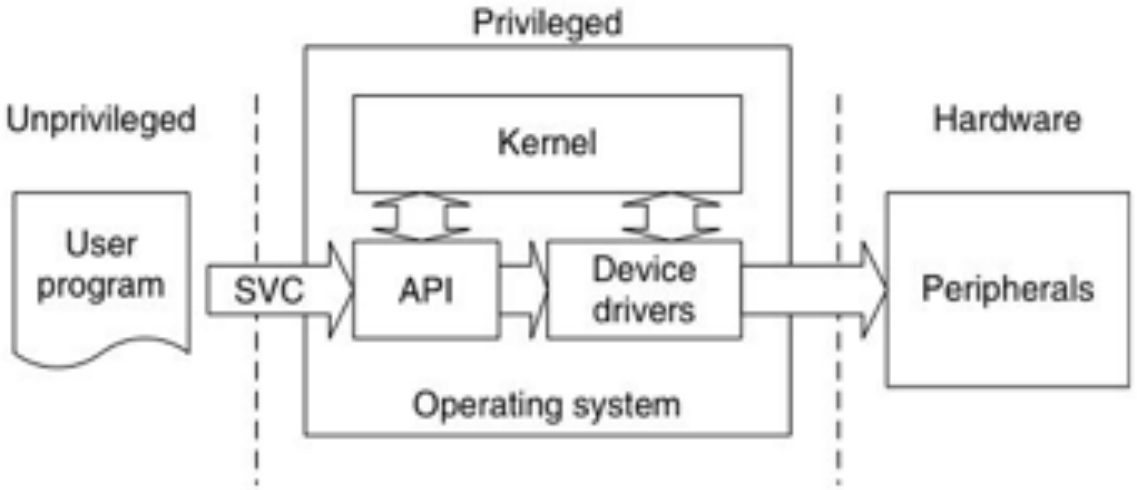
Supervisor Calls: SVC

- Systems with high reliability requirements
  - Some hardware set up to be privileged access only
  - Embedded system is more robust and secure



Supervisor Calls: SVC

- Application tasks
  - Run in unprivileged access level
  - Can only access protected resources via services from OS
  - Cannot gain unauthorized access to critical hardware
  - No need to know programming details of underlying hardware
  - Can be developed independently of OS (don't need to know exact address of OS service functions – only the SVC service number and parameters)
  - Hardware level programming handled by device drivers



SVC Handling

- SVC instruction (`__svc`)
  - Needs immediate value (`SVC #0x3; Call SVC function 3`)
  - SVC exception handler extracts parameter and knows which action to perform
- Procedure
  - Determine if the calling program was using main stack or process stack?
  - Read the program counter value from appropriate stack
  - Read instruction from that address (masking out unnecessary bits)

```
SVC_Handler      ; NOTE: Code written in ARM Assembler!
  TST lr, #4           ; Test bit 2 of EXC_RETURN
  ITE EQ
  MRSNE R0, MSP   ; if 0, stacking using MSP, copy to R0
  MRSEQ R0, PSP   ; if 1, stacking using PSP, copy to R0
  LDR R0, [R0, #24] ; get stacked PC from stack frame
        ; stacked PC = address of instruction after SVC)
  LDRB R0, [R0, #-2] ; get first byte of the SVC instruction
        ; now SVC number is in R0
```

Source : J. Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

McGill

SVC Handling

- In C, we break it into two parts
- Can't check the value of LR (EXC_RETURN) in C
- Use assembly inline (`__asm`)

```
__asm void SVC_Handler(void)
{
   TST LR, #4          ; Test bit 2 of EXC_RETURN
   ITE EQ
   MRSNE R0, MSP    ; if 0, stacking using MSP, copy to R0
   MRSEQ R0, PSP    ; if 1, stacking using PSP, copy to R0
   B _cpp(SVC_Handler_C)
   ALIGN 4
}
```

Source : J. Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

SVC Handling

```
void SVC_Handler_C(unsigned int * svc_args)
{
  uint8_t svc_number;
  unint32_t stacked_r0, stacked_r1, stacked_r2, stacked_r3;

  svc_number = ((char *) svc_args[6])[-2];
  // Memory[(Stacked PC)-2]
  stacked_r0 = svc_args[0];
  stacked_r1 = svc_args[1];
  stacked_r2 = svc_args[2];
  stacked_r3 = svc_args[3];

  // other processing
  …
  // Return result (e.g. sum of first two elements)
  svc_args[0] = stacked_r0 + stacked_r1;
  return;
}
```

Source : J. Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

SVC Handling

- Passing the address of the stack frame allows the C handler to extract any information it needs
- Essential if you want to pass parameters to an SVC service and get a return value

- A higher priority interrupt could be executed first and change the values of R0, R1, etc.
- Using the stack frame ensures your SVC handler gets the correct input parameters

PendSV

- Pended Service Call (exception type 14)
  - Programmable priority level
  - Triggered by writing to Interrupt Control and Status Register (ICSR)
  - Unlike SVC, it is not precise. Pending status can be set in a higher priority exception handler
- Execution of OS kernel (and context switching) triggered by
  - SVC call from an application task (e.g. task is stalled because it is waiting for data or an event)
  - Periodic SysTick exception
- PendSV is lowest priority exception
  - Context switching delayed until all other IRQ handlers have finished
  - OS can set pending status of PendSV & carry out context switching in PendSV exception handler

Exclusive Access

- Multi-tasking system: tasks need to share limited resources
  - For example, one console output
- Tasks need to "lock" a resource and then "free" it after use
  - Usually based on software variables
  - If lock variable is set, other tasks can see that the resource is locked
- Lock variable is called a semaphore
  - If only one resource is available, also called Mutual Exclusive (MUTEX)
  - In general, semaphores can support multiple tokens
    - e.g. one for each channel of a communication stack
    - Semaphore implemented as a token counter
    - Each task decrements the semaphore when it needs the resource

Exclusive Access (2)

- Decrement of counter is not atomic
  - One instruction to read variable
  - One instruction to decrement it
  - One instruction to write back to memory

- Context switching may occur between read and write
  - Simplest approach: disable context switching when handling semaphores
    - Can increase latency and only works for single processor designs
    - Multi-processor: tasks on different processors can try to decrement semaphore variable at same time

Exclusive Access: Local Monitor (LL/SC)

- Cortex-M4 supports feature called exclusive access
  - Semaphores read and written by exclusive load and exclusive store
  - If during store, access cannot be guaranteed to be exclusive, exclusive store fails
- Processor has small hardware unit called the local monitor
  - Normally in Open Access state
  - Exclusive load: switches to Exclusive Access state
  - Exclusive store: only if local monitor in Exclusive Access state
  - STREX fails if
    - CLREX has been executed (switching local monitor back to Open)
    - Context switch has occurred (interrupt)
    - No LDREX was executed earlier
    - External hardware returns an exclusive fail status
  - Multiprocessor systems need a global exclusive access monitor