Operating System Lab Assignment 02: Design and Deploy SysTick, Interrupt and Syscall

Dr. Mosaddek Tushar, Professor
Dr. Upama Kabir, Professor
Computer Science and Engineering, University of Dhaka,
Version 1.0
Due Date: Following week

Aug 23, 2022

Contents

1 Objectives and Policies

1.1 General Objectives

The objectives of the lab assignment are to understand and have hands-on training to understand

1.2 Assessment Policy

The assignment has three level objectives (i) primary objectives, (ii) advanced objectives, and (iii) optional boost objectives. Every student must complete the primary objective; however, they can attempt advanced and optional Boost-up objectives. The advanced objective will be a primary objective in the subsequent assignment. Further, you can achieve five (5) marks for completing the optional objectives and add these marks at the end of the semester with your total lab marks. However, you cannot get more than 100% assigned for the labs. The current lab does not contain advanced or optional boost objectives.

2 What to do?

- Implement Systick Service routines
- Implement Interrupt Service routine and
- Deploy OS service call using SVC and SVCPend

2.1 Systick Implementation

Create two files sys.h and sys.c that contains all the codes for this assignment

1. sys.h in 'src/kern/arch/stm32f446re/include/sys' directory and
2. sys.c in 'src/kern/arch/stm32f446re/sys' directory

2.1.1 SysTick Timer Registers

Find the details of SysTick registers and design hints in the programming manual from pages 228 to 233. You may find the programming manual on the course website under the 'Reference and Reading Material' section.

2.1.2 SysTick Data Structure

sys.h file contains data structure and function prototyping for implemetation of SysTick

```
typedef struct{
        volatile uint32_t CTRL; //enable SysTick, clock source, systick interrupt and count flag
        volatile uint32_t LOAD; // 24 bits — Reload Register; maximum count
        volatile uint32_t VAL;  // Current count value similiar to 'CNT' of timer
        volatile uint32_t CALIB;  // Calibration Register
}SysTick_Typedef;
```

SysTick address to access: 0xE000E010 (Starting Address)

2.1.3 SysTick Functions need to implement

- Systick Configuration:'void sysTick_init(uint32_t);'
  This function configure the system timer with interrupt. This function must disable the SysTick timer, set '0' to VAL register content, and load the reload value so the SysTick timer can interrupt every 10ms. Initialize a global variable 'mscount' to '0'. The 'mscount' variable keeps track of the time in milliseconds from the SysTick timer's beginning. Then enable the SysTick timer.

- Systick Enable: 'void sysTick_enable(void);'
  This function enables the disabled SysTick timer. The function does not modify an active SysTick timer and 'mscount' global variable. However, if the timer is currently disabled, the function must initialize the 'mscount' to '0' before enabling it.

- Systick Disable: 'void sysTick_disable(void);'
  The sysTick_disable function disables the SysTick timer if it is currently disabled.

- Get current tick count: uint32_t getSysTickCount(void);
  This function returns the current content of the VAL register.

- Update SysTick reload value: void updateSysTick(uint32_t); The update function disables the timer, initializes the reload value to the LOAD register, and finally initializes the 'mscount' variable.

- Get current time (in ms): 'uint32_t getTime(void)'
  This function returns milliseconds by combining the 'mscount' and VAL-Register values. You can use this function to get the execution time required for a code block.

2.1.4 SysTick_Handler

Modify the SysTick_Handler function to update the 'mscount' value.

2.1.5 Testing

Write suitable test code in the 'kmain' function. The code must be able to test all the above functions effectively. We test your developed functions to verify if the functions are working correctly. We are not proving any test codes; you must design your test to cover all aspects of the SysTick timer implementation. Add comments on the developed functions and test codes.

2.1.6 What to submit

Submit four updated files (1) sys.h and (2) sys.c, (3) kmain.c (test), and (4) kmain.h and the updated SysTick_Handler function.

2.2 Implementation of NVIC Interrupt services

*****Important: You may need 'STM32F-Programming Manual'**

NVIC – nested interrupt vector provides configurable interrupt abilities to the processor. The processor facilitates low latency exceptions and interrupts handling and control power management. ARM Cortex-M has several exceptions and interrupts to handle system events for particular tasks

and error conditions. ARM Cortex-M4 has 15 exceptions and 240 interrupts. The first exceptions are internal to the processor, and the remaining 240 interrupts are to accomplish tasks for external events, including peripherals and the outside world. The NVIC supports 256 levels of programmable dynamic priorities. ARM Cortex-M further segregates interrupt-priority bits into groups (preemptive) and subgroups. The interrupts assign the same group or preemption with higher priority subgroup never suspended while executing.

The program running in privileged mode has full access to the NVIC. But you can cause interrupts to enter a pending state in user mode if you enable this capability using the Configuration Control Register. Any other user mode access causes a bus fault. Unless otherwise stated, you can access all NVIC registers using byte, halfword, and word accesses. NVIC registers reside within the SCS (System Control Space). All NVIC registers and system debug registers are little-endian regardless of the endianness state of the processor. You configure the number of interrupts and bits of interrupt priority during implementation. The software can choose only to enable a subset of the configured number of interrupts and can decide how many bits of the configured preferences to use.

We already define the exceptions, interrupts, and handlers in stm32_statup.h file. However, the definitions of the interrupts handlers are weakly defined and map to the default handler. The system developers must write the task of an interrupt or exception according to their needs. Nevertheless, the prerequisites are developing a kernel function set to enable, disable, and prioritize the interrupt assignment. To do this, you must acquire knowledge of interrupt vectors and the configuration of the registers. The functions to use the interrupts in the ARM Cortex-M processor that you need to implement are listed below.

2.2.1 NVIC functions to do

Before going to the function detail, you need to define the data structure for SCB, SCS, NVIC, and IRQn_Type to access the registers. You must express all data structure and address definitions in the 'sys.h' file and function details in 'sys.c'.

1. **SCB data structure:** The starting address for SCB is '0xE000ED00'

```
typedef struct
{
        volatile uint32_t CPUID;   // CPUID Base Register 0x0
        volatile uint32_t ICSR;    // Interrupt Control and State Register 0x4
        volatile uint32_t VTOR;    // Vector Table Offset Register 0x8
        volatile uint32_t AIRCR;   // Application Interrupt and Reset Control Register 0xC
        volatile uint32_t SCR;     // System Control Register   0x10
        volatile uint32_t CCR;     // Configuration and Control Register 0x14
        volatile uint8_t SHPR[12]; // Exception priority setting for system exceptions
        volatile uint32_t SHCSR;   // System Handler Control and State Register  0x24
        volatile uint32_t CFSR;    // Configurable Fault Status Register combined of MemManage
                        // Fault Status Register, BusFault Status Register, UsageFault Status Register 0x28
        volatile uint32_t HFSR;    // HardFault Status Register 0x2C
        volatile uint32_t DFSR;    //Hint information for causes        of debug events
        volatile uint32_t MMFAR;    // MemManage Fault Address Register  0x34
        volatile uint32_t BFAR;    // BusFault Address Register  0x38
        volatile uint32_t AFSR;    // Auxiliary Fault Status Register 0x3C
        volatile uint32_t PFR[2];  // Read only information on available processor features
        volatile uint32_t DFR;           // Read only information on available debug features
        volatile uint32_t AFR;           // Read only information on available auxiliary features
        volatile uint32_t MMFR[4]; // Read only information on available memory model features
        volatile uint32_t ISAR[5]; //
        uint32_t RESERVED1[5];    //
        volatile uint32_t CPACR;   // Coprocessor access control register 0x88
 } SCB_TypeDef;
```

2. **NVIC Data Structure**: Starting address: '0xE000E100'. Verify with the 'Cortex-M4 programming manual' if the data structure is acceptable to the byte spacing.

```
typedef struct
{
        //define NVIC register compenenets -- use volatile data type
        volatile uint32_t ISER[8]; /*!< Offset: 0x000  addr: 0xE000E100 Interrupt Set Enable Register*/
        uint32_t RESERVED0[24];
        volatile uint32_t ICER[8]; /*!< Offset: 0x080  addr: 0xE000E180 Interrupt Clear Enable Register*/
        uint32_t RSERVED1[24];
        volatile uint32_t ISPR[8]; /*!< Offset: 0x100  addr: 0xE000E200 Interrupt Set Pending Register*/
        uint32_t RESERVED2[24];
        volatile uint32_t ICPR[8]; /*!< Offset: 0x180  addr: 0xE000E280 Interrupt Clear Pending Register*/
        uint32_t RESERVED3[24];
        volatile uint32_t IABR[8]; /*!< Offset: 0x200  addr: 0xE000E300 Interrupt Active bit Register*/
        uint32_t RESERVED4[56];
        volatile uint8_t  IP[240]; /*!< Offset: 0x300  addr: 0xE000E400 Interrupt Priority Register (8Bit wide) */
        uint32_t RESERVED5[644];
        volatile  uint32_t STIR;   /*!< Offset: 0xE00  addr: 0xE000EF00 Software Trigger Interrupt Register*/
}NVIC_TypeDef;
```

3. **Priority gouping**: Register AIRCR in SCB is used to determine the number of bits for priority prouping. See lecture slides
4. You also need **PRIMASK**, **FAULTMASK** and **BASEPRI** register to mask or unmask interrupts
5. Interrupt Data Structure

```
enum IRQn_TypeDef {
        NonMaskableInt_IRQn = -14,
        HardFault_IRQn = -13,
        MemoryManagement_IRQn = -12,
        BusFault_IRQn = -11,
        UsageFault_IRQn = -10,
        SecureFault_IRQn = -9,
        SVCall_IRQn = -5,
        DebugMonitor_IRQn = -4,
        PendSV_IRQn = -2,
        SysTick_IRQn = -1,
        WWDG_STM_IRQn = 0,
        PVD_STM_IRQn = 1,
        ......
        ......
}
```

1. void __NVIC_SetPriority (IRQn_TypeDef IRQn,uint32_t priority): This function takes two arguments (i) interrupt number and sets the priority to the interrupt. Note that priority in the above NVIC register is 8-bit. The priority puts the preference to the ISR executing before the lower (higher number) priority interrupts.
2. uint32_t __NVIC_GetPriority(IRQn_TypeDef IRQn): Return the priority set to the interrupt.
3. void __NVIC_EnableIRQn(IRQn_TypeDef IRQn): enable interrupt given as argument or interrupt number (IRQn_typeDef) – data structure (enumerator) defined earlier.
4. void __NVIC_DisableIRQn(IRQn_TypeDef IRQn): Disable interrupt.
5. void __disable_irq(): Masking interrupts (‘__disable_irq()’) – all interrupts other than HardFault, NMI, and reset.
6. void __set_BASEPRI(uint32_t value): __set_BASEPRI(uint32_t value) function mask interrupt number greater and equal to the given interrupt priority as an argument.
7. void __enable_irq(): Enable (unmask) all interrupts
8. void __unset_BASEPRI(uint32_t value): __unset_BASEPRI(uint32_t value) function unmask interrupts greater or equal to the given argument/priority number.

9. void __set_PRIMASK(uint32_t priMask): Prevent all interrupt without non-maskable interrupt.
10. uint32_t get_PRIMASK(void): Return value of the PRIMASK register
11. void __enable_fault_irq(void) and void __set_FAULTMASK(uint32_t faultMask): Enable all interrupt including FaultMask.
12. void __disable_fault_irq(void): Disable or prevent all interrupt including FaultMask
13. uint32_t __get_FAULTMASK(void): This function will return the status of the masking value of FaultMask register
14. void __clear_pending_IRQn(IRQn_TypeDef IRQn): Clear interrupt pending bit
15. uint32_t __get_pending_IRQn(IRQn_TypeDef IRQn): It returns the pending status of an interrupt.
16. uint32_t __NVIC_GetActive (IRQn_TypeDef IRQn): This function return the active status of the interrupt.

Note: Some of the functions may be semantically identical. However, different expressions embellish your code and enhance usability and interpretation.

2.2.2 Bootom Line

In this assignment segment, you need extensive coding and debugging to ensure the elegant implementation of NVIC activities and enable the NVIC features for higher-level developers. Split your work and share it with your group member. It undoubtedly increases your efficiency and gives an essence of the group work. You can discuss it with anyone; however, write the code yourself. This development process helps you to understand the cooperation among your team member and other group doing the same work. Let us know if you have any queries or difficulties with understanding.

2.2.3 Testing

Write appropriate test code in the 'kmain()' function. The code must be able to test all the above NVIC functions effectively. We test your developed functions to ascertain if the functions are functioning perfectly. The test program must efficiently verify the errors (if any) in the execution of the ISR. To test the interrupt, you can reuse the program developed in the last semester.

2.2.4 What to Submit

Similar to the earlier part of the assignment, you must submit (i) kmain.c for testing, (ii) sys.h, and (iii) sys.c along with the SysTick code. We prefer comments in your code, which improve readability and is an industry-standard coding style. The comments help your fellow and legatee start from where you finished.

2.3 Implementation of Syscall Exception (SVC) and PendSV

Alert: You can discuss with your classmates, however, do not copy code from others.