EMBEDDED SYSTEMS

PROJECT - AN OTHELLO GAME

REPORT OF 1ου MILESTONE

12 April 2019
Professor: A. Dollas

Working Group LAB41140573
Vasilakopoulos Panagiotis A.M:2015030048
Maragkaki Maria A.M:2015030153

PROJECT PURPOSE

The purpose of this Project is the implementation of the Othello game where it can play a game against another human player or against another STK500. The Othello game is played on a 8x8 chessboard with cells of the same color. On the horizontal axis the columns are numbered from 1 to 8 and on the vertical axis the lines named from A to H. There are two-colored checkers, black and white. Each player choose one color. At the start of the game 4 checkers are placed in the middle of the chessboard. Two white in the positions D4 and E5 and two black in the positions D5 and E4. The first player places a checker near of the opponent's checker. If the opponent's checkers are closed into the first's player checkers, then they change color. The goal is at the end of the game one of the players to have the most checkers on his color. If the number of checkers is the same then the game is tie. The game ends when all of the squares have a checker or when there are no valid moves.

PURPOSE OF MILESTONE 1

The purpose of the first milestone is a first contact with the Ohello game and the implementation of basic modules that will be needed for the project. Requests of this Milestone were the following:

- Serial port interface implementation for which supports the instruction set and the answers of the instructions
- Implementation of lighting a LED
- Chessboard implementation in the SRAM with the data structures for initialize, declaration of an empty cell, declaration of unavailable cell with black or white

SERIAL PORT INTERFACE IMPLEMENTATION

➔ Serial Port Implementation

First of all was implemented the serial port communication for send and receive ASCII characters. First of all, was defined the system's Baud Rate and with the function $UBBR = \frac{f_{osc}}{16*BAUD} - 1$, was implemented the value to be stored in the UBBR register. In this case the Baud Rate was 9600 bits/sec. During system initialization the bit UDRE($5^{th}\ bit$) of UCSRA register, is activated until the UDR register is used for sending data, which is suggests that the transmit buffer is read to receive data. Subsequently, the bit TXCIE($6^{th}\ bit$) and TXEN($3^{th}\ bit$) of UCSRB register were enabled, of which the first enabled the interrupt that the Transmitter finished the transmission and the second enables the

transmitter. Finally, the bit UCSZ01 and UCSZ00 of the UCSRC were enabled, where according to the table below, from the ATmega16 datasheet, declare that the system's data frame is 8 bit. Related to Receiver the bit RXIE(7^{th} bit) and RXEN(4^{th} bit) of the UCSRB register were enabled that they are respectively with the Transmiter's. The first activates the interrupt that the Receiver has finished the receive and the second indicates that the receiver is active. Receiver reads the UDR register's data where in this case is ASCII characters, and gives it as input on readByte function. ReadByte function store every byte into a buffer and increment a counter. The function where writes the data for transmit is the wirteByte function which stores into a buffer the data.

In the serial- communication part, two buffers (Txbuffer) and (Rxbuffer) were used with static size in bytes (for example 16 bytes) in order to achieve asynchronous byte - stream transaction. With this design there is no need to wait for the transaction to complete in order to continue the processing. The data for the transaction were inserted in the appropriate buffer and the processing was proceeding individually from the serial communication, because the processes which made the interface between stream of data and the internal RS232, were driven by interrupts. With this design was achieved fast processing speed with the disadvantage of using a large amount of memory (SRAM), specifically the double buffer - size .

Table 17-7. UCSZn Bits Settings

UCSZn2	UCSZn1	UCSZn0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Figure 1: Table of UCSZn bits Settings

➔ Instruction Set Implementation

In the second phase, a communication protocol was implemented. Each instruction is written at the terminal, received from AVR, is decoded and the PC transmits the right data

to the screen of the PC. For the decoding of instructions is used the function examinInsr which is takes as a definition the buffer which is reading from RX and checks one by one the letters where are stored into the buffer. If the letters matches with one of the following instruction it execute the correspond code. The instruction set is the following:

- *Instruction AT<CR>*

In this instruction the PC sends AT to AVR, AVR receive AT and sends back to PC the answer OK. The decoding was done as follows. With ReadByte the bytes stored into a buffer as described above, and the examinInstr function is called. If the letters matches with AT then the WriteByte is called with OK as definition.

- *Instruction RST<CR>*

This instruction reset the game and sends to PC the answer OK.

- *Instruction SP<SP>{B,W}<CR>*

This instruction sets the players checker to black or white and sends to PC the answer OK.

- *Instruction NG<CR>*

This instruction starts a new game and sends to PC the answer OK.

- *Instruction EG<CR>*

This instruction ends a game and sends to PC the answer OK.

- *Instruction ST<SP>[1-9]<CR>*

This instruction sets the system's timer by default in 2 sec as will described below and sends to PC the answer OK.

- *Instruction MV<SP>{[A-H] , [1-8]}<CR>*

This instruction accept a move finds the cells of chessboard which the opponent want to put his checker and write in this address of memory the color of the player. The finding of the memory address will described below. FInally AVR sends to PC the answer OK.

- *Instruction for myMove*

This instruction writes the AVR's color in the SRAM and sends to the PC MM<SP>{[A-H],[1-8]}<CR>. The [A-H] and [1-8] describes the location of the cell on the chessboard.

- *Instruction PS<CR>*

This instruction sends to AVR pass which means that the PC has no valid move to play so it's AVR's turn and sends to PC the answer OK.

- *Instruction for myPass*

This instruction sends to PC MP<CR> which means that AVR has no valid move to play so it's PC's turn.

- *Instruction for AVR's Win*

This instruction sends to PC WN<CR> which means that the AVR has win the game.

➢ *Instruction for AVR's Lose*

This instruction sends to PC LS<CR> which means that the AVR has lost the game.

➢ *Instruction for Tie*

This instruction sends to PC TE<CR> which means that the game is tie.

➢ *Instruction for Illegal Move*

This instruction sends to PC IL<CR> which means that the opponent has done an illegal move and PC sends back PL<CR> for rejection or OK<CR> for accept the request. An illegal move in which the player has placed a checker not near from another opponent's checker or has placed a checker near of his checker.

➢ *Instruction for Illegal Time*

This instruction sends to PC IT<CR> which means that the opponent has exceeded the time limit. PC sends back PL<CR> for rejection or OK<CR> for accept the request.

➢ *Instruction for Finish the Game with AVR lose*

This instruction sends to PC QT<CR> which means that AVR accepts the lose and ends the game. PC sends back PCOK<CR>.

➢ *Instruction for AVR's win*

This instruction sends to AVR WN<CR> which means that either the opponent quit the game, either the opponent does an invalid move or exceeded the time limit so the winner of the game is AVR. AVR sends back to PC OK<CR>

In this phase of the project not all the instructions are fully operational.

IMPLEMENTATION OF LIGHTING A LED

In this project were currently used the LED0, LED1 and LED2 for the game and the LED5 only for debugging reasons. The assertion of each led represented each final state of the game. For example, the activation of led0 means win and respectively led1 and led2 for lose and draw.

IMPLEMENTATION OF TIMING EVENT

There was a need to monitoring the time which was spending from the opponent in each move with purpose of examining that the opponent does not spend more time than the predefined time limit. For this purpose, the timer1 was setted to fire an interrupt every second. In this phase we didn't have access the crystal oscillator of 10 Mhz, so we set up the timer and the UART for the internal clock source of 8Mhz. The mode of the interrupt was clear timer on compare match. In order to succeed 1 sec period, the prescaler must divide the clock by 256, so in one period the timer has done 31250 steps. The timer value was compared with the immediate 31250 and if were equal then the interrupt was triggered. In

addition with the timer-overflow interrupt there is no need to reassign the 16-bit start value in the timer every time the specific interrupt handler is called. When this interrupt-handler is called a variable was incremented by , which represented the passed time in seconds and then compared with the setted timing limit. For debugging reasons when the time surpassed the timing limit, then the led5 was toggled. This was very useful to examine the functionality of the instruction 'set timer'.

Table 47. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1X	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Figure 2: Mode Bite Description

Timer/Counter1 Control Register B – TCCR1B

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure:3 Timer Control Register B

Table 48. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)

Figure 4: Prescaler Bit Description

CHESSBOARD IMPLEMENTATION IN SRAM

The access in memory has done as following. An 64 table was defined for implement the 8x8 chessboard. Each cell of chessboard corresponds to a memory cell. As mentioned earlier the chessboard has 8 columns numbered from 1 to 8 and 8 lines named from A to H so for the memory table A1 corresponds to Table[0], A2 corresponds to Table[1],....,A8 corresponds to Table[7], B1 corresponds to Table[8], B2 corresponds to Table[9],....,B8 corresponds to Table[15] etc. To calculate the address of the memory corresponds each cell, is known that the first address of SRAM is 0x0060 so to find the position of the letter subtract the number 65 which is the ASCII number for letter A, from the correspond number of the input letter. After this the result of this substruction is shifted left logical by 3 which means that the result multiplied with $2^3 = 8$, and on this results is added the number of the column minus 1. For example if the cell which is going to be written is A5 is known that the correspond memory cell is Table[4]. So with the iterations the Ascii of A is 65 so $i = 65 - 65 = 0, addr = (i << 3) + j - 1 = 0 + 5 - 1 = 4$ so the memory cell is 4. When the address is calculated then in this position of the Table will be written the data. Similarly is implemented and the read function. This implementation of the chessboard in RAM is more expensive in memory but it's easily accessible, cheaper on time and less complexity. An another way to implement the chessboard on the RAM is to correspond 4 cells of chessboard into 1 cell of memory. The upper 4 bits of the memory will represent the color of the cell and the lower 4 bits will represent if each cell is valid or not. This way of implementation is cheaper on memory but it's more complexity and expensive on time. The table of 64 positions is defined for safety reasons because 64 bytes memory is allocated for this table and block of memory can not be overwritten by the compiler instructions.

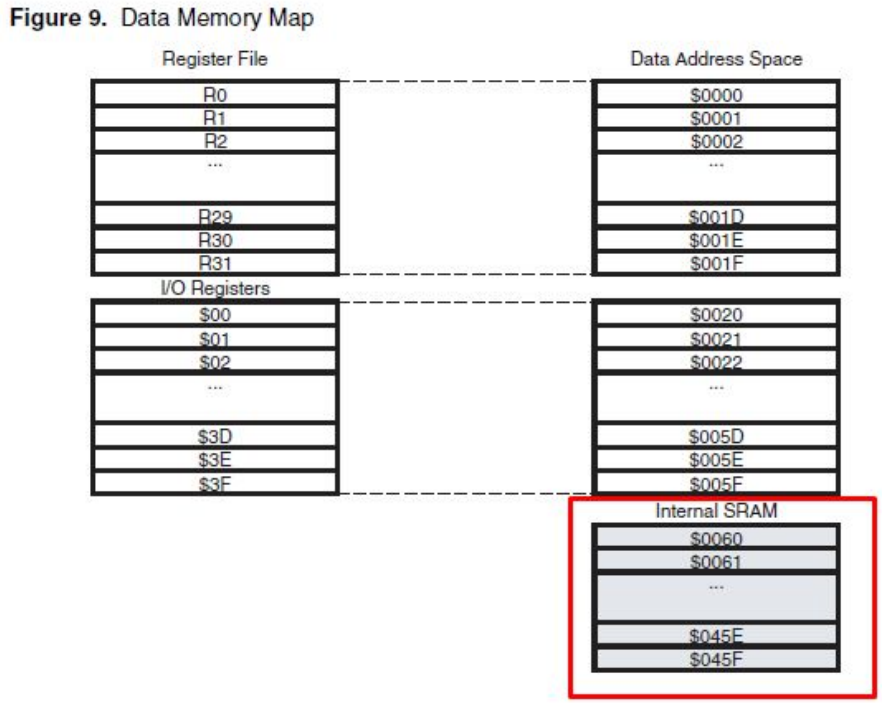

Figure 5: Memory Map

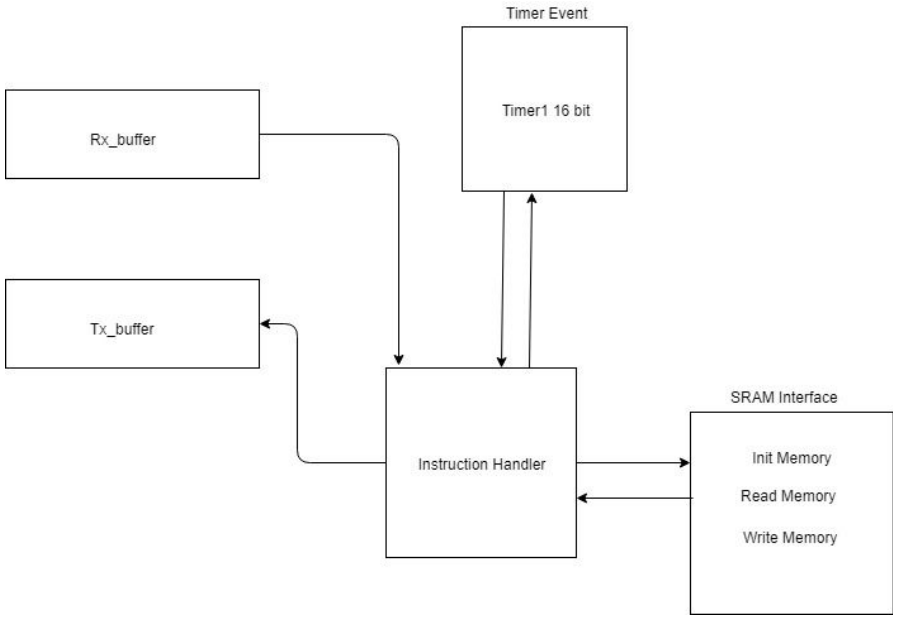

Figure 6: Block DIagram