### ECSE-426

### Lab NOTES – LAB 2

*Simon Foucher (260 223 197), Alexandru Susma (260 235 940)*

Lab Group Number 07

**1. FuNCTIONAL SPECIFICATIONS**

**1.A – GAME LOGIC**

The game follows a simple finite state machine flow. State transition is generated either by flags or interrupts. The game takes its inputs from the amazing ‘home made’ keyboard provided in the MicroP kit. It also uses both the 8M clock and the real time 32K clock embedded in the McGump board. The output of the game is sent out to the UART interface which converts it to a serial bit signal sent out to the RS232 port.

Notes on I/O: Since P3.4 and 3.5 are hard wired to the RS232 port Tx and Rx pins, we didn’t use them for the keyboard. Similarly for pin 1.0, which is wired to the board’s LED.

**1.B – OUTPUT: UART**

For this experiment we chose to use the UART0 interface to pass characters from the board to the terminal via the RS232 port. The characters to be sent are written in the transmit register and an interrupt is generated while the character transfer is being accomplished. The UART keeps shifting bits out of the UART buffer into pin 3.4, which is hard wired to the RS232 Tx pin, connected to the computer’s Terminal interface.

**1.C – INPUT: KEYPAD INTERFACE**

* Keyboard Input Pins (High logic): P6 (We avoided pin 3 because of its connection with the RS232 port, and avoided pin2 since it is Interrupt Enabled and this function does not require interrupts. This would facilitate ‘recycling’ this design by adding interrupt driven features without having to change pinnout)
* Keyboard Output Pins (Scanned during Polling): P1.1, 1.2, 1.3 (Chosen because they are interrupt driven, we avoided pin 1.0 because it is wired to the board’s LED)

The keypad is used in this experiment to interface with the game menu and to play the game. The keypad is 3x4 with 3 columns and 4 rows. On the back of the keypad there are 8 pins: 1 ground, 3 columns and 4 rows. Interrupts are triggered whenever a button is pressed. The ISR calls Timer A for debouncing (more details in ‘Implementation’), then converts the ‘button pressed’ into an integer value. We chose integers representing the printout of the keyboard (i.e. button ‘1’ is represented by integer 0x01), 10 for ‘\*’ and 11 for ‘#’.

**2. IMPLEMENTATION**

First we coded the game in C with the computer’s keyboard as input and the ‘printf’ buffer as an output. It was designed with the requirements in mind, using the system time instead of actual timers. In parallel, we developed the UART interface and tested it by printing various things on the screen, then modified the game by replacing ‘printf’ by ‘send to UART’. Similarly, we designed and troubleshooted the keyboard, then implemented it in the game by replacing ‘getchar()’ with ‘read from the keyboard’. (Since the getchar function had the proprety to ‘pause’ the game while waiting for a user input, we created a flag called ‘KEYPRESSED’, which gets set in the keyboard’s ISR. We could the ‘pause’ the game while waiting for a keyboard input by using a while(!KEYPRESSED) loop)

2.1 - GAME LOGIC

The game was implemented as a finite state machine using a combination of while loops to keep the sequential program ‘frozen’ in a state.

Few design decision:

Random seed: Once the game is launched if the user asked for a random game, we grab the value of timer B as a random seed (32k possibilities!). If the user asked for a rigged game, we just run the random function without seeding it. This results in a ‘random’ spread for the first game, but identical distributions for every subsequent games.

What is to be randomized: Instead of running through every location sequentially and selecting a random element, we selected elements sequentially then randomly found a location. For performance, if the random location was already taken, the game tries to put the element at the next available row position. If the row runs out, it shifts to the next column. When the columns run out, the position is sent to (0,0) and the scan restarts sequentially until we find an empty location. We chose this approach instead of continuously choosing a random location until an empty spot was found in order to guarantee us to find a spot in at most O(size^2) time.

Cheating: To facilitate ‘cheating’ for debugging and for the TA, as the elements are inserted the element as well as both locations where it can be found are printed on the screen.

Seizability: The size of the board is controlled by a single variable ‘SIZE’ defined at the top of main.c.

Keeping track of game: The game goes on for as long as there are elements left uncovered. A flag is set to SIZE2­ as the game starts and is decreased by a factor of 2 whenever 2 elements are uncovered. The elements are stored in a 2D array of characters: ‘board[SIZE][SIZE+1]’ (the +1 added to account for the terminating null character in a string. Even though we are only using chars in this exercise, we have the capacity to store actual strings). A second array is used: ‘showC[SIZE][SIZE]’, which contains a flag indicating whether the element should be displayed (1) or hidden (0).

Game sequence:

1. Initialize UART, timer B and Keyboard control registers
2. Print a menu then wait for user input (random game, rigged game or exit)
3. Function initialize is called:
	1. Cleat the screen
	2. Format the game by writing ‘\0’ as every board elements and 1 as every showC locations (set to 1 to display the board to the user)
	3. If user selected random game, randomize the rand() seed
	4. Populate the board (see above comment for procedure)
	5. Show the user the board, then wait a few seconds
	6. Set all the showC flags to 0, clear screen the print the board (all ‘\*’ at this point)
	7. Return to main
4. Function playGame is called:
	1. Freeze in play game state using while(stuffleft) flag, which counts how many items are left to uncover
	2. Call function formatkeys: resets the data in key input buffers (KEYS[2] and KEY)
	3. Wait for user press by locking into a while(1) loop.
		1. As the user presses keys, the new key is recorded in the buffer KEY.
		2. Then KEYS[0] <- KEYS[1], and KEYS[1] <- KEY, such that the game records only the last 2 keys pressed by the user.
		3. If the user enters a key larger than the max row/column, issue a warning.
		4. If the user presses ‘#’, exit while loop (as long as user is not trying to uncover an element already marked as uncovered)
		5. If the user presses ‘\*’, the game exits
5. Set the flag showC for the combination entered as 1 and launch a printboard function
6. Format keys and repeat procedure for second element
7. If both elements are identical, return to top of while(stuffLeft) loop
8. Otherwise, wait a few seconds, reset showC for both elements, clearscreen and printboard
9. When all the elements are uncovered, exit loop, print game time and exit funtion

2.2 O/I

2.2.B – UART

UART0 was used to pass information from the board to the terminal. In this experiment we had to use a 57600 baud rate, 8 data bits, 1 stop bit, no parity check and no flow control. To set these values we set the respective registers from the UART component. To generate the baud rate we used the 8MHz clock and used a divider of 138.8 (0x8B). Port3 is used to pass the information from the UART interface to the RS232 Rx/Tx pins.

We designed 4 output functions, all using the UART.

* printChar: Writes an ASCII character into the UART register and ‘pauses’ in a while loop until all the data has been sent serially.
* printString: Receives a string and breaks it up into characters, which are fed to printChar() setially
* printInt: Receives an integer and breaks it up indo digits. The digits are then incremented by 48 (Since ASCII ‘0’ = 48 in decimal) then fed sequentially as ASCII characters into the printChar function
* printBoard: scans all the elements in board[i][j] as well as showC[i][j]. The latter conatains a logical flag (1 or 0) that keeps track of weather an element should be shown or not (set when should be shown to simplify coding). If(showC[i][j]), board[i][j] is sent to printChar. Otherwise, the function sends out a ‘\*’ character.

2.2.C - KEYPAD

To complete this component we used the schematics provided. Three 4.7Kohm resistors were used as a link from the column pins to the ground. We wired all columns to port1 to make sure we get the interrupts when a button is pressed. After that, we connected the rows to port6 to be able to drive them high. The idea is that the rows will always be high and when a button will be pressed the circuit will close and that specific column will generate an interrupt. Using timerA we made the program wait for debounce purposes. The exact value of Timer A was determined by physical trial and error with the keyboard.

For this part of the code we used 2 interrupt service routines. The first one waits on a key press and it starts timer A once it receives an interrupt. The second interrupt service routine waits on timerA to reach a set value then it starts scanning for the key pressed.

Here is the flow of events:

1. All out pins are high, all in pins are listening.
2. Button pressed; interrupt is detected in one of the 4 in pins
3. Pin 1 ISR: reset interrupt flag and fire up Timer A, then return to game
4. Wait for timer A interrupt […] ms so nothing happens
5. Timer A ISR:
	1. Set the logic flag KEYPRESSED
	2. Turn off all out pins
	3. Turn them back on, sequentially until all 3 have been turned on individually and listen to input pins for signal. The system will detect the occurrence of 2 or more signals (if any) and report an error
	4. Convert the signals into the corresponding value.
	5. Reset interrupt flags and exit

2.3 – GAME TIME

Game time is kept using the 32K real time clock on timer B, which is initialized when the game starts. A global variable called ‘TIME’ is set to 0 when the user decides on the game and the game start. Every second, timer B interrupts and ‘TIME’ is incremented by 1 to keep track of the elapsed seconds.

**3. Performance Analysis**

In this experiment we had to test each component individually as well as the whole system at the end. First the game logic was tested with different values input from the terminal keyboard. After that we tested the UART by manually overwriting the receive register and waiting on it to generate the interrupt and transmit the data. The keypad had to be tested both from a software point of view (code) and from hardware point of view (wires and connections). Many times we realized the wires were not connected or they were shorting each other. The oscilloscope was a very handy tool that we used to determine if signals were correct on wires and pins.

After the whole system was assembled we did a lot of testing. This proved to be a very good strategy since we found a lot of errors in logic and design that we were able to spot before the demo. The final program was tested using both random generated boards as well as rigged boards. Different cheating techniques were tried and as some of them bypassed the system, we had to go back and fix the problems.

**RefERENCES**

-msp430 header file

-keypad application note posted on webct (assembly code)

-msp430 user guide

-msp430x14x.h file: used to ‘look’ at values associated with acronym flags