Lab 9. Distributed Data Acquisition System

Preparation

Read Sections 8.2, and 11.4 on the UART

Read Sections 11.1, 11.2, and 11.3 on communication systems and the FIFOs

Download these examples FIFO_xxx.zip, UART_xxx.zip, and UARTInts_xxx

(*****WARNING: YOU ARE RESPONSIBLE FOR EVERY LINE OF CODE IN YOUR SOLUTION except for the LCD software in ST7735.c/Nokia5110.c and any submodules the software in ST7735.c/Nokia5110.c calls *****).

You will copy your **Lab 8** solution into the **Lab 9** folder.

Teams

To fully test your system you will need two boards. However, you can perform initial testing by connecting the transmitter output to the receiver input on one board. We suggest you find another group with whom you will finish testing your system. You will not be checking out with the same team as a team with whom you debug. Both groups will have to design, build and test separate systems. That is you *will not* split the lab into two parts with one team working on the Sender and other working on the Receiver. Each team is responsible for their own system. At the time of checkout, the TA will select another group with whom you will be checked out. It is also possible for there to be a TA board with which you will check out. **During checkout your system must operate as both a transmitter and a receiver.** You cannot share code with other EE319K teams past, present, or future.

Purpose

The objectives of this lab are to:

1. learn how the UART works using both busy-wait and interrupt synchronization;
2. employ a FIFO to buffer data;
3. study synchronization issues in a distributed data acquisition system. This lab is demonstrated on two TM4C123 boards. There is no simulation in Lab 9.

System Requirements

You will extend the system from Lab 8 to implement a distributed system as can be seen in Figure 9.1. In particular, each TM4C123 will sample the data at 40 Hz, transmit the data to the other system, and the other system will display the results on its LCD. Basically the hardware/software components from Lab 8 will be divided and distributed across two TM4C123 microcontrollers. Communication will be full duplex, so the slide potentiometer position will be displayed to the other system.

Therefore the full system will:

- use two (2) TM4C123 microcontrollers
- sample the slide potentiometer using one of the microcontrollers
- display the position in centimeters on the other microcontroller
- use UART to communicate between the microcontrollers
- each of the two microcontrollers must be capable of sampling and sending ADC data, and displaying the results from the other microcontroller

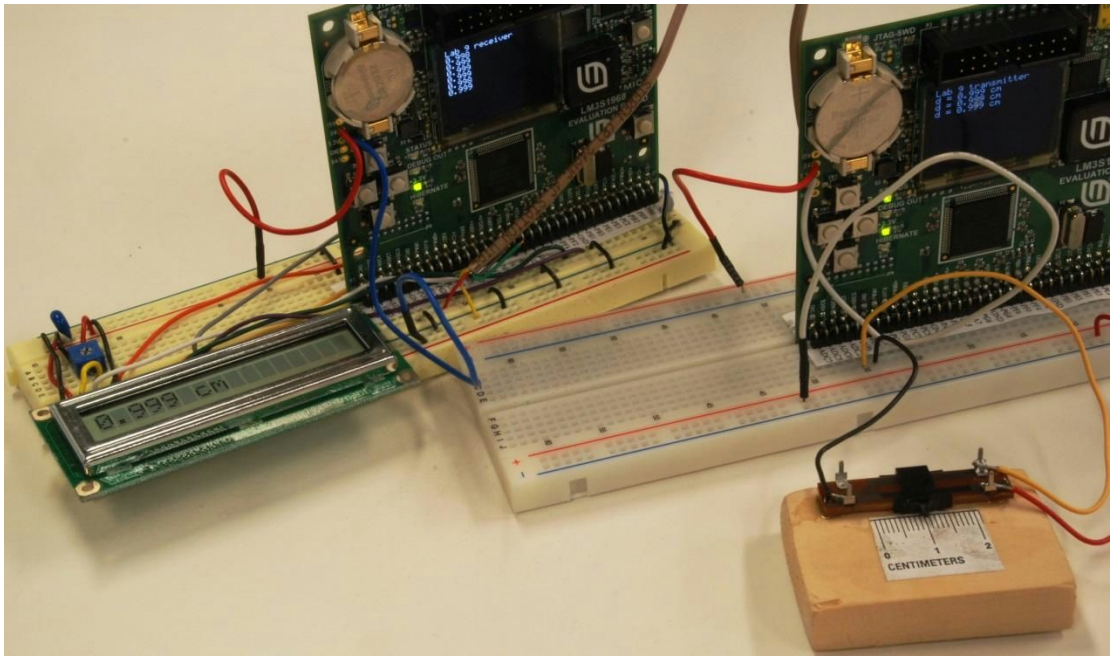

*Figure 9.1. The LM3S1968 on the right measures position and the one on the left displays the results. Both systems show the optional debugging monitors on the OLEDs. (***needs a new photo, but the basic idea is the same***)*

Procedure

The procedure below describes the process as if you are using UART1, but using UART1 is not required. The procedure would not significantly change if you decide to use one of the other UARTs.

Both teams of students are expected to participate in the development of the receiving and transmitting code bases, and may be asked questions regarding the design and implementation of both embedded systems. Further, you will turn in the lab with a different team than you tested with.

Overview of the Full System

As the distributed system will be run across different teams of students with different programs, to ensure that all of the teams can communicate, the protocol of the UART message is fixed. The format of the UART message is displayed below, where each message is exactly eight (8) serial frames at 100,000 bits/sec.

Start (0x02)	Digit 1	'.' (0x2E)	Decimal 1	Decimal 2	Decimal 3	'\r' (0x0D)	End (0x03)

For example if the distance is 1.234 cm:

0x02	'1' (0x31)	'.' (0x2E)	'2' (0x32)	'3' (0x33)	'4' (0x34)	'\r' (0x0D)	0x03

To help you understand the distributed system you are about to design and implement, we have provided an example data flow graph, Figure 9.2, and an example call graph, Figure 9.3. Your embedded systems may look very similar to the examples provided below.

As can be seen from the data flow graph in Figure 9.2, the transmitting and receiving will be done on the background, with an interrupt, and on the foreground, respectively. Data flows from the sensor on one computer to the LCD on the other computer. SysTick interrupts are used to trigger the real-time ADC sampling. You will use a 3-wire serial cable to connect the two UART ports. The full scale range of distance is exactly the same as Lab 8 and depends on the physical size of the slide potentiometer. Shown here as UART1, but any available UART can be used.

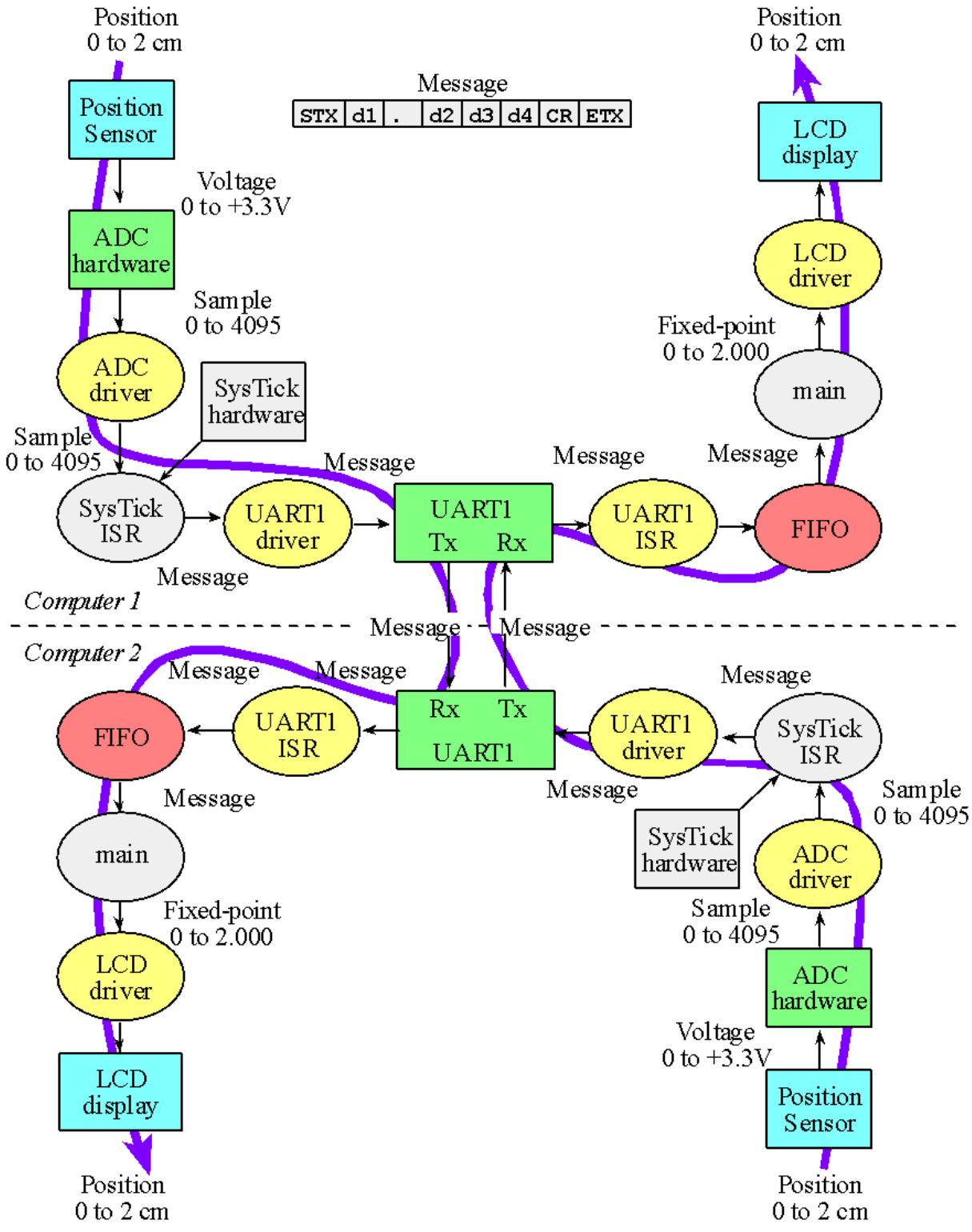

Figure 9.2. An example data flow graph of a completed system.

As you can see in the call graph in Figure 9.3, the transmitter and receiver are implemented on the same microcontroller. Further, the UART on the transmission will be busy-wait synchronized and the receiving will be interrupt synchronized.

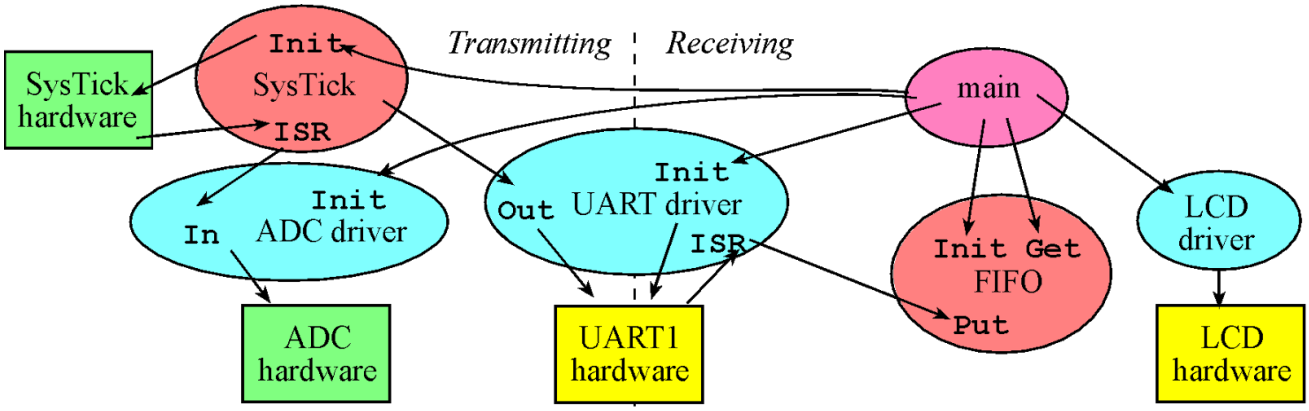

Figure 9.3. A call graph showing the modules used by the distributed data acquisition system.

Transmitter Tasks

All tasks listed under this section should be completed on both microcontrollers. Again, both teams should participate in the transmitter software tasks, and both teams are expected to understand the software produced by the transmitter tasks.

Part a - Initialization

You must select a pair of UART pins to use for the transmitter and receiver. Several pin assignments are listed below:

UART1	PC4, PC5
UART2	PD6, PD7 (PD7 requires unlocking)
UART3	PC5, PC6
UART4	PC4, PC5
UART7	PE0, PE1

Write a C function: **`UART1_Init`** that will initialize your UART1 transmitter. This will be similar to book Program 8.1, but with UART1 and with 100,000 bps baud rate. Busy-wait synchronization will be used in the transmitter. The steps for this function are listed below. Place all the UART1 routines in a separate UART1.s or UART1.c file.

1. enable UART1 transmitter without interrupts
2. set the baud rate

Part b - Send One Byte

Write a C function: **`UART1_OutChar`** for the transmitter that sends one byte using busy-wait synchronization. Place all the UART1 routines in a separate UART1.s or UART1.c file.

1. Wait for TXFF in `UART1_FR_R` to be 0

2. Write a byte to `UART1_DR_R`

Part c - Send Measurements Periodically

Create a simple array to store the 8 bytes needed for the message. Modify the SysTick interrupt handler from Lab 8 so that it samples the ADC at 40 Hz and sends the 8-frame message to the other computer using UART1. The SysTick interrupt service routine performs these tasks:

1. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
2. sample the ADC
3. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
4. convert to distance and create the 8-byte message
5. send the 8-byte message to the other computer (calls `UART1_OutChar` 8 times)
6. increment a `TxCounter`, used as debugging monitor of the number of ADC samples collected
7. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
8. return from interrupt

The format of the UART message is reproduced below, where each message is exactly eight (8) serial frames at 100,000 bits/sec.

Start (0x02)	Digit 1	'.' (0x2E)	Decimal 1	Decimal 2	Decimal 3	'\r' (0x0D)	End (0x03)

For example if the distance is 1.234 cm:

0x02	'1' (0x31)	'.' (0x2E)	'2' (0x32)	'3' (0x33)	'4' (0x34)	'\r' (0x0D)	0x03

Toggling the heartbeat three times allows you to see the LED toggle every 25ms, as well as measure the total execution time of the ISR using an oscilloscope attached to the heartbeat. To understand the transmitter better answer these questions

Q1. The UART transmitter has a built-in hardware FIFO. How deep is the FIFO? (How many frames can it store?)
Q2. At a baud rate of 100,000 bits/sec (1 start, 8 data, 1 stop) how long does it take to send 8 frames?
Q3. The SysTick ISR runs every 25 ms, and each ISR calls **`UART1_OutChar`** 8 times. Does this hardware FIFO ever fill? In other words will the busy-wait loop in **`UART1_OutChar`** ever loop back?
Q4. Assuming we use the same protocol for the UART and we transmit one message every 1/40th of a second, what is the slowest baud rate we could use?

Part d - Test The Transmitter

Write a temporary main program for testing the transmitter, which initializes the PLL, SysTick, ADC, UART1, heartbeat, and enables interrupts. After initialization, this transmitter test main (foreground) performs a do-nothing loop. This main program will only be used for testing, and will be replaced by a more complex main once the receiver is built. The ADC sampling and UART1 transmissions occur in the SysTick interrupt service routine running in the background. It is possible, but not required to debug the system in simulation mode. Figure 9.4 shows the transmitter being debugged in simulation mode. Notice the various representations of the 1.424 cm position signal, the analog input voltage 1.5 V on the ADC, and the 02 31 2e 34 32 34 0d 03 message in the UART debugging window. (Simulation for lab 9 isn't available right now. You'll have to use the oscilloscope to debug the lab.)

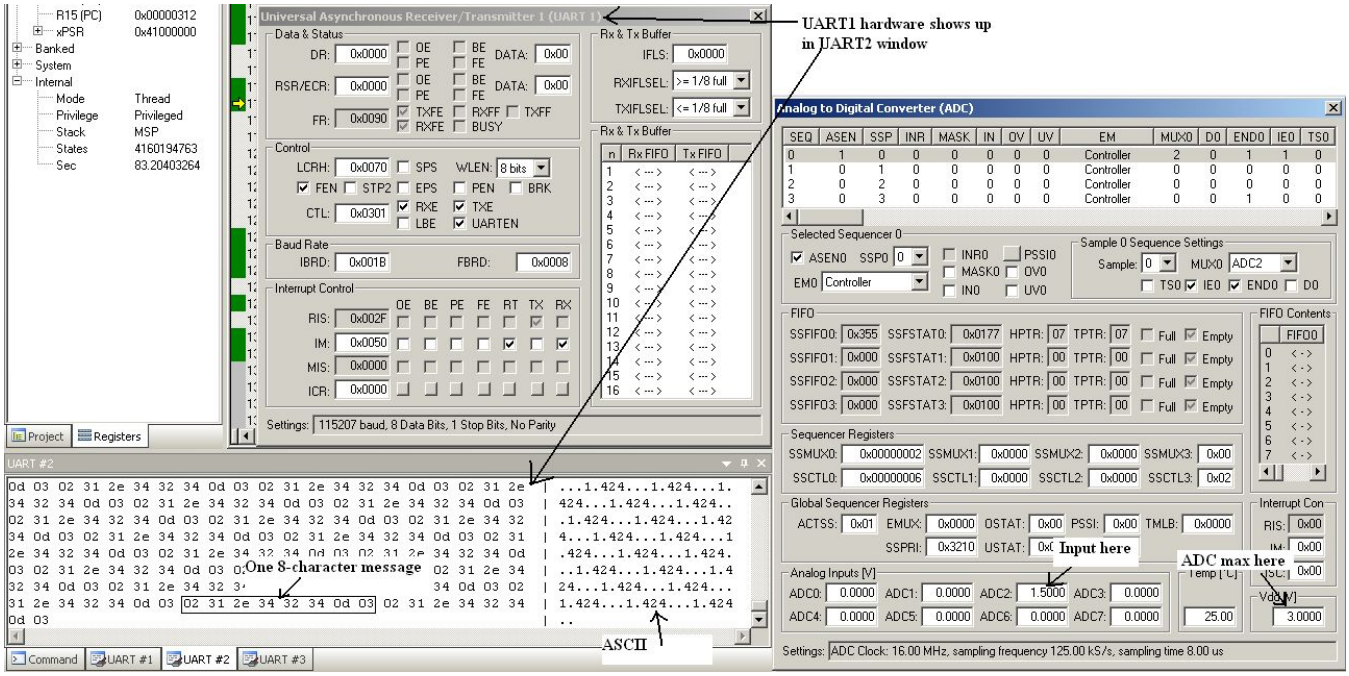

Figure 9.4. Example screenshot showing the transmitter system in simulation mode. We are currently unable to simulate the UART in interrupting mode .

To debug the transmitter on the real TM4C123, you can place the ADC sample and the interrupt counter in a debugger Watch window. The interrupt **`TxCounter`** should increase by 40 every second, and the 12-bit ADC sample should respond to changes in the slide pot. Attach a dual-channel oscilloscope to the heartbeat and UART outputs, you will see the 8 frames being transmitted at 100,000 bits/sec every 25 ms. Take two photographs of the scope output, one zoomed in to see the first two frames, and one zoomed out to see the 25ms time between messages, as shown in Figures 9.5 and 9.6.

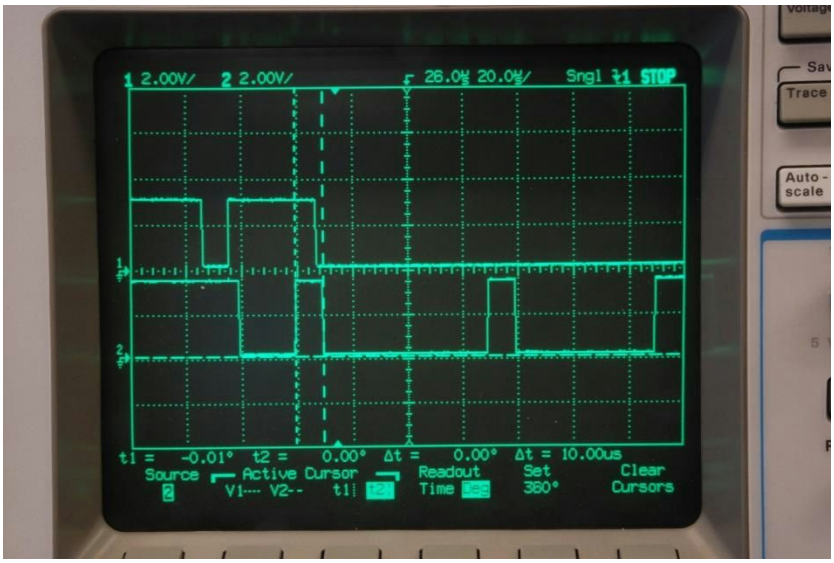

Figure 9.5. Example scope output measured on the transmitter. The top trace is heartbeat showing the SysTick interrupt has occurred and the total execution time of the SysTick ISR is 42µs. The bottom trace is U1Tx showing the

first frame (STX =0x02, frame is start, bit0, bit1, bit2, bit3, bit4, bit5, bit6, bit7, and stop). The width of bit1 is 10µs, showing the baud rate is 100,000 bits/sec.

Questions to think about:

Q1. If the SysTick ISR sampled the ADC and performed 8 UART outputs, why did this ISR finish in only 42 µs?
Q2. Using the photo captured in Figure 9.5, approximately how long does it take to sample the ADC?

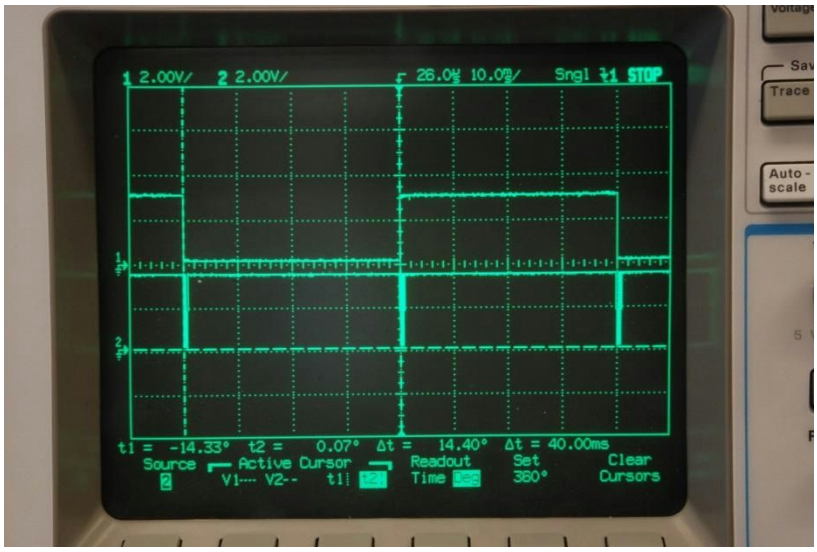

Figure 9.6. Example scope output measured on the transmitter. The top trace is PG2 showing the SysTick interrupt is occurring every 40 ms. The bottom trace is PD2 (U1Tx). The 8 frames of PD2 occur in 800 µs, so on this scale one cannot see the individual frames. (This data is from Spring 2013. Your measurements should show SysTick occurring every 25ms)

Receiver software tasks

All tasks listed under the this section should be completed on both microcontrollers. Again, both teams should participate in the transmitter software tasks, and both teams are expected to understand the software produced by the transmitter tasks.

Part a - Implement a Queue

Design, implement and test a C language module implementing a statically allocated FIFO. You are allowed to look at programs in the book and posted on the internet, but you are required to write your own FIFO. Examples of how to make your FIFO different from the ones in the book include:

1. store the data so the oldest is on top and one size counter defines the FIFO status;
2. implement it so 5, 6, or 8 bytes are put and get at a time;
3. reverse the direction of the pointers so both PutPt and GetPt decrement rather than increment;

Make sure your FIFO is big enough to hold all data from one message. Place the FIFO code in a separate **`fifo.c`** file. Add a **`fifo.h`** file that includes the prototypes for the functions.

The software design steps may include:

1. Define the names of the functions, input/output parameters, and calling sequence. Type these definitions in as comments that exist at the top of the functions.
2. Write pseudo-code for the operations. Type the sequences of operations as comments that exist within the bodies of the functions.
3. Write C code to handle the usual cases. I.e., at first, assume the FIFO is not full on a put, not empty on a get, and the pointers do not need to be wrapped.
4. Write a main program in C to test the FIFO operations. An example main program is listed below.
5. Iterate steps 3 and 4 adding code to handle the special cases.

The example test program assumes the FIFO can hold up to 6 elements. This main program has no UART, no LCD and no interrupts. Simply make calls to the three FIFO routines and visualize the FIFO before, during and after the calls. Debug either in the simulator or on the real board. In this example, Put and Get return a true if successful, Put takes a call by value input, and Get takes a return by reference parameter. You are free to design the FIFO however you wish as long as it is different from the ones in the book, and you fully understand how it works.

```
#include "fifo.h"
int Status[20];             // entries 0,7,12,19 should be false, others true
unsigned char GetData[10];  // entries 1 2 3 4 5 6 7 8 should be 1 2 3 4 5 6 7 8
int main(void){
  Fifo_Init(); // initializes a FIFO that holds 6 elements
  for(;;){
    Status[0]  = Fifo_Get(&GetData[0]);  // should fail,    empty
    Status[1]  = Fifo_Put(1);            // should succeed, 1
    Status[2]  = Fifo_Put(2);            // should succeed, 1 2
    Status[3]  = Fifo_Put(3);            // should succeed, 1 2 3
    Status[4]  = Fifo_Put(4);            // should succeed, 1 2 3 4
    Status[5]  = Fifo_Put(5);            // should succeed, 1 2 3 4 5
    Status[6]  = Fifo_Put(6);            // should succeed, 1 2 3 4 5 6
    Status[7]  = Fifo_Put(7);            // should fail,    1 2 3 4 5 6
    Status[8]  = Fifo_Get(&GetData[1]);  // should succeed, 2 3 4 5 6
    Status[9]  = Fifo_Get(&GetData[2]);  // should succeed, 3 4 5 6
    Status[10] = Fifo_Put(7);            // should succeed, 3 4 5 6 7
    Status[11] = Fifo_Put(8);            // should succeed, 3 4 5 6 7 8
    Status[12] = Fifo_Put(9);            // should fail,    3 4 5 6 7 8
    Status[13] = Fifo_Get(&GetData[3]);  // should succeed, 4 5 6 7 8
    Status[14] = Fifo_Get(&GetData[4]);  // should succeed, 5 6 7 8
    Status[15] = Fifo_Get(&GetData[5]);  // should succeed, 6 7 8
    Status[16] = Fifo_Get(&GetData[6]);  // should succeed, 7 8
    Status[17] = Fifo_Get(&GetData[7]);  // should succeed, 8
    Status[18] = Fifo_Get(&GetData[8]);  // should succeed, empty
    Status[19] = Fifo_Get(&GetData[9]);  // should fail,    empty
  }
}
```

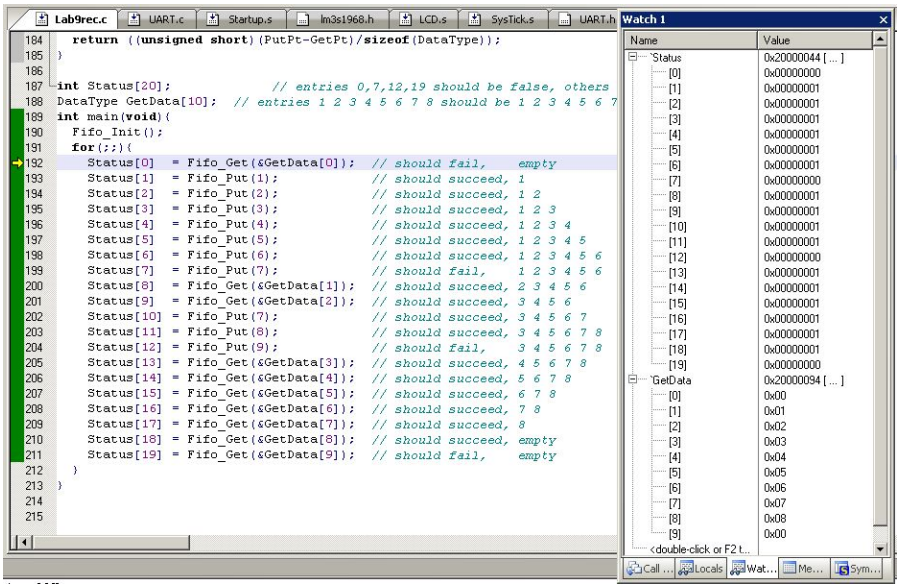

Figure 9.7. Example simulator screen shot debugging the FIFO.

Part b - Initialize UART

Write a C function, **`UART1_Init,`** that will initialize the UART1 receiver for the receiver. This function should:

1. clear a global error count, initialize your FIFO (calls `Fifo_Init`)
2. enable UART1 receiver (arm interrupts for receiving)
3. set the baud rate to 100,000 bits/sec

In general, one must enable interrupts in order for UART1 interrupts to occur. It is good style however to finish all initializations, then enable interrupts from the high-level main program. The receiver initialization with interrupts should be added to the transmitter initialization using busy-wait.

There are many options for arming interrupts for the receiver. However, we require you to arm just RXRIS with ½ full FIFO. This way you get one interrupt for each 8-frame message received. One method, as illustrated in the book Program 11.6, but not allowed in Lab 9, arms both RXRIS (receive FIFO 1/8 full) and RTRIS (receiver idle). This way the bytes are passed one at a time through the FIFO from ISR to main. The main program would get the bytes one at a time and process the message. Another method, good approach but not allowed in Lab 9, is to arm just RTRIS. This way you get one interrupt a short time after all 8 bytes have been received.

Part c - Receive UART Data

Write the UART interrupt handler in C that receives data from the other computer and puts them into your FIFO. Your software FIFO buffers data between the ISR receiving data and the main program displaying the data. If your software FIFO is full, increment a global error count. If you get software FIFO full errors, then you have a bug that must be removed. Do not worry about software FIFO empty situations. The software FIFO being empty is not an error. Define the error count in the UART.c file and increment it in the UART ISR.

Arm RXRIS receive FIFO ½ full (interrupts when there are 8 frames): the interrupt service routine performs these tasks:

1. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
2. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
3. as long as the RXFE bit in the UART1_FR_R is zero

    a. Read bytes from UART1_DR_R
    b. Put all bytes into your software FIFO (should be exactly 8 bytes, but could be more)
    c. t
    d. the message will be interpreted in the main program
4. Increment a `RxCounter`, used as debugging monitor of the number of UART messages received
5. acknowledge the interrupt by clearing the flag which requested the interrupt
    a. UART1_ICR_R = 0x10; // this clears bit 4 (RXRIS) in the RIS register
6. toggle a heartbeat (change from 0 to 1, or from 1 to 0),
7. return from interrupt

Toggling heartbeat three times allows you to see the LED flash with your eyes, as well as measure the total execution time of the ISR using an oscilloscope attached to heartbeat. Place all the UART1 routines in a separate **`UART1.s`** file (or **`UART1.h`** and **`UART1.c`**). This driver is not the same as the UART driver in computer 1.

Part d - Display Received Measurement Data

The body of the main program reads data from the FIFO and displays the measurement using the same LCD routines developed in Lab 7 and used in Lab 8. The main program in this data acquisition system performs these tasks:

1. initialize PLL
2. initialize transmitter functions (ADC, heartbeat, SysTick, UART transmission busy-wait)
3. initialize receiver functions (FIFO, LCD, heartbeat, UART receiver with interrupts)
4. enable interrupts
5. calls your FIFO get waiting until new data arrives. (wait until you see the start byte)
6. output the fixed-point number (same format as Lab 8) with units.
    a. The next five characters gotten from the FIFO should be the ASCII representation of the distance
7. repeat steps 5,6 over and over

FIFO should return an Empty condition if the main program in the receiver calls get and the FIFO is empty. For more information on how the ISR and main use the FIFO, read Section 11.3 in the book and look at the left side of Figure 11.6. Also read Section 11.4 and look the the function **`UART_InChar`** in Program 11.6**.** I

The format of the UART message is reproduced below, where each message is exactly eight (8) serial frames at 100,000 bits/sec.

Start (0x02)	Digit 1	'.' (0x2E)	Decimal 1	Decimal 2	Decimal 3	'\r' (0x0D)	End (0x03)

For example if the distance is 1.234 cm:

0x02	'1' (0x31)	'.' (0x2E)	'2' (0x32)	'3' (0x33)	'4' (0x34)	'\r' (0x0D)	0x03

Debugging both Transmitter and Receiver

To debug the system on one TM4C123, connect the transmitter output to the receiver input. In this mode, the system should operate like Lab 8, where the sensor position is displayed on the LCD. Use the oscilloscope to visualize data like Figures 9.5 and 9.6.

To debug the distributed system on the real TM4C123, use two PC computers close enough together so you can connect the two TM4C123 LaunchPads with the special cable, yet run **uVision4** on both computers. You can place strategic variables (e.g., ADC sample, interrupt counter) in the Watch windows on both PCs. If you start both

computers at the same time all four interrupt counters should be approximately equal, and the data on both computers should respond to changes in the slide pot. For the lab to work completely, the transmitter and receivers must synchronize properly regardless of which computer is turned on first, so try turning the boards on/off in different orders and at different times.

Demonstration

(both partners must be present, and demonstration grades for partners may be different)

You will show the TA your program operation on the two TM4C123 boards. Your TA will connect a scope to either transmitter heartbeat and TxD→RxD or receiver heartbeat and TxD→RxD and expect you to explain the relationship between your executing software and the signals displayed on the scope. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. How do you initialize UART? How do you input and output using UART? What is the difference between busy-wait and interrupt synchronization? What synchronization method does the transmitter UART use? What synchronization method does the receiver UART use? What sets RXFE, TXFF, RXRIS, RTRIS and when are they set? What clears RXFE, TXFF, RXRIS, RTRIS and when are they cleared? What does the PLL do? Why is the PLL used? There are both hardware and software FIFOs in this system. There are lots of FIFO code in the book and on the web that you are encouraged to look at, but you are responsible for knowing how your FIFO works. What does it mean if the FIFO is full? Empty? What should your system do if the FIFO is full when it calls PUT? Why? What should your system do if the FIFO is empty when it calls GET? Why?

Deliverables

(Items 1, 2 are one pdf file uploaded to SVN, have this file open during demo.)

0. Lab 9 grading sheet. You can print it yourself or pick up a copy in lab. You fill out the information at the top.
1. A circuit diagram showing position sensor, the UART, and the LCD.
2. Two photographs of a scope trace showing serial transmission data, like Figures 9.5 and 9.6.
3. Optional Feedback : http://goo.gl/forms/rBsP9NTxSy

Commit to SVN the final version of the software that implements both transmitting and receiving on one computer. All source files that you have changed or added should be committed to SVN. Please do not commit other file types.

Hints

1. Remember the port name **UART1_DR_R** links to two hardware FIFOs at the same address. A write to **UART1_DR_R** is put into the transmit hardware FIFO and a read from **UART1_DR_R** gets from the receive hardware FIFO
2. You should be able to debug your system with any other EE319K group. You are not allowed to look at software written by another group, and you certainly are not allowed to copy software from another group. You are allowed to discuss strategies, flowcharts, pseudocode, and debugging techniques.
3. This system can be debugged on one board. Connect TxD to RxD.
4. This hint describes in more detail how to create a message. Assume the distance is 1.424 and your software calculates the appropriate integer 1424. First you convert the digits to 1, 4, 2, 4 in a similar way as LCD_OutFix. Next, you make the digits ASCII 0x31, 0x34, 0x32, 0x34. Add the STX(02), the decimal point (2E), the CR (0D) and ETX (03), and the message will be 0x02, 0x31, 0x2E, 0x34, 0x32, 0x34, 0x0D, 0x03. To send this message you call **`UART_OutChar`** 8 times using busy-wait. These 8 bytes will go in the hardware FIFO of the UART transmitter. The 8 bytes will be sent one at a time, one byte every 100usec.

5. In order to synchronize the computers regardless of which one is turned on first, the receiver must search a variable number of input frames until it finds an STX. After receiving an STX, the next 7 frames are a message. Furthermore, the receiver should discard messages that are not properly formatted (e.g., no decimal point, decimal digits, CR, or ETX)

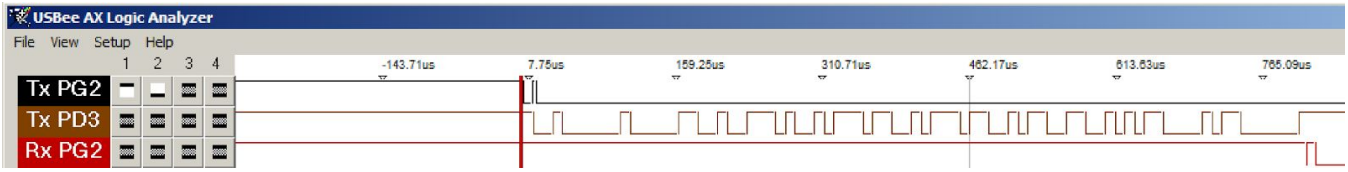

Figure 9.8. Logic analyzer trace showing heartbeat in the transmitter ISR, the serial line, and heartbeat in the receiver ISR. Time to execute the transmitter ISR is 14us, the time to send the message is 800us, and the time to execute the receiver ISR is 10us.

FAQ.

1. How many elements should our FIFO be (from fifo.c)?

I would suggest atleast 9 because if your FIFO is simlar to the one in class, it will have a empty array element. A message is 8 elements, so therefore, we would need atleast a size of 9 for the empty element buffer.

2.In the lab manual for the transmission portion it states for you to create an array. What is the array for? It also says to send the 8 byte message. What is this message? Does it have anything to do with the ADC?

The transmission of distance measurements follows a universal (for lab#9) MESSAGE PROTOCOL as described on Page #3 of the manual. i.e., every transmission ans reception of distance measurement happens as a group of 8-bytes, where each BYTE is sent using the UART protocol. Thus, the array is to store a copy of this MESSAGE. The specification of this contents of this MESSAGE array is described on page#3. There are always four fixed bytes, 1st, 3rd, 7th and 8th. Remaining 4 Bytes correspond to the converted distance measurement output.

So, yes it is indirectly related to the ADC measurements. i.e, 2nd, 3rd, 4th, 5th, 6th bytes = the individual bytes that your LCD_OutFix(Convert(ADC_In())) was sending to the LCD in Lab#8.

3.When I'm trying to send the four digits for part c of the lab my program gets stuck in the outchar subroutine. My program doesn't crash when I halt it but it's also not getting past that subroutine. It outputs the first character that we want which is 0x02.

enable the transmit and receive bits in the ctl register.
#define UART_CTL_TXE 0x00000100 // UART Transmit FIFO enable
#define UART_CTL_RXE 0x00000200 // UART Receive FIFO enable
And this command in your init function:
UART1_CTL_R |= (UART_CTL_UARTEN|UART_CTL_TXE|UART_CTL_RXE); // enable UART

4.For part 2 part b, the lab manual says to write a UArt1_Init function for the receiver, but i have already written that function for part 1. Should i just call UArt1_init and call Fifo_Init inside of it?

You can first write a different function and test the Init for receiver separately if that helps you debug. But eventually your program will be able to do both receive and transmit so yes, you can simply call Fifo_Init() in your UART1_Init().

5.Is there suppose to be two different UART1.c files? One for the transmitter and one for the receiver (ie. UART1Tx.c and UART1Rx.c)? If not, does the additional code for enabling the Rx interrupt simply just go into the UART1 Init function?

No. There is no need for separate files. You should add the code for Rx to your previous UART_Init() routine. A possibility is to connect your transmitter (provided you have already made it to run) to your receiver and simply debug your receiver using your own transmitter.

6.Do we have to use an array to save our message before transferring it to the UART?
Couldn't we output them to the UART as we are converting (like a modified LCD_OutDec that outputs to UART instead of the LCD) instead of sending them to an array to be outputted at a later point in the code?

Functionality wise yes they will do the same thing. I think the point was to ensure that you always follow the same structure throughout your code, that it's easier for others to read your code (they can directly see that one message contains 8 bytes with 0x02 start and 0x03 end etc.), and that it's easier to debug through the watch window (you can directly see the content of the array and see what is being sent).

7.My transmitter code works on the oscilloscope, but now every time I try to open the debugger, it will open for about 4 seconds and automatically goes back to code editing mode. Two other people have also had this problem. I had to reflash my board because of a power cycling problem today, just for background. After I fixed that, the code flashes on the microcontroller, but as i said, debugger won't stay open. Does anyone know how to solve this problem? I will have no way to step through and watch my other code works until I figure out how to get debugging working again.

If you're running Windows 8 or 10, try this:

http://users.ece.utexas.edu/~valvano/Volume1/Window8KeilDebuggerFix.htm

8.To clarify, for the final product are we flashing the exact same Lab9 project onto each microcontroller? If so, is it necessary to have different heartbeat init and UART init functions for receiving vs. transmitting, or can we just have one UART Init and one heartbeat Init that initialize everything for both receiving and transmitting?

You will be flashing your microcontroller with your code. During the demo, your Tx/Rx will be connected to the Rx/Tx of a SECOND microcontroller board. The SECOND microcontroller board would be flashed with a different teams (randomly selected) Lab 9 code or a TA/Instructor Lab 9 code.

9.How exactly are we supposed to test the transmitter part of the code. Previous answer say that we cannot use the simulator, instead we have to use an oscilloscope. I assume we connect one of the probes to the UART transmitter port, and connect the other to ground, but what exactly does this show us?

You should understand the UART standard and how it works. When a byte is sent through UART, the 8 bits are sent serially along with a start bit of 0 (voltage LOW) that signifies the beginning of a byte, and a stop bit of 1 (voltage HIGH) that signifies the end of a byte. For example, when a 0xAB is sent, the voltage toggles in the following way per unit time: 0 1 0 1 0 1 0 1 1 1, where the first 0 is the start bit, and the last 1 is the stop bit. When nothing is being sent, the voltage stays high. You can look into the lecture slides (Lec12) for more details. When you look at the oscilloscope, you should check whether if the waveform matches what you are expecting to be sent.

10.Our strings to print end with a carriage return '\r', which moves the cursor to the next line when it is printed. This results in the measurements quickly being printed in new lines from the top to bottom of the

screen. Are we supposed to manually move the cursor back to the first line after every print to keep the measurement in the same spot?

Just don't print the carriage return! (or move the cursor back, your choice).

11.Just for clarification, does the debugging simulator work for *any* part of Lab 9 (ie. transmitter part, receiver part)? Or just some parts?

Watch windows can work. There is no simulator for the UART channels. This why teams need to test with other teams. Debugging without the software simulator is a skill you will need as you can't really do Lab 10 in the simulator.

12.