Interrupt Latency

ENCE361 Embedded Systems 1

Course Coordinator: Ciaran Moore (ciaran.moore@Canterbury.ac.nz)

Lecturer: Le Yang (le.yang@canterbury.ac.nz)

Department of Electrical and Computer Engineering

Where we’re going today

- **Interrupt performance measure**
- Interrupt priority and preemption
- More on ISR coding
- Homework

Measures of Interrupt Performance (1)

- How do we quantify the performance of a foreground/background processing model?
  - Interrupt **latency**: interval between interrupt event and start of activity related to servicing it (usually context switch)
    - Finish current instruction, critical section, wait for other ISR to complete …
  - Interrupt **response**: interval between interrupt event and start of executing the ISR

    response = latency + context switch time
  - Interrupt **recovery**: time for microprocessor to return to interrupted code

Measures of Interrupt Performance (2)

- Above definitions are for the course, not necessarily universally used ···
- Interrupt are asynchronous and worst-case analysis is resorted to

$$\boxed{\text{Interrupt Response} = T_{response} = \max\{\boldsymbol{T_{disabled}}, \boldsymbol{T_{instr}}\} + \boldsymbol{T_{csw}}}$$

$\max\{\boldsymbol{T_{disabled}}, \boldsymbol{T_{instr}}\}$: interrupt latency

$\boldsymbol{T_{disabled}}$: the longest period during which interrupts are disabled (due to e.g., critical section, servicing other interrupts)

$\boldsymbol{T_{instr}}$: the longest execution time for any instruction

$\boldsymbol{T_{csw}}$: context switch time (part of it might be in the prologue of ISR)

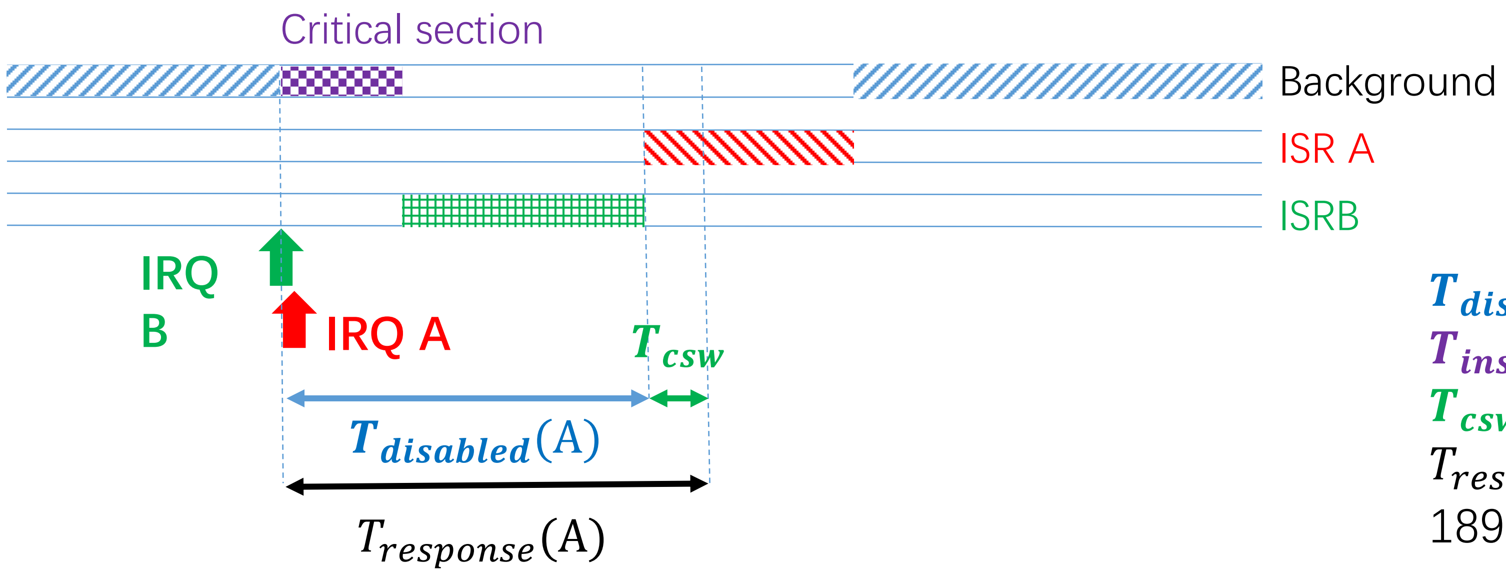
Example

- What are the worst-case interrupt responses for IRQs A and B?
  - MCU services <u>two interrupts (IRQs A and B)</u> of same priority
  - MCU instructions take <u>4 – 10 clock cycles</u> to execute ($\boldsymbol{T_{instr}}$ = 10)
  - For each ISR, <u>context switch takes 25 clock cycles</u> ($\boldsymbol{T_{csw}}$ = 25)
  - <u>Worst-case executing times</u> for ISRs A and B are 85 & 108 clock cycles
  - Background program has <u>a critical section</u> that requires 56 clock cycles to complete

$\boldsymbol{T_{disabled}}$ = ?

Analysis and Solution

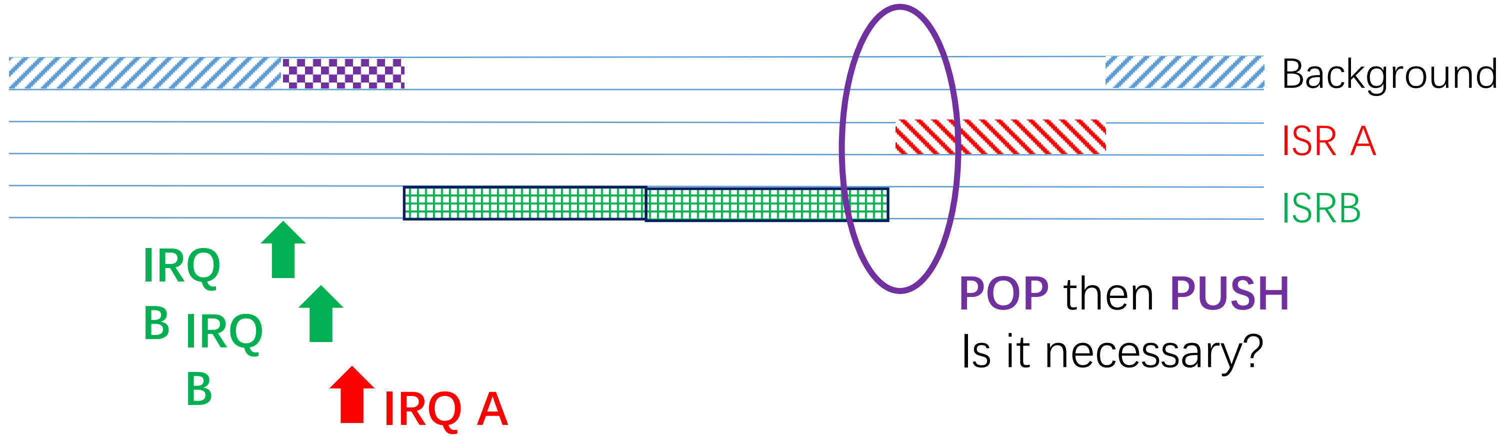
- $T_{disabled}$ = ?
  - Two interrupts have the same priority
  - ISRs A and B have worst-case executing times of 85 & 108 clock cycles
  - Background program has a critical section requiring 56 clock cycles to complete
- For IRQ A, the worst case is



$T_{disabled}(\mathbf{A}) = \mathbf{56} + \mathbf{108} =$ 164
T_{instr} = 10
T_{csw} = 25
$T_{response}(A)$=max{164, 10} +25 = 189

Assumptions

- Above analysis makes the following two important assumptions
  - Interrupts occurs not so frequently
    - What if IRQ B happens frequently



$T_{disabled}(\mathrm{A})$ can be very large!

  - No tail chaining …

Tail Chaining

- In ARM Cortex-M4, PUSH and POP operations both require 12 clock cycles
  - When servicing back-to-back interrupts, conventional use of POP and PUSH operations needs 24 clock cycles

- Tail chaining enables servicing back-back interrupted to be performed without complete context saving and restoration, requiring 6 clock cycles
  - Reduced interrupt response time

- Consider servicing IRQs A and B in sequence
  - Before running ISR A, PUSH the context on the stack
  - After executing ISR A, switch to servicing IRQ B with the context intact
  - After executing ISR B, POP the context from the stack → Tail chaining is safe

Where we’re going today

- Interrupt performance measure
- **Interrupt priority and preemption**
- More on ISR coding
- Homework

Preemptive vs. Non-Preemptive

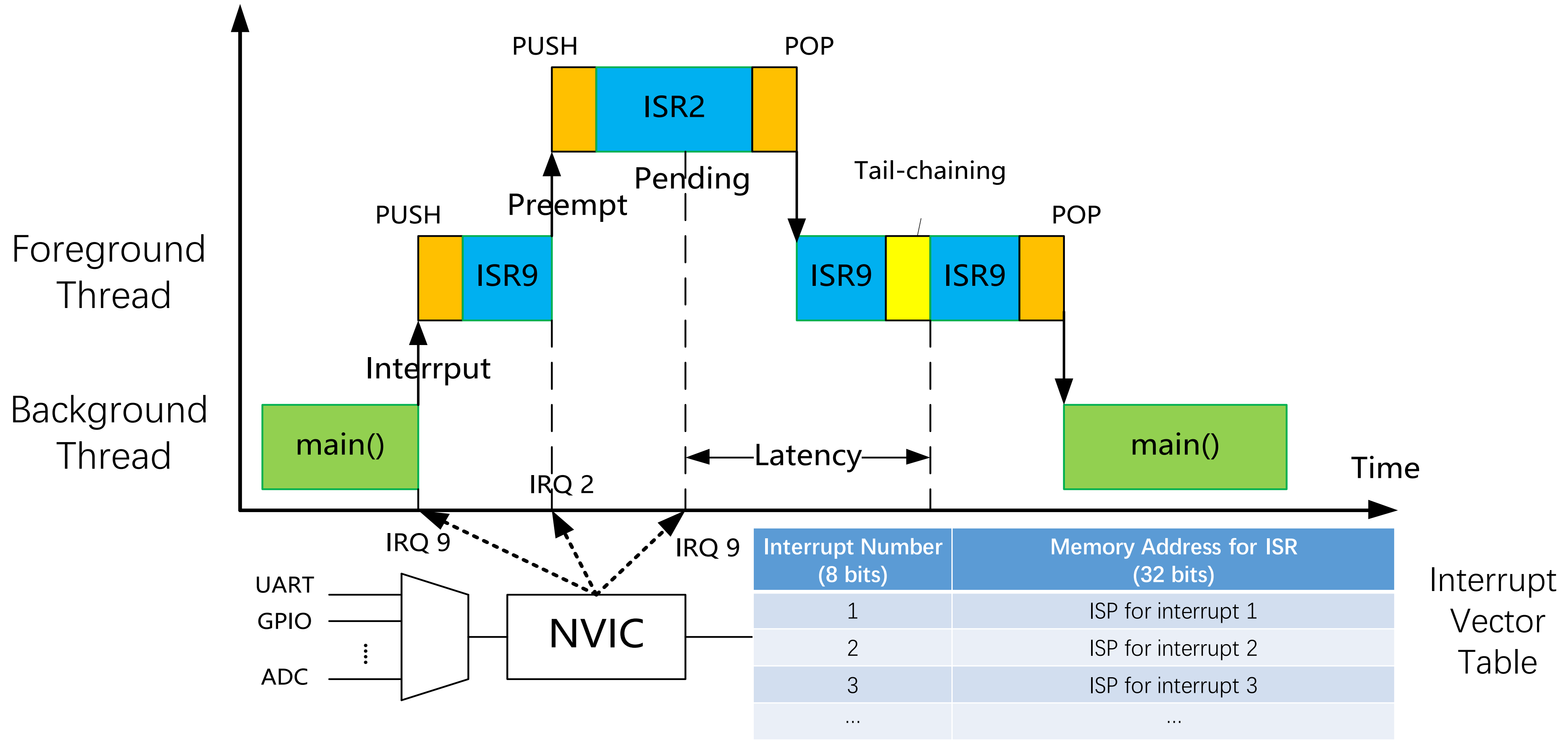
- Preemptive
  - Within an ISR, interrupts of lower and equal priority are masked
  - Interrupts with higher priority can preempt and get serviced before ISR completes

- Non-preemptive
  - Within an ISR, all other interrupts are globally disabled until ISR completes
  - Usually the default mode of many MCUs

- For ARM Cortex-M4, all user-defined interrupts, by default, have equal priority

Preemption

- Interrupts with higher priority preempt interrupts with lower priority via Nested Vectored Interrupt Control (NVIC)



Interrupt Number (8 bits)	Memory Address for ISR (32 bits)
1	ISP for interrupt 1
2	ISP for interrupt 2
3	ISP for interrupt 3
…	…

Interrupt Priority in ARM Cortex-M4 (1)

- For ARM Cortex-M4, each interrupt is associated with a priority value
  - The lower the priority value, the higher the interrupt priority

  - Three interrupts have fixed priority values
    - RESET: -3 (highest among all interrupts)
    - NMI: -2
    - Hard fault: -1

  - Other interrupts have configurable priority values 0-7
    - By default, their priority values are equal to 0 (non-preemptive mode)

From Table 2-8 on Page 103 in TM4C123GH6PM.pdf

Interrupt Priority in ARM Cortex-M4 (2)

- Set interrupt priority using API functions

  void IntPrioritySet (uint32_t ui32Interrupt, uint8_t ui8Priority)

  uint32_t IntPriorityGroupingSet (uint32_t ui32Bits)

  void IntPriorityMaskSet (uint32_t ui32PriorityMask)

- Configurable priority value has the range 0 – 7
  - Only the 3 highest bits of the priority value are considered
    - ui8Priority = 0x1F or 0x00 has the same effect (both means priority value 0)
    - IntPriorityMaskSet (0x40) and IntPriorityMaskSet (0x4F) both mask interrupts with priority values ≥ 2

From Chapter 17 in TivaWare Peripheral Driver Library Users

Interrupt Priority in ARM Cortex-M4 (3)

- **Example**

```
    IntPrioritySet (IRQ_A, 0x60); // IRQ A has a priority value = 0 11
    IntPrioritySet (IRQ_B, 0x20); // IRQ B has a priority value = 0 01
  IntPriorityGroupingSet (1);     // Bit 1 of priority value is preemptable priority
                                  // Bits 2-3 are sub-priority levels
```

- NVIC groups interrupts according to preemptable priority
  - <u>Interrupts cannot be preempted by interrupts in the same group</u>

- Interrupts with same preempt priority level are ordered according to sub-priority levels
  - IRQ B will be handled first when IRQs A and B occur simultaneously

- What if IntPriorityGroupingSet (2) is used instead?

From Chapter 17 in TivaWare Peripheral Driver Library Users

Where we’re going today

- Interrupt performance measure
- Interrupt priority and preemption
- **More on ISR coding**
- Homework

Function calls in ISR (1)

From ADCdemo1.c

```
// ******************************************************************
// The handler for the ADC conversion complete interrupt.
// Writes to the circular buffer.
//******************************************************************
void  ADCIntHandler(void)
{
		uint32_t ulValue;
									:

		// Get the single sample from ADC0.  ADC_BASE is defined in inc/hw_memmap.h
		ADCSequenceDataGet(ADC0_BASE, 3, &ulValue);
		//
		// Place it in the circular buffer (and advance the write index (windex))
		writeCircBuf (&g_inBuffer, ulValue);
		//
		// Clean up, clearing the interrupt
		ADCIntClear(ADC0_BASE, 3);
}
```

Function calls in ISR (2)

```
void ADCIntHandler(void)
{
        uint32_t ulValue;
        //
        // Get the single sample from ADC0.  Alternative to call ADCSequenceDataGet ().
        ulValue = ADC0_SSFIFO3_R & UL_LS_10BITS_MASK;
        //
        // Place it in the circular buffer (advance index). Alternative to call writeCircBuf ().
        g_inBuffer.data[g_inBuffer.windex] = (int) ulValue;
        g_inBuffer.windex++;
        if (g_inBuffer.windex >= g_inBuffer.size)
                g_inBuffer.windex = 0;
        //
        // Clean up, clearing the interrupt  Alternative to call  ADCIntClear ().
        ADC0_ISC_R = ADC_ISC_IN3;
}
```

Function calls in ISR (3)

Version 1:

```
;*************************************************************
;* FUNCTION NAME: ADCIntHandler                              *
;*                                                           *
;*   Regs Modified  : A1,A2,A3,A4,V1,V9,SP,LR,SR             *
;*   Regs Used      : A1,A2,A3,A4,V1,V9,SP,LR,SR             *
;*   Local Frame Size  : 0 Args + 4 Auto + 8 Save = 12 byte  *
;*************************************************************
```

Version 2:

```
;*************************************************************
;* FUNCTION NAME: ADCIntHandler                              *
;*                                                           *
;*   Regs Modified  : A1,A2,A3,A4,V1,SP,SR                   *
;*   Regs Used      : A1,A2,A3,A4,V1,SP,LR,SR                *
;*   Local Frame Size  : 0 Args + 0 Auto + 8 Save = 8 byte   *
;*************************************************************
```

With vs. Without Function Calls in ISRs

- With function calls in ISRs
  - Extra overhead associated with the calls
    - The set of registers which need to be saved is larger
  - Prologue and epilogue intervals (thus the interrupt *response*) are likely to increase

- Without function calls in ISRs
  - **The code is often harder to understand and maintain** (because the hardware interface is less abstracted)
  - There will be less nesting in stack use

- **Suggestion**: unless timing is very critical, use function calls in ISRs

Homework

1. With reference to Slide 10, what does it mean to "mask" an interrupt?

2. With reference to Slide 18, explain carefully why the version which uses API calls within the ISRs is slower to service an interrupt.

3. What happens if the *prototype* for a function, for which the implementation appears in a source file 'fancy_functions.c', does not appear in 'fancy_functions.h'?

4. In the case mentioned in Q3, where else, other than in a header file, might a prototype for a function correctly appear and why?