NUS
National University
of Singapore

Cortex-M0+
Exceptions and Interrupts

Ravi Suppiah
Lecturer, NUS SoC

The Architecture for the Digital World®

ARM

Overview

NUS National University of Singapore

- Exception and Interrupt Concepts
  - Entering an Exception Handler
  - Exiting an Exception Handler
- Cortex-M0+ Interrupts
  - Using Port Module and External Interrupts
- Timing Analysis
- Program Design with Interrupts
  - Sharing Data Safely Between ISRs and Other Threads

- Sources
  - Cortex M0+ Device Generic User Guide - DUI0662
  - Cortex M0+ Technical Reference Manual - DUI0484

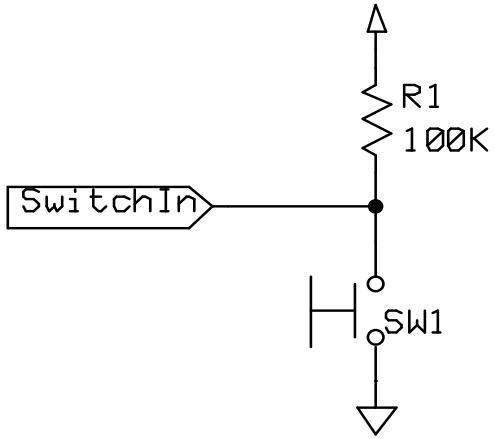
ARM

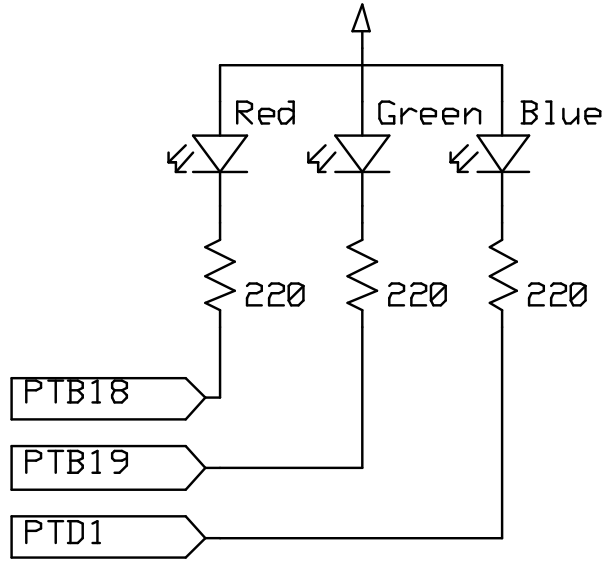
NUS
National University
of Singapore

EXCEPTION AND INTERRUPT CONCEPTS

ARM

Example System with Interrupt





- Goal: Change color of RGB LED when switch is pressed
- Will explain details of interfacing with switch and LEDs in GPIO module later
- Need to add external switch

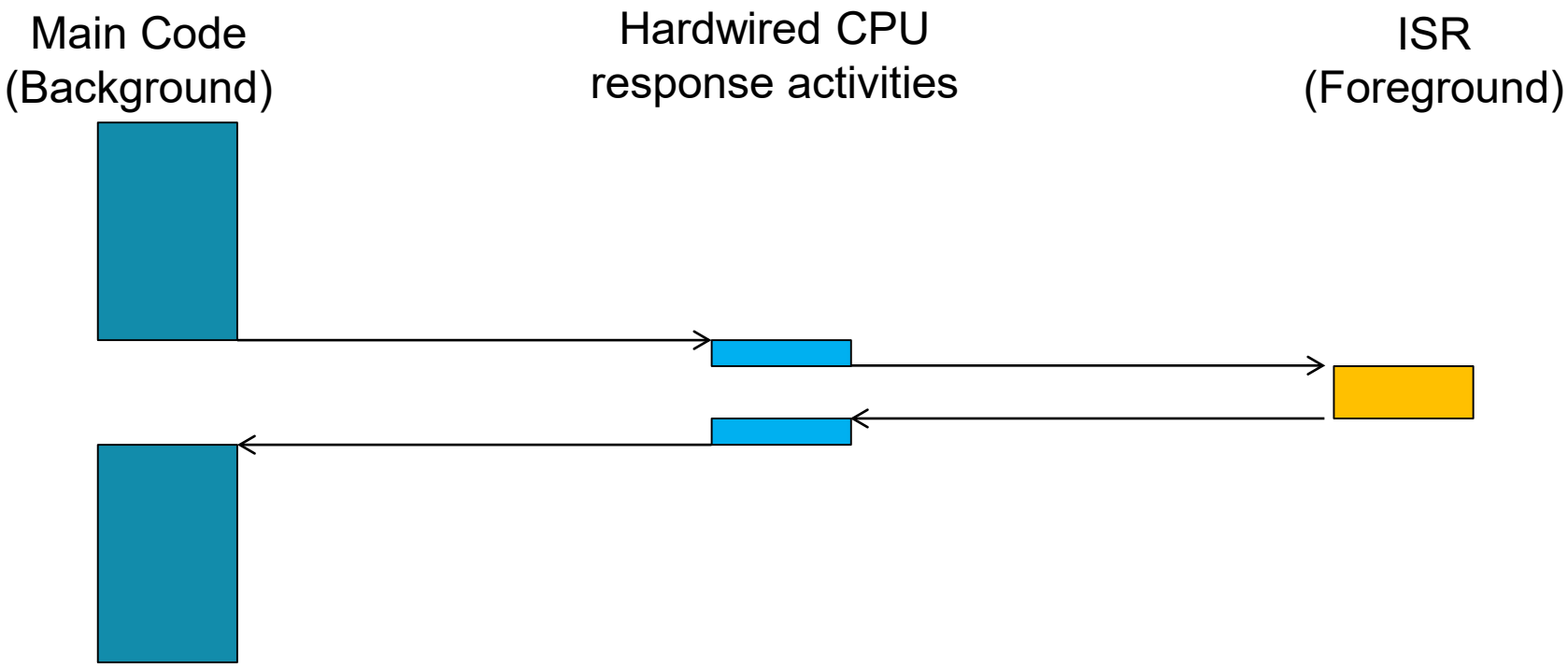
How to Detect Switch is Pressed?

- Polling - use software to check it
  - Slow - need to explicitly check to see if switch is pressed
  - Wasteful of CPU time - the faster a response we need, the more often we need to check
  - Scales badly - difficult to build system with many activities which can respond quickly. Response time depends on all other processing.

- Interrupt - use special hardware in MCU to detect event, run specific code (*interrupt service routine* - ISR) in response
  - Efficient - code runs only when necessary
  - Fast - hardware mechanism
  - Scales well
    - ISR response time doesn't depend on most other processing.
    - Code modules can be developed independently

Interrupt or Exception Processing Sequence

- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code



Interrupts

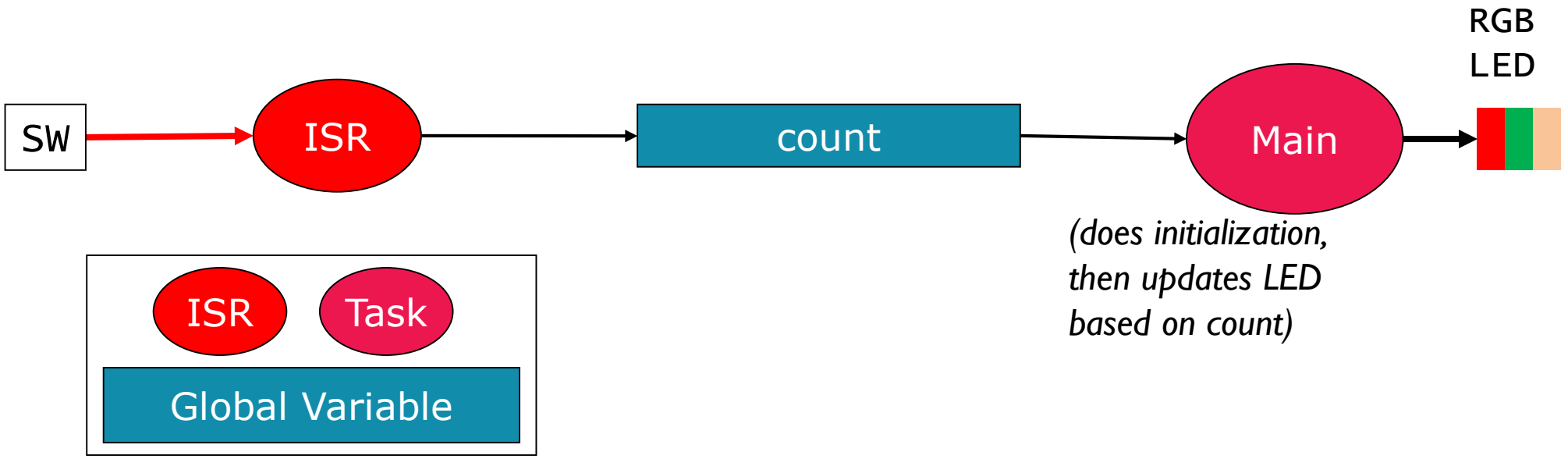
NUS National University of Singapore

- Hardware-triggered asynchronous software routine
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous - can happen anywhere in the program (unless interrupt is disabled)
  - Software routine - Interrupt service routine runs in response to interrupt

- Fundamental mechanism of microcontrollers
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events regardless* of program state, complexity, location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

ARM

Example Program Requirements & Design



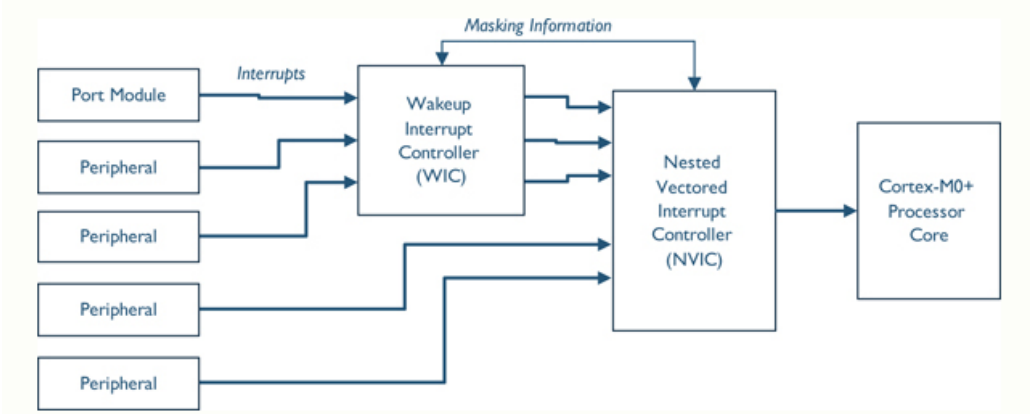
- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)

CORTEX-M0+ INTERRUPTS

Microcontroller Interrupts

- Types of interrupts
  - Hardware interrupts
    - ***Asynchronous***: not related to what code the processor is currently executing
    - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
  - Exceptions, Faults, software interrupts
    - ***Synchronous***: are the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed *(maskable)*, others are *non-maskable*
- Interrupt service routine (ISR)
  - Subroutine which processor is ***forced to execute*** to respond to a ***specific event***
  - After ISR completes, MCU goes back to previously executing code

Nested Vectored Interrupt Controller



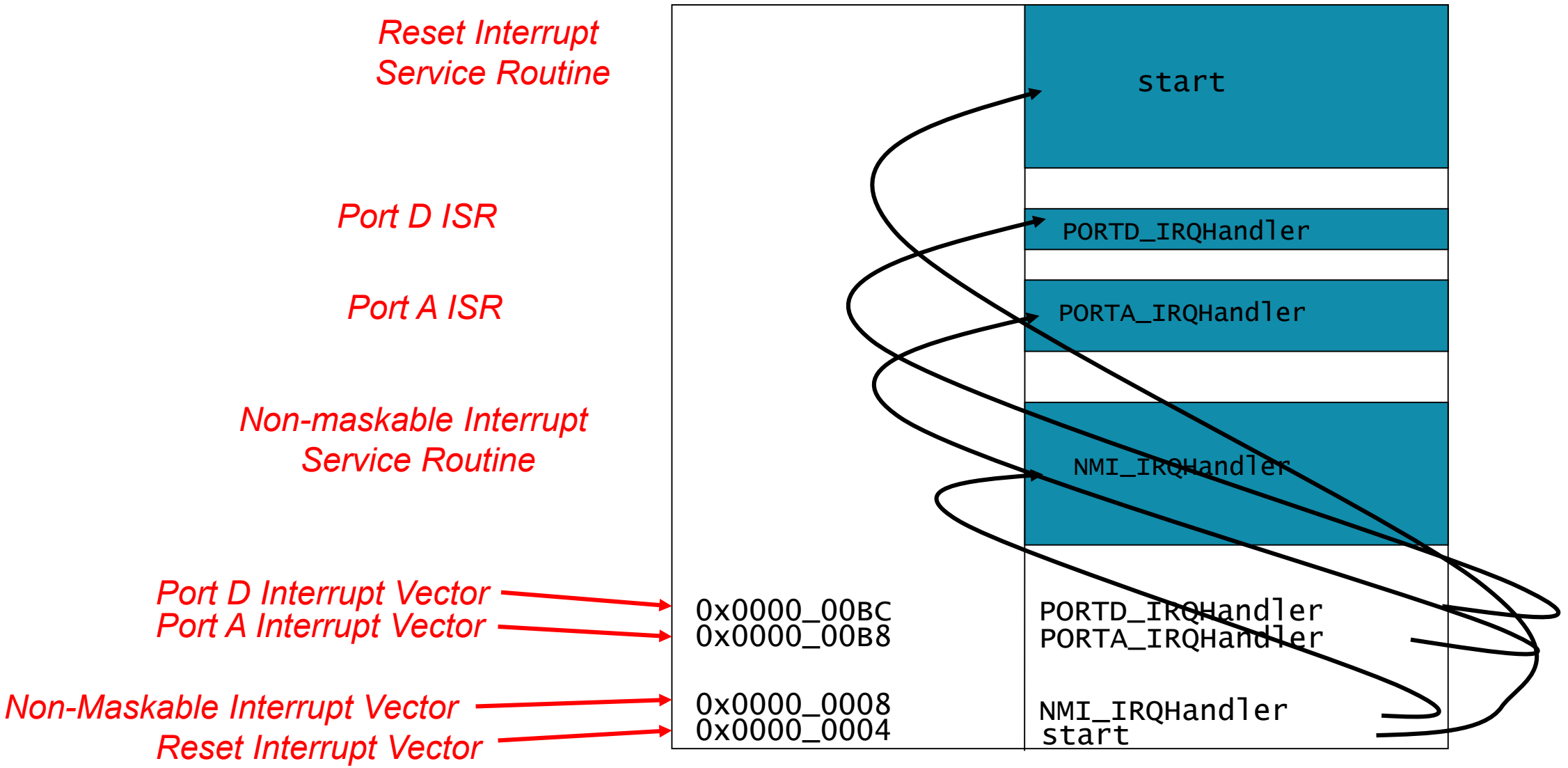
- NVIC manages and prioritizes external interrupts for Cortex-M0+
- Interrupts are types of exceptions
  - Exceptions 16 through 16+N
- Modes
  - Thread Mode: entered on Reset
  - Handler Mode: entered on executing an exception
- Privilege level
- Stack pointers
  - Main Stack Pointer, MSP
  - Process Stack Pointer, PSP
- Exception states: Inactive, Pending, Active

Some Interrupt Sources (Partial)

Vector Start Address	Vector #	IRQ	Source	Description
0x0000_0004	1		ARM Core	Initial program counter / Reset
0x0000_0008	2		ARM Core	Non-maskable interrupt
0x0000_0040-4C	16-19	0-3	Direct Memory Access Controller	Transfer complete or error
0x0000_0058	22	6	Power Management Controller	Low voltage detection
0x0000_0060-64	24-25	8-9	I^2C Modules	Status and error
0x0000_0068-6C	26-27	10-11	SPI Modules	Status and error
0x0000_0070-78	28-30	12-14	UART Modules	Status and error
0x0000_00B8	46	30	Port Control Module	Port A Pin Detect
0x0000_00BC	47	31	Port Control Module	Port D Pin Detect

Up to 32 non-core vectors, 16 core vectors
From KL25 Sub-Family Reference Manual, Table 3-7

What's in the Vector Table?



NVIC Registers and State

Bits	31:30	29:24	23:22	21:16	15:14	13:8	7:6	5:0
IPR0	IRQ3	reserved	IRQ2	reserved	IRQ1	reserved	IRQ0	reserved
IPR1	IRQ7	reserved	IRQ6	reserved	IRQ5	reserved	IRQ4	reserved
IPR2	IRQ11	reserved	IRQ10	reserved	IRQ9	reserved	IRQ8	reserved
IPR3	IRQ15	reserved	IRQ14	reserved	IRQ13	reserved	IRQ12	reserved
IPR4	IRQ19	reserved	IRQ18	reserved	IRQ17	reserved	IRQ16	reserved
IPR5	IRQ23	reserved	IRQ22	reserved	IRQ21	reserved	IRQ20	reserved
IPR6	IRQ27	reserved	IRQ26	reserved	IRQ25	reserved	IRQ24	reserved
IPR7	IRQ31	reserved	IRQ30	reserved	IRQ29	reserved	IRQ28	reserved

- Priority - allows program to prioritize response if both interrupts are requested simultaneously
  - IPR0-7 registers: two bits per interrupt source, four interrupt sources per register
  - Set priority to 0 (highest priority), 64, 128 or 192 (lowest)
  - CMSIS: NVIC_SetPriority(IRQnum, priority), with priority set as 0, 1, 2 or 3

NVIC Registers and State

- Enable - Allows interrupt to be recognized
  - Accessed through two registers (set bits for interrupts)
    - Set enable with NVIC_ISER, clear enable with NVIC_ICER
  - CMSIS Interface: NVIC_EnableIRQ(IRQnum), NVIC_DisableIRQ(IRQnum)

- Pending - Interrupt has been requested but is not yet serviced
  - CMSIS: NVIC_SetPendingIRQ(IRQnum), NVIC_ClearPendingIRQ(IRQnum)

Core Exception Mask Register

- Similar to "Global interrupt disable" bit in other MCUs

- PRIMASK - Exception mask register (CPU core)
  - Bit 0: PM Flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exceptions
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity

- CMSIS-CORE API
  - void __enable_irq() - clears PM flag
  - void __disable_irq() - sets PM flag
  - uint32_t __get_PRIMASK() - returns value of PRIMASK
  - void __set_PRIMASK(uint32_t x) - sets PRIMASK to x

Prioritization

- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)

- Priorities of some exceptions are ***fixed***
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1

- Priorities of other (peripheral) exceptions are ***adjustable***
  - Value is stored in the interrupt priority register (IPR0-7)
  - 0x00
  - 0x40
  - 0x80
  - 0xC0

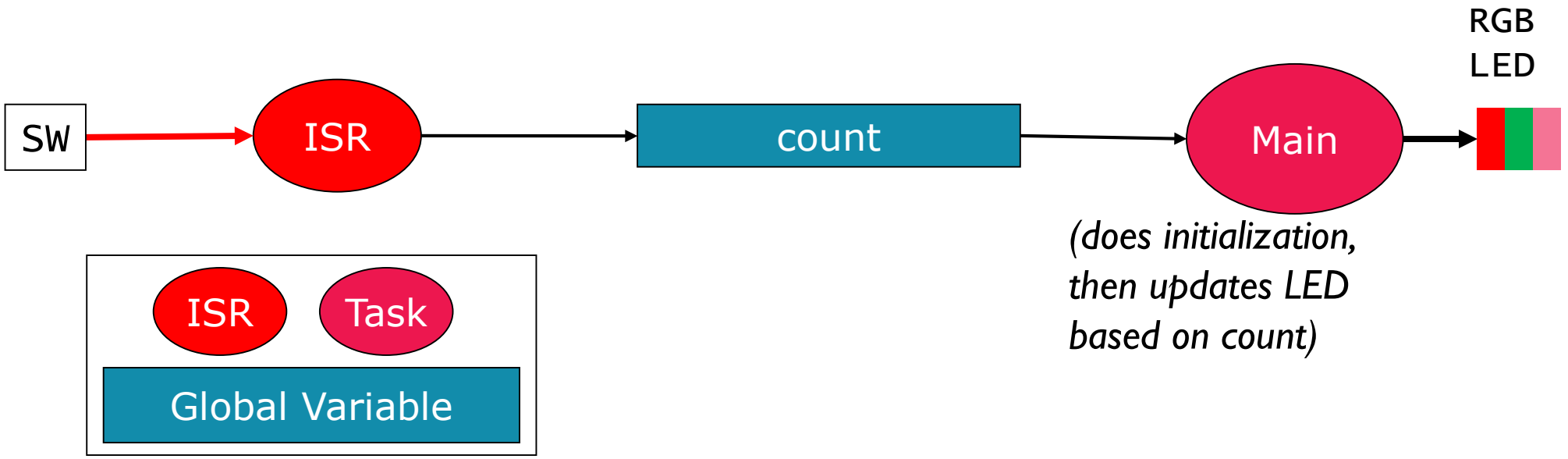
Special Cases of Prioritization

- Simultaneous exception requests?
  - Lowest exception type number is serviced first

- New exception requested while a handler is executing?
  - New priority higher than current priority?
    - New exception handler **preempts** current exception handler
  - New priority lower than or equal to current priority?
    - New exception held in **pending state**
    - Current handler continues and completes execution
    - Previous priority level restored
    - New exception handled if priority level allows

EXAMPLE USING PORT MODULE AND EXTERNAL INTERRUPTS

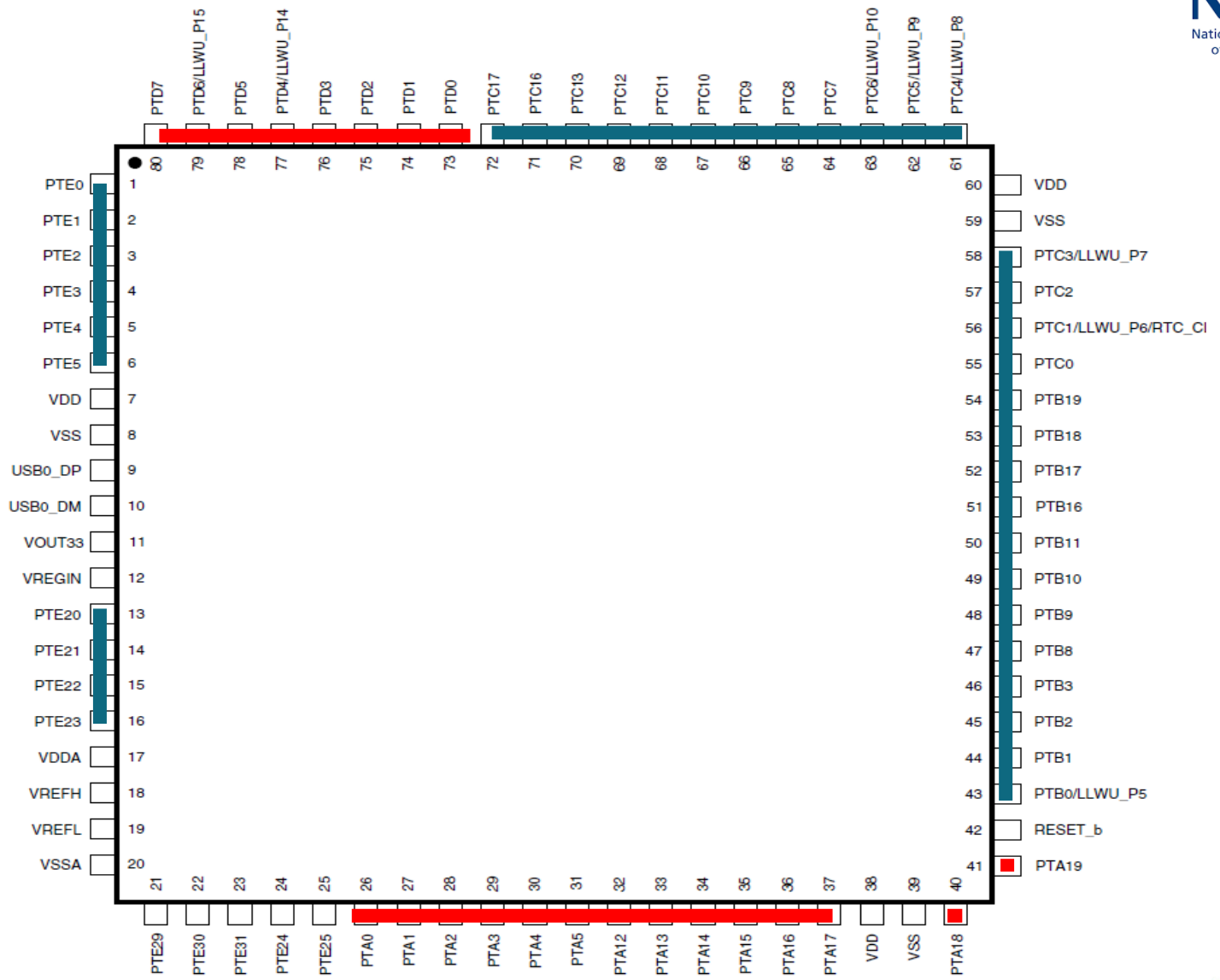
Refresher: Program Requirements & Design



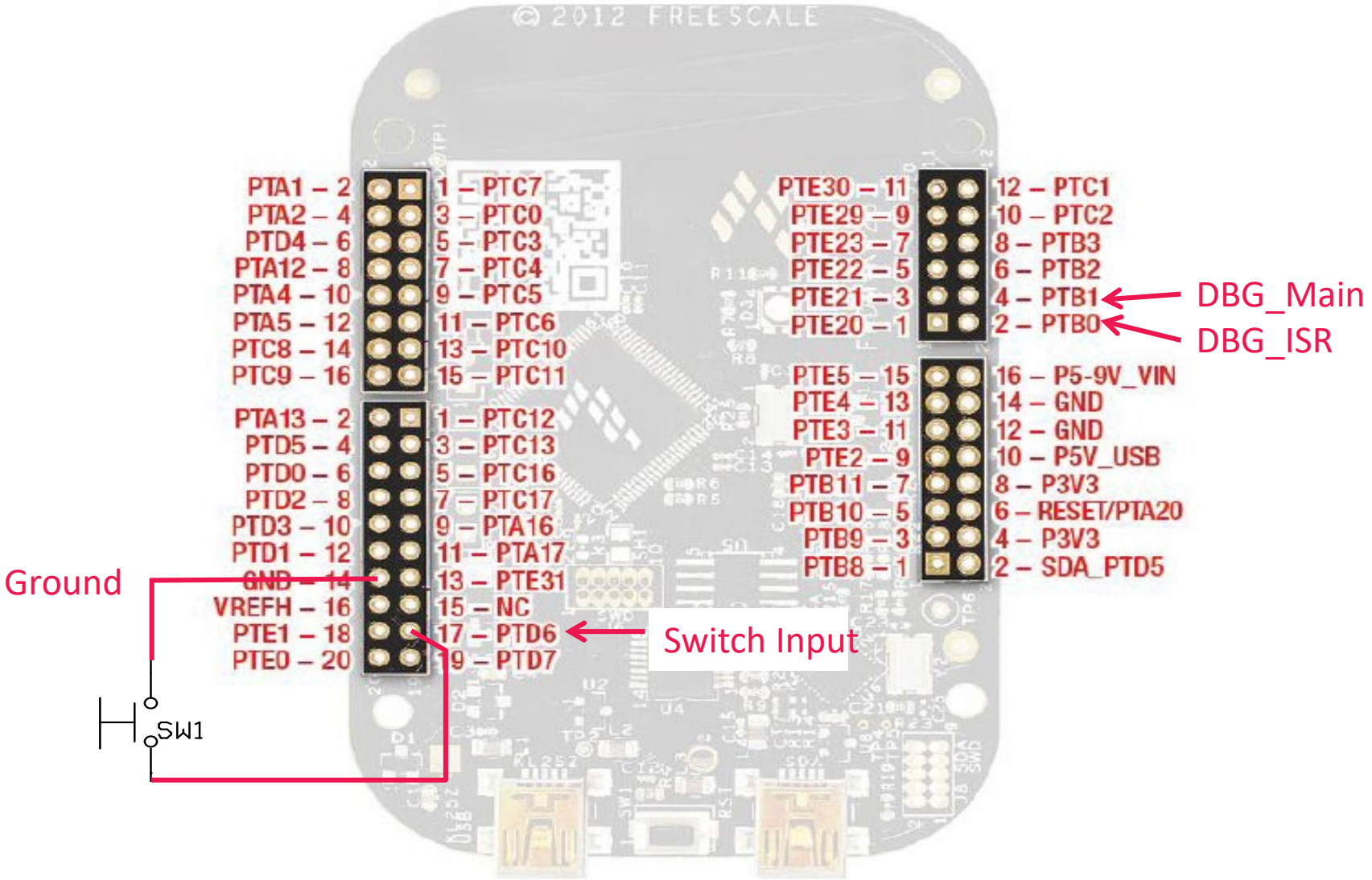
- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- *Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- *Req4: ISR will raise its debug line DBG_ISR (and lower main's debug line DBG_MAIN) whenever it is executing

KL25Z GPIO Ports with Interrupts

- Port A (PTA) through Port E (PTE)
- Not all port bits are available (package-dependent)
- Ports A and D support interrupts



FREEDOM KL25Z Physical Set-up



Building a Program – Break into Pieces

- First break into threads, then break thread into steps
  - Main thread:
    - First initialize system
      - initialize switch: configure the port connected to the switches to be input
      - initialize LEDs: configure the ports connected to the LEDs to be outputs
      - initialize interrupts: initialize the interrupt controller
  - Then repeat
    - Update LEDs based on count
  - Switch Interrupt thread:
    - Update count

- Determine which variables ISRs will share with main thread
  - This is how ISR will send information to main thread
  - Mark these shared variables as *volatile* (more details ahead)
  - Ensure access to the shared variables is *atomic* (more details ahead)

Where Do the Pieces Go?

- main
  - top level of main thread code
- switches
  - #defines for switch connections
  - declaration of count variable
  - Code to initialize switch and interrupt hardware
  - ISR for switch
- LEDs
  - #defines for LED connections
  - Code to initialize and light LEDs
- debug_signals
  - #defines for debug signal locations
  - Code to initialize and control debug lines

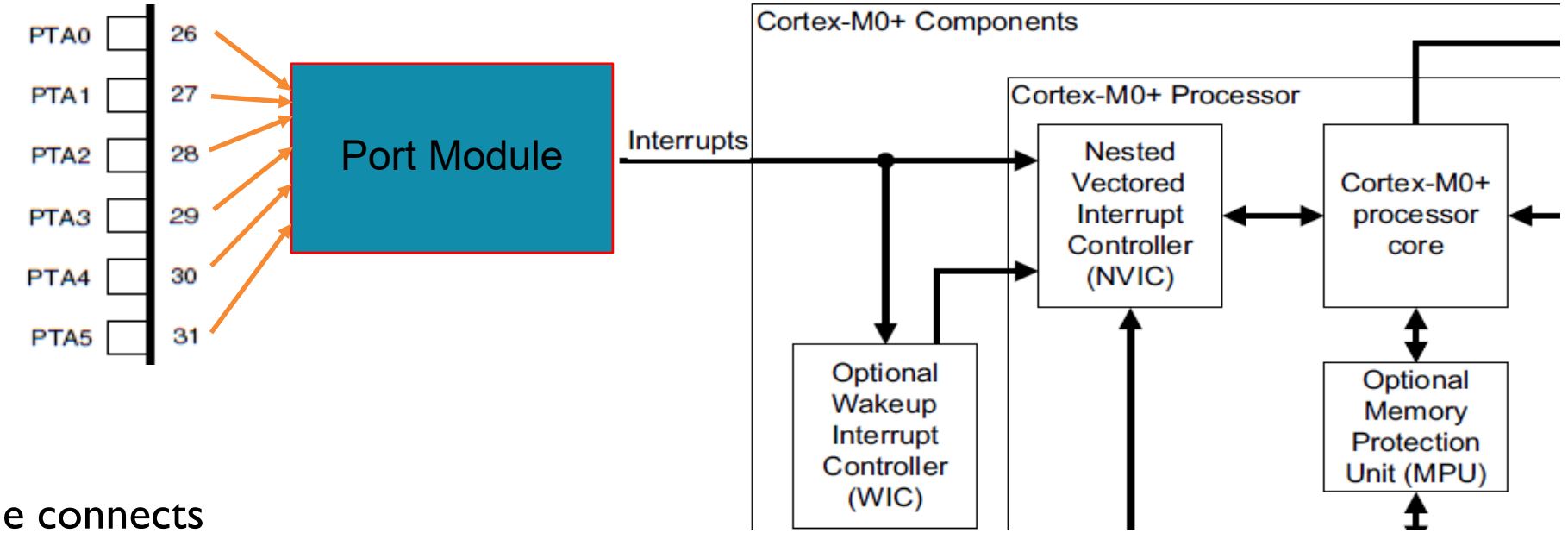
Configure MCU to Respond to the Interrupt

NUS National University of Singapore

- Set up peripheral module to generate interrupt
  - We'll use Port Module in this example
- Set up NVIC
- Set global interrupt enable
  - Use CMSIS Macro __enable_irq();
  - This flag does not enable all interrupts; instead, it is an easy way to ***disable*** interrupts
    - Could also be called "don't disable all interrupts"

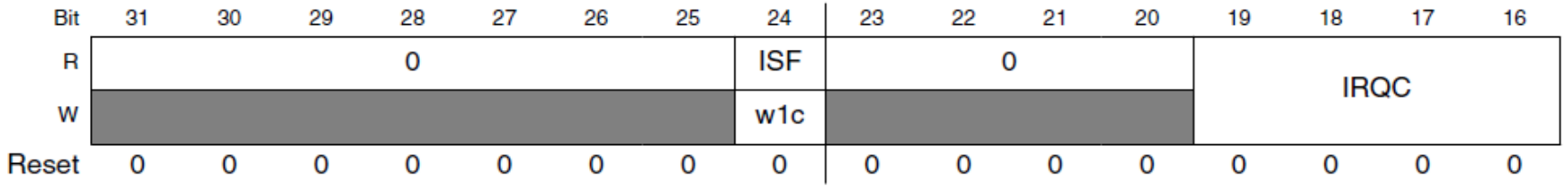
ARM

Port Module



- Port Module connects external pins to NVIC (and other devices)
- Relevant registers
  - PCR - Pin control register (32 per port)
    - Each register corresponds to an input pin
  - ISFR - Interrupt status flag register (one per port)
    - Each bit corresponds to an input pin
    - Bit is set to 1 if an interrupt has been detected

Pin Control Register



- ISF indicates if interrupt has been detected - different way to access same data as ISFR
- IRQC field of PCR defines behavior for external hardware interrupts
- Can also trigger direct memory access (not covered here)

IRQC	Configuration
0000	Interrupt Disabled
….	DMA, reserved
1000	Interrupt when logic zero
1001	Interrupt on rising edge
1010	Interrupt on falling edge
1011	Interrupt on either edge
1100	Interrupt when logic one
…	reserved

ARM

Let's examine the Code…

- Setup LED's and Switch
  - Clock Gating
  - MUX Setting
  - Data Direction
  - Pull-Up Enable
  - Set Interrupt Priority
  - Enable Interrupt

Write Interrupt Service Routine

- No arguments or return values – void is only valid type
- Keep it short and simple
  - Much easier to debug
  - Improves system response time
- Name the ISR according to CMSIS-CORE system exception names
  - PORTD_IRQHandler, RTC_IRQHandler, etc.
  - The linker will load the vector table with this handler
- Clear pending interrupts
  - Call NVIC_ClearPendingIRQ(IRQnum)
- Read interrupt status flag register to determine source of interrupt
- Clear interrupt status flag register by writing to PORTD->ISFR

ISR

```
void PORTD_IRQHandler(void) {

      // clear pending interrupts
      NVIC_ClearPendingIRQ(PORTD_IRQn);

      if ((PORTD->ISFR & MASK(SW_POS))) {
            count++;
      }
      // clear status flags
      PORTD->ISFR = 0xffffffff;

}
```

Evaluate Basic Operation

NUS National University of Singapore

- Build program
- Load onto development board
- Start debugger
- Run
- Press switch, verify LED changes color

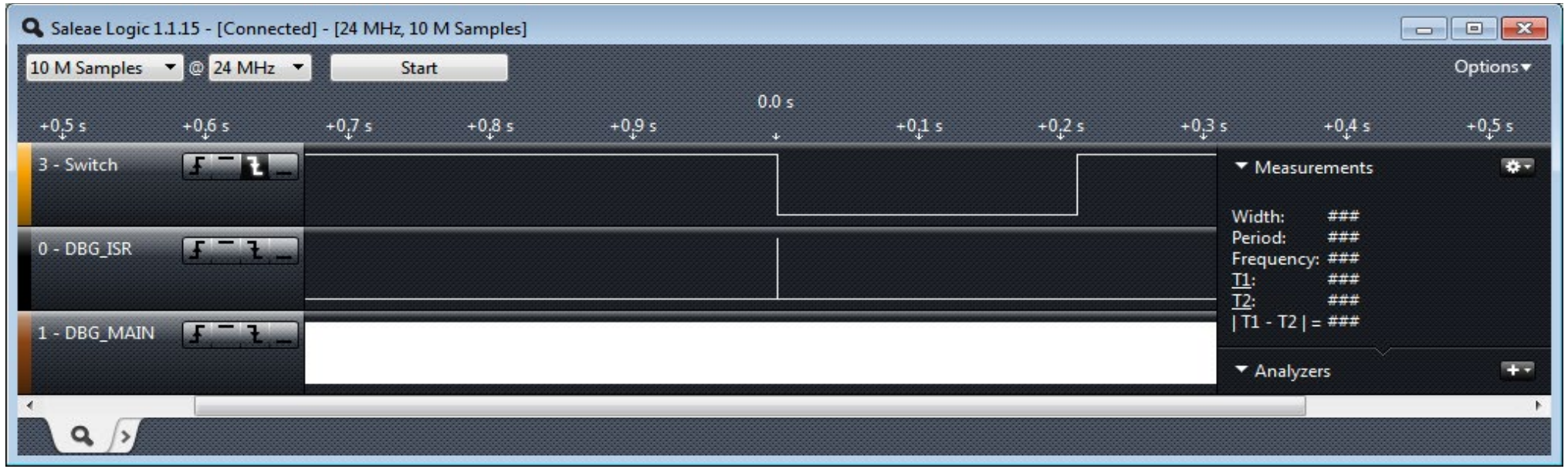
ARM

Examine Saved State in ISR

- Set breakpoint in ISR
- Run program
- Press switch, verify debugger stops at breakpoint
- Examine stack and registers

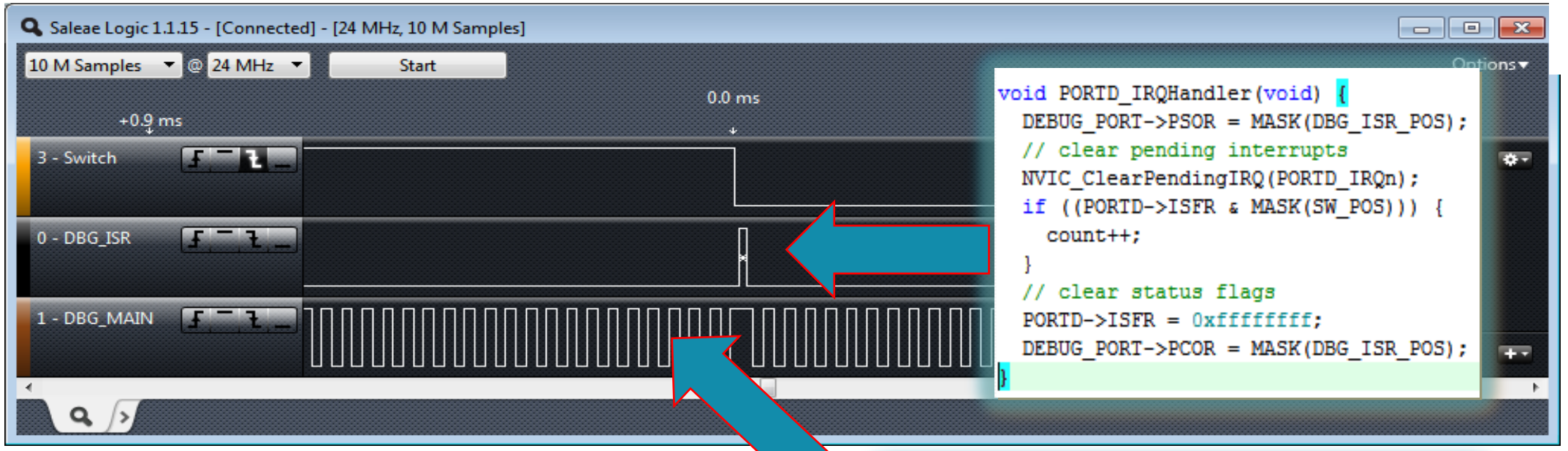
TIMING ANALYSIS

Big Picture Timing Behavior



- Switch was pressed for about 0.21 s
- ISR runs in response to switch signal's falling edge
- Main seems to be running continuously (signal toggles between 1 and 0)
  - Does it really?

Visualizing the Timing of Processor Activities




```
void PORTD_IRQHandler(void) {
  DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
  // clear pending interrupts
  NVIC_ClearPendingIRQ(PORTD_IRQn);
  if ((PORTD->ISFR & MASK(SW_POS))) {
    count++;
  }
  // clear status flags
  PORTD->ISFR = 0xffffffff;
  DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
}
```

- Indicate CPU activity by controlling debug output signals (GPIO)

```
while (1) {
  DEBUG_PORT->PTOR = MASK(DBG_MAIN_POS);
  control_RGB_LEDs(count&1, count&2, count&4);
}
```

Interrupt Response Latency

- Latency = time delay

- Why do we care?
  - This is overhead which wastes time, and increases as the interrupt rate rises
  - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform

- How long does it take?
  - Finish executing the current instruction or abandon it
  - Push various registers on to the stack, fetch vector
    - $C_{IntResponseOvhd}$: Overhead for responding to each interrupt)
  - If we have external memory with wait states, this takes longer

Maximum Interrupt Rate

- We can only handle so many interrupts per second
  - F_{Max_Int}: maximum interrupt frequency
  - F_{CPU}: CPU clock frequency
  - C_{ISR}: Number of cycles ISR takes to execute
  - $C_{Overhead:}$ Number of cycles of overhead for saving state, vectoring, restoring state, etc.
  - $F_{Max\_Int} = F_{CPU}/(C_{ISR+} C_{Overhead})$
  - Note that model applies only when there is one interrupt in the system

- When processor is responding to interrupts, it isn't executing our other code
  - U_{Int}: Utilization (fraction of processor time) consumed by interrupt processing
  - $U_{Int} = 100\%*F_{Int}* (C_{ISR}+C_{Overhead})/ F_{CPU}$
  - CPU looks like it's running the other code with CPU clock speed of $(1-U_{Int})*F_{CPU}$

NUS
National University
of Singapore

PROGRAM DESIGN WITH INTERRUPTS

ARM

Program Design with Interrupts

- How much work to do in ISR?
- Should ISRs re-enable interrupts?
- How to communicate between ISR and other threads?
  - Data buffering
  - Data integrity and race conditions

How Much Work Is Done in ISR?

- Trade-off: Faster response for ISR code will delay completion of other code
- In system with multiple ISRs with short deadlines, perform critical work in ISR and buffer partial results for later processing

NUS
National University
of Singapore

SHARING DATA SAFELY BETWEEN ISRS AND OTHER THREADS

ARM

Overview

- Volatile data – can be updated outside of the program's immediate control
- Non-atomic shared data – can be interrupted partway through read or write, is vulnerable to race conditions

Definitions

- Race condition: Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of example code depends on the *relative timing* of the read and write operations.

- Critical section: A section of code which creates a possible race condition. The code section can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

The End!

- Thank You!

ARM