

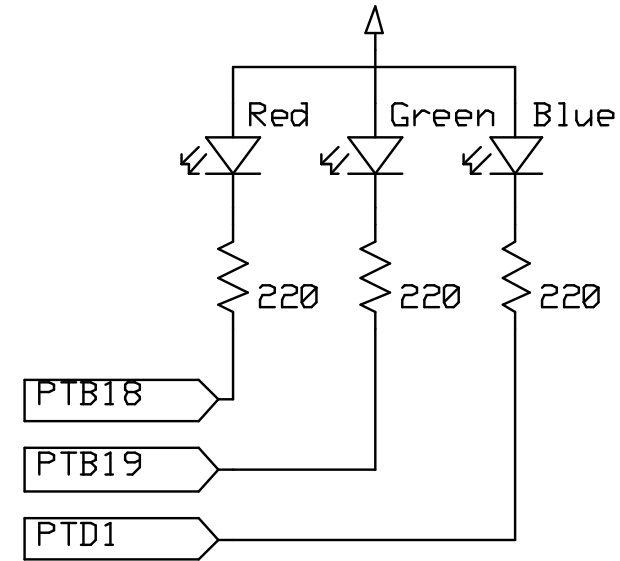
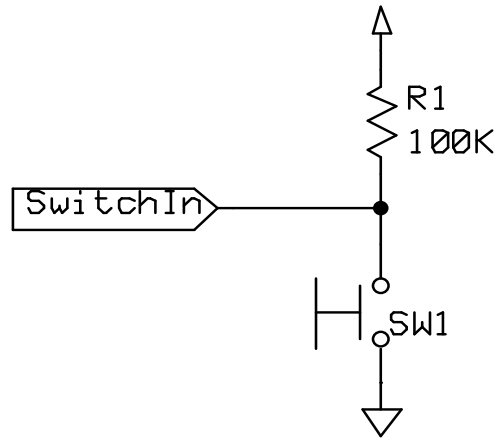
Module 3 - Cortex-M0+ Exceptions and Interrupts

Overview

- 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

EXCEPTION AND INTERRUPT CONCEPTS

Example System with Interrupt



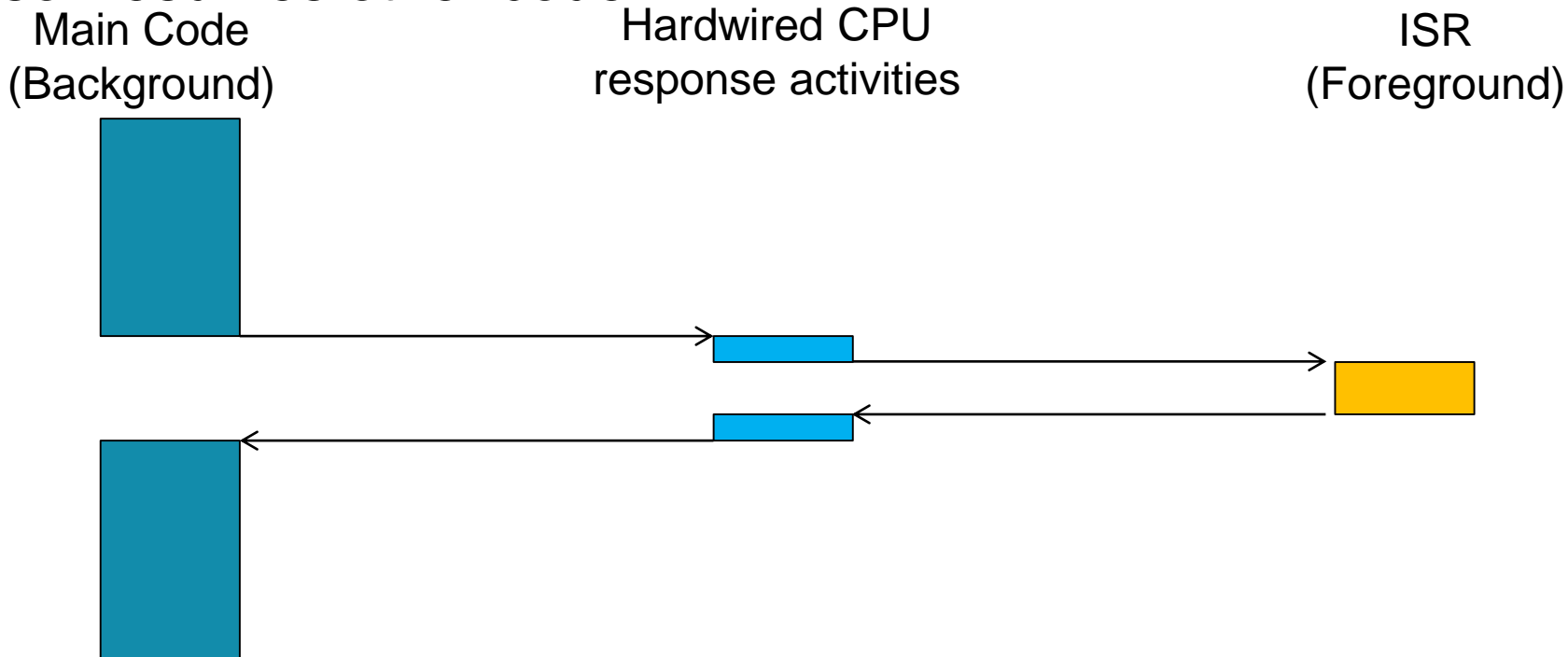
- Goal: Change color of RGB LED when switch is pressed
- Need to add external switch

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

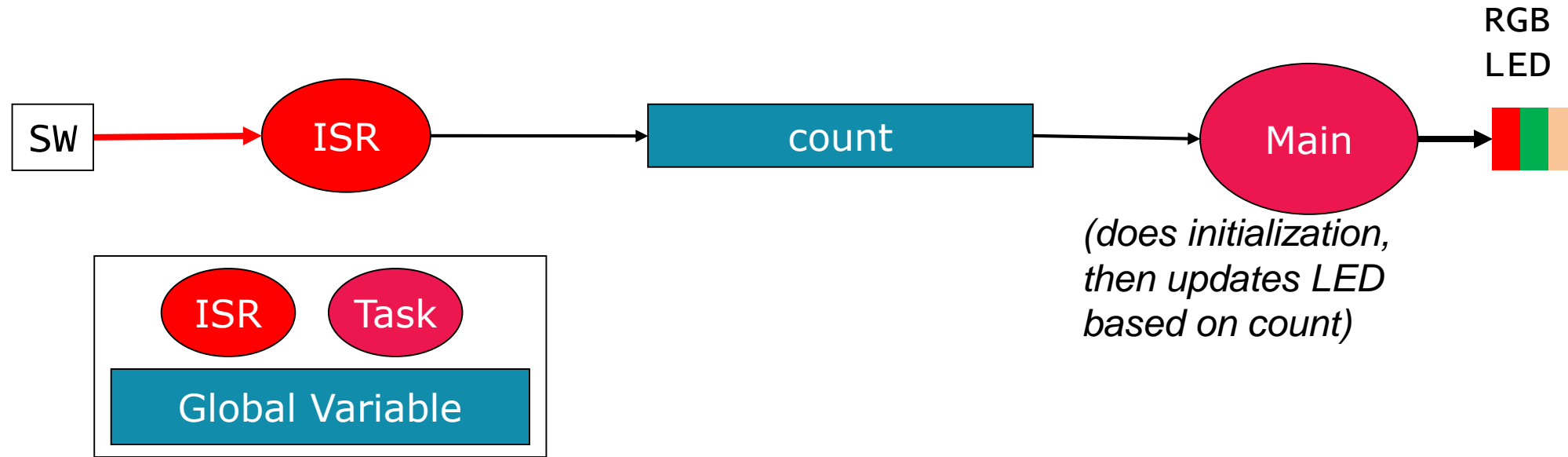
- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code



Interrupts

- 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)

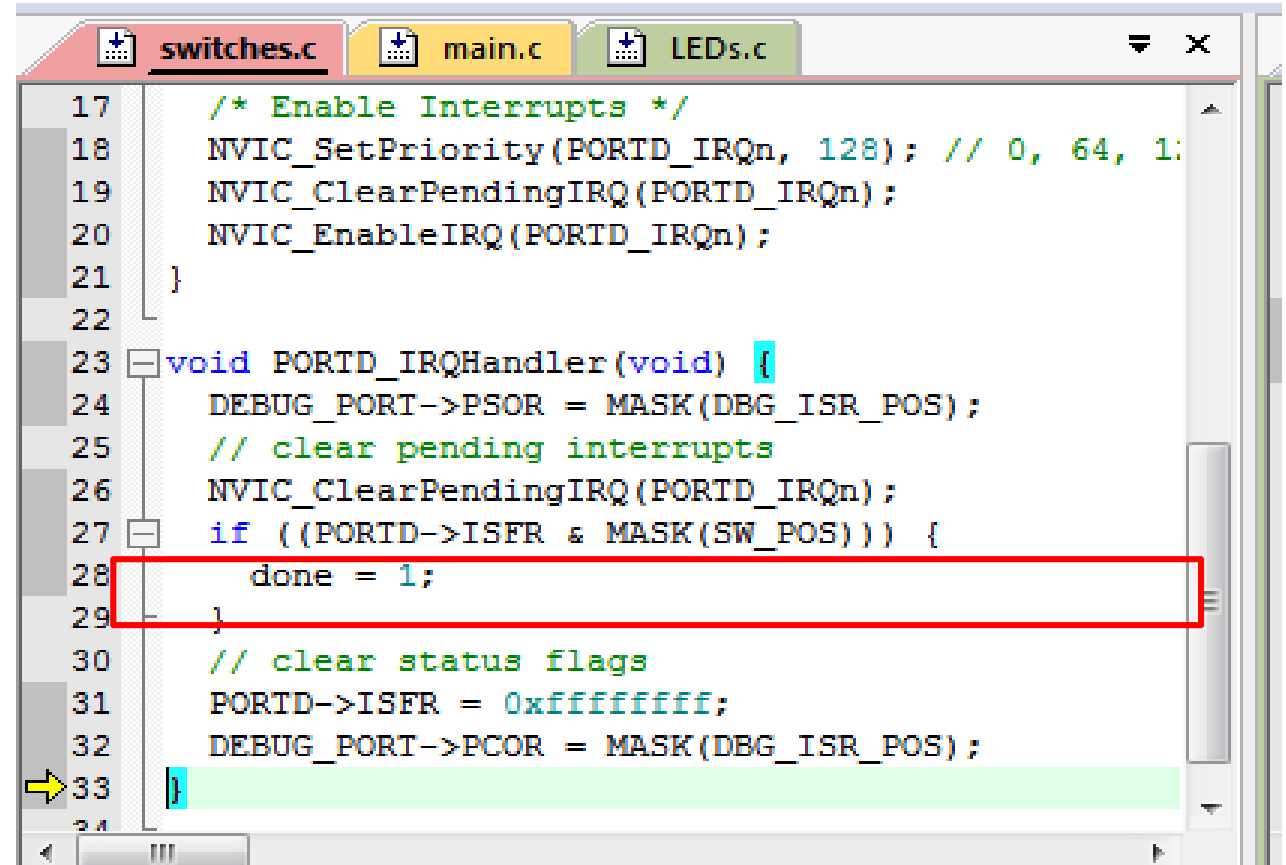
Example Program Requirements & Design



- 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)
- Req3: Main code will toggle its debug line each time it executes
- Req4: ISR will raise its debug line (and lower main's debug line) whenever it is executing

Example Exception Handler

- We will examine processor's response to exception in detail



```
switches.c  main.c  LEDs.c
17  /* Enable Interrupts */
18  NVIC_SetPriority(PORTD_IRQn, 128); // 0, 64, 1:
19  NVIC_ClearPendingIRQ(PORTD_IRQn);
20  NVIC_EnableIRQ(PORTD_IRQn);
21  }
22
23  void PORTD_IRQHandler(void) {
24      DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
25      // clear pending interrupts
26      NVIC_ClearPendingIRQ(PORTD_IRQn);
27      if ((PORTD->ISFR & MASK(SW_POS))) {
28          done = 1;
29      }
30      // clear status flags
31      PORTD->ISFR = 0xffffffff;
32      DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
33  }
```

Use Debugger for Detailed Processor View

- Can see registers, stack, source code, disassembly (object code)
- Note: Compiler may generate code for function entry (see address 0x0000_0454)
- Place breakpoint on Handler function declaration line in source code (23), not at first line of function code (24)

The screenshot displays a debugger interface with four main panes:

- Registers:** A table showing the state of various registers. The PC (R15) is at 0x00000456.
- Disassembly:** A list of machine code instructions with their addresses. Address 0x00000456 is highlighted, corresponding to the MOV instruction.
- Source Code:** The C source code for the `PORTD_IRQHandler` function. Line 23, where the function is declared, is highlighted with a red dot and a breakpoint.
- Memory:** A view of the stack memory starting at address 0x1FFFF3E8.

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFEE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFFF3E8
R14 (LR)	0xFFFFF9
R15 (PC)	0x00000456
xPSR	0x0100002F

```
23: void PORTD_IRQHandler(void) {
24:     DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
25:     // clear pending interrupts
26:     NVIC_ClearPendingIRQ(PORTD_IRQn);
27:     if ((PORTD->ISFR & MASK(SW_POS))) {
28:         done = 1;
29:     }
30:     // clear status flags
31:     PORTD->ISFR = 0xffffffff;
32:     DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
33: }
```

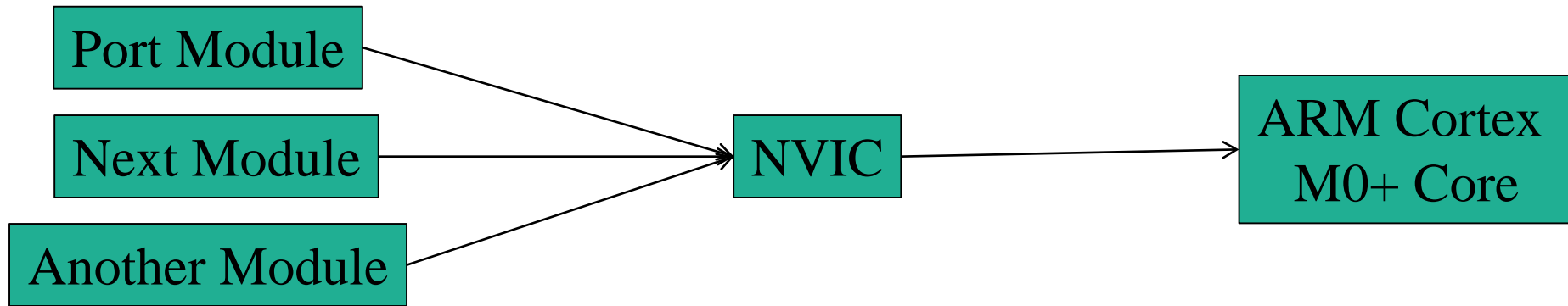
Address	Value
0x1FFFF3E8	00000462 FFFFFFFF 00000000 00000000
0x1FFFF3FC	00000002 00000000 0000034F 00000352
0x1FFFF410	66178BE3 57AC823B F0AA6C90 4B8BCE07
0x1FFFF424	EFC30BDF AA0A209D 4EA993EB 4093A563
0x1FFFF438	EA1ADA60 7E7F3B8B 4FE50B6E 7B9F7C9E

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, A&P

Some Interrupt Sources (Partial)

Vector Start Address	Vector #	IRQ	Source	Description
0x0000_0004	1		ARM Core	Initial program counter
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 ² C 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 3-6

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)

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

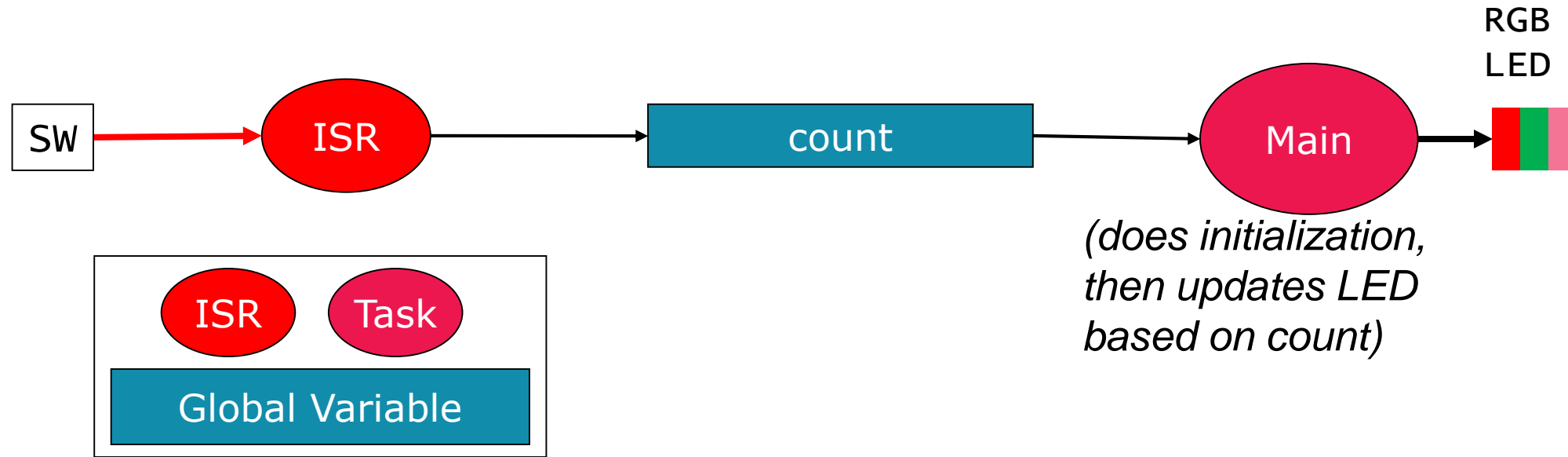
- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are ***fixed***
 - Reset: -3, highest priority
 - NMI: -2
 - Hard Fault: -1
- Priorities of other (peripheral) exceptions are ***adjustable***
 - Value is stored 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

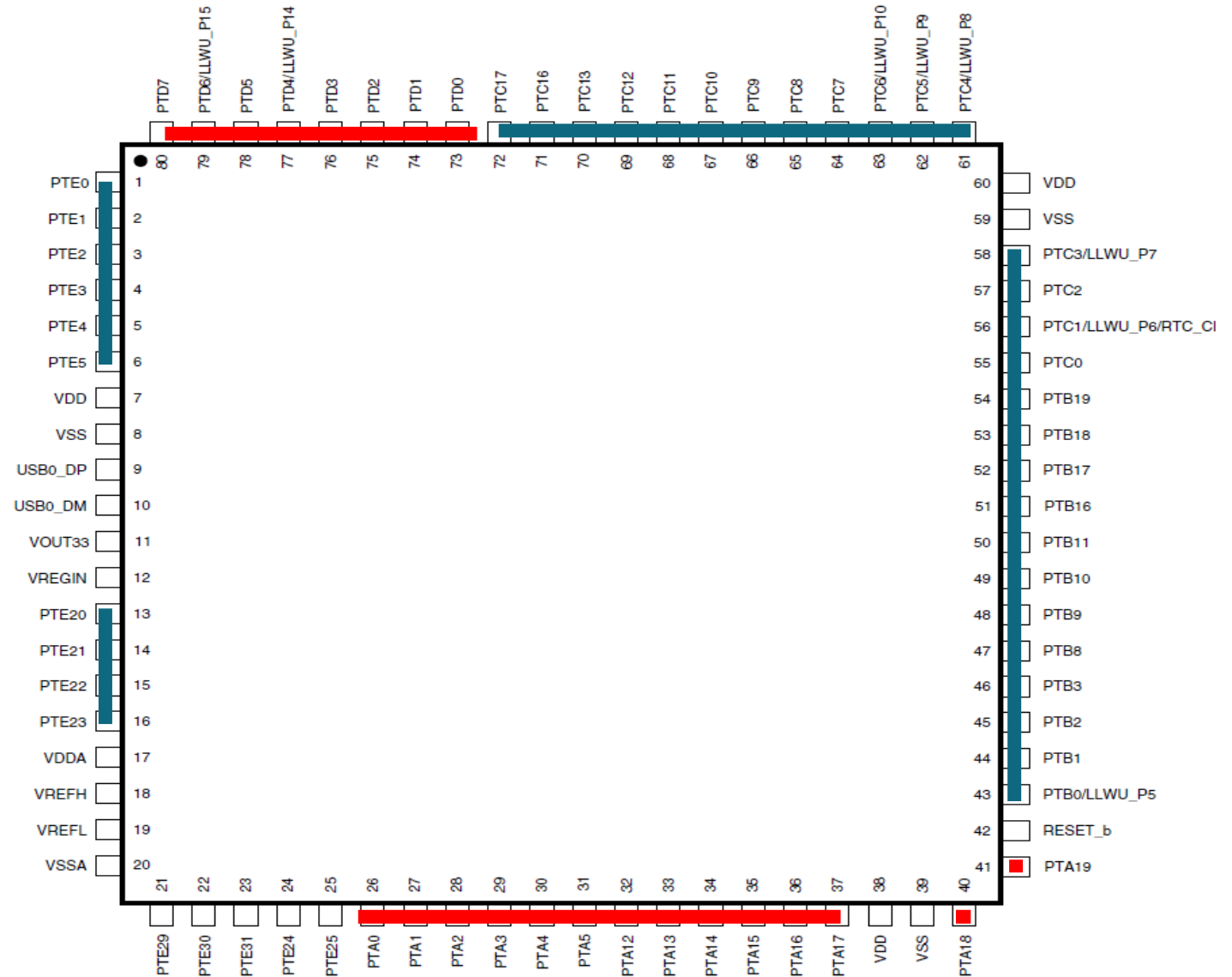
Refresher: Program Requirements & Design



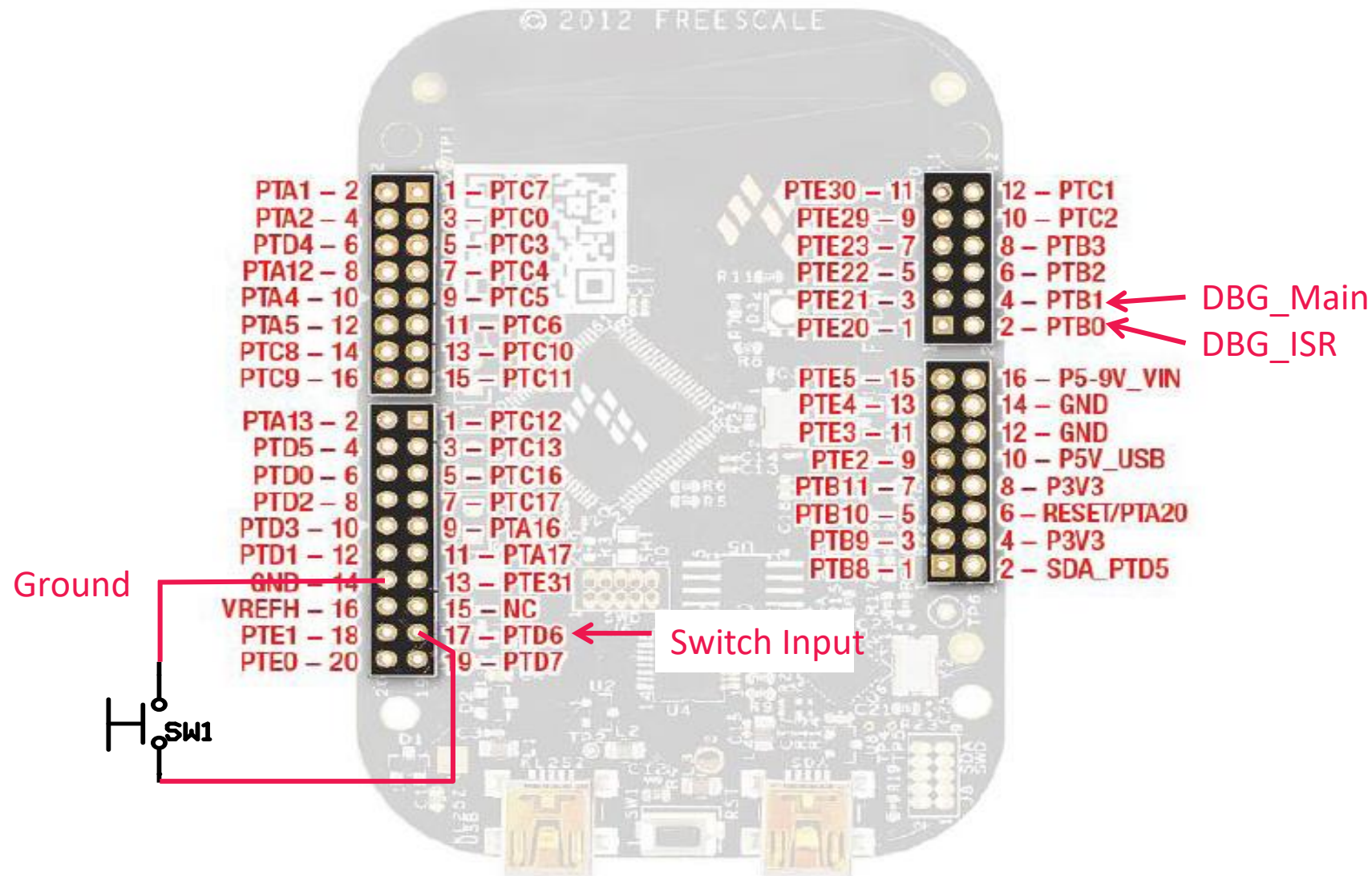
- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- Req4: ISR will raise its debug line DBG_ISR (and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

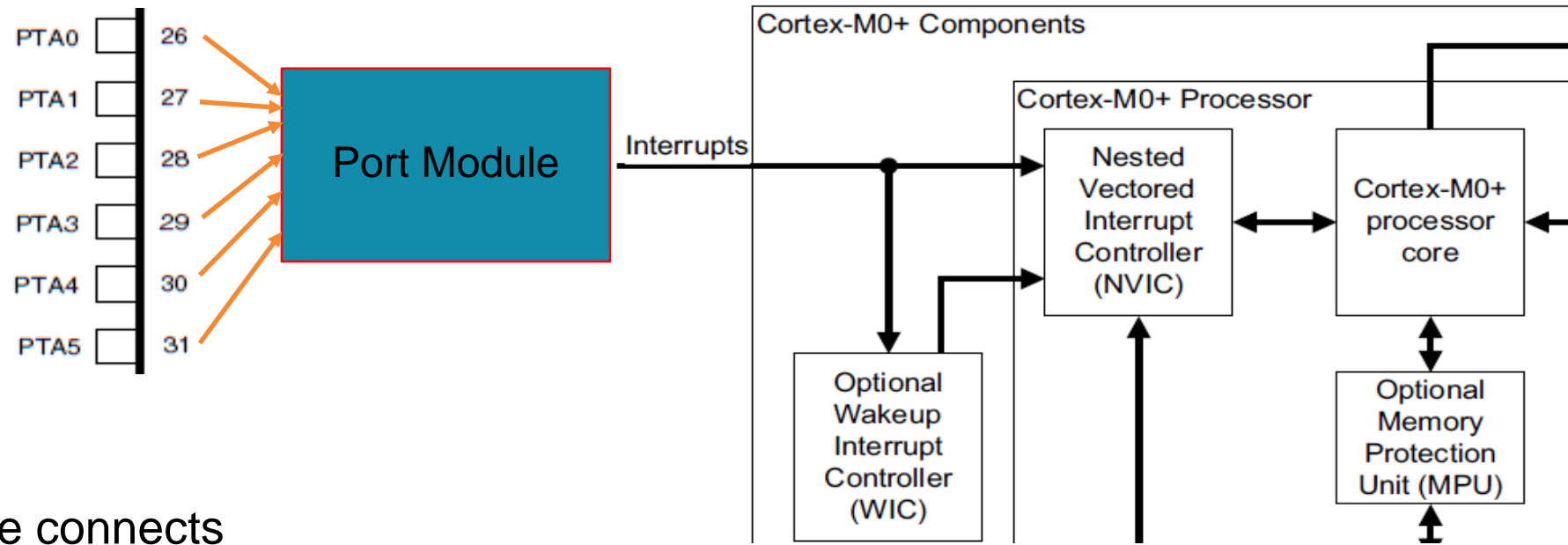
Where Do the Pieces Go?

- 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

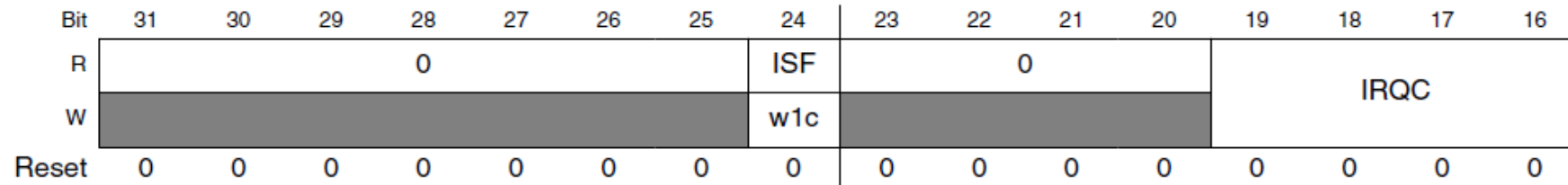
- Set up peripheral module to generate interrupt
 - We'll use Port Module in this example
- Set up NVIC
- Set global interrupt enable
 - Use CMSIS Macro `__enable_irq()`;
 - This flag does not enable all interrupts; instead, it is an easy way to **disable** interrupts
 - Could also be called “don't disable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

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

CMSIS C Support for PCR

- MKL25Z4.h defines PORT_Type structure with a PCR field (array of 32 integers)

```
/** PORT - Register Layout Typedef */
typedef struct {
    __IO uint32_t PCR[32]; /** Pin Control Register n, array offset: 0x0, array step:
0x4 */
    __IO uint32_t GPCLR;    /** Global Pin Control Low Register, offset: 0x80 */
    __IO uint32_t GPCHR;    /** Global Pin Control High Register, offset: 0x84 */
    uint8_t RESERVED_0[24];
    __IO uint32_t ISFR; /** Interrupt Status Flag Register, offset: 0xA0 */
} PORT_Type;
```

CMSIS C Support for PCR

- Header file defines pointers to PORT_Type registers

```
/* PORT - Peripheral instance base addresses */  
/** Peripheral PORTA base address */  
#define PORTA_BASE (0x40049000u)  
/** Peripheral PORTA base pointer */  
#define PORTA ((PORT_Type *)PORTA_BASE)
```

- Also defines macros and constants

```
#define PORT_PCR_MUX_MASK 0x700u  
#define PORT_PCR_MUX_SHIFT 8  
#define PORT_PCR_MUX(x)  
(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT)) &PORT_PCR_MUX_MASK)
```

Switch Interrupt Initialization

```
#include <MKL25Z4.H>
#include "switches.h"
#include "LEDs.h"
volatile unsigned count=0;
void init_switch(void) {
    /* enable clock for port D */
    SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK;
    /* Select GPIO and enable pull-up resistors and
       interrupts on falling edges for pin connected to switch */
    PORTD->PCR[SW_POS] |= PORT_PCR_MUX(1) | PORT_PCR_PS_MASK | PORT_PCR_PE_MASK
| PORT_PCR_IRQC(0x0a);
    /* Set port D switch bit to inputs */
    PTD->PDDR &= ~MASK(SW_POS);
    /* Enable Interrupts */
    NVIC_SetPriority(PORTD_IRQn, 128);
    NVIC_ClearPendingIRQ(PORTD_IRQn);
    NVIC_EnableIRQ(PORTD_IRQn);
}
```

Main Function

```
int main (void) {  
  
    init_switch();  
    init_RGB_LEDs();  
    init_debug_signals();  
    __enable_irq();  
  
    while (1) {  
        DEBUG_PORT->PTOR = MASK(DBG_MAIN_POS);  
        control_RGB_LEDs(count&1, count&2, count&4);  
        __wfi(); // sleep now, wait for interrupt  
    }  
}
```



```
void Control_RGB_LEDs(int r_on, int g_on, int b_on) {  
    if (r_on)  
        PTB->PCOR = MASK(RED_LED_POS);  
    else  
        PTB->PSOR = MASK(RED_LED_POS);  
    if (g_on)  
        PTB->PCOR = MASK(GREEN_LED_POS);  
    else  
        PTB->PSOR = MASK(GREEN_LED_POS);  
    if (b_on)  
        PTD->PCOR = MASK(BLUE_LED_POS);  
    else  
        PTD->PSOR = MASK(BLUE_LED_POS);  
}
```

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 rather than the default 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) {  
    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);  
}
```

Volatile Data

- Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief
 - *Don't reload a variable from memory if current function hasn't changed it*
 - Read variable from memory into register (faster access)
 - Write back to memory at end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail
 - Example: reading from input port, polling for key press
 - while (SW_0) ; will read from SW_0 once and reuse that value
 - Will generate an infinite loop triggered by SW_0 being true
- Variables for which it fails
 - Memory-mapped peripheral register – register changes on its own
 - Global variables modified by an ISR – ISR changes the variable
 - Global variables in a multithreaded application – another thread or ISR changes the variable

The Volatile Directive

- Need to tell compiler which variables may change outside of its control
 - Use volatile keyword to force compiler to reload these vars from memory for each use
`volatile unsigned int num_ints;`
 - Pointer to a volatile int
`volatile int * var; // or`
`int volatile * var;`
- Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
- Good explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001