ACE411 - Embedded Systems

LAB 3

7-Segment Display Decoder w/ 8 Digit Multiplexing & USART input

Odysseas Stavrou 2018030199

October 2021

Technical University of Crete

Prof.: A. Dollas

The 3rd Lab Excersice was a variation of the 2nd Excersice with the addition that the microcontroller now supports a really basic Haze commant set, that is to be read from the USART port. For this Lab once again, AVR assembly was used.

Command	Arguments	Action	Reply
AT<CR><LF>	No Arguments	Attention	OK<CR><LF>
C<CR><LF>	No Arguments	Clear Display[1]	OK<CR><LF>
N0123CR><LF>	0123[2]	Display Argument	OK<CR><LF>

1. **Timer Configuration**

   Since this was explained in great extent in the past two Lab exercices the Frequency Derivation will not be covered here. Timer0 was used with an initial value of 99 and a prescaler of $\frac{clk}{256}$ (4) for a Frequency of 250Hz.

2. **Decoder**

   The same decoder was used with the following two differences:

   i. The values now lie in the Code Section rather than the SRAM.

   ii. There has been an addition of another 'digit', 0x0A, with the sole purpose of decoding to 0xFF, setting all segments to HIGH, effectively turning off the whole Display

3. **USART Configuration**

   In Order to set up a usable USART Port, the first thing to do is define a baud rate (bits/s). For the USART Port to operate as a specific baud rate the UBRR Register has to be loaded with the appopriate value, calculated using this formula:

   $$UBRR = \frac{sys_{clk}}{16 \cdot baudrate} - 1$$

   With a system clock of 10 MHz and a baudrate of 9600 bps the contents of the UBRR Register are appox 64.10.

   In order to be able to Recieve and Transmit Data the RXEN and TXEN bits have to be set in the UCSRB register, along with the RXCIE bit, so that an Interrupt arises when the Port finishes recieving data.

[1] Turn off All Digits

[2] Up to 8 Digits

4. **Writing to USART Port**

   For writting a byte to the Port, a check is performed in order to ensure that the UDR Register is empty, by checking if the UDRE bit is set in the UCSRA Register. If as such, then the UDR is written with the prefered byte, else the routine rechecks indefinetely until the UDR Register is empty. This technique is called 'polling' and the call to the 'putc' subroutine is a blocking call, meaning that the program will halt until the call is finished.

```
putc:
in r18, UCSRA
sbrs r18, UDRE
rjmp putc

; out UDR, r19
out TCNT2, r19
ret
```

   Note: Since this will not be uploaded to an actual AVR Microproccessor, the UDR Register is replaced by the unused TCNT2 Register for monitoring Purposes.

   The only time writting to the USART Port happens is when the Microproccessor has to sent the Reply 'OK'. When replying, the program makes a call to the 'reply' subroutine which in turn call the 'putc' subroutine four times in order to write four bytes 'O', 'K', <CR>, <LF>, as output. The reply string is stored alongside the decoder values in FLASH.

5. **Reading from USART Port**

   Unlike the approach of writing to the USART, reading is implemented in a way that does not block. As soon as the USART_RXC Interrupt arises, the program will read from the UDR Register in order to consume the RXC flag.

   Then the program will act accordingly, depending on the recieved byte:

Byte	Action	Reply
'A', 'T', <CR>	None	None
'C', 'N'	Clear Display	None
<LF>	None	OK<CR><LF>
'0' .. '9'	Write to Data	None

   Keep in mind that this is an ad-hoc solution to the recieving problem. If the input is not as described above, then this approach will fail! It is Assumed that the input is **ALWAYS** correct and within spec. A better solution will be to store the recieved Data, Process it, and then Reply.

6. **Parsing Input**

   Nothing special is happening here, the program will call the 'clr_data' and the 'reply' subroutines depending on what byte is recieved. The only caviat is when a digit is recieved. In such case the whole data has to be shifted to a higher memory address and then store the recieved byte on the lowest address. The numbers address has to point to the lowest digit (Little Endian)

```
shift:
; load address of 2nd to last byte
ldi xL, low(data)
add xL, r19
; save byte
ld r0, x
; inc xL pointer so it points to last byte
mov r18, xL
inc r18
mov xL, r18
; store 2nd to last byte
st x, r0
; decreace offset
dec r19
cpi r19, -1
brne shift

; when shift ends, store recieved byte
ldi xL, low(data)
st x, r16
```

Also, since the numbers are the ASCII printable numbers, number 1 is expressed as 0x31 instead of 0x01, so an ANDI is performed against a mask of 0x0F in order to convert it to 0x01. (A SUBI of 0x30 would also work)

7. **Testing Using Stimuli Files**

In order to confirm the above working behaviour, a Stimuli File is Constructed, defining the contents of some registers, the amount of cycles bewteen changes and which registers to monitor for changes. For Example, for testing if the program replies correctly, after the AT command is recieved, the values 'A', 'T', <CR>, <LF> are loaded in order into R1. To simulate the RXC Interrupt, UCSRA is set to a value of 0x80 after every change of R1.

If all went as planned, in the specified log file there should be 4 changes of the value in TCNT2 register, more specifically TCNT2 should take the values (in order): 'O', 'K', <CR>, <LF>.

The debugging methodology that was used was really simple:

i. Load the stimfile with proper funcionality

ii. Break each time after reading from "UDR" inside the Interrupt Handler

iii. Monitor Memory changes or Program Flow depending on read Byte

Note: The cycles between each change were set to 1000 in order to give enough time for the interrupts to be handled so that the change wont be caught inside the interrupt.

There is only one stimfile provided, but it checks all the funcionality the program needs to perform (Clear, Reply, Write in proper order).