

## Lab 10 SCI Serial Interface

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

- Goals**
- Design the low-level communication interface between two or more microcomputers,
  - Analyze the synchronization problems that occur when two computers are interfaced,
  - Investigate low-level methods for error detection,
  - Implement half-duplex serial communication.

- Review**
- Valvano Chapter 7,
  - Data sheets for SP483,
  - The chapter on the SCI port in the Motorola Reference Manual for your microcomputer.

- Starter files**
- SCI12A.C, MASTER.C, SLAVE.C, RXFIFO.C, RXFIFO.H, TXFIFO.C, TXFIFO.H

### Background

The half-duplex communication system between two or more 6812 computers will be designed in two layers. The first layer, the physical layer, will allow single characters to be transmitted. A half-duplex asynchronous physical channel should be constructed using open collector logic. The second layer (the next lab) will consist of a simplified binary synchronous communication protocol (BSC).

There are many ways to check for transmission errors. Since it is half-duplex, the frame goes to all the Rx/D serial inputs including the same computer that sent it. In this lab, we will check the echo of each transmitted byte to detect for collisions. In the next lab, we will add error checking at the next higher layer that will detect errors that occur in the channel, that might affect the data received by the receiver(s), but is still correctly echoed into the transmitter. E.g., this class of error (to be handled in the next lab) could arise if the receiver is disconnected.

You will use the asynchronous serial interface, SCI, with interrupt synchronization. Half-duplex communication allows information to travel both directions but is cheaper than full duplex. It is simpler and easier to implement more than 2 nodes on the network with half-duplex. The two wires in the half-duplex channel should be twisted pair. The hardware diagram is shown in Fig. 10.1. The asynchronous serial protocol will be 8 bits, no parity, 1 stop bits, and 9600 baud. You are free to adjust these parameters as long as all groups on your network agree. Use the 6812 internal half-duplex mode (LOOPS=1, RSRC=1) and the open collector mode (SWOM=1)

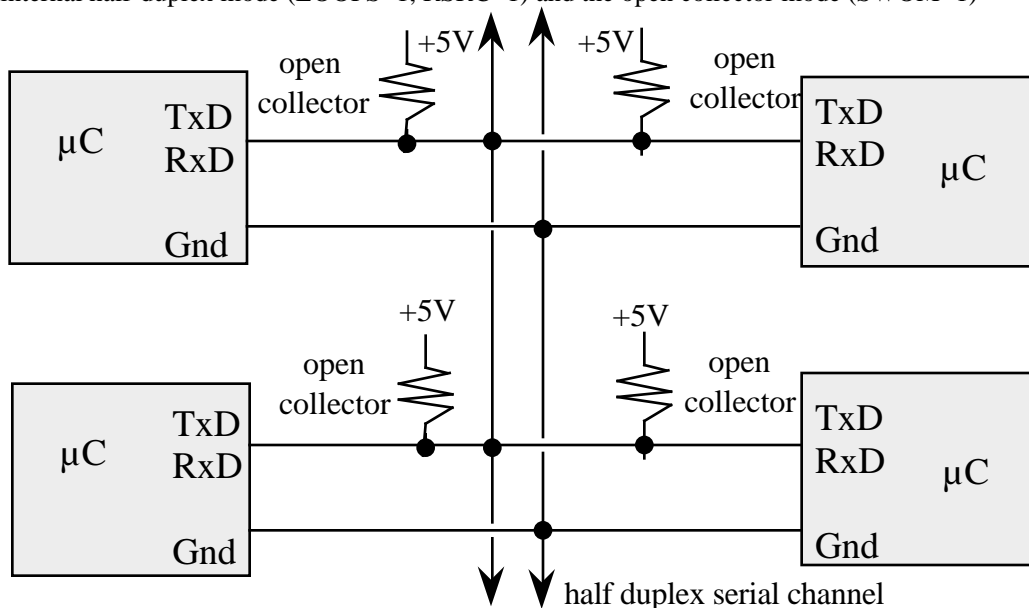


Figure 10.1. Block diagram of an open-collector half-duplex asynchronous serial channel.

A second option is to use the built-in RS485 interface of the Adapt812, as shown in Fig. 10.2. The Sipex SP483 chip creates the RS485 channel. RS485 is more robust than open-collector digital. To enable the RS485 driver, make PE6 and PE5 outputs and set PE6 to high and PE5 to low. The normal mode for the SCI1 can then be used. In particular, for this hardware configuration we set LOOPS, WOMS and RSRC all to zero.

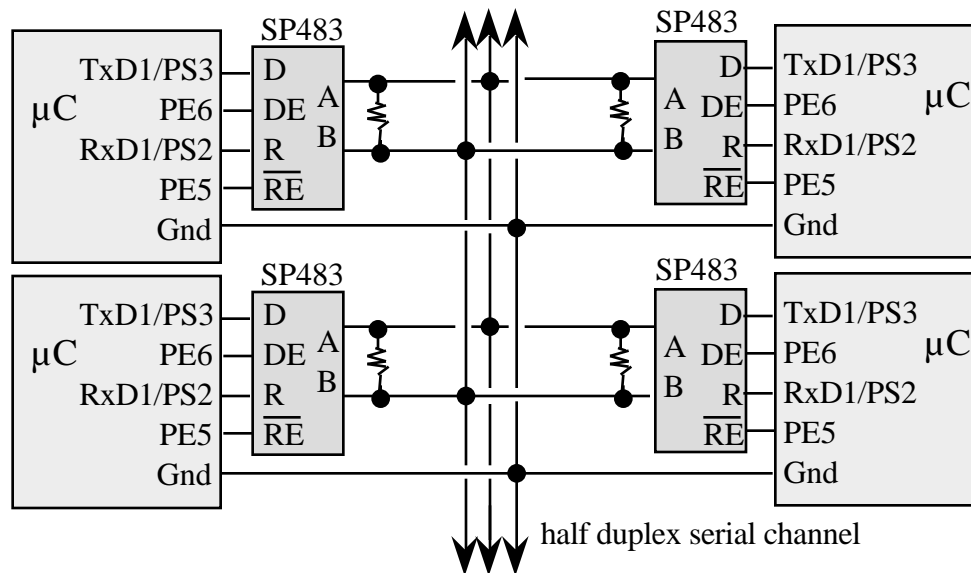


Figure 10.2. Block diagram of a RS485 half-duplex asynchronous serial channel.

The software for this lab will be divided into two parts. The low-level “device driver” software will provide support for initialization, transmitting and receiving individual bytes across the network. Both the receive and transmit I/O threads must be interrupt driven. Two FIFO data structures will link the background threads (receive and transmit interrupts) and foreground thread (the main program and the functions it calls). Collision detection and reporting is built into this layer. At this level collision is detected at the transmitter when the echoed data frame received does not match the data frame transmitted. On the receiver end, a collision may result in a NF, or FE error. Organize this device driver software in **Network.h** and **Network.c** files to simplify reusing it in later labs. Use a naming convention to identify all components of the low-level device driver (e.g., start all function and variable names with Network).

The second part is the main program that tests the low-level network. In this lab, it will accept keyboard input from the PC (**InChar()**). Each character will not be echoed immediately on the PC terminal window. Instead the data received from the network (using your network receive function) will be displayed in the PC terminal window. If all goes well, it should behave like a simple typewriter, because the channel is half-duplex. If both computers attempt to transmit simultaneously a collision error will occur. The objective of this part of the lab is to send single characters between computers. Please note that both nodes on the network are not driven by identical software. Each group writes unique and separate code. The main program will look something like the following. You are free to modify it as long as the overall effect is similar.

```
void main(void){ unsigned char TxData, RxData, ErrorCode;
  NetworkInitialize(); // hardware, software initialization
  while(1){
    if( InStatus() ) { // skip if no PC keyboard input
      TxData=InChar(); // receive input from PC keyboard
      NetworkSendByte(TxData); // begin process of transmitting data
      NetworkSendByte(TxData+1); // transmit a second character data
      NetworkSendByte(TxData+2); // transmit a third data
    }
    if( NetworkRecvByte(&RxData) ) // skip if no incoming network data
      OutChar(RxData); // display on PC terminal window
    if( ErrorCode=NetworkError() ){ // check for network errors
      OutString("The error code is ");
      OutUdec(ErrorCode); // message on PC terminal window
      OutChar(13);
      NetworkReset(); // recover from error
    }
  }
}
```

Note that the **NetworkSendByte()** and **NetworkRecvByte()** do not perform I/O directly to the SCI interface (because they are part of the foreground thread), rather they communicate with the two FIFO's. You could implement the system such that **NetworkSendByte()** and **NetworkRecvByte()** return error codes rather than having a separate **NetworkError()** check error function. It is very important that no I/O to the PC (e.g., **InStatus()**, **InChar()**, **OutChar()**, **OutUdec()**, **printf()**, **OutString()**, etc.) occur in any of the **Networkxxxx()** functions. In

other words, I/O to the PC exists only in the highest most `main()` program of the foreground thread. This modularity provides for easier reuse of the device driver routines.

**Preparation**

Show the required hardware connections. Show the syntax-free software for both the low-level device driver and the foreground main program that tests the network.

**Procedure**

You will not need two PC computers (and two microcomputer boards) until you are very close to finishing the lab. Virtually all your work can be done at one terminal.

You should find another lab group and test your hardware/software system with the other group. IT IS VERY IMPORTANT NOT TO SHARE IMPLEMENTATION DETAILS (just specifications.) YOU MAY NOT SHARE SOFTWARE. The two software solutions must be very different in style and approach. You are not to work as a group of four, and achieve one optimal solution. You can communicate with other groups about channel specifications and network policies (no sharing source code.)

**Checkout**

Demonstrate the transmission of individual characters between two microcomputer systems. You must demonstrate error detection on both sides during a collision. You should create network errors by using a third open collector driver on the transmission line. Use it to send short zero pulses during a transmission (your systems should detect the error). You must demonstrate the detection of various errors to your TA.

**Hints**

- 1) Try using the debugging techniques developed in Lab 2.
- 2) There are interrupting SCI examples on the web site (SCI12.H, SCI12A.C), in the Example/MC6812/Chap7.C file on the TExaS CD and in the ICC12 folder after TExaS is installed (see the TUT4.C example).
- 3) There are FIFO examples on the web site (RXFIFO.C RXFIFO.H), in the Example/MC6812/Chap4.C file on the TExaS CD and in the ICC12 folder after TExaS is installed (see the TUT4.C example).