ECSE426 Microprocessor Systems Fall 2009

Experiment 2: Memory game on the microcontroller – Asynchronous Communication interfaces

Demos: Oct. 1, 2

Lab Notes: Oct. 4

1 Objectives

This exercise will familiarize you with the Universal Synchronous/Asynchronous Receiver/Transmitter (USART) peripheral interface and interfacing with a simple external input device (keypad). Use of timers will be needed to provide accurate timing of the game.

2 Background

Before you start coding, please review the appropriate information from the MSP430x1xx Family Guide (focus on Chapter 4, Chapter 11 and Chapter 13).

The software program may be implemented in MSP430 assembly code, in C language or a combination of both languages. You will be required to justify your design language choice.

USART

The USART peripheral interface supports two serial modes with one hardware module. These modes are the asynchronous (UART) mode (see Chapt. 13 of the MSP430 user guide), and the synchronous (SPI) mode (see Chap. 14 of the MSP430 user guide). In this lab, you will make use of the asynchronous mode. In asynchronous mode, the UART connects the MSP430 to an external system via two external pins, URXD and UTXD. The UART mode allows the transmission of 7 or 8 bit data (with parity checking); it provides separate transmit and receive buffer registers, and independent interrupt capability for receive and transmit. See the attached document for more background on UART.

Timer

Chapter 11 should be your starting point where you will find necessary information about the timer and how to access its features. The MSP430F149 has 2 timer modules. In this lab, you will use TimerA. You will not need to use all its features, only the mode capable of generating an interrupt at specific intervals. The timer will be needed to keep the timing of the game. The clock source for the TimerA module should come from the low speed (32.768 kHz) oscillator.

Design Notes

For anything that is not explicitly specified, you should decide the best way this system should behave to enhance its robustness. As in many design applications, there is not necessarily one right answers. You are given broad specifications and you should work out the details to produce an elegant solution to the given problem. Since there is a substantial amount of work involved in this lab, estimate the workload, divide the work, and pace yourself carefully. Please ask questions early in the design process, as you will find that it is much less stressful to change your system's architecture while you are still in the design phase than the day before the demo…

3 Task

Part 1: Use the serial communication with the terminal via the UART interface to implement a simple memory game. A grid of 8x8 cells should be displayed with an initial configuration of '*'. Then, using the keypad, the player will enter a row and column to display the letter behind that cell. The player tries to reveal matching pairs. Once two cells have been displayed for 0.5s, if they are not identical, the grid is printed again with '*'. If the two cells are identical, the revealed pairs are left displayed on the grid. The game finished when all the cells have been paired up. At this point, the elapsed time for the game must be displayed.

Functional Requirements:

(1) The game must be usable. The interface should be made such that its easy to use and intuitive.

(2) The game time must be accurately tracked. This means that the timers must be kept running during the game duration. The overall accuracy for a complete game should be 1 second. At the end of the game, the total time for the game should be displayed.

(3) The coordinates in the game must be entered using an external keypad. The terminal keyboard should also be capable of sending coordinates to the game. The coordinates are sent by typing the Row, then the Column, then the # key. For example, typing 34# on the keyboard or keypad should reveal cell at Row=3, Column=4. If one makes a mistake keying the entry, they can re-type the coordinates. Only the last 2 key presses are kept and the # key is used to accept the value. The keypad must be interrupt driven (i.e. use the interrupt features of the I/O module).

(4) The serial link should operate at 57600 bauds, no parity, 8 data bits and one stop bit.

(5) The demo should not depend on the presence of the Rowley tool. Your board should have only the serial line, keypad and AC adapter present during the demo (no parallel port connection). Basically, it should be plug and play.

4 Demonstration

The demonstration involves showing your completed source code and a working system conforming to the aforementioned specifications. Grading will be based on how well the system satisfies the specifications, robustness, design elegance, and how sensible and well-considered the design decisions were. There will also be user-interface assessment. Please also prepare a program flow-chart and system schematics to aid explanations during the meetings and demonstration sessions. Make sure to record and have available all your performance test results !

5 Lab Notes

You do not need to hand in a report for this lab, but you do need to submit lab notes. These can be handwritten (but legible) or typed, as you prefer. These lab notes should consist of 4 sections: (i) function specifications, (ii) a flow diagram, (iii) methodology/implementation notes, (iv) performance testing method and results. Each of these sections should be approximately a page; you should be able to write the notes during lab hours (and they will be marked as though this is the case).

However, make sure you express yourself concisely and clearly, and keep the presentation reasonably neat. For example, flow diagrams should still be reasonably professional – not freehand sketches.

Appendix

Design tips and considerations

* The game should ideally start with a random array of letter pairs. You are free to use the random generator the way you want, but an easy way is to simply sample the value in a free running counter by asking the user to type 2 keys with a delay in between. If the counter is running fast and the user is taking some time between key presses, the delay will be random. You should have a way to overwrite this random "seed" with a fixed value such that you can test your game with the same grid multiple times.

* Those routines from stdlib.h are likely to be useful in this experiment:

```
int rand(void);  // Returns a random number when called
void srand(unsigned int seed); // Seed the random number generator
```

* An efficient method to prepare an array with known elements, but in a random order is to first fill the array with all the valid elements in order. Then, randomly select two elements in the array and perform a permutation. Repeat a few times and the array of elements will be "shuffled". A function call similar to this will be quite handy:

```
rnd_element = rand() % NUM_ELEMENTS;  // % is the C modulo operator
```

* You can develop the game in 'C' on your PC. You simply have to make a console project and the input/output using only getchar() and putchar(). Then you could write a putchar() and getchar() routine on the MSP430 which have the same functionality. The putchar() and getchar() routines on the MSP430 have to use the UART (i.e. this is not debug_printf() or its equivalent). You may use debug_print() at some point to help you design, but in the demo, you have to removed all those calls.

* An application note will be posted to detail how to interface with a hardware keypad. Its suggested that all the game logic be debugged and working before integrating the keypad inputs. This way, you can separate the routines and keep the complexity under control.

* The program can be built around a constant defining the number of element in the grid. Debugging your program with a smaller (e.g. 4x4) grid is much more efficient (the game should be a lot easier too...). You only need to recompile and test later for the final 8x8 grid once the code is stable.

* ANSI control sequences can make screen erasing and cursor control a lot easier and allow nicer user interfaces. From the UART point of view, its just a regular sequence of characters sent. However, when received by the terminal program, they affect the way the cursor moves or modify the text color. For example, sending "<ESC>[2J" (in C : "\x1B[2J") will clear the screen. More details on ANSI sequences can be found at:

http://www.termsys.demon.co.uk/vtansi.htm

Notes on USART module (by Milos Prokic)

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART reassembles the bits into complete bytes.

There are two primary forms of serial transmission: Synchronous and Asynchronous. MSP430 that McGumps features supports both Synchronous and Asynchronous, hence USART. In this lab you will be using the Asynchronous mode – Chapter 13 of the UG.

There are two USART – 0 and 1. I suggest you use UART0 that connects to the Dsub9 connector on the McGumps board via the MAX3223 chip. The chip is used to convert MSP430 voltage levels to RS232 compatible. If you decide to use UART1 you'll have to connect the board via the 3 pin header located next to the Dsub9 connector – J11.

Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word which are used to synchronize the sending and receiving units. UxCTL register in MSP430 is your friend for this lab ☺. You should make sure that all the bits in this register are properly set.

Any baud rate is acceptable, as long as it is above**(not equal)** to 19200 bits/sec and below 115200 bits/sec. For this lab I suggest **No Parity, Data – 8 bits, 1 stop bit, and no flow control** (CTS, DTS, DTR etc signals are not present so hardware control is not possible…).

Read about UxTCTL and how to set which of the three clock sources will be feeding the UART module. Once that is done, make sure you divide it properly to achieve required bits/sec by writing to UxBR0 and UxBR1. Details about setting the proper values can be found on pages 272-273 of the UG.

I suggest you take a look at the TI FET examples for MSP430x14x provided within the

Crossworks IDE. In particular USART0 – UART 19200 Echo HF XTAL…

As you can see from the example receive is handled by the ISR. This is the desirable way to do it because you shouldn't loop and wait on the receive flag thus stalling the microcontroller and preventing it from doing useful work or going into low power mode. The receive characters come from the user terminal producing huge delays in terms of the micro controller.

On the other hand transmit can easily done without the ISR, by looping on a flag, why is that? How do Transmit and Receive interrupts differ?

The MSP430 has two registers that are used for reception and transmission – UxRXBUF and UxTXBUF. These are one byte registers that you can read the data from (RXBUF) or just write to TXBUF what you want to be transferred. You do not have to send out bit by bit – it is a hardware feature. Data from the or to the TXBUF and RXBUF respectively are shifted out or in by shift registers.

To connect to McGumps you can use any of the following terminal programs:

Built-in Crossworks terminal emulator ==> View->Terminal Emulator

TinyTerm

HyperTerminal

Read below for some extra info regarding the UART (you should be familiar with this from ICE2);

When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter. These two clocks must be accurate enough to not have the frequency drift by more than 10% during the transmission of the remaining bits in the word. (This requirement was set in the days of mechanical teleprinters and is easily met by modern electronic equipment.)

After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver "looks" at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a 1 or a 0. For example, if it takes two seconds to send each bit, the receiver will examine the signal to determine if it is a 1 or a 0 after one second has passed, then it will wait two seconds and then examine the value of the next bit, and so on.

The sender does not know when the receiver has "looked" at the value of the bit. The sender only knows when the clock says to begin transmitting the next bit of the word.

When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity Bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter.

When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver looks for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. The usual cause of a Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted.

Regardless of whether the data was received correctly or not, the UART automatically discards the Start, Parity and Stop bits. If the sender and receiver are configured identically, these bits are not passed to the host.

If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent.

Because asynchronous data is "self synchronizing", if there is no data to transmit, the transmission line can be idle.