ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY
EXPERIMENT REPORT

EXPERIMENT NO	:	6
EXPERIMENT DATE	:	27.11.2019
LAB SESSION	:	WEDNESDAY - 13.30
GROUP NO	:	G10

GROUP MEMBERS:

150170062	:	Mehmet Fatih YILDIRIM
150180704	:	Cihat AKKİRAZ
150180705	:	Batuhan Faik DERİNBAY
150180707	:	Fatih ALTINPINAR

FALL 2019-2020

Contents

1 INTRODUCTION

In this experiment, using 7-segment display and MSP430 microcontroller desired tasks are implemented in the experiment booklet. An interrupt handler(ISR), an timer interrupt handler(TISR) are used to perform these tasks.

2 MATERIALS AND METHODS

This experiment is conducted via using MSP430G2553 microprocessor. This microprocessor is programmed using Code Composer Studio according to desired tasks on the experiment handout. During coding below sources are used:

- MSP430 Education Board Manual [1]
- MSP430 Architecture Chapter 4 [2]
- MSP430 Instruction Set [3]
- Supplementary Chapter 6 General Purpose [?]
- MSP430 User Guide - Chapter 8 [?]

2.1 Part 1

In the first part of the experiment, the team was asked to write an infinite loop as the main program code in order to light up all four digits of the 7-segment display panel simultaneously, achieving an output as follows "0123" (See Figure 1).

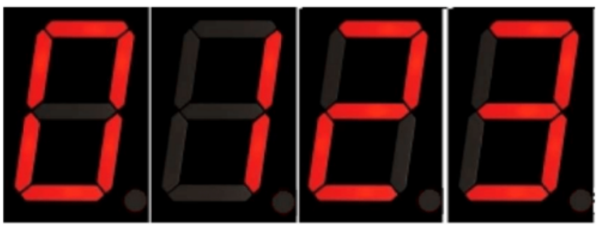

Figure 1: Sample Output of 4 Digit 7-Segment Display

To accomplish the assigned task, the code given in Figure 2.1 was written. Let's go through the code and analyze the implementation:

- Line 2-4: Initial setup of the ports 1 and 2 are done. All bits of port 1 and first 4 bits of port 2 are enabled. Then all the bits of port 1 are cleared.

- Line 5-11: We set four registers each representing a digit of the display. From left to right R7, R6, R5 and R4 holds the values that is printed on the corresponding digits of the display. Even though the aim could have been achieved using only one register, because the next part involves modification of the values of each digit, this approach was chosen for flexibility. After moving the address of arr, that holds the binary values to output for each decimal number from zero to nine, to each register, their values were incremented respectively.

- Line 13-17: First the value stored in the memory address which is held by the corresponding register -in this case R4- is written to the port 1. Then the digit to display the value on port 1 is chosen using port 2. Notice here that hex value $0x08$ is $1000b$ and activates the rightmost digit of the display. It's also commented on the very first line in order to minimize unnecessary confusion. After selection of the outputs a no operation "NOP" instruction is executed in order to increase the delay between two digits. Without it, assuming the refresh rate of the digits are not able to catch up with the execution speed of the micro-controller, the display gets very low dark and unreadable. Then the output ports are cleared and got ready for the next digit to be displayed.

- Line 18-32: Same actions for the first digit are repeated for the remaining three digits in these lines. Note that the only differences are the aforementioned registers and its corresponding digit on port 2.

- Line 33: The program jumps back to the Main, completing a full cycle. Because it jumps back to the beginning without any exit condition, an infinite loop is achieved.

It is important to emphasize that because the data line for the display is shared among the digits, the illusion of a constant display is achieved by frequently flashing each digit of the display for a brief moment. None of the digits are light up exactly at the same time in order to achieve a constant display, else-wise, the number on the digits would have been the same.

```
;r4 = 1, r5 = 10, r6 = 100, r7 = 1000, r15 = timer
Setup                   bis.b   #0FFh,  &P1DIR
                        bis.b   #00Fh,  &P2DIR
                        bic.b   #0FFh,  &P1OUT
                        mov.w   #arr,   r4
                        add     #3,     r4
                        mov.w   #arr,   r5
                        add     #2,     r5
                        mov.w   #arr,   r6
                        add     #1,     r6
                        mov.w   #arr,   r7

Main                    mov.b   @r4,    &P1OUT
                        mov.b   #08h,   &P2OUT
                        nop
                        clr     &P1OUT
                        clr     &P2OUT
                        mov.b   @r5,    &P1OUT
                        mov.b   #04h,   &P2OUT
                        nop
                        clr     &P1OUT
                        clr     &P2OUT
                        mov.b   @r6,    &P1OUT
                        mov.b   #02h,   &P2OUT
                        nop
                        clr     &P1OUT
                        clr     &P2OUT
                        mov.b   @r7,    &P1OUT
                        mov.b   #01h,   &P2OUT
                        nop
                        clr     &P1OUT
                        clr     &P2OUT
                        jmp Main

arr .byte   00111111b, 00000110b, 01011011b, 01001111b, 01100110b
   , 01101101b, 01111101b, 00000111b, 01111111b, 01101111b

```

Figure 2: Main Loop - Part 1

2.2 Part 2

In this part, a chronometer which counts up continuously is implemented. When P2.5 is pushed or overflow occurs, the chronometer resets the time back to 0 and continues counting up.

In order to perform these operations, three code blocks are coded.

- Timer Interrupt Subroutine
- Interrupt Subroutine
- BCD Convertion Subroutine

Let us go through the code and analyze the implementation.

- Line 1-6: This is the setup sequence to enable interrupt functionality. Port 2's 7^{th} bit is set to 1, enabling interrupt when 7^{th} button is pressed. Remaining bits are set to 0 so I/O functions are selected for corresponding pins. Interrupt flag is set on a high-to-low transition for the 7^{th} bit. Then interrupt flags are cleared and interrupt is enabled for the micro-controller.

- Line 8-11: Initial setup of the ports 1 and 2 are done. All bits of port 1 and first 4 bits of port 2 are enabled. Then all the bits of port 1 are cleared.

- Line 13-18: `Set_timer` function assigns the necessary values to TA0CTL, TA0CCR0 and TA0CCTL0 to have the timer exactly as needed. Let us examine them one by one.
  TA0CTL is the configuration of the timer. It has 16 bits, the bits 15-10 of which is unused. Bits 9-8 should be 10 as using SMCLK is asked in lab booklet. Bits 7-6 should be 00 in order to set the input divider as /1 which will be okay in this case. Bits 5-4 should be 01 because "up mode" will be used.
  TA0CCR0 consists of 16 bits and holds the value, up to which the timer will count. SMCLK's frequency is 1048576 Hz, meaning that at each second it operates that many times. The interrupts should be done once a centisecond, therefore the timer should count up to 1048576 divided by 100, which yields approximately 10486. Therefore, this value is assigned to TA0CCR0.
  TA0CCTL0 consists of 16 bits as well and is the configuration of comp/cap mechanism. Bit 8 is set to 0 to choose compare mode. Bit 4 is set to 1 to enable interrupt request.

```
setup_INT    bis.b        #040h,           &P2IE
             and.b        #0BFh,           &P2SEL
             and.b        #0BFh,           &P2SEL2
             bis.b        #040h,           &P2IES
             clr                           &P2IFG
             eint
;r4 = 1, r5 = 10, r6 = 100, r7 = 1000,
Setup        bis.b        #0FFh,           &P1DIR
             bis.b        #00Fh,           &P2DIR
             bic.b        #0FFh,           &P1OUT
             mov.b        #001h,           &P2OUT

Set_timer        ; TA0CTL  15-10..100001x010
                          ; TA0CCR0       #10486d
                          ; TA0CCTL0   00??x?x00011x?x0
                          mov.w   #01000010000b,   TA0CTL
                          mov.w   #10486d,         TA0CCR0
                          mov.w   #0000000000010000b,      TA0CCTL0


```

Figure 3: Setup sequences for Part 2

- Line 28-53: Main function firstly calls BCD2Dec function to have the necessary values pointed by registers R4, R5, R6 and R7. Then, for each digit, the value stored in the memory address which is held by the corresponding register is written to the port 1. Then the digit to display the value on port 1 is chosen using port 2. Notice here that for instance, for the first digit, hex value $0x08$ is $1000b$ and it activates the rightmost digit of the display. After selection of the outputs, a no operation "NOP" instruction is executed in order to increase the delay between two digits. Without it, giving the refresh rate of the digits that are not able to catch up with the execution speed of the micro-controller, the display is dark and unreadable. Then the output ports are cleared and became ready for the next digit to be displayed. At line 53, the program jumps back to the Main, completing a full cycle. Because it jumps back to the beginning without any exit condition, an infinite loop is achieved.

```
28 Main            call    #BCD2Dec
29                         mov.b   @r4,            &P1OUT
30                         mov.b   #08h,           &P2OUT
31                         nop
32                         nop
33                         clr             &P1OUT
34                         clr             &P2OUT
35                         mov.b   @r5,            &P1OUT
36                         mov.b   #04h,           &P2OUT
37                         nop
38                         nop
39                         clr             &P1OUT
40                         clr             &P2OUT
41                         mov.b   @r6,            &P1OUT
42                         mov.b   #02h,           &P2OUT
43                         nop
44                         nop
45                         clr             &P1OUT
46                         clr             &P2OUT
47                         mov.b   @r7,            &P1OUT
48                         mov.b   #01h,           &P2OUT
49                         nop
50                         nop
51                         clr             &P1OUT
52                         clr             &P2OUT
53                         jmp             Main
54
```

Figure 4: Main Loop - Part 2

- Line 55-60: The interrupt service routine (ISR) is called when Port 2 receives an interrupt signal. INT03 interrupt vector is instantiated and the interrupt handler routine in the memory location ISR is called. ISR disables the interrupts and zeros out the sec and csec values. Then clears the interrupt flag, re-enables the interrupts and returns from interrupt.

```
55 ISR                 dint
56                     mov.b   #00h,           sec
57                     mov.b   #00h,           csec
58                     clr             &P2IFG
59                     eint
60                     reti
61
62
```

Figure 5: Interrupt Subroutine - Part 2

- Line 63-94: The timer interrupt service routine (TISR) is called when the timer sends an interrupt signal once a csec. INT09 interrupt vector is instantiated and the interrupt handler routine in the memory location TISR is called. TISR firstly disables the interrupts. Then, increments csec. After that, for each digit of csec value (in function names, they are referred to as csec, decisec(decsec), sec, decasec(deksec)), it does the following:
  If the relevant digit is being incremented from the value 9, it jumps to the function which increments the digit right next to it (which is the rightmost digit that is greater in significance). Otherwise, it jumps to TISRend directly. TISRend reenables the interrupts and returns from interrupt.
  While doing all these operations, in order not to lose the value in R15, at top, its value is pushed to stack and at the very end, its value is popped back.
  Note that even though generally the interrupt flag has to be cleared before returning from interrupt, it is not necessary in this case, because it is already automatically done.

```
63  TISR        dint
64                      push    r15
65                      add.b   #1b,            csec
66                      mov.b   csec,           r15
67                      bic.b   #0F0h,          r15
68                      cmp             #0Ah,           r15
69                      jz              ADDDecSec
70                      jmp             TISRend
71
72  ADDDecSec   add.b   #010h,          csec
73                      bic.b   #00Fh,          csec
74                      mov.b   csec,           r15
75                      cmp             #0A0h,          r15
76                      jz              ADDSec
77                      jmp             TISRend
78
79  ADDSec      add.b   #001h,          sec
80                      bic.b   #0FFh,          csec
81                      mov.b   sec,            r15
82                      cmp             #0Ah,           r15
83                      jz              ADDDekSec
84                      jmp             TISRend
85
86  ADDDekSec   add.b   #010h,          sec
87                      bic.b   #00Fh,          sec
88                      mov.b   sec,            r15
89                      cmp             #0A0h,          r15
90                      jz              RESET
91
92  TISRend     pop     r15
93                      eint
94                      reti
95
96
```

Figure 6: Timer Interrupt Subroutine - Part 2

- Line 100-132: BCD2Dec function is called in Main to get the values of csec and sec which are set by TISR in BCD format, turn them into decimal format and set R4,

R5, R6 and R7 so that they hold the addresses of values to be used when printing. This is achieved by firstly, taking csec and sec (2 times each); secondly, if it is the second least significant bit of csec or sec that is being dealt with, performing right shift 4 times to get the second least significant bit of that variable (otherwise, least significant bit will already be had); thirdly, masking the value so that only the desired digit will be had and lastly, adding this value to be shown to the address of arr to have the address of the binary value in arr to be used to print the value to be shown and assigning it to the relevant register among R4, R5, R6 and R7.
While doing all these operations, in order not to lose the value in R14, at top, its value is pushed to stack and at the very end, its value is popped back.

```
100 BCD2Dec         push    r14
101
102                 mov.b   csec,   r14
103                 bic.b   #0F0h,  r14
104                 mov.w   #arr,   r4
105                 add.w   r14,    r4
106
107                 mov.b   csec,   r14
108                 rra.b   r14
109                 rra.b   r14
110                 rra.b   r14
111                 rra.b   r14
112                 bic.b   #0F0h,  r14
113                 mov.w   #arr,   r5
114                 add.w   r14,    r5
115
116
117                 mov.b   sec,    r14
118                 bic.b   #0F0h,  r14
119                 mov.w   #arr,   r6
120                 add.w   r14,    r6
121
122                 mov.b   sec,    r14
123                 rra.b   r14
124                 rra.b   r14
125                 rra.b   r14
126                 rra.b   r14
127                 bic.b   #0F0h,  r14
128                 mov.w   #arr,   r7
129                 add.w   r14,    r7
130
131                 pop             r14
132                 ret
133
134
```

Figure 7: BCD Conversion Subroutine - Part 2

- Line 135-140: In these lines, necessary definitions are done. Firstly, an array of 10 elements is defined which holds the bit-wise values that turn on the corresponding LEDs of the 7-segment display in order to display decimal numbers. Then, 2 more variables sec and csec are initialized as 0 whose values will be displayed.

```
;0, 1, 2, 3, 4, 5, 6, 7, 8, 9
arr                     .byte   00111111b, 00000110b, 01011011b, 01001111b, 01100110b, 01101101b, 01111101b, 00000111b, 01111111b, 01101111b
arr_end
                        .data
sec                     .byte   00h
csec            .byte   00h

```

Figure 8: Definitions, variables - Part 2

3 RESULTS

At the end of the first task, the team is succeeded in lit different digits of 7-segment display panel simultaneously. How this is implemented are mentioned detailed in the methods part. Although the brightness at first was slightly lower, brightness was increased by playing with the delay which enables every digit to stay visible just a little longer.

At the second part of the experiment, a chronometer is implemented. How this chronometer is implemented mentioned detailed in the methods part. At the end of the experiment, the team had a chronometer that has functionality counting up continuously. Also has a functionality that when pressed a declared reset button, sets timer to 0 and starts counting up again.

Team also gathered how the timer works in MSP430, how to set it up and use it.

4 DISCUSSION

In the first part, lighting up all of the digits at once is possible by showing all digits one after the other rapidly. Which is enough to trick human eye to think all of them are lit at the same time. This is how all of the screens work in today's world. Old televisions used to scan the image and display it on the screen as little parts going from top left to top right then second line and so on.

Interrupts caused by the timer was the focus in the second part of the experiment. A timer outside of the processor creates much better solutions for delaying program executions, computing are done related to time and so on. Before a delay can be achieved by writing correct value into a register and counting down from it in the main program. This creates a lot of problems. Since the counting down is done by the processor your program cannot run simultaneously and you have to calculate the value that you are going to write to the register after changing your program. Since number of clock cycles that your program uses will change every time you add or remove an instruction. Another downside of simulating delay in the main program is that power inefficiency since processor will be running all the time rather than sleeping when unnecessary or running other useful calculations.

5 CONCLUSION

As always, the team has successfully completed all tasks. A few problems are faced in during the lab session were: changing the timer flag during timer interrupt service routine and using byte instructions rather than word during moving or adding address values. Timer interrupt flag is risen by itself, interrupt is called then it goes down by itself again. We used to change it in the beginning of the session and timer used to stop counting after first iteration. Took us quite a while to figure this one out.

Another lesson that should be taken from the experiment is remembering that address values are always 16-bit words. Doing byte operations on them will result those values lose first 8-bit which results inaccurate calculations.

REFERENCES

[1] Texas Instruments. Msp430 education board document. 2009.

[2] Texas Instruments. Msp430 architecture. 2009.

[3] Texas Instruments. Msp430 architecture. December 2004 - Revised July 2013.

[4] Overleaf documentation https://tr.overleaf.com/learn.

[5] Detailed info on writing reports https://projects.ncsu.edu/labwrite/res/res-studntintro-labparts.html.