Lab 10c Peer-to-Peer Serial Network

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Develop a layered communication system,
- Design and implement a hardware/software interconnect protocol.
- Use communication skills to work effectively as team.

Review
- Valvano Chapter 7 on RS232 and SCI operation,
- Valvano Chapter 14 on communication systems.

Starter files
- **SCIA** project

Background

Figure 10.1 shows I/O system that uses interrupts for both input and output. When the main program wishes to output, it calls **SCI_OutChar**, which will put the data in the **TxFifo** and arm the output device. When the main program wishes to input, it calls **SCI_InChar**, which will get data from the **RxFifo**. This example has been implemented as the **SCIA** project.

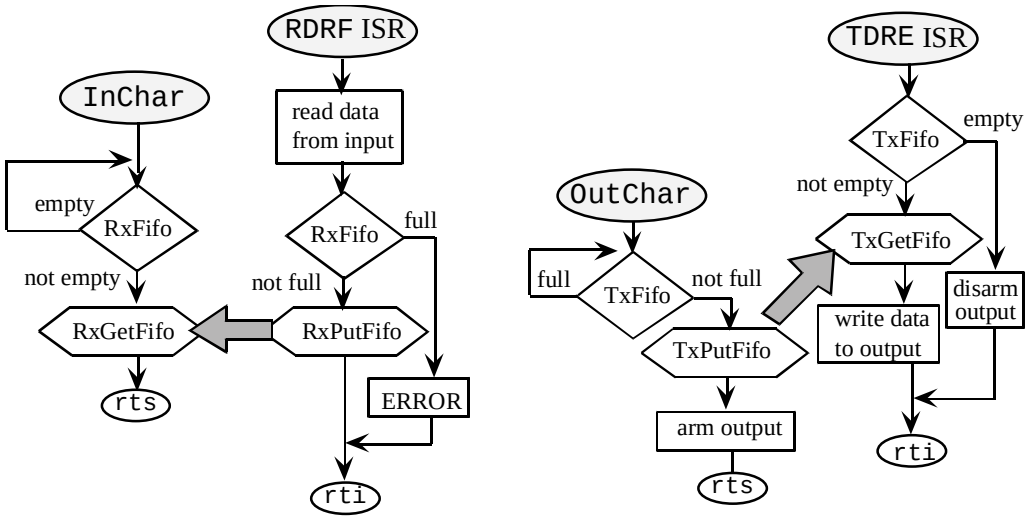

Figure 10.1. FIFO queues can be used to pass data between threads.

The incoming serial data will set **RDRF**, requesting an interrupt. The ISR (background) will accept the data and put it in the **RxFifo**. The **RxFifo** buffers data between the input hardware and the main program that processes the data. If the **RxFifo** becomes full, then data will be lost. FIFO full errors will always occur if the average input rate (number of bytes arriving per second from the input hardware) exceeds the average processing rate (number of bytes processed per second by the main program.) In this situation, either the output rate must be increased (by using a faster computer or by writing a better software processing algorithm), or the input rate must be decreased (by slowing down the arrival rate of data.) The second way the **RxFifo** could become full is if there is a temporary increase in the arrival rate or a temporary decrease in the process rate. For this situation, the full errors could be eliminated by increasing the size of the **RxFifo**.

TDRE signals the output device is idle, ready to output more data. If there is data available in the **TxFifo**, the ISR will get it and write it to the output device. If the **TxFifo** is empty, the output device is disarmed. The main program puts data in **TxFifo** and gets data from the **RxFifo** as desired. If the **TxFifo** becomes full, then it is appropriate to wait for the interrupts to make room. It is inefficient, but not catastrophic for

the main program to wait on a full **TxFifo**. Efficiency can be improved for the buffered output problem by increasing the **TxFifo** size. It is also inefficient, but not catastrophic for the main program to wait on an empty **RxFifo**. Efficiency can be improved for the buffered input problem by performing other tasks while waiting for data.

It is important to study the timing behavior of the I/O hardware and software processing when designing an interrupting interface. One simple way to study a problem is to measure the number of elements in the **RxFifo** when new data is entered by the input ISR. If the time for the software to read and process the data is much faster than the time for the input device to create new input, then there will be very few elements in the **RxFifo**. For most systems, the producer and consumer rates fluctuate, but during the times when the software waits for the I/O hardware, the system is classified as **I/O bound**. For an I/O bound input interface the **RxFifo** has either 0 or 1 entry, and the use of interrupts does not enhance the bandwidth over the busy-waiting implementations. Even with an I/O-bound input device however, it may be more efficient to utilize interrupts because it provides a straight-forward approach to servicing multiple devices.

If the input device generates a burst of high bandwidth activity, then there will be many elements in the **RxFifo**. As long as the interrupt service routine is fast enough to keep up with the input device and as long as the **RxFifo** does not become full, no data is lost. Recall the ISR doesn't have to process the input data, just read it and save it in the **RxFifo**. In this situation, the overall bandwidth is higher than it would be with a busy-waiting implementation, because the input device does not have to wait for each data byte to be processed. This is the classic example of a "buffered" input, because data enters the system (via the interrupts) is temporarily stored in a buffer (put into the **RxFifo**) and the data is processed later (by the main program, get from the **RxFifo**.) During the times when the I/O device is faster than the software, the system is called **CPU-bound**. A system will work only if the producer rate only temporarily exceeds the consumer rate (a short burst of high bandwidth input). If the external device sustained the high bandwidth input rate, then the **RxFifo** would become full and data would be lost.

For an output device, we will count the number of elements in the **TxFifo** when data is removed by the output ISR. If the rate for the software to generate new data is much slower than the rate for the output device to send data, then there will be very few elements in the **TxFifo**. During this time the system is called CPU-bound. In this situation, the **TxFifo** has either 0 or 1 entry, and the use of interrupts does not enhance the bandwidth over the busy-waiting implementations. Even with a CPU-bound output device however, it may be more efficient to utilize interrupts because it provides a straight-forward approach to servicing multiple devices.

If the main program generates a burst of output activity, then there will be many elements in the **TxFifo**. In this situation, the overall bandwidth is higher than it would be with a busy-waiting implementation, because the main program does not have to wait for each data byte to be outputted. This is the classic example of a "buffered" output, because data enters the system (via the main program) is temporarily stored in a buffer (put into the **TxFifo**) and the data is processed later (by the output ISR, get from the **TxFifo**.) During the time when the main program is faster than the output hardware, the system is called I/O-bound. Just like the input situation, a system will work only if the producer rate temporarily exceeds the consumer rate. If the main program sustained the output rate, then the **TxFifo** would become full and main program would then have to wait. Again, the output situation is most efficient if the **TxFifo** is big enough to avoid full errors.

An axiom with interrupt synchronization is that the interrupt program should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should come in clean, perform that function, and return right away. Placing backward branches (busy-waiting loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be minimized. For an input device, the **interface latency** is the time between when new input is available, and the time when the software reads the input data. We can also define device latency as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it take to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle, and the time when the software writes new data. A **real-time** system is one that can guarantee a worst case interface latency.

Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. We will limit our discussion with communication with devices within the same room. A *full duplex* channel allows data to transfer in both directions at the same time. In a *half duplex* system, data can transfer in both directions but only in one direction at a time. Half-duplex is popular because it is less

expensive (2 wires) and allows the addition of more devices on the channel without change to the existing nodes. If the distances are short, half-duplex can be implemented with simple *open collector* or *open-drain* TTL-level logic. Open collector logic has two output states: low and off. In the off state the output is not driven high or low, it just floats. The 10 kΩ pull-up resistor will passively make the signal high if none of the open collector outputs are low. The 6812 can make its **`TxD`** serial outputs be open collector. This mode allows a half-duplex network to be created without any external logic (although pull-up resistors are often used). Three factors will limit the implementation of this simple half-duplex network: 1) the number nodes on the network, 2) the distance between nodes; and 3) presence of corrupting noise. In these situations a half-duplex RS485 driver chip like the SP483 made by Sipex or Maxim can be used.

The master-slave system connects the master transmit output to all slave receive inputs, see Figure 10.2. This provides for broadcast of commands from the master. All slave transmit outputs are connected together using wire-or open collector logic, allowing for the slaves to respond one at a time. The 6812 **`WOMS`** bit (**`SC0CR1`** bit 6) in the slaves should be set to activate open collector mode on **`PS1`**.

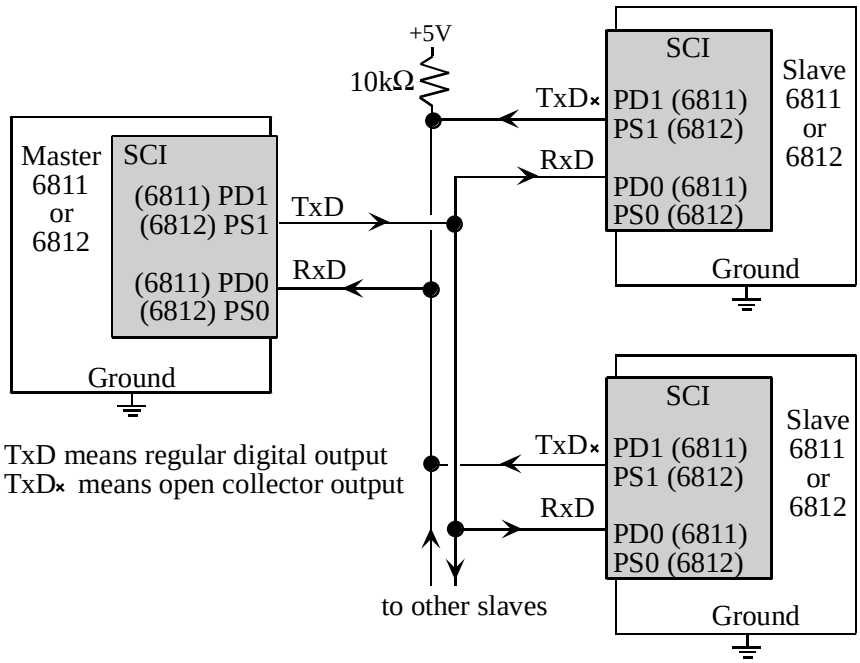

Figure 10.2. A master-slave network implemented with multiple microcomputers.

The ring network is a simple distributed approach, because it can be constructed using standard serial ports, by chaining the transmit and receive lines together in a circuit, as shown in Figure 10.3. Messages will include source address, destination address and information. If computer A wishes to send information to computer C, it sends the message to B. The software in computer B receives the message, notices it is not for itself, and it resends the message to C. The software in computer C receives the message, notices it is for itself, and it keeps the message.

A very common approach to distributed embedded systems is called multi-drop, as shown in Figure 10.4. To transmit a byte to the other computers, the software activates the SP483 driver and outputs the frame. Since it is half-duplex, the frame is also sent to the receiver of the computer that sent it. This echo can be checked to see if a collision occurred (two devices simultaneously outputting.) If more than two computers exist on the network, we usually send address information first, so that the proper device receives the data.

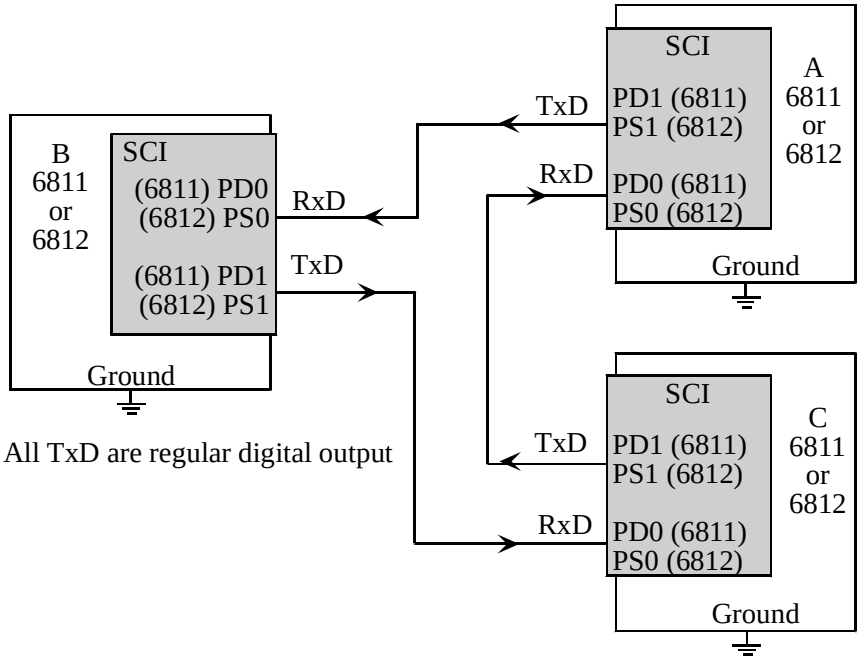

Figure 10.3. A ring network implemented with 3 microcomputers.

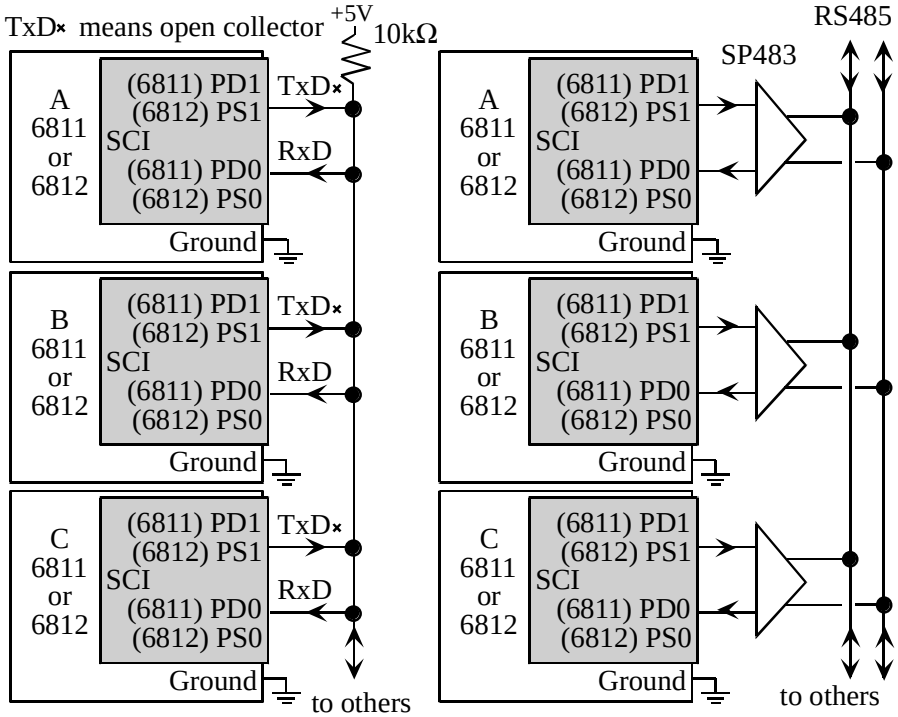

Figure 10.4. Two multi-drop networks implemented with 3 microcomputers.

Specifications

The overall goal of this lab is to design, implement and test a peer-to-peer communication system. Peer-to-peer means people on two computers communicate without the people on the other computers seeing the information. The system must use the serial channel, must use interrupt-driven I/O, and must have a layered software configuration. The actual implemented system should work for 1, 2, or 3 computers, but the approach should workable for networks with more nodes. The lowest-level software performs serial I/O. The middle-level software sends message packets from one computer to another. The highest level software interfaces with the human operator (keyboard/LCD) and provides a mechanism to create a peer-to-peer connection. In a layered system, software in one layer can only call routines within that layer or the layer immediately below it, as shown in Figure 10.5 You need a way to see who is on the network, and a way to request/accept/terminate connection between two operators. Local operator input/output will occur via the keyboard and LCD as developed in the calculator lab. You may assume all nodes on the system are willing to cooperate and not perform malicious activity.

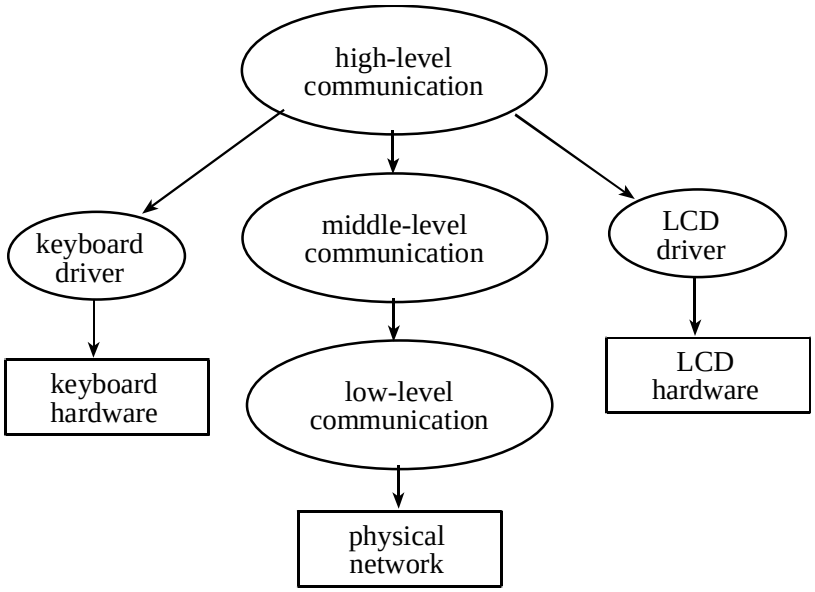

Figure 10.5. Call graph for a layered system.

Preparation (do this before your lab period)

1: Design the hardware-level communication interconnection. One possibility uses the existing RS232 channel connected in a ring, similar to Figure 10.3. You need to construct the cable/connector hardware that allows 1, 2, or 3 devices to communicate. For example, if you have a ring network, then a simple male DB9 plug can be inserted into any ring sockets that have no computer attached. DB9 connectors and 4-wire cable are available for this lab. Ask your TA where you can get the materials needed to build the physical part of your network.

2: Write the low-level serial I/O device driver. You should be able to use the existing SCIA project with very few changes. You should be able to run at a faster baud rate than used to communicate with the PC.

3: Next, design the middle-level software system that implements message passing.

4: Finally, write the high-level software that implements the operator input and communication system.

Procedure (do this during your lab period)

Write separate main programs to test each module of the system. In you are testing bottom-up, start with the lowest level and work up. If you test top-down, write snubs for the lower level modules. One of the hardest

aspects of this lab will be interacting as a large group. It will be important to organize and delegate. Elect a manager, a secretary (records meetings and decisions). Divide the project into pieces that can be developed in parallel. Provide time in the schedule for possibility that decisions you make may be wrong.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)

B) Hardware Design

Detailed circuit diagram of the physical layer of the network (preparation 1)

C) Software Design (no software printout in the report)

Draw figures illustrating the major data structures used

A call-graph illustrating the modularity of the software components

D) Measurement Data

Measure the network bandwidth

E) Analysis and Discussion

Include minutes (date, time, duration, attendance, topics) for each team meeting

Document major features of the network

F) Post-mortem concerning team member interactions (attached to the report)

1) Each team member evaluates each other team member including oneself

Simply list one or two weaknesses.

Simply list two or three strength characteristics.

2) Major failures in the way the team interacted (if any)

3) Major successes in the way the team interacted

G) Peer Review (each student submits independently and confidentially directly to the TA)

Classify each team member including oneself as:

- worked harder than average (explain), worked an average amount, worked less than average (explain)

Checkout (show this to the TA)

You should demonstrate the major features of your communication system to the TA.

Your software files will be copied onto the TA's zip drive during checkout.