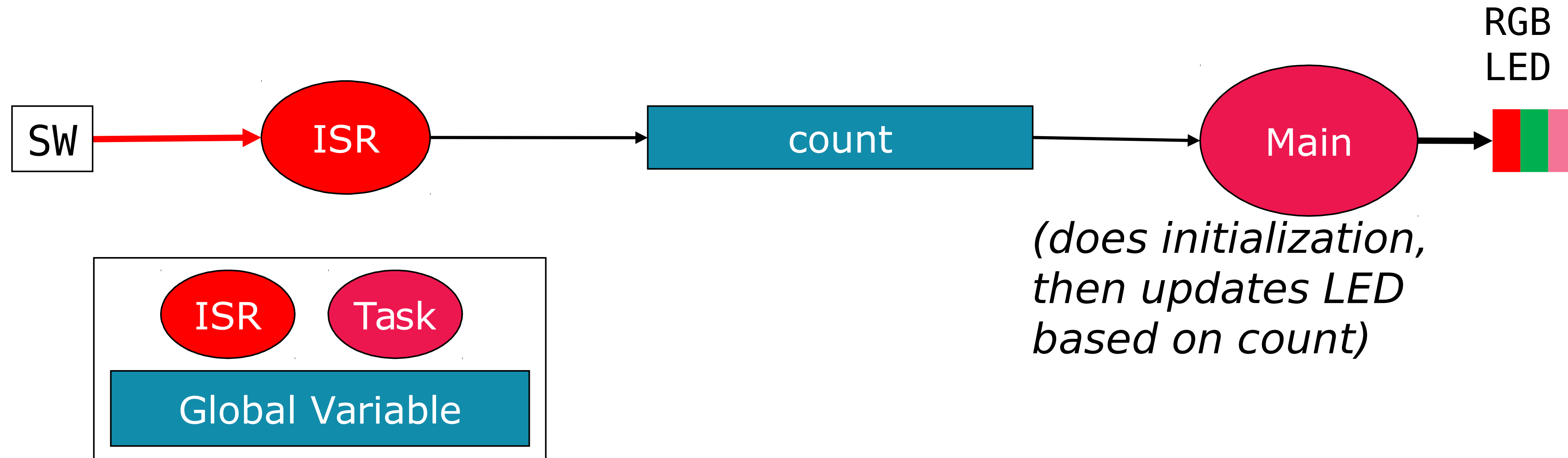
EXTERNAL INTERRUPTS EXAMPLE USING A GPIO PORT

ARM

Refresher: Program Requirements & Design

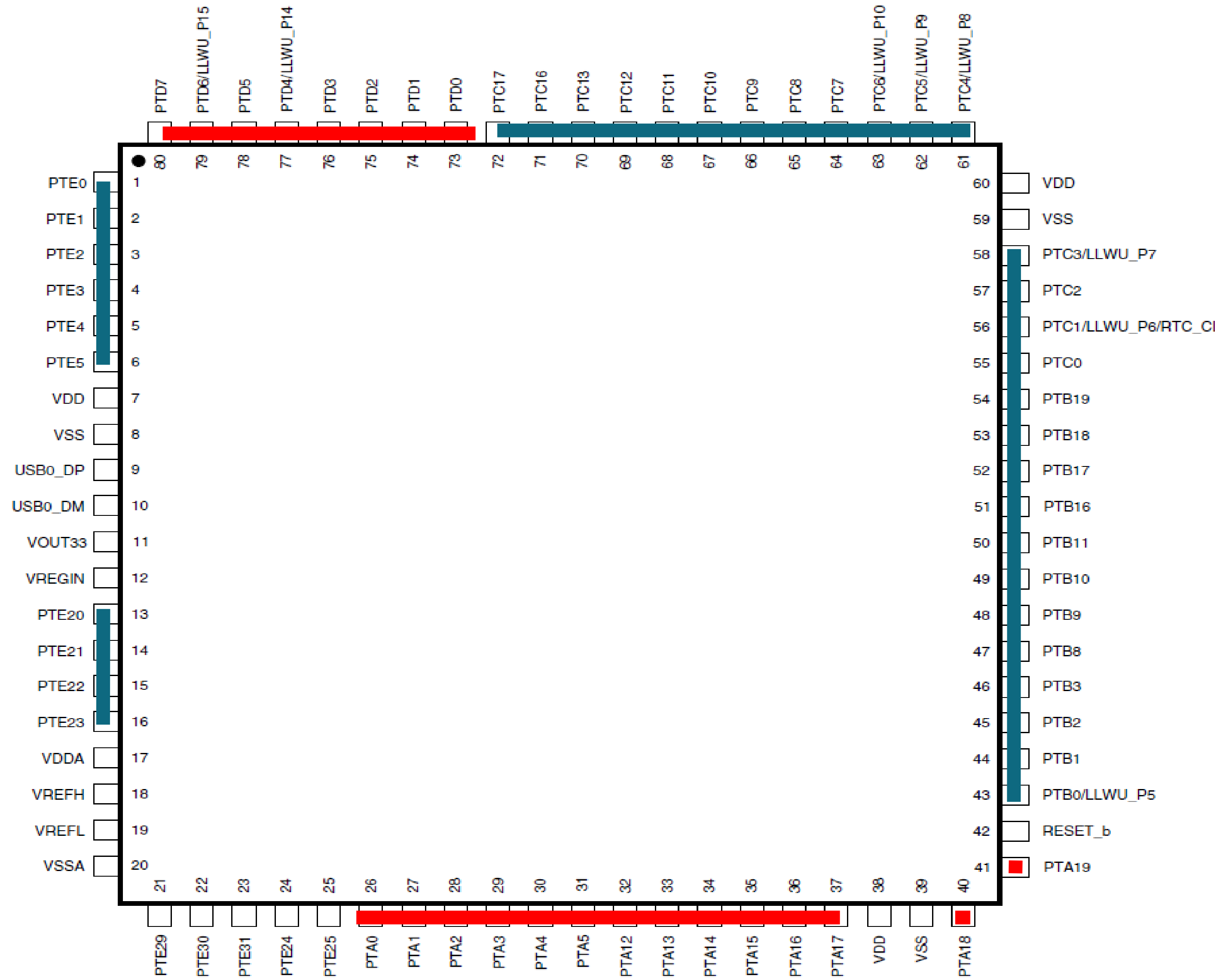


- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- Req4: ISR will raise its debug line DBG_ISR (and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

ARM

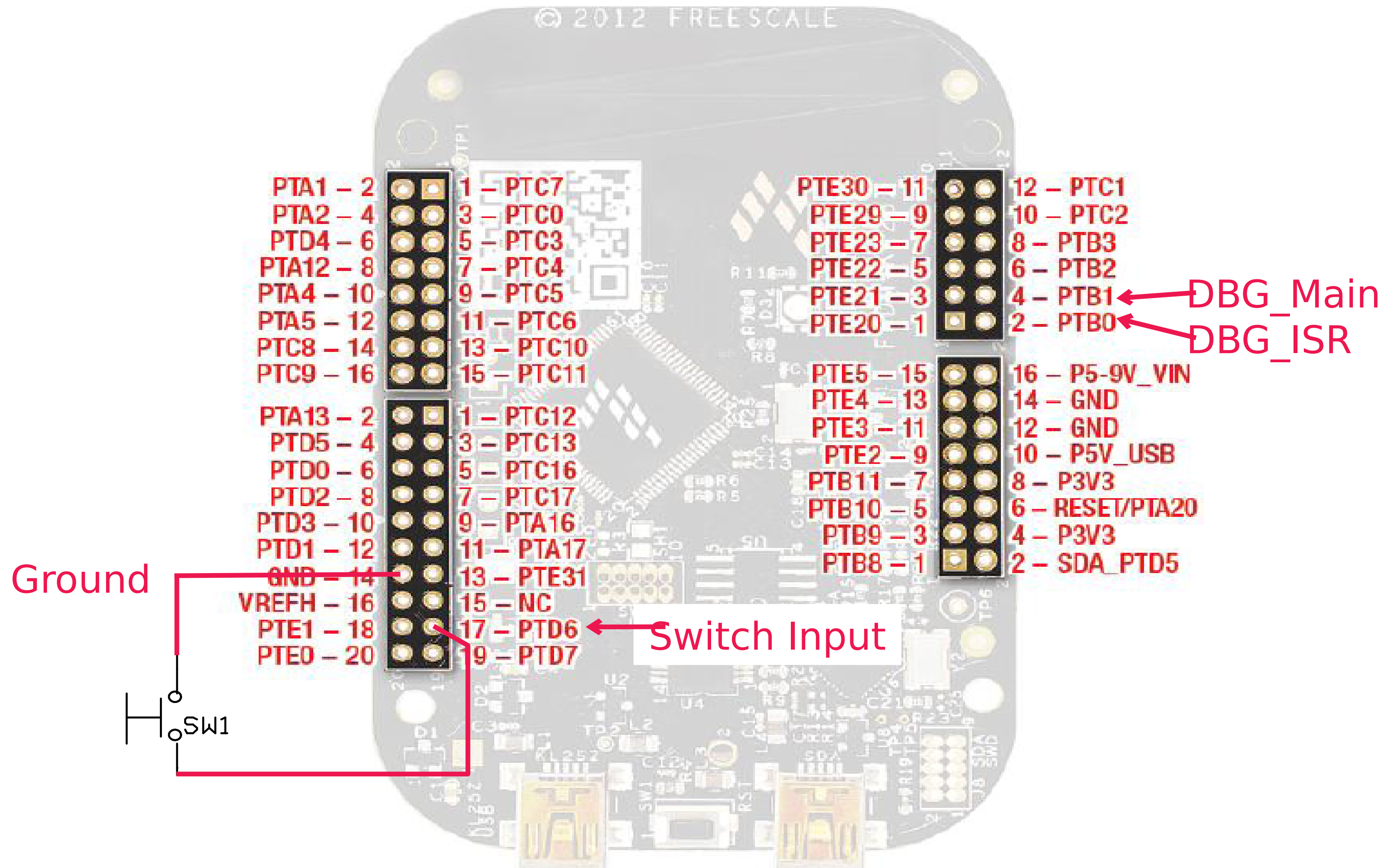
KL25Z GPIO Ports with Interrupts

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



ARM

FREEDOM KL25Z Physical Set-up



ARM

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)

ARM

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

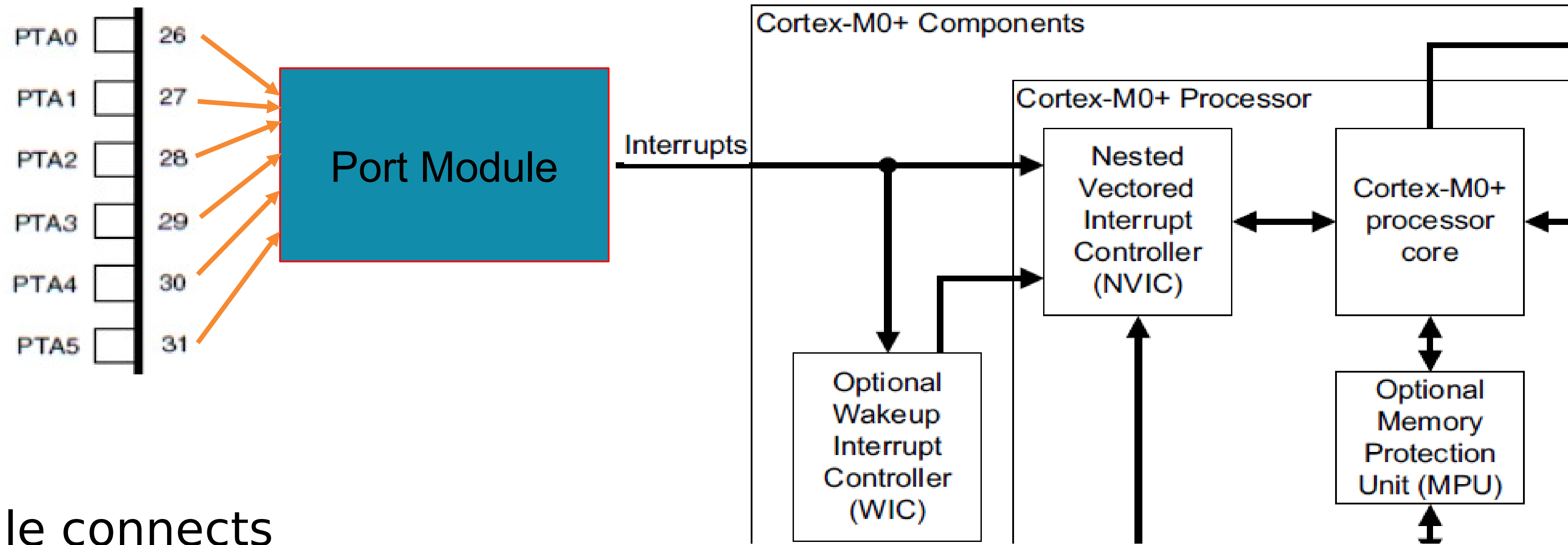
ARM

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”

ARM

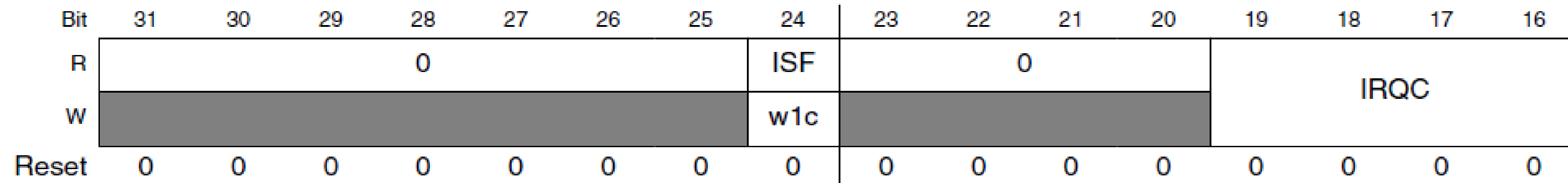
Port Module



- 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

ARM

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

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 */
  __O  uint32_t GPCLR;    /** Global Pin Control Low Register, offset: 0x80 */
  __O  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;
```


ARM

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)
```


ARM

CMSIS C Support for PCR

(Same slide as previous. Text size adjusted)

- 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)
```

3.3.2.1 Interrupt priority levels

This device supports 4 priority levels for interrupts. Therefore, in the NVIC each source in the IPR registers contains 2 bits. For example, IPR0 is shown below:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	IRQ3		0	0	0	0	0	0	IRQ2		0	0	0	0	0	0	IRQ1		0	0	0	0	0	0	IRQ0		0	0	0	0	0	0
W																																

3.3.2.2 Non-maskable interrupt

The non-maskable interrupt request to the NVIC is controlled by the external $\overline{\text{NMI}}$ signal. The pin the $\overline{\text{NMI}}$ signal is multiplexed on, must be configured for the $\overline{\text{NMI}}$ function to generate the non-maskable interrupt request.

ARM

Table 3-7. Interrupt vector assignments (continued)

Address	Vector	IRQ[1]	NVIC IPR register number[2]	Source module	Source description
0x0000_0000	0	—	—	ARM core	Initial Stack Pointer
0x0000_0004	1	—	—	ARM core	Initial Program Counter
0x0000_0008	2	—	—	ARM core	Non-maskable Interrupt (NMI)
0x0000_000C	3	—	—	ARM core	Hard Fault
0x0000_0010	4	—	—	—	—
0x0000_0014	5	—	—	—	—
0x0000_0018	6	—	—	—	—
0x0000_001C	7	—	—	—	—
0x0000_0020	8	—	—	—	—
0x0000_0024	9	—	—	—	—
0x0000_0028	10	—	—	—	—
0x0000_002C	11	—	—	ARM core	Supervisor call (SVCall)
0x0000_0030	12	—	—	—	—
0x0000_0034	13	—	—	—	—
0x0000_0038	14	—	—	ARM core	Pendable request for system service (PendableSrvReq)
0x0000_003C	15	—	—	ARM core	System tick timer (SysTick)

ARM

Non-Core Vectors					
0x0000_0040	16	0	0	DMA	DMA channel 0 transfer complete and error
0x0000_0044	17	1	0	DMA	DMA channel 1 transfer complete and error
0x0000_0048	18	2	0	DMA	DMA channel 2 transfer complete and error
0x0000_004C	19	3	0	DMA	DMA channel 3 transfer complete and error
0x0000_0050	20	4	1	—	—
0x0000_0054	21	5	1	FTFA	Command complete and read collision
0x0000_0058	22	6	1	PMC	Low-voltage detect, low-voltage warning
0x0000_005C	23	7	1	LLWU	Low Leakage Wakeup
0x0000_0060	24	8	2	I^{2}C0	
0x0000_0064	25	9	2	I^{2}C1	
0x0000_0068	26	10	2	SPI0	Single interrupt vector for all sources
0x0000_006C	27	11	2	SPI1	Single interrupt vector for all sources
0x0000_0070	28	12	3	UART0	Status and error
0x0000_0074	29	13	3	UART1	Status and error
0x0000_0078	30	14	3	UART2	Status and error
0x0000_007C	31	15	3	ADC0	
0x0000_0080	32	16	4	CMP0	
0x0000_0084	33	17	4	TPM0	
0x0000_0088	34	18	4	TPM1	
0x0000_008C	35	19	4	TPM2	

ARM

Table 3-7. Interrupt vector assignments (continued)

Address	Vector	IRQ[1]	NVIC IPR register number[2]	Source module	Source description
0x0000_0090	36	20	5	RTC	Alarm interrupt
0x0000_0094	37	21	5	RTC	Seconds interrupt
0x0000_0098	38	22	5	PIT	Single interrupt vector for all channels
0x0000_009C	39	23	5	—	—
0x0000_00A0	40	24	6	USB OTG	
0x0000_00A4	41	25	6	DAC0	
0x0000_00A8	42	26	6	TSI0	
0x0000_00AC	43	27	6	MCG	
0x0000_00B0	44	28	7	LPTMR0	
0x0000_00B4	45	29	7	—	
0x0000_00B8	46	30	7	Port control module	Pin detect (Port A)
0x0000_00BC	47	31	7	Port control module	Pin detect (Port D)

1. Indicates the NVIC's interrupt source number.
2. Indicates the NVIC's IPR register number used for this IRQ. The equation to calculate this value is: IRQ div 4

Switch Interrupt Initialization

```
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);
}
```


ARM

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
   }
}
```


ARM

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

ARM

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);
}
```

Note: DEBUG_PORT is not defined in the system header file. It is which ever port you choose likewise, DBG_ISR_POS is not pre-defined, it is the port bit you choose

ARM

Evaluate Basic Operation

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

ARM

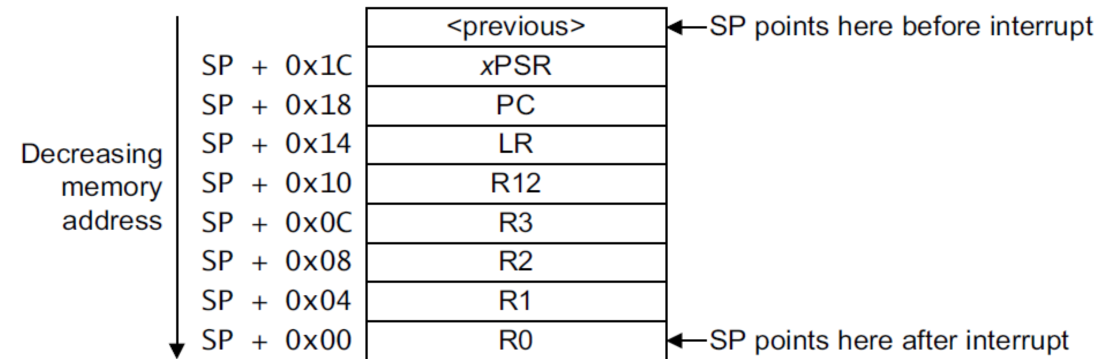
Examine Saved State in ISR

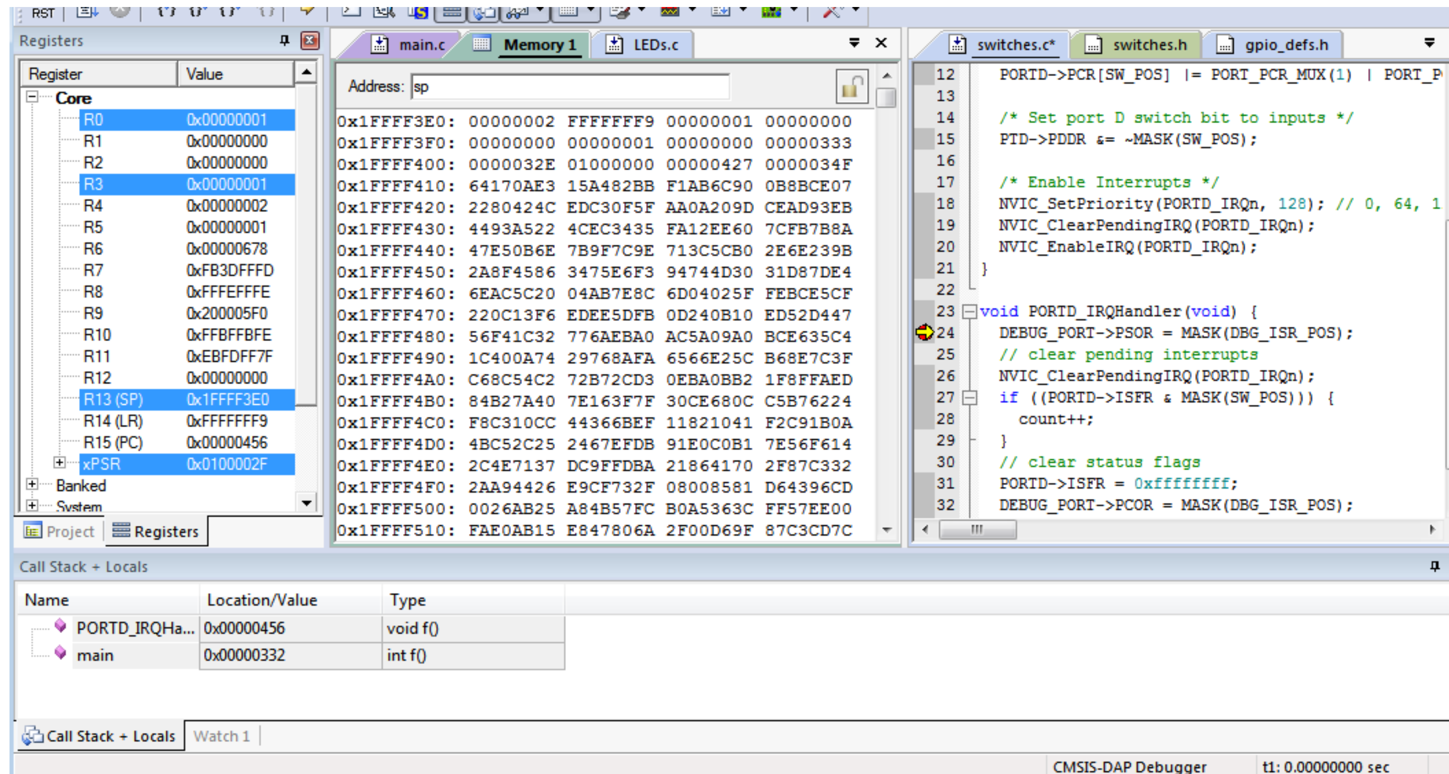
- Set breakpoint in ISR
- Run program
- Press switch, verify debugger stops at breakpoint
- Examine stack and registers

ARM

At Start of ISR

- Examine memory
- What is SP's value? See processor registers window





ARM

Step through ISR to End

- PC = 0x0000_048C
- Return address stored on stack: 0x0000_0333

Registers

Register	Value
Core	
R0	0x00000001
R1	0x400FF040
R2	0xE000E280
R3	0x00000001
R4	0x00000002
R5	0x00000001
R6	0x00000678
R7	0xFB3DFFFD
R8	0xFFFEFFFE
R9	0x200005F0
R10	0xFFBFFBFE
R11	0xEBFDFF7F
R12	0x00000000
R13 (SP)	0x1FFFF3E0
R14 (LR)	0x00000465
R15 (PC)	0x0000048C
xPSR	0x2100002F
Banked	
System	

Project | Registers

main.c | Memory 1 | LEDs.c

Address: sp

```
0x1FFFF3E0: 00000002 FFFFFFF9 00000001 00000000
0x1FFFF3F0: 00000000 00000001 00000000 00000333
0x1FFFF400: 0000032E 01000000 00000427 0000034F
0x1FFFF410: 64170AE3 15A482BB F1AB6C90 0B8BCE07
0x1FFFF420: 2280424C EDC30F5F AA0A209D CEAD93EB
0x1FFFF430: 4493A522 4CEC3435 FA12EE60 7CFB7B8A
0x1FFFF440: 47E50B6E 7B9F7C9E 713C5CB0 2E6E239B
0x1FFFF450: 2A8F4586 3475E6F3 94744D30 31D87DE4
0x1FFFF460: 6EAC5C20 04AB7E8C 6D04025F FEBCE5CF
0x1FFFF470: 220C13F6 EDEE5DFB 0D240B10 ED52D447
0x1FFFF480: 56F41C32 776AEBA0 AC5A09A0 BCE635C4
0x1FFFF490: 1C400A74 29768AFA 6566E25C B68E7C3F
0x1FFFF4A0: C68C54C2 72B72CD3 0EBA0BB2 1F8FFAED
0x1FFFF4B0: 84B27A40 7E163F7F 30CE680C C5B76224
0x1FFFF4C0: F8C310CC 44366BEF 11821041 F2C91B0A
0x1FFFF4D0: 4BC52C25 2467EFDB 91E0C0B1 7E56F614
0x1FFFF4E0: 2C4E7137 DC9FFDBA 21864170 2F87C332
0x1FFFF4F0: 2AA94426 E9CF732F 08008581 D64396CD
0x1FFFF500: 0026AB25 A84B57FC B0A5363C FF57EE00
0x1FFFF510: FAE0AB15 E847806A 2F00D69F 87C3CD7C
```

switches.c* | switches.h | gpio_defs.h

```
14   /* Set port D switch bit to inputs */
15   PTD->PDDR &= ~MASK(SW_POS);
16
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     count++;
29   }
30   // clear status flags
31   PORTD->ISFR = 0xffffffff;
32   DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
33 }
34 SP
```

Call Stack + Locals

Name	Location/Value	Type
PORTD_IRQHa...	0x00000456	void f()
main	0x00000332	int f()

Call Stack + Locals | Watch 1

CMSIS-DAP Debugger | t1: 0.00000000 sec

ARM

Return from Interrupt to Main function

- PC = 0x0000_0332

Registers

Register	Value
Core	
R0	0x00000001
R1	0x00000000
R2	0x00000000
R3	0x00000002
R4	0x00000002
R5	0x00000001
R6	0x00000678
R7	0xFB3DFFFD
R8	0xFFFEFFFE
R9	0x200005F0
R10	0xFFBFFBFE
R11	0xEBFDFF7F
R12	0x00000000
R13 (SP)	0x1FFFF408
R14 (LR)	0x00000333
R15 (PC)	0x00000332
xPSR	0x01000000
Banked	
System	

Project | Registers

startup_MKL25Z4.s | Memory 1 | LEDs.c

Address: sp

```
0x1FFFF408: 00000427 0000034F 64170AE3 15A482BB
0x1FFFF418: F1AB6C90 0B8BCE07 2280424C EDC30F5F
0x1FFFF428: AA0A209D CEAD93EB 4493A522 4CEC3435
0x1FFFF438: FA12EE60 7CFB7B8A 47E50B6E 7B9F7C9E
0x1FFFF448: 713C5CB0 2E6E239B 2A8F4586 3475E6F3
0x1FFFF458: 94744D30 31D87DE4 6EAC5C20 04AB7E8C
0x1FFFF468: 6D04025F FEBCE5CF 220C13F6 EDEE5DFB
0x1FFFF478: 0D240B10 ED52D447 56F41C32 776AEBA0
0x1FFFF488: AC5A09A0 BCE635C4 1C400A74 29768AFA
0x1FFFF498: 6566E25C B68E7C3F C68C54C2 72B72CD3
0x1FFFF4A8: 0EBA0BB2 1F8FFAED 84B27A40 7E163F7F
0x1FFFF4B8: 30CE680C C5B76224 F8C310CC 44366BEF
0x1FFFF4C8: 11821041 F2C91B0A 4BC52C25 2467EFDB
0x1FFFF4D8: 91E0C0B1 7E56F614 2C4E7137 DC9FFDBA
0x1FFFF4E8: 21864170 2F87C332 2AA94426 E9CF732F
0x1FFFF4F8: 08008581 D64396CD 0026AB25 A84B57FC
0x1FFFF508: B0A5363C FF57EE00 FAE0AB15 E847806A
0x1FFFF518: 2F00D69F 87C3CD7C 2021E807 AFF38EF8
0x1FFFF528: 2551072C 7632F6A4 13680DB0 EF3250D3
0x1FFFF538: 29808977 6AE5B18E C88C8828 C5777BC5
```

main.c | switches.c* | switches.h | LEDs.h

```
8
9   extern volatile unsigned switch_pressed;
10  extern void init_debug_signals(void);
11
12  /*----------------------------------------------
13    MAIN function
14   *----------------------------------------------
15  int main (void) {
16
17    init_switch();
18    init_RGB_LEDs();
19    init_debug_signals();
20    __enable_irq();
21
22    while (1) {
23      DEBUG_PORT->PTOR = MASK(DBG_MAIN_POS);
24      control_RGB_LEDs(count&1, count&2, count&4);
25    }
26  }
27
28
```

Call Stack + Locals

Name	Location/Value	Type
main	0x00000332	int f()

Call Stack + Locals | Watch 1

ARM

PROGRAM DESIGN WITH INTERRUPTS

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

ARM

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

ARM

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

ARM

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

ARM

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

ARM

Non-Atomic Shared Data

- Want to keep track of current time and date
- Use 1 Hz interrupt from timer
- System
  - TimerVal structure tracks time and days since some reference event
  - TimerVal’s fields are updated by periodic 1 Hz timer ISR

```
void GetDateTime(DateTimeType * DT){
 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
}
```

```
void DateTimeISR(void){
 TimerVal.second++;
 if (TimerVal.second > 59){
   TimerVal.second = 0;
   TimerVal.minute++;
   if (TimerVal.minute > 59) {
     TimerVal.minute = 0;
     TimerVal.hour++;
     if (TimerVal.hour > 23) {
    TimerVal.hour = 0;
       TimerVal.day++;
       … etc.
      }
```


ARM

Example: Checking the Time

- Problem
  - An interrupt at the wrong time will lead to half-updated data in DT
- Failure Case
  - TimerVal is {10, 23, 59, 59} (10th day, 23:59:59)
  - Task code calls GetDateTime(), which starts copying the TimerVal fields to DT: day = 10, hour = 23
  - A timer interrupt occurs, which updates TimerVal to {11, 0, 0, 0}
  - GetDateTime() resumes executing, copying the remaining TimerVal fields to DT: minute = 0, second = 0
  - DT now has a time stamp of {10, 23, 0, 0}.
  - ***The system thinks time just jumped backwards one hour!***
- Fundamental problem – “race condition”
  - Preemption enables ISR to interrupt other code and possibly overwrite data
  - Must ensure ***atomic (indivisible)*** access to the object
    - Native atomic object size depends on processor’s instruction set and word size.
    - Is 32 bits for ARM

ARM

Examining the Problem More Closely

- Must protect any data object which both
  - (1) requires multiple instructions to read or write (non-atomic access), and
  - (2) is potentially written by an ISR

- How many tasks/ISRs can write to the data object?
  - One? Then we have one-way communication
    - Must ***ensure the data isn't overwritten partway through*** being ***read***
      - Writer and reader don't interrupt each other
  - More than one?
    - Must ***ensure the data isn't overwritten partway through*** being ***read***
      - Writer and reader don't interrupt each other
    - Must ***ensure the data isn't overwritten partway through*** being ***written***
      - Writers don't interrupt each other

ARM

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.

ARM

Solution: Briefly Disable Preemption

- Prevent preemption within critical section
- If an ***ISR can write*** to the shared data object, need to ***disable interrupts***
  - save current interrupt masking state in m
  - disable interrupts
- Restore ***previous state*** afterwards (interrupts may have already been disabled for another reason)
- Use CMSIS-CORE to save, control and restore interrupt masking state
- Avoid if possible
  - Disabling interrupts delays response to all other processing requests
  - Make this time as short as possible (e.g. a few instructions)

```
void GetDateTime(DateTimeType * DT){
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
 __set_PRIMASK(m);
}
```


ARM

Summary for Sharing Data

- In thread/ISR diagram, identify shared data
- Determine which shared data is too large to be handled atomically by default
  - This needs to be protected from preemption (e.g. disable interrupt(s), use an RTOS synchronization mechanism)
- Declare (and initialize) shared variables as volatile in main file
  - ***volatile*** int my_shared_var=0;
- Update extern.h to make these variables available to functions in other files
  - ***volatile*** extern int my_shared_var;
  - ***#include "extern.h"*** in every file which uses these shared variables
- When using long (non-atomic) shared data, save, disable and restore interrupt masking status
  - CMSIS-CORE interface: __disable_irq(), __get_PRIMASK(), __set_PRIMASK()

ARM