Department of Computer Engineering

BLG 351E
Microcomputer Laboratory
Experiment Report

Experiment No : 5

Experiment Date : 13.11.2017

Group Number : Monday - 8

Group Members :

ID	Name	Surname
150150114	EGE	APAK
150140066	ECEM	GÜLDÖŞÜREN
150150701	YUNUS	GÜNGÖR

Laboratory Assistant : Ahmet ARIŞ

1. INTRODUCTION

In this experiment, we have learned about and improved our understanding of 7-segment hex display and interrupt subroutines. We have carried out this experiment in two parts.

In the first part, we have implemented a basic counter using registers, which counts from 0 to 10 (excluding 10) and wraps back to zero, and displayed the value of the counter.

In the second part of the experiment, we have changed counting mechanism of the counter and made it increment its value by two and enabled it to count either even or odd numbers, which can be toggled via the Port#2.6.

2. EXPERIMENT

2.1. COUNTER

In this part, we have built a counter with a period of 1 second. The code we wrote can be seen in Picture 2.1:

```
;-------------------------------------------------------------------------------
setup       mov.b #0FFh, P1DIR
            mov.b #01h, P2DIR
            mov.b #01h, P2OUT

count$str   mov.w #00h, R15
count$dsp   mov.w R15, R14
            add.w #array, R14
            mov.b @R14, P1OUT
            push.w R15
            call #Delay
            pop R15
            inc.w R15
            cmp.w #0Ah, R15
            jz count$str
            jmp count$dsp

Delay       mov.w #0Ah, R14
L2          mov.w #07A00h, R15
L1          dec.w R15
            jnz L1
            dec.w R14
            jnz L2
            ret

array .byte 00111111b, 00000110b, 01011011b, 01001111b, 01100110b, 01101101b, 01111101b, 00000111b, 01111111b, 01101111b
lastElement
;-------------------------------------------------------------------------------
; Stack Pointer definition
```

Picture 2.1: Source code of Part#1

First of all, we set all pins on Port#1 as output pins, since they define which parts of the 7-segment display lights up. After that, we set the Port#2.1 as an output and set its out value to 1 because Port#2.1 enables first 7-segment display.

After setting the 7-segment hex display up, we start the counter from 0. For displaying the value of the counter, we added the value of counter to #array, which is the address of the first element defined in array, which contains the values of segments that should be lit for values between 0 and 9 consecutively. Then we move this value into Port#1 in order to display the current value.

After displaying the current counter value, we push counter into stack in order to prevent data loss due to usage of same registers in Delay function and then call the Delay function to wait for a second. After the delay, we pop the value back into register and check its value to determine whether it reached 10 or not, after incrementing it. If the counter reached 10, we wrap it back to zero and start over or keep counting if it hasn't reached 10 yet.

2.2. Even-Odd Counter

In this part of the experiment, we manipulated the counter we have built in the first part to make it count either even or odd numbers, toggled by the button in Port#2.6. The source code of this part is as follows:

```

init_INT    bis.b  #040h, &P2IE
    and.b   #0BFh,&P2SEL
    and.b   #0BFh,&P2SEL2

    bis.b   #040h,&P2IES
    clr &P2IFG
    eint

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
setup       mov.b #0FFh, P1DIR
            mov.b #01h, P2DIR
            mov.b #01h, P2OUT
            mov.b #00h, R13

count$str   mov.w R13, R12
count$dsp   mov.w R12, R14
            add.w #array, R14
            mov.b @R14, P1OUT
            call #Delay
            add.b #02h, R12
            cmp.w #0Ah, R12
            jge count$str
            jmp count$dsp

Delay       mov.w #0Ah, R14
L2          mov.w #07A00h, R15
L1          dec.w R15
            jnz L1
            dec.w R14
            jnz L2
            ret

ISR dint
    xor.b   #01h, R13
    dec.b   R12
    clr     &P2IFG
    eint
    reti

array .byte 00111111b, 00000110b, 01011011b, 01001111b, 01100110b, 01101101b, 01111101b, 00000111b, 01111111b, 01101111b
lastElement
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
            .sect   ".int03"
            .short  ISR
```

Before doing anything, we setup the interrupt options and flags. First, we enable interrupts on Port#2.6 and select Port#2 interrupt method as I/O. Then, we set Port#2.6 to trigger on rising edge and clear the interrupt flag.

Port#1 and #2 setup are same as the previous part. Additionally, we choose R13 as flag representing whether odd or even values are counter (0 means even, 1 means odd). This time, instead of starting the counter from 0, we start it from the even-odd flag, since it also acts as a starting value of even and odd numbers, based on its value which also means that we are free of additional compares to decide the number we are starting from.

Rest of the adding functionality is almost same except for using different register instead of saving the value to stack in order to manipulate the current counter from the interrupt and instead of counting in steps of one, we count in steps of two this time.

In the interrupt subroutine we first disable global interrupts for preventing nested interrupts. After that, we toggle the even-odd flag and decrement our counter by one to switch to odd-mode if it is even-mode and vice versa. At the end of our interrupt subroutine, we clear the interrupt flag to prevent it from entering the interrupt subroutine endlessly and enable the global interrupt flag once more and return from interrupt with the command **reti** instead of usual **ret**.

Last but not least, we define the interrupt vector at the end with **.sect ".int03"** and define which label to jump to in case of an interrupt.

2.3. ADVANTAGES/DISADVANTAGES OF INTERRUPT

In the previous experiment (Experiment-4) we have also implemented detection of button press on Port#2.6 but we have continuously checked the value of P2IN instead of interrupts. That approach proved to be buggy since it didn't catch the button presses time to time due to logic of checking the rising edge of the button. However, when we used interrupt to check the button press, we haven't faced with this issue at all, since the logic is built into the hardware and there aren't any risks of de-synchronization of edge checks and button presses; the hardware simply performs a set action when the predetermined conditions are met.

Another advantage of interrupt is the written code is concerned only about the logic of the button press (what should be performed) instead of both the logic and the detection of the button press, which results in much cleaner and understandable code.

3. CONCLUSION

In this experiment, we have learned a lot about interrupts: How to define them, how to configure the ports to accept interrupts, why does the interrupt flag needs to be cleared and what happens when we forget to clear it etc. Apart from the interrupts, we have improved ourselves on assembly in general and got more familiar with MSP430 Education Board.

This experiment was not too hard and we haven't faced any serious issues apart from a bug that caused the counter to skip a value only when switching to odd-mode from even-mode but we have solved that problem rather quickly.