ECSE 426 - MICROPROCESSOR SYSTEMS

LAB 2 NOTES

MEMORY GAME ON THE MICROCONTROLLER – ASYNCHRONOUS COMMUNICATION INTERFACES

Group 6

Hossam Al-Saati	260169285
Vinod Sridharan	260256868

1. INTRODUCTION

This lab involved designing a memory game using the microprocessor, keypad, and the USART module. The objective of the game was to identify all matching pairs of characters in an 8x8 grid. The grid was initially displayed to the user as a network of stars. The user typed in 2 sets of row and column numbers on the keypad, which displayed the respective characters on the grid. If the characters matched, they remained on the grid, and the user had to identify further pairs. If the pairs did not match, then those characters were replaced with stars, and the user continued to play the game until all the characters were matched.

2. FUNCTIONAL SPECIFICATIONS

Function: void main(void)
Purpose: The main entry point into the game. It contained the finite state machine, which determined the control flow of the game.
Inputs: None
Output: None
Special Cases/Error Conditions:

- Pressing the reset button would restart the main method (and hence the game) from the beginning.

Function: void genValues(char *values)
Purpose: The method fills an array of size 64 (memory pre-allocated) with 32 pairs of characters, sorted in pseudo-random order.
Inputs: *values: a pointer to an array of size 64, to fill in.
Output: The function does not return a value, but the array pointer provided is modified to contain 32 pairs of characters from ASCII 64 to ASCII 95 sorted in random order.
Special Cases/Error Conditions:

- If the array has less than 64 elements, the method fails as it goes out of the bounds of the array.

Function: void printArray(char *values)
Purpose: Prints the contents of the array *values (which is of size 64) through the UART.
Inputs: *values: a pointer to an array of size 64, to print out through the UART.
Output: While the function does not return anything explicit, the contents of the array are printed out on the terminal that is connected to the board, using the UART module.
Special Cases/Error Conditions:

- If the array has less than 64 elements, the method fails as it goes out of the bounds of the array.

Function: void getKey(void)
Purpose: Gets a valid row and column number from the user to display in the array
Inputs: The function does not have an input, but expects inputs from the UART or Keypad. Note that this method uses the keyPress() method to obtain each key from the keypad or the UART.
Output: While the function does not explicitly return a value, it sets the value of a global variable which corresponds to an index in the input array (between 0 and 63) based on the row and column number entered.
Special Cases/Error Conditions:

- If the user enters 00 consecutively, the game exits.
- If the user enters a value other than a number between 0 and 8, or the '#' character, it is ignored by the method
- If the user enters '#', then the last 2 digits entered are checked. If the last two digits are valid (between 1 and 8), then it returns. Otherwise, it waits until the next '#' is entered.

Function: void keyPress(void)
Purpose: Gets a valid character from the keypad or UART.
Inputs: The function does not have an explicit input, but expects an input character from the keypad or UART.
Output: A number between 0 and 11, representing numbers between 0 and 9, 10 for '*', and 11 for '#'.
Special Cases/Error Conditions:

- If the user enters a character other than those on the keypad using the UART, it is ignored by the keyPress method.
- If the user enters multiple characters, the keyPress registers the first key pressed.

Function: void uart_init(void)
Purpose: This function is responsible for setting up and initializing the UART. The initialization will include choosing the clock, setting up the baud rate and enabling interrupts.
Inputs: None
Output: None
Special Cases/Error Conditions: None

Function: void sendStr(char *string)
Purpose: sendStr uses TXBUF0 to send a string through the UART to the Hyper Terminal
Inputs: *string – a pointer to the string that needs to be sent
Output: While the function does not explicitly return a value, it sends the string, character by character, to the terminal
Special Cases/Error Conditions: None

3. IMPLEMENTATION

The main memory game was implemented using a moore finite state machine. There were a total of seven states, representing the various stages of the game. The different states and the actions inside each state are highlighted below. Furthermore, the state transitions are shown in the state transition diagram (Figure1). Note that the input array here refers to the array of characters arranged in random order, while the output array refers to the array of characters that the user can see.

FSM STATE	**Purpose and tasks**
INIT	• Initialize the game time, the UART and TIMER A • Initialize the input character array, and the output character array (*s) • Print the array of *s on the screen to begin the game
KEY1	• Ask the user to enter a key • Call the getKey() method to return an array index to access, and store it as 'key1'.
KEY2	• Ask the user to enter a key • Call the getKey() method to return an array index to access, and store it as 'key2'; Display that character along with key1 on the output array
TSTEQ	• Check if the input values at key1 and key2 are equal, and key1 and key2 are distinct indices • If the above condition is true, set a flag that makes the state transition to REPL to replace the values on the output array; otherwise set a flag that makes the state transition to CHKTOT. • Wait for 1 second, and inform the user that there has been a match or mismatch in their selection

REPL	• Replace the character at the index key1 in the output array with the input value at key1, and replace the character at the index key2 in the output array with the input value at key2. Increment the number of characters found by 2.
CHKTOT	• Check if all 64 characters have been found. If yes, set the FSM state to DONE. If not, then go back to state KEY1
DONE	• Output the total game time, congratulate the user, and enter Low Power Mode.

Table 1: Finite State Machine: State actions

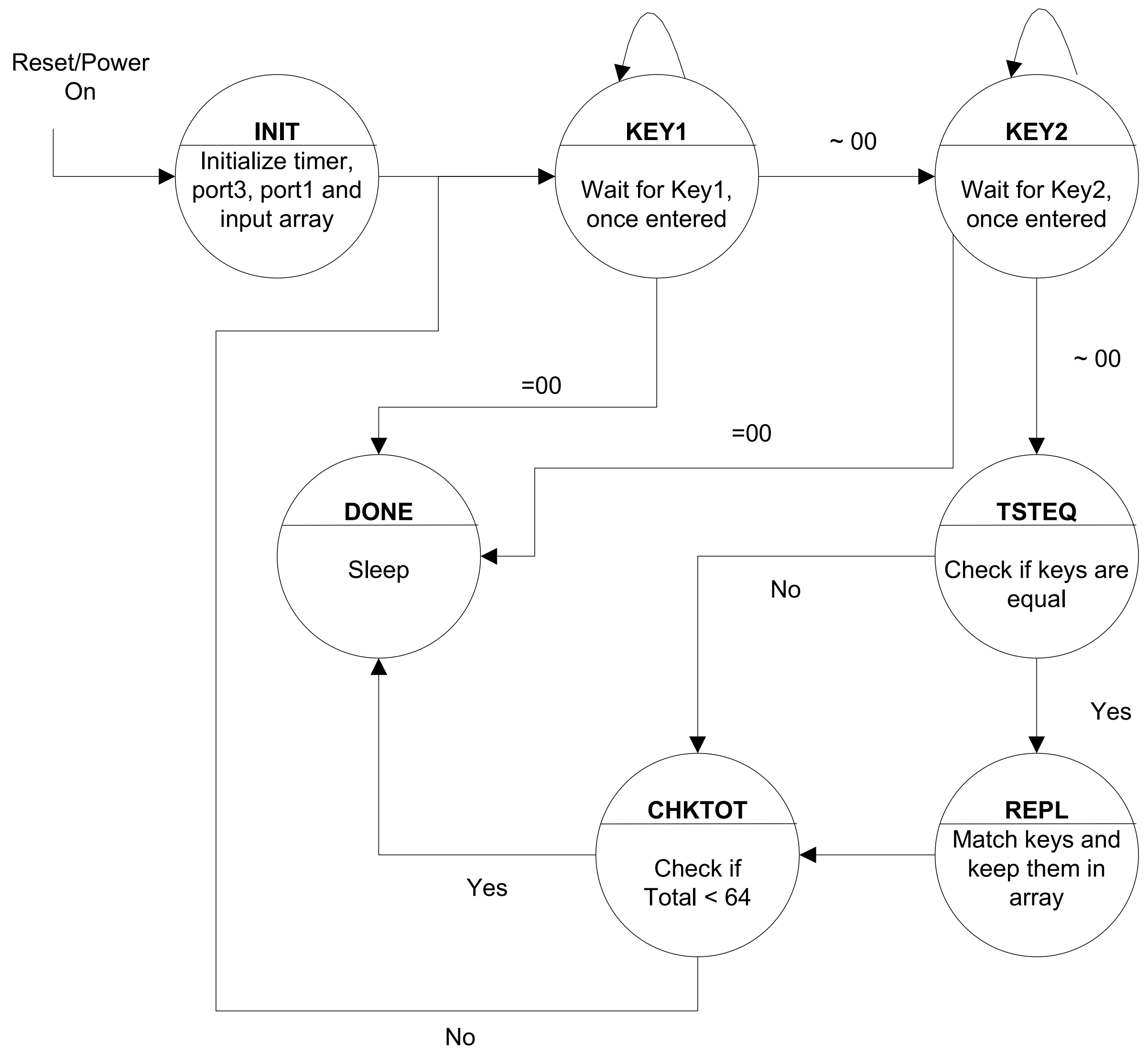
Reset/Power On
INIT
Initialize timer, port3, port1 and input array
KEY1
Wait for Key1, once entered
~ 00
KEY2
Wait for Key2, once entered
=00
=00
~ 00
DONE
Sleep
TSTEQ
Check if keys are equal
No
Yes
CHKTOT
Check if Total < 64
REPL
Match keys and keep them in array
Yes
No

Figure 1: Finite State Machine - Flowchart

The Finite State Machine made frequent calls to the getKey() method to obtain the index of the array to consider. The program flow for the getKey() method is highlighted below.

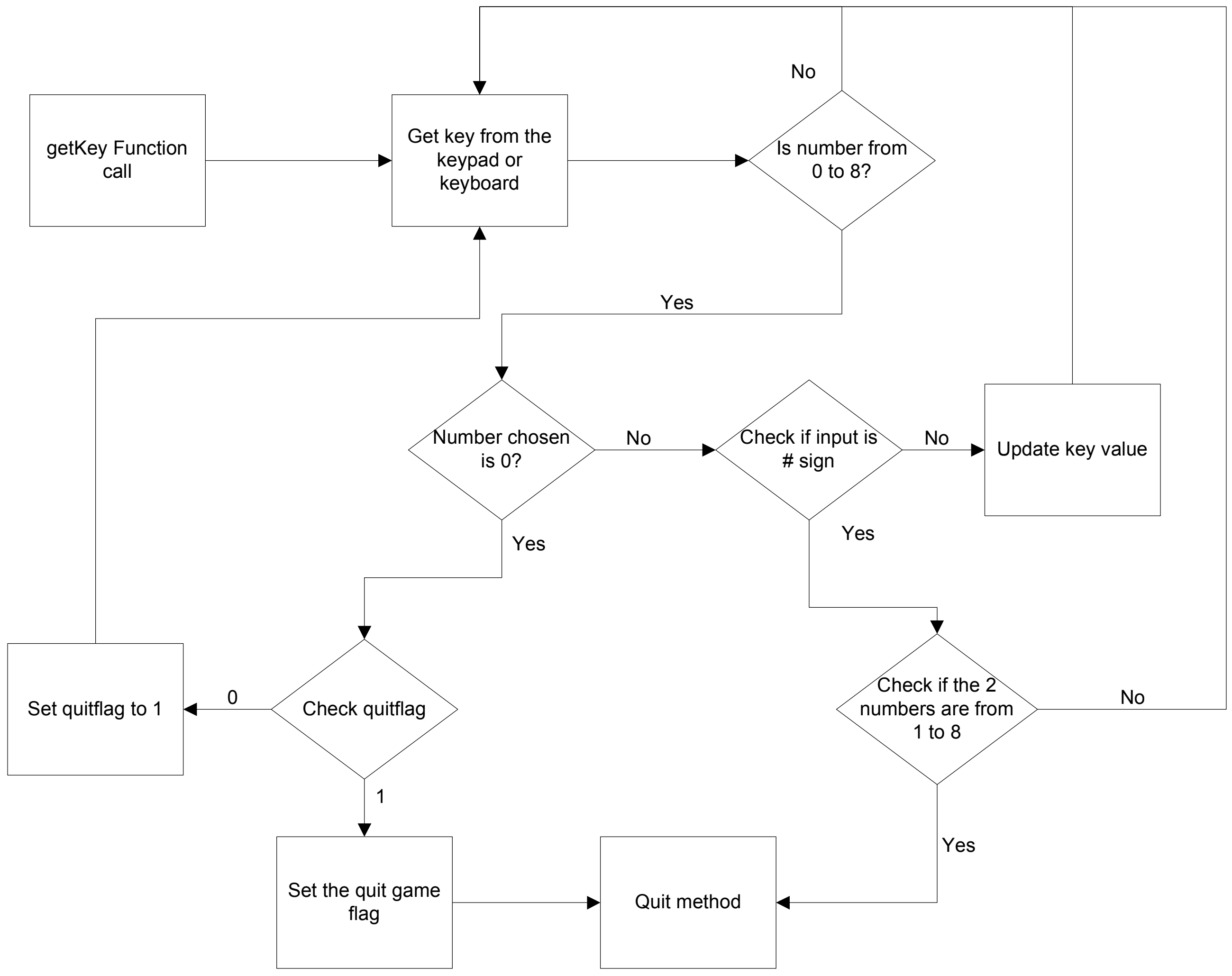

Figure 2: Program flow of getKey()

The UART communication consisted of an interrupt driven system to transfer inputs to the terminal. It consisted of a main screen that displayed the start of a game, and allowed users to input a row and column to display the letter in the coordinates chosen. The UART implementation accepted communication through the terminal keyboard. In this lab, we chose to use a baud rate of 115200 bits/sec and used 8 data bits, 1 stop bit, no parity and no flow control to configure UART mode. The figure below briefly explains the interrupt methodology used for the UART communication.

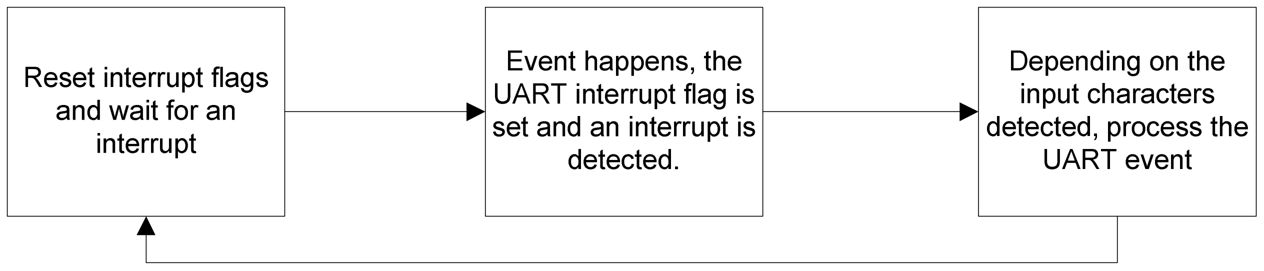
Reset interrupt flags and wait for an interrupt
Event happens, the UART interrupt flag is set and an interrupt is detected.
Depending on the input characters detected, process the UART event

Figure 3: UART Implementation Diagram

One of the functionalities of the game required an accurate track of the game time. The INIT stage of the FSM configured TIMER A to enable interrupts, and with interrupts triggered every second. Consequently, during the game, the Timer A ISR was called, which incremented a variable every second. Due to this, at the end of the game, this variable contained the number of seconds that had elapsed since the game started.

4. PERFORMANCE ANALYSIS

The testing performed in this lab included testing the FSM/Game and testing inputs from the keypad and the UART. Validating the game functionality included several test cases to check if the inputs entered return the desired output. Below is a table listing a few test cases that were used for the purpose specified above.

Input	**Output**	**Test Case Purpose**	**Pass or Fail**
123456#	56	The game should always take the last 2 inputs entered by a user	Pass
29#1#	21	9 is not a valid input	Pass
060	-	Exiting the game will require 2 consecutive 0's	Pass
40#04#	44	The game should always take the last 2 inputs entered by a user, even if keys in the middle include invalid characters	Pass
1*4#	14	* is not a valid character	Pass

Table 2: Inputs test cases for the game

Testing the keypad required finding out the digit-error-rate (the number of times the input from the keypad was erroneous). Six digits from different rows and columns were picked at random and were pressed 25 times consecutively each, and the number of errors in the output were counted. The table below summarizes the error rates that we found.

Min error rate (number of keys failed)	**Mean error rate (number of keys failed)**	**Max error rate (number of keys failed)**	**Average percent error (%)**
0	1	3	4

Table 3: Results of digit-error-rate test

We also verified the integrity of the characters received by the UART/Keyboard. This included sending patterns of characters that were not recognized by the getKey() method. This involved sending streams of characters which were not included in the character sets accepted by the getKey() method ([0- 9 | # | *]). The following table summarizes the text streams sent via the keyboard and the keypad, and the output we received on the terminal for each input:

Input sequence on UART/Keypad	**Output sequence on terminal**	**Output**
T4#w*091#	4#*091	41
999999T405#	999999405	45
#######JKLMN00#	#######00	QUIT
Gkdlskdkfkennal	-	-
12t#	12	12
92#	92	- (waiting for second key)

Table 4: invalid character-test results

Finally, we had to test the case where we had several keys pressed at once. The keyPress() method cut off the current to all other rows than the one on which the key was recognized. Consequently, we only needed to worry about several keys being pressed on the row of the key that was considered. Therefore, over 10 test runs, we pressed multiple keys on the same row. We then let go of the keys one by one and checked the output on the terminal. In all cases, the wait-for-release sequence made sure that we never got more than one output at a time.

4. REFERENCES

[1] 304-426A Microprocessor Systems Fall 2009 Experiment 2: Memory game on the microcontroller – Asynchronous Communication interfaces, McGill University, Fall 2009
[2] Texas Instruments, *MSP430x1xx Family User's Guide*, Mixed Signal Products, 2006
[3] Texas Instruments, *Implementing an Ultralow-Power Keypad Interface With the MSP430*, Mike Mitchell, 2002