ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY
EXPERIMENT REPORT

EXPERIMENT NO	:	6
EXPERIMENT DATE	:	20.12.2024
LAB SESSION	:	FRIDAY - 14.30
GROUP NO	:	G7

GROUP MEMBERS:

150210087 : HİLAL KARTAL

150210100 : SELİN YILMAZ

150210067 : ALPER DAŞGIN

FALL 2024

Contents

1 INTRODUCTION [10 points]

This experiment used the MSP430 to carry out a series of activities in order to investigate basic microcontroller operations. There were two sections to the experiment:

Part 1: Using a seven-segment display panel, the sequence "0123" is shown concurrently over four numbers.

Part 2: Using interrupt-driven programming to create a clock with sophisticated functions like start, stop, reset, and save best time.

Timer setups, interrupt vectors, and GPIO manipulation were used to maximize the capabilities of the MSP430 microcontroller. This allowed for fine control over hardware components and gave practical experience with assembly-level programming. The experiment focused on precise timing, real-time responsiveness, and useful uses of binary-coded decimal (BCD) conversion for hardware displays. Part 1 demonstrated the synchronization capabilities of GPIO ports by using high-frequency flashing of the 7-segment display components to produce the appearance of a continuous light. Interrupt service routines (ISRs) were included in Part 2 to facilitate user interaction and guarantee the chronometer's proper operation. Every phase of the experiment demonstrated how crucial methodical debugging, effective resource management, and a thorough comprehension of microcontroller design are to achieving desired results.

2 MATERIALS AND METHODS [40 points]

2.1 MATERIALS

- MSP430
- Selin's Laptop

2.2 METHODS

2.2.1 FIRST PART

The purpose of this part is to lit different digits of the 7-segment display panel simultaneously by an infinite loop. We used the logic: "If a light flashes in a high frequency, we can not see flashes but only constant light.".

We first defined the array we would use for the output.

```
            .data
```

```
array        .word 00111111b, 00000110b, 01011011b, 01001111b
lastElement      ; 0      1    2      3
```

Then, we did the *Setup*.

```
1 Setup        mov.b   #00000000b, &P2SEL
2              mov.b   #11111111b, &P1DIR
3              mov.b   #00001111b, &P2DIR
4              mov.b   #00000000b, &P1OUT
5              mov.b   #00000000b, &P2OUT
6              mov.w   #array, r13
```

Listing 1: Part 1 - Setup

Line 1: Set P2SEL for GPIO mode.

Line 2, 3: Select all of the P1DIR as output for 7-segment display. Select P2DIR as output for digit selection from P2.0 to P2.3.

Line 4, 5: Initialize P1OUT and P2OUT as low.

Line 6: Load the starting address of the array into R13.

We wrote the *Mainloop*.

```
1  Mainloop     mov.b   0(r13), &P1OUT
2               mov.b   #1, &P2OUT
3               call    #ShortDelay
4
5               mov.b   2(r13), &P1OUT
6               mov.b   #2, &P2OUT
7               call    #ShortDelay
8
9               mov.b   4(r13), &P1OUT
10              mov.b   #4, &P2OUT
11              call    #ShortDelay
12
13              mov.b   6(r13), &P1OUT
14              mov.b   #8, &P2OUT
15              call    #ShortDelay
16
17              jmp     Mainloop
```

Listing 2: Part 1 - Mainloop

Line 1: Load the first element (0) of the array to P1OUT.

Line 2: Enable the first digit of P2OUT (P2.0).

Line 3: Call the *ShortDelay* subroutine. This creates a high frequency to observe constant lit.

Line 5, 6, 7: Load the second element (1) of the array to P1OUT. Enable the second digit (P2.1). Call the *ShortDelay* subroutine.

Line 9, 10, 11: Load the third element (2) of the array to P1OUT. Enable the third digit (P2.2). Call the *ShortDelay* subroutine.

Line 13, 14, 15: Load the fourth element (3) of the array to P1OUT. Enable the fourth digit (P2.3). Call the *ShortDelay* subroutine.

Line 17: Jump to *Mainloop* to repeat the sequence.

Finally, we wrote the *ShortDelay* to create a high frequency for visibility.

```
1 ShortDelay   mov.w    #0010h, r14
2 DelayLoop    dec.w    r14
3              jnz      DelayLoop
4              ret
```

Listing 3: Part 1 - ShortDelay

Line 1: Load a small value to R14 to make it a counter.

Line 2, 3: Decrement it until it becomes 0.

Line 4: Return from the subroutine.

2.2.2 SECOND PART

The purpose of this part was to design a chronometer with some features using the buttons.

- Reset - When it is pressed, time should be reset back to 0. (P2.7)
- Stop - When it is pressed, the counter should stop counting. (P2.6)
- Start - When it is pressed, the counter should start counting. (P2.5)
- Save Best Time - When the stop and start buttons are pressed together if the current time is the longest duration, then it must be saved.

We added the necessary interrupt vectors for this buttons.

```
             .sect    ".int09"
             .short   TISR

             .sect    ".int07"             ; Vector for P2.7 (Reset)
             .short   ISR_reset

             .sect    ".int06"             ; Vector for P2.6 (Stop)
             .short   ISR_stop

             .sect    ".int05"             ; Vector for P2.5 (Start)
```

```
            .short  ISR_start
```

Listing 4: Part 2 - Interrupt Vectors

Then, we defined the array and second, centisecond.

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

sec      .byte   00h
csec     .byte   00h
```

Then, we configured the settings for interupt.

```
1 setup_INT:
2       bis.b    #0F0h, &P2IE
3       and.b    #00Fh, &P2SEL
4       and.b    #00Fh, &P2SEL2
5       bis.b    #0F0h, &P2IES
6       clr      &P2IFG
7       eint
```

Listing 5: Part 2 - setup_INT

Line 1: Enables interrupts for P2.7, p2.6, P2.5, and P2.4 (normally save but we could not do it).

Line 2, 3: Configures P2.7, P2.6, P2.5, P2.4 as GPIO. Difference between those lines is P2SEL register selects whether a pin is used for a peripheral function or as GPIO. P2SEL2 works in conjunction with P2SEL to determine the pin function when multiple peripheral options are available.

Line 4: Set the edge select for Port2 interrupt sources. *#0F0h* is a bitmask where the upper 4 bits (P2.7, P2.6, P2.5, P2.4) are set to 1, and the lower 4 bits are 0. Doing this causes those pins (P2.7-P2.4) to trigger an interrupt on the high-to-low transition.

Line 5: Clear the interrupt flags for Port 2.

Line 6: This is used to enable global interrupts.

We set up GPIO and Timer.

```
Setup:
    bis.b    #0FFh, &P1DIR
    bis.b    #00Fh, &P2DIR
    bic.b    #0FFh, &P1OUT
    mov.b    #001h, &P2OUT
```

```
6      mov.b #2d, r10
7
8  Set_timer:
9      mov.w   #01000010000b, TA0CTL
10     mov.w   #10486d, TA0CCR0
11     mov.w   #0000000000010000b, TA0CCTL0
```

Listing 6: Part 2 - Setup

Line 2, 3: All P1 pins are set as output. The lower nibble of P2 is set as output.

Line 4, 5: Clear P1OUT. Initialize P2.0 as 1.

Line 6: Prepare R10 register with the mode variable.

Line 9: Configure Timer A *TA0* with control *TA0CTL*.

Line 10: Compare vale *TA0CCR0*. The timer will trigger an interrupt when its value matches *TA0CCR0*.

Line 11: Capture/compare control *TA0CCTL0*.

We created the loop structure.

```
Main:
    call    #decidemode
    call    #BCD2Dec
    mov.b   @r4, &P1OUT
    mov.b   #08h, &P2OUT
    nop
    nop
    clr     &P1OUT
    clr     &P2OUT
    mov.b   @r5, &P1OUT
    mov.b   #04h, &P2OUT
    nop
    nop
    clr     &P1OUT
    clr     &P2OUT
    mov.b   @r6, &P1OUT
    mov.b   #02h, &P2OUT
    nop
    nop
    clr     &P1OUT
    clr     &P2OUT
    mov.b   @r7, &P1OUT
    mov.b   #01h, &P2OUT
    nop
    nop
    clr     &P1OUT
```

```
27     clr      &P2OUT
28     jmp      Main
29
30 decidemode  cmp #0, r10
31             jeq Setup
32             cmp #1, r10
33             jeq stop
34             cmp #2, r10
35             jeq continue
36 continue    ret
37
38 stop        nop
39             jmp decidemode
```

Listing 7: Part 2 - Main

Line 2: Calls *decidemode* to check whether a button is pressed.

Line 3: Then, main calls BCD2Dec to gain the values pointed by R4-R7.

Line 4: Write what is written in the address pointed by R4 into P1OUT.

Line 5: The rightmost digit of the display is activated.

Line 6, 7: Increase delay between instructions.

Line 8, 9: Output ports are cleaned.

Line 10: Write what is written in the address pointed by R5 into P1OUT.

Line 11: The next digit of the display is activated.

Line 28: Jump to the beginning of *main*.

Line 30, 31: Compares R10 with 0. If they are equal, branch to *Setup*.

Line 32, 33: Compare R10 with 1. If they are equal, branch to *stop*.

Line 34, 35: Compare R10 with 2. If they are equal, branch to *continue*.

Line 36: Return to the caller function.

Line 38, 39: Jump to *decidemode*.

The interrupt service routine is implemented.

```
1 ISR:
2     dint
3     mov.b    #00h, sec
4     mov.b    #00h, csec
5     clr      &P2IFG
6     eint
7     reti
```

Listing 8: Part 2 - ISR

Line 1: Disable interrupts.

Line 2, 3: Clear *sec* and *csec*.

Line 4: Clear interrupt flag.

Line 5: Re-enable interrupts.

Line 6: Return to the caller instruction.

We implemented the time interrupt service routine.

```
TISR:
    dint
    push    r15
    add.b   #1b, csec
    mov.b   csec, r15
    bic.b   #0F0h, r15
    cmp     #0Ah, r15
    jz      ADDDecSec
    jmp     TISRend

ADDDecSec:
    add.b   #010h, csec
    bic.b   #00Fh, csec
    mov.b   csec, r15
    cmp     #0A0h, r15
    jz      ADDSec
    jmp     TISRend

ADDSec:
    add.b   #001h, sec
    bic.b   #0FFh, csec
    mov.b   sec, r15
    cmp     #0Ah, r15
    jz      ADDDekSec
    jmp     TISRend

ADDDekSec:
    add.b   #010h, sec
    bic.b   #00Fh, sec
    mov.b   sec, r15
    cmp     #0A0h, r15
    jz      RESET

TISRend:
    pop     r15
    eint
    reti
```

Listing 9: Part 2 - TISR

Line 2: Disable interrupts.

Line 3: Save the R15 to stack.

Line 4: Add 1 to the lower byte of the *csec*.

Line 5: Move the csec to R15.

Line 6: Clear the upper nibble (4 bits) of R15 by bitwise AND with the complements of 0xF0. Ensure only the last digit of csec is being checked.

Line 7, 8, 9: Compare R15 with 10. If equal jump to *ADDDecSec*. If not jump to *TISRend*.

Line 12: Add 16 to csec.

Line 13: Clear the lower nibble of csec to reset the units place to zero.

Line 14: Move the last version of csec to R15.

Line 15, 16, 17: Compare R15 with 160. If equal, jump to *ADDSec*. If not, jump to *TISRend*.

Line 20: Add 1 to the *sec*.

Line 21: Clears all bits in *csec*.

Line 22: Move sec to R15.

Line 23, 24, 25: Compare R15 with 10. If equal, jump to *ADDDekSec*. If not, jump to *TISRend*.

Line 28: Add 16 to *sec*.

Line 29: Clear the lower nibble of sec.

Line 30: Move sec to R15.

Line 31, 32: Compare R15 with 160. If equal, jump to the *RESET* routine (defined in the interrupt vectors).

Line 35: Restore the original value of R15.

Line 36: Re-enable interrupts.

Line 37: Return from the interrupt service routine to the caller instruction.

Then, we implemented a Binary-Coded-Decimal (BCD) to Decimal conversion for the csec and sec.

```
BCD2Dec:
    push    r14

    mov.b   csec, r14
    bic.b   #0F0h, r14
    mov.w   #arr, r4
    add.w   r14, r4

    mov.b   csec, r14
    rra.b   r14
```

```
11      rra.b   r14
12      rra.b   r14
13      rra.b   r14
14      bic.b   #0F0h, r14
15      mov.w   #arr, r5
16      add.w   r14, r5
17
18      mov.b   sec, r14
19      bic.b   #0F0h, r14
20      mov.w   #arr, r6
21      add.w   r14, r6
22
23      mov.b   sec, r14
24      rra.b   r14
25      rra.b   r14
26      rra.b   r14
27      rra.b   r14
28      bic.b   #0F0h, r14
29      mov.w   #arr, r7
30      add.w   r14, r7
31
32      pop     r14
33      ret
```

Line 2: Save R14 to stack.

Line 4: Move *csec* to R14.

Line 5: Clear the upper nibble of R14.

Line 6: Move the starting address of *arr* array to R4.

Line 7: Add R14 to R4.

Line 9: Move *csec* to R14 again.

Line 10, 11, 12, 13: Logically shift right (divide by 2) R14 four times.

Line 14: Clear the upper nibble of R14.

Line 15: Move the starting address of the *arr* array to R5.

Line 16: Add R14 to R5.

Line 18: Move *sec* to R14.

Line 19: Clear the upper nibble of R14.

Line 20: Move the starting address of the *arr* array to R6.

Line 21: Add R14 to R6.

Line 23: Move *sec* to R14.

Line 24, 25, 26, 27: Logically shift right (divide by 2) R14 four times.

Line 28: Clear the upper nibble of R14.

Line 29: Move the starting address of the *arr* array to R7.

Line 30: Add R14 to R7.

Line 32: Restore the original value of R14 from the stack.

Line 33: Return from the subroutine to the caller instruction.

With these codes, we managed to see the chronometer. So we continued to the buttons part. However, we were able to implement only the "stop" button. Other ones did not work. But here are the codes we wrote for all of the buttons. And also we did not think about the save-button.

```
1 ISR_reset   dint
2             ; Handle Reset (P2.7)
3             mov.w #0, r10
4             eint
5             reti
6
7 ISR_stop    dint
8             ; Handle Stop (P2.6)
9             mov.w #1, r10
10            eint
11            reti
12
13 ISR_start  dint
14            ; Handle Start (P2.5)
15            mov.w #2, r10
16            eint
17            reti
```

Listing 10: Part 2 - ISR_button

Line 1-5: Disable interrupts. Move 0 to R10. Re-enable interrupts. Return from the isr.

Line 7-11: Disable interrupts. Move 1 to R10. Re-enable interrupts. Return from the isr.

Line 13-17: Disable interrupts. Move 2 to R10. Re-enable interrupts. Return fromt the isr.

3 RESULTS [15 points]

3.1 FIRST PART

Figure 1: Result for part 1

3.2 SECOND PART

We were able to see the chronometer and work the stop interrupt. However, we were not able to implement the other interrupts.

4 DISCUSSION [25 points]

4.1 PART 1

In order to display the sequence "0123" over all four digits of a seven-segment display, we first put up a software. The information acquired includes:
A delay subroutine was implemented to provide synchronization and smooth transitions between displays, enhancing understanding of loop-based timing in assembly. The subroutine is shown below:
GPIO Setup and Control of the Seven-Segment Display: Appropriate GPIO initialization was required in order to effectively control the segments. To guarantee correct display output, precise mapping between GPIO ports and 7-segment display pins was necessary. A crisp and synchronized display was produced by managing to show one of the numerals (0, 1, 2, or 3) simultaneously on each digit of the seven-segment display.

```
Delay           mov.w   #0Ah, r14L2             mov.w   #07A00h, r15
L1              dec.w   r15
jnz     L1
dec.w   r14
jnz     L2
```

```
ret
```

4.2 PART 2

An interrupt was used to implement the even and odd counting modes, showing real-time responsiveness to outside inputs. Through the activation of a maskable interrupt on GPIO Port 2, the system dynamically switched between odd and even modes. Below is the interrupt service routine (ISR):

```
ISR             dint
xor.b   #11111111b, r11
call    #r_counter
clr     &P2IFG
eint
reti
```

Significant aspects of this implementation consist of:
→ switching between even and odd modes using a Boolean variable that is kept in a register.

→ The Boolean state was used to access arrays that included values for even and odd digits.

→ Accurate handling of button toggles was achieved by properly initializing the interrupt vector.

→ In order to prevent recurrent interruptions, flags were cleared inside the ISR and interrupts were triggered via port P2.

Additionally, this section highlighted the ideas of:

→ Use assembly directives to interrupt configuration and initialization.

→ effective use of system resources by using registers to switch between memory operations.

→ Useful information about real-time microcontroller applications and interrupt handling.

4.3 Details and Implementation of Timer A

Timer A was set up to use the SMCLK signal as the input clock source in order to produce periodic interruptions at set times. Among the configuration registers were:

TA0CTL: Set the clock source and timer mode.

TA0CCR0: Configure the compare value to generate interrupts.

TA0CCTL0: The timer compare event's interrupts were enabled.

4.4 Implementing Four Interrupt Vectors

For handling different functionalities, four interrupt vectors were put into place:

Timer Interrupt (TISR): Accuracy was maintained within centiseconds.

Stop/Start Interrupt: Toggled timer states by handling button presses.

Reset Interrupt: The chronometer was reset by clearing the time values.

Save as much time as possible. Interrupt: Updated as necessary after comparing the current time with stored values.

Every interrupt vector was set up and controlled to guarantee smooth transitions and dependable operation.

4.5 Conversion to BCD

Binary time values were converted into formats compatible with 7-segment displays using the binary-coded decimal (BCD) conversion procedure. This required breaking down the data in centiseconds and seconds into separate digits using:

```
.Bytes 00000110b, 01011011b, 00111111b,...
```

Direct display on the hardware was made possible by mapping the digits to their corresponding segments.

4.6 TISR Interrupt

The centisecond and second counters were updated by the Timer Interrupt Service Routine (TISR). In order to prevent several triggers, the procedure made sure that updates were made on a regular basis and cleared interrupt flags.The chronometer's ability to keep time was based on this pattern.

4.7 Stop Interrupt

The stop interrupt handled button presses to halt or resume timer counting. By toggling a Boolean flag, the system alternated between active and paused states. Proper flag management ensured responsive and consistent behavior.

5 CONCLUSION [10 points]

- From first part, we learned how to use the four of the 7 segments at the same time
- From the second part, we learned how to initialize 4 interrupts and how to use them in an ISR, we also learned how to write a BSD conversion Subroutine

- However, we were not able to finish the last part as we could not make our interrupts work as intended. We were only able to make the chronometer and the stop interrupt.

REFERENCES

[1] Microcomputer Lab. Micro_experiment_6. *Lab Booklet*, 2024.

[2] Texas Instruments. *MSP430x2xx Family User's Guide.* Texas Instruments, revised july 2013 edition, December 2004.

[3] Digital Logic Programming. 12.2(d) - msp430 timers - overflow example using smclk, April 2020.