California State University, Northridge

Department of Electrical & Computer Engineering

Lab 10

Interrupts

December 8, 2022

ECE 425L

Assigned by: Neda Khavari

Written By: Pathum Addarapathirana & Cristian Robles

Introduction:

In lab 10 we are introduced to interrupts. Interrupts are a very important tool for handling large applications and multitasking. Here we use specifically the SoftWare Interrupt (SWI), Interrupt ReQuest (IRQ), and Fast Interrupt ReQuest (FIQ).

Procedure:

Equipment Used

- Keil uVision4
- Keil Debugger
- LPC2148 Education Board

Description of Procedure

1. We first set up the startup code for this lab. We initially start using the same startup code from lab 3. We are asked to modify the startup code so that the stack initialization is performed in the reset handler for all processor modes. This is done by first setting the number of bytes attached or used, to the stack at each mode as shown below.

```
; Standard definitions of Mode bits and Interrupt (I & F) flags in PSR s
Len_SVC_Stack   EQU     0x100 ; # of bytes assigned to the stack in Supervisor mode
Len_FIQ_Stack   EQU     0x200 ; # of bytes assigned to the stack in FIQ mode
Len_IRQ_Stack   EQU     0x90 ; # of bytes assigned to the stack in IRQ mode
Len_UND_Stack   EQU     0x80 ; # of bytes assigned to the stack in Undefined mode
Len_ABT_Stack   EQU     0x80 ; # of bytes assigned to the stack in Abort mode
```

2. Then defining the mode bits from the manual as shown below.

```
Mode_USR        EQU     0x10 ; Mode bits for USR mode
Mode_SVC        EQU     0x13 ; Mode bits for Supervisor mode
Mode_UND        EQU     0x1B ; Mode bits for Undefined mode
Mode_IRQ        EQU     0x12 ; Mode bits for IRQ mode
Mode_FIQ        EQU     0x11 ; Mode bits for FIQ mode
Mode_ABT        EQU     0x17 ; Mode bits for ABT mode
```

3. Then adding these last three lines for the stack in the definitions. This will be more clear in the next step.

```
;Defintions of User Mode Stack and Size
SRAM_BASE       EQU     0x40000000 ; Starting address of the SRAM
I_Bit           EQU     0x80 ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40 ; when F bit is set, FIQ is disabled
USR_Stack_Size  EQU     0x00000100
SRAM            EQU     0X40000000
Stack_Top       EQU     SRAM+USR_Stack_Size
```

4. Then for the final part of task 1 we load the descending stack for each mode. This is done by loading r0 with the SRAM+BASE and adding the corresponding mode stack. Then we move the value from the selected special-purpose register, which is the mode+ I_Bit + F_Bit into CPSR_c. Finally move r0 to sp and load user_code to PC.
   This process is repeated for all modes as shown below.

```
Reset_Handler
                ;;Supervisor mode
                LDR     r0, =SRAM_BASE+ Len_SVC_Stack ; Descending stack
        ; Enter each mode in turn and set up the stack pointer
                MSR     CPSR_c, #Mode_SVC+I_Bit+F_Bit
                MOV     sp, r0
                LDR     PC, =user_code

                ;;FIQ mode?
                LDR     r0, =SRAM_BASE+ Len_FIQ_Stack
        ; Enter each mode in turn and set up the stack pointer
                MSR     CPSR_c, #Mode_FIQ+I_Bit+F_Bit
                MOV     sp, r0
                LDR     PC, =user_code

                ;;IRQ mode
                LDR     r0, =SRAM_BASE+ Len_IRQ_Stack
        ; Enter each mode in turn and set up the stack pointer
                MSR     CPSR_c, #Mode_IRQ+I_Bit+F_Bit
                MOV     sp, r0
                LDR     PC, =user_code

                ;;Undefined mode
                LDR     r0, =SRAM_BASE+ Len_UND_Stack
        ; Enter each mode in turn and set up the stack pointer
                MSR     CPSR_c, #Mode_UND+I_Bit+F_Bit
                MOV     sp, r0
                LDR     PC, =user_code

                ;;Abort mode
                LDR     r0, =SRAM_BASE+ Len_ABT_Stack
        ; Enter each mode in turn and set up the stack pointer
                MSR     CPSR_c, #Mode_ABT+I_Bit+F_Bit
                MOV     sp, r0
                LDR     PC, =user_code
```

5. For Task 2 we are write a complete ARM assembly program which calls a single Software Interrupt to light up all 8 LEDs connected to P0.15 - P0.8, in the service routine.
   This is done by first setting up GPIO as shown below.

```
PINSEL0     EQU     0xE002C000      ;pin function for port 0, equate symbolic name PINSEL0 as address 0xE002C000

                                    ;;Selecting funcion as GPIO by writing all zeros to given address
            MOV     r0,#0           ;moves #0 into register r0
            LDR     r1,=PINSEL0     ;puts what is stored in register 0xE002C000 (PINSEL0) in r1 register; CANNOT PUT MOV, outputs error in kei
            STR     r0,[r1]         ;copies value stored in r0 to memory address specified by r1
```

6. Then set IO Direction and the initial clear and set of the LEDs we simply created a loop for these tasks as you will see in the next steps, some tasks require doing these steps again. So for a more clear and shorter code we implemented a loop.

```
            ;Initially Assigning B constant to zero
            MOV B_var,#0

            ;set IODIRECTION
            BL      setIODIR            ;;Selecting signal direction of each port pin, we put '1' in each to bit to make it an output pi

            BL      clrset              ;initial clr and set
```

7. Now after implementing a small delay for clarity when testing, we add the first exception which is SWI#1 that is meant to turn on all the lights.

```
;-----------------------------------------------------------------------------------------------------
task2                               ;Turn on lights using delay from another file
            LDR     r5,=delayone
            BL      RTN             ;delay subroutine here

            SWI #1
```

8. The step above causes the PC to jump to the exception which is the SWI Handler. Shown below is the SWI Handler where we first decrement the link register by 4 so it will step back to the next line in the main program. R9 is loaded with the SWI value where we simply use a CMP function to BEQ to the desired action loop.

```
SWIHandler
            LDRB    r9,[LR,#-4]
            CMP     r9, #0
            BEQ     turnoff
            CMP     r9, #1
            BEQ     lighton
            CMP     r9, #2
```

9. For this particular task it will branch to the loop shown below.

```
lighton                                 ;turning on ALL LEDS
            MOV     r0,#0x0000FF00
            LDR     r1,=IO0BASE
            STR     r0,[r1,#IO0CLR]
            MOVS    PC, LR
```

10. Then for task 3 we are to make three different software interrupts which does 3 different functions with the lights. This is done in the same way as task 2 except we set a different number for each interrupt. Shown below are those respective interrupts and their functions in the main file.

```
task3
            LDR     r5,=delayone
            ;BL     RTN                 ;delay subroutine here

            SWI #0                      ;to turn off all lights
            SWI #2                      ;to turn on first four lights

            LDR     r5,=delayone
            ;BL     RTN                 ;delay subroutine here

            SWI #3                      ;to turn on last four lights

            LDR     r5,=delayone
            ;BL     RTN                 ;delay subroutine here

            SWI #0                      ;turns it all off
            B       endref
```

11. Shown below is the respective branch in the SWI Handler that achieves the tasks explained in the main.

```

turnoff                                 ;turning off ALL LEDs, (forcing a high) (binary 1111111100000000)
            MOV     r0,#0x0000FF00      ;LDR for this?
            LDR     r1,=IO0BASE
            STR     r0,[r1,#IO0SET]
            MOVS    PC,LR

lighton                                 ;turning on ALL LEDS
            MOV     r0,#0x0000FF00
            LDR     r1,=IO0BASE
            STR     r0,[r1,#IO0CLR]
            MOVS    PC, LR

firstfour
            ;LDR R0,[LR,#-3]
            MOV     r0,#0x0000F000      ;turns on first 4 (0000)111100000000
            LDR     r1,=IO0BASE
            STR     r0,[r1,#IO0CLR]
            MOVS    PC, LR

lastfour
            ;LDR R0,[LR,#-3]
            MOV     r0,#0x00000F00      ;turns on last 4 1111(0000)00000000
            LDR     r1,=IO0BASE
            STR     r0,[r1,#IO0CLR]
            MOVS    PC, LR
```

12. For the first task we are to set the lights to turn on one by one using loops and a 1 second delay. We calculated the delay for one second in ARM is about 4 million loops (as shown in step 1 screenshot, the 'delayone' hex number). We first load the delay to a register then we branch link to another file that we named RTN. This loop follows the same logic as the previous lab where we subtract 1 from the imported number in r5. Then once that is done, PC is loaded with the next line or link register of the previous file so it continues with the next step after the loop.

That step is a loop where we do a logical left shift of bit 8 in a loop 8 times for all 8 LEDs. This is shown in the code below.

```
task1
            CMP     B_var,#8        ;B_var = 8 BEQ next loop for task 2
            BEQ     task2

turnLow
            LDR     r5,=delayone    ;delaying onesec
            BL      RTN             ;branching to external file delay_arm.s
                                    ;for logical shift
            LDR     r0,=0x00000100  ;binary 100000000 (bit 8)
            LSL     r0,B_var        ;Logical shift left of B_var

            LDR     r1,=IO0BASE     ;forcing low
            STR     r0,[r1,#IO0CLR]

            ADD     B_var,B_var,#1  ;+1 B_var

            B       task1
```

```
        GLOBAL  RTN
        AREA    mycode,CODE,READONLY
        ENTRY
        ;STMFD  sp!,{LR}



RTN         SUBS    r5,#1
            BNE     RTN         ;jump to given address if not zero
            MOVEQ   pc,lr
            ;LDMFD  sp!,{PC}    ;works without LDMFD and STMFD

        B RTN
        END
```

Results:

Task 1

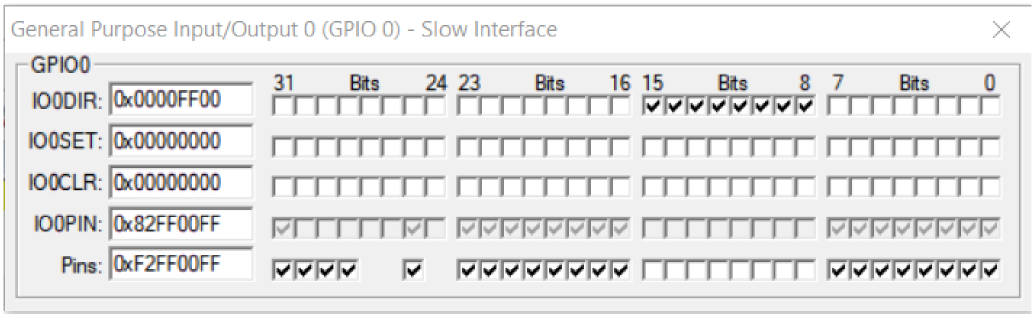
```
LDR r0,=SRAM+FIQ_Stack
MSR CPSR_c,#Mode_FIQ+I_Bit+F_Bit
MOV sp,r0

LDR r0,=SRAM+IRQ_Stack
MSR CPSR_c,#Mode_IRQ+I_Bit+F_Bit
MOV sp,r0

LDR r0,=SRAM+Len_SWI_Stack
MSR CPSR_c,#Mode_SWI+I_Bit+F_Bit
MOV sp,r0
```

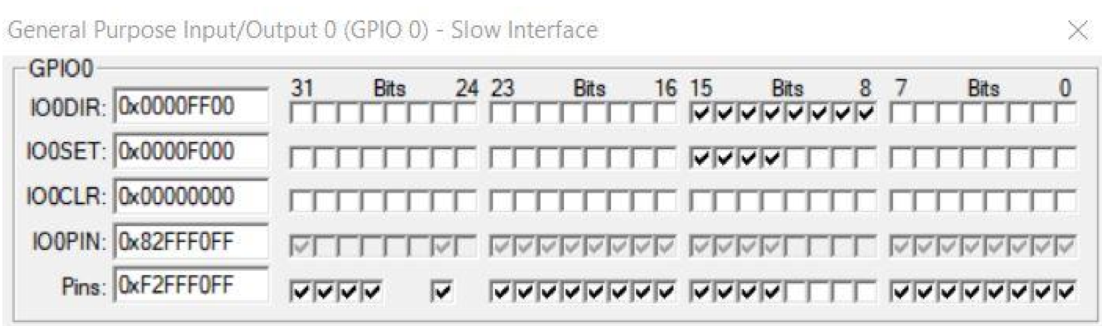
In task 1 all stack initializations were performed in the reset handler for the processor modes needed in this experiment

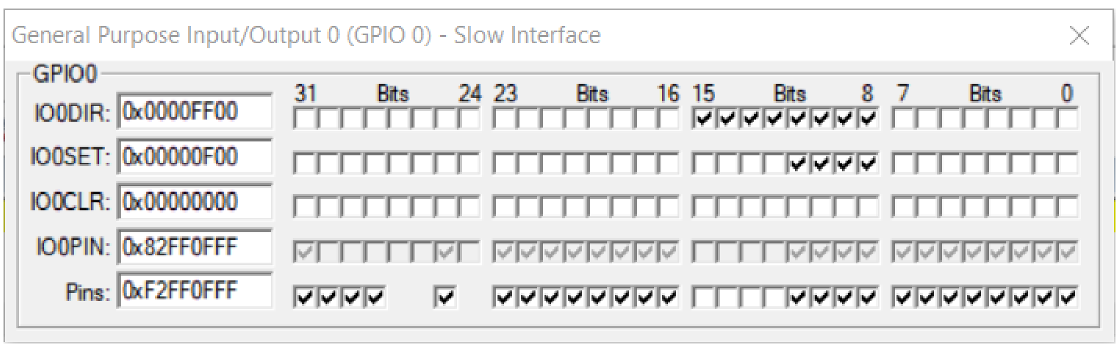
Task 2

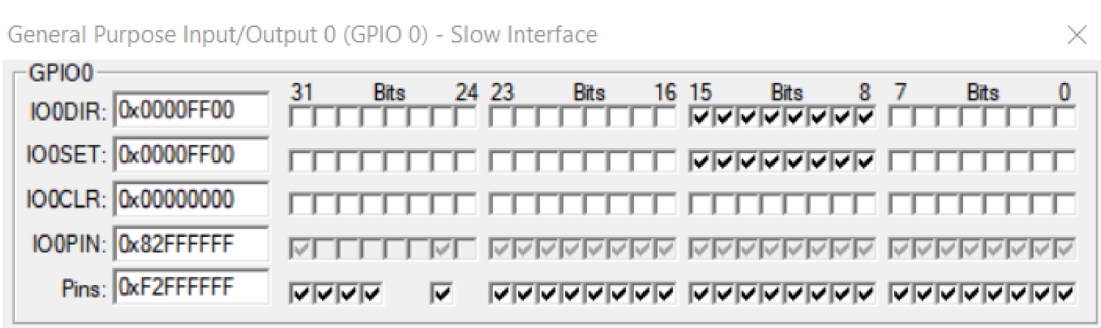


In task 2 LEDs on P0.8-P0.15 were turned on by a single software interrupt call

Task 3







In task 3 we use 3 software interrupt calls.The first turns on LEDs P0.8-P0.11, the second turns on P0.12-P0.15, and the third turns off all LEDs that were turned on by the previous two software interrupts.

Conclusion:

In conclusion, we were able to successfully handle the different types of interrupts in assembly. In task 1 we initialized the stack for all different processor modes in the reset handler and this allowed us to later have the ability to handle both software and external interrupts. Task 2 and 3 were very similar and they both ran successfully in simulation and on the physical board as well. Task 4 was extra credit and we managed to successfully complete the set up for both the IRQ and FIQ interrupts but we were not able to exit the interrupts after we had serviced the interrupts. One way this lab could have gone better is for us to have exited the external interrupts in task 4 properly so that the board wouldn’t get stuck in an infinite loop. This lab introduced us to both software and external interrupts and furthered our understanding of using pins as both inputs and outputs.