Keywords: serial, ethernet, network microcontrollers, uController, serial to ethernet, embedded networking, microcontroller, micros

Oct 07, 2002

APPLICATION NOTE 704

Asynchronous Serial-to-Ethernet Device Servers

Abstract: This article explores an easy and economical way to migrate standalone serial devices to the Ethernet by retrofitting legacy systems that build on top of the Tiny InterNet Interface (TINI®) platform using the DS80C390 or DS80C400 network microcontrollers. Once a device connects to the Ethernet, implementing TINI web services such as an HTTP server is very straightforward.

Overview

The sheer number of devices that use a serial port as a means for communicating with other electronic equipment is staggering. In fact, for many, a serial port provides the sole mechanism of communicating with the outside world - including thermostats, point-of-sale systems, remote monitors, barcode readers, receipt printers, RFID transceivers, blood-pressure meters, and many more in fields as diverse as legacy test tools to the latest in building automation. Such devices have no direct means of participating in a larger computer network, yet new applications demand TCP/IP connectivity and Ethernet capabilities. Often, an expensive and time-consuming redesign is not an option.

This article explores an easy and economical way to migrate stand-alone serial devices to the Ethernet by retrofitting legacy systems that build on top of the TINI® platform using the DS80C390 or DS80C400 microcontrollers. Once a device is connected to the Ethernet, implementing TINI web services such as an HTTP server is very straightforward.

RS-232 Serial Port

The asynchronous serial communication discussed in this article is based on the RS-232-C standard that dates back to the earliest days of recorded computer history; RS-232-C was published in 1969[1]. Most modern serial ports do not support all of the signals defined in the standard - and the signals that are implemented are used in a fashion that is merely "fairly close" to that defined in the standard. We'll ignore the purely historical definitions and concentrate on the way RS-232 is used today.

Space and Mark

RS-232-C specifies a voltage level of +3V to +25V as "SPACE" (binary 0) and -3V to -25V as "MARK" (binary 1). The region between -3V and +3V is the "switching region." Many UARTs (Universal Asynchronous Receiver Transmitters) use the more modern (in relative terms) TTL voltage levels of 0V and +5V for 0 and 1. Special-purpose level translators, like the famous MAX-232, convert between TTL and RS-232 levels. Since the serial ports on the DS80C390/DS80C400 are TTL-level, no level translators are needed when interfacing to another TTL-level UART.

DCE and DTE

DCE (data communications equipment) and DTE (data terminal equipment) are the two endpoints of a communications channel. The main difference is the serial connector pinout (a so-called null-modem can be used to convert between the two).

Table 1 shows the signals on a DB-9 DTE serial connector and the corresponding signals on another DTE when using a null-modem.

Table 1. DB-9 DTE serial connector signals

DTE PIN	SIGNAL NAME	NULL-MODEM
1	CD (Carrier Detect)	4 (DTR)
2	RD (Receive Data)	3 (TD)
3	TD (Transmit Data)	2 (RD)
4	DTR (Data Terminal Ready)	6 (DSR) and 1 (CD)
5	Common (Signal Ground)	5 (Common)
6	DSR (Data Set Ready)	4 (DTR)
7	RTS (Request To Send)	8 (CTS)
8	CTS (Clear To Send)	7 (RTS)
9	RI (Ring Indicator)	N/C

Flow Control

Serial communication can be realized by sending on one pin (TD) and listening on another (RD). However, when two devices that communicate over RD, TD transmit at will, one might overrun the other, resulting in data loss. There are two ways flow control is commonly implemented:

- XON/XOFF (often loosely termed software flow control)

- RTS/CTS (often loosely termed hardware flow control)

The XON/XOFF flow control scheme transmits in-band characters that cause the other side to pause (XOFF, 13h) and resume (XON, 11h) transmission. The XON and XOFF characters must be escaped in software by the sender and unwrapped by the receiver if they occur in a binary data stream.

RTS/CTS uses extra signaling lines. RTS (request to send) is asserted by the sender. The receiver responds with CTS (clear to send) when it is ready to receive data and clears CTS when its receive buffer is full.

Some devices support flow control, some don't. The defaults are, therefore, usually set to "no flow control," which should be overridden if a device is known to implement flow control.

Speed, Data Bits, Stop Bits, and Parity

Other parameters that have to be set correctly in order for communications to be successful are, of course, the transmission speed (bit rate), the number of data and stop bits, and the type of parity checking (if any). Most new devices use a setting of "8N1," which means 8 data bits, no parity, and 1 stop bit. However, legacy systems are known to use the full range of possibilities, so the correct setting actually might not be that trivial.

TINI and Networking

TINI (Tiny InterNet Interfaces) is a technology platform developed by Dallas Semiconductor to allow rapid development on the DS80C390 and DS80C400 microcontrollers. Specifically, TINI encompasses a chipset definition, and an embedded operating system integrated with a highly optimized Java™ runtime environment. Using Java, programmers benefit from powerful features not commonly found in embedded development: multithreading, garbage collection, inheritance, virtualization, cross-platform capabilities, powerful networking support, and, last but not least, a multitude of free development tools. TINI users are usually shielded from

assembly language coding. However, native language subroutines are supported and encouraged to optimize speed-critical paths or low-level hardware access (the TINI operating system is written in native code, resulting in serial I/O throughput not significantly different from modern PCs).

In addition to full support of the *java.net* package, the TINI Java runtime also contains an implementation of the *javax.comm* subsystem. Since both TCP/IP and the serial ports are effortlessly accessible from Java, the TINI system easily lends itself to implementing Serial-to-Ethernet bridges.

The TINIm390 verification module on an E10 socket (also called DSTINIS-005) used in the following examples is the hardware portion of the DS80C390 TINI development platform (the TINIm400 uses the DS80C400). In addition to SRAM, flash memory, Ethernet, CAN-bus, 1-Wire®, etc., the system also has four serial ports; two of the UARTs are internal to the DS80C390 (called *serial0* and *serial1*). Two ports are external (using a 16550 build option). It is important to note that both serial connectors on the E10 socket are wired to *serial0* and just differ in DTE/DCE pin assignment.

The TINI environment is documented in great detail in *The TINI Specification and Developer's Guide* (Addison-Welsey, 2001). A PDF copy can be downloaded from www.maxim-ic.com/TINIguide.

Examples

We'll start with two concrete applications and then present a short excerpt from a generic Serial-to-Ethernet program that can be modified to suit almost any particular application. The examples are built using the TINIm390/400 verification modules.

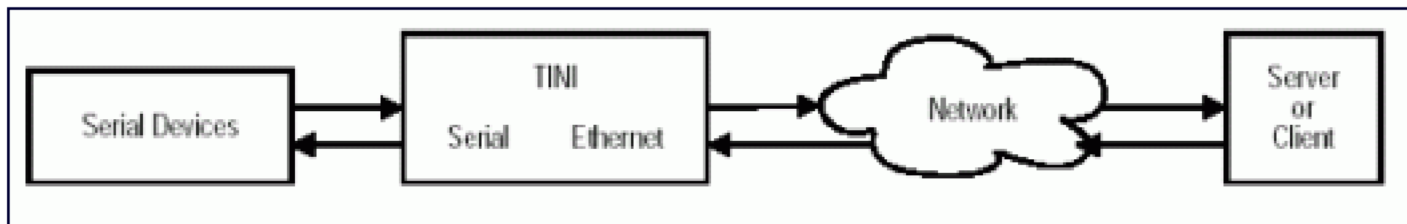

Figure 1.

The TINI verification module can be used as "black box" to connect multiple serial devices to the Ethernet. Depending on the needs of the end equipment, the TINI can either pass the data straight through or parse, interpret and modify the data stream.

Note that although you can run the examples from the *slush* developer's shell on the TINIm390/400, a more polished application would reside in flash, be self-starting in the event of a power loss, and use other TINI construction techniques to make the finished product virtually indestructible.

Some basic networking knowledge and programming experience are required to be able to modify the examples. Working sample code is also downloadable from the Dallas ftp site.

Virtual Modems

The first example, a "Virtual Modem,"[2] uses the TINIm390/400 to replace a physical modem and telephone line with TCP/IP connectivity. Assume a legacy device like a factory "machine status monitor" that uses a modem to dial into a central server several times a day to report machine status, load and efficiency data. To eliminate the need for an ever-growing modem bank on the server side and to be able to use an existing LAN instead of phone lines to the equipment, one could

- rewrite the server software to be TCP/IP based and

- use TINI virtual modems to replace the original modems at each machine

The machine status monitors, however, don't have to be modified since the virtual modem behaves like a real

modem as far as the end equipment is concerned!

Virtual modems can of course also be used in pairs instead of the configuration described above. When using two virtual modems, no server software needs to be changed at all and the TINI modules are a drop-in replacement for existing modems.

Behind the scenes, a virtual modem establishes a TCP connection whenever it receives the "ATD" modem dial command. An "ATH" disconnect command closes the TCP connection. The software also implements a number of other classic AT modem commands and is recognized as a true modem by Microsoft® Windows® networking, for example. In addition, a virtual modem listens on a TCP port itself and can answer incoming "calls" that are signaled by a "RING" to the end equipment.

The following code fragments show how to initialize a serial port on the TINIm390:

```
public static void main(String args[])
{
   TINIOS.setSerialBootMessagesState(false);
   TINIOS.setDebugMessagesState(false);
   TINIOS.setConsoleOutputEnabled(false);

   System.out.println("Connecting to serial0 at 9600bps, "
                      "listening on TCP port 8001");

   try {
      CommPortIdentifier portId = CommPortIdentifier.getPortIdentifier("serial0");
      SerialPort port = (SerialPort) portId.open("VModemTINI", 10000);

      TINIOS.setRTSCTSFlowControlEnable(1, false);
      TINIOS.setRTSCTSFlowControlEnable(0, true);

      TCPSerialVirtualModem modem = new TCPSerialVirtualModem(port,
                            /* Comm speed */ 9600, /*TCP Port */ 8001);

      modem.processInput();
   }
   catch (Exception e) {
   System.out.println("Exception: "+e.toString());
   }
}
```

The code first disables all TINI OS debug output, standard practice on the TINI. After getting a port identifier, the port is then opened (the second parameter tells open how long to wait if the port is currently in use by another application). Next, the state of the hardware flow control is set. Since the TINIm390 only has one set of RTS/CTS lines for serial ports 0 and 1, a program should always disable flow control on the other port before enabling it on the desired port. Next, a Java virtual modem is instantiated.

The virtual modem class consists of an AT command interpreter (not shown here, although by far the largest part of the example) and networking code. The following code sets the serial port bit rate, data and stop bits as well as parity and shows how easy it is to handle inbound connections:

```
/** Creates a new VirtualModem connected to a serial port on
 * one end and a TCP port on the data side.
 * serial -- the serial port this VirtualModem talks to.
 * speed -- the speed the serial port should be set to.
 * tcpport -- the TCP port this VirtualModem listens on.
 * throws IOException when there's a problem with the serial or TCP port.
 */
public TCPSerialVirtualModem(SerialPort serial, int speed, int tcpport)
       throws IOException
```

```
{
   super(serial);

   try {
      serial.setSerialPortParams(speed, SerialPort.DATABITS_8,
                                 SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
   }
   catch (UnsupportedCommOperationException e) {
      throw new IOException();
   }

   ...

   serverSock = new ServerSocket(tcpport, 1); // backlog of one
   listenThread = new listenInbound();
   listenThread.start();
}
```

Finally, the following excerpt of the `listenThread()` accepts an incoming connection request:

```
public void run()
   {
      int rc;
      Socket s;

      while (running) {
         s = null; // No incoming connection request
         try {
            answered = false;
            s = serverSock.accept();

            // Discard incoming connection if already connected
            if (connected)
               throw new IOException();

            sock = s; // for answer()
            ...
```

UPS Monitor

The second example connects a TINIm390/400 to a serial port of an uninterruptible power supply. The software implements the Network UPS Tools protocol[3], allowing a variety of clients on a variety of platforms to monitor the UPS state and health (this project originated from the need to monitor an existing UPS from a new Macintosh computer without any serial ports).

There are two basic kinds of UPS devices: So-called "smart" ones and simple (or "dumb") ones. A simple UPS signals its state on several serial pins, it does not actually send any ASCII data. Due to the fact that there are not terribly many serial pins, it can only report a very limited set of information, for example:

SIGNAL	MEANING
RTS (*from UPS*)	Low Battery
TD (*from UPS*)	On Battery
CTS (*from UPS*)	Kill UPS Power

The `javax.comm.notifyOn...()` methods can be used in Java to easily implement code that reacts to status changes, for example:

```
    ...
    // Listen for DTR changes
    try {
       port.addEventListener(this);
    } catch (TooManyListenersException e) {
       ...
    }
    port.notifyOnDSR(true);
    ...

public void serialEvent(SerialPortEvent ev)
{
    try {
       if (ev.getEventType() == SerialPortEvent.DSR)
          ...
    } catch ...
    ...
}
```

A smart UPS is more interesting, since it implements a serial protocol and can return values like the battery charge percentage or the temperature. Protocols are vastly different between different vendors and, often enough, undocumented.

The following code shows how to receive UDP requests and send out UPS status information over UDP.

```
// Listen to incoming UDP requests
private class listenUDPThread extends Thread
{
    private DatagramSocket sock;
    private byte[] buffer;
    private DatagramPacket dp;

    public listenUDPThread(DatagramSocket s)
    {
       sock = s;
       buffer = new byte[BUF_SIZE];
       dp = new DatagramPacket(buffer, buffer.length);
    }

    public void run()
    {
       while (running) {
          try {
             sock.receive(dp);
             byte[] data = parseCommand(buffer, dp.getLength());
             sock.send(new DatagramPacket(data, data.length,
                       dp.getAddress(), dp.getPort()));
          }
          catch (Exception e) {
          }
       }
       try {
          sock.close();
       }
       catch (Exception e) {
       }
    }
}
```

Due to the powerful networking support built into Java, this example is almost self-explanatory. The code in the `while()` loop waits until it receives a UDP request, parses it and sends out an answer to the originator of the request (using `getAddress()` on the incoming packet).

Generic Serial-to-Ethernet Application

A complete Serial-to-Ethernet example is beyond the scope of this article. (A complete example is shown and explained in the *The TINI Specification and Developer's Guide*.) However, the following code fragment shows how to efficiently use multithreading to transfer data between the serial and networking portions of a Serial-to-Ethernet bridge. The serial and TCP ports are abstracted as Input/OutputStreams dataIn and dataOut, so this layer of the code does not actually need to know anything about the network at all and could also bridge data between the CAN and 1-Wire, for example.

```
public GenericBridge()
{
   ...
   running = true;
   dcThread = new dataCopy();
   dcThread.start();
}

// Thread that copies everything from dataIn to dataOut
private class dataCopy extends Thread
{
   public void run()
   {
      int r = 0;
      while (running && r >= 0) {
         try {
            synchronized (threadLock) {
               r = dataIn.read(dataBuffer);
               if (r > 0)
                  dataOut.write(dataBuffer, 0, r);
            }
         }
         catch (Exception e) {
            r = -1;
            ...       // Handle error
         }
      }
   }
}
```

Conclusion

Many legacy devices only support asynchronous serial communications, yet current applications demand Ethernet connectivity and TCP/IP networking. Using the powerful Java runtime and the TINI technology on the DS80C390 and DS80C400 microcontrollers, developing a Serial-to-Ethernet converter is easy and can be done in a matter of hours.

References

1. NASA has trouble deciphering computer tapes from this area, so the comparison is valid.
2. Refer to *Application Note 196: Designing a Virtual Modem Using TINI*.
3. See http://www.networkupstools.org/

1-Wire is a registered trademark of Dallas Semiconductor Corp.
Microsoft is a registered trademark of Microsoft Corp.
TINI is a registered trademark of Dallas Semiconductor Corp.
Java is a trademark of Sun Microsystems, Inc.

Dallas Semiconductor is a wholly owned subsidiary of Maxim Integrated Products, Inc.

Application Note 704: http://www.maxim-ic.com/an704

More Information
For technical questions and support: http://www.maxim-ic.com/support
For samples: http://www.maxim-ic.com/samples
Other questions and comments: http://www.maxim-ic.com/contact

Related Parts

DS80C390: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS80C400: QuickView -- Full (PDF) Data Sheet -- Free Samples

AN704, AN 704, APP704, Appnote704, Appnote 704
Copyright © by Maxim Integrated Products
Additional legal notices: http://www.maxim-ic.com/legal