iJET EXPLORE International Journal of Engineering & Technology
CENTRE OF PROFESSIONAL RESEARCH PUBLICATIONS
ISSN 2049-3444

International Journal of Engineering and Technology Volume 3 No. 10, October, 2013

Efficient Networking of TINI for Real-Time Weather Data Logging & Deployment over Ethernet and Serial Communication Links

Vincent A. Akpan[1], Reginald O. A. Osakwe[2] and Amaku Amaku[3]
[1]Department of Physics Electronics, The Federal University of Technology, P.M.B. 704 Akure, Ondo State, Nigeria.
[2]Department of Physics, The Federal University of Petroleum Resources, P.M.B. 1221 Effurun, Delta State, Nigeria.
[3]Department of Computer Science, College of ICT, Salem University, P.M.B. 1060 Lokoja, Kogi State, Nigeria.

ABSTRACT

This paper presents an efficient technique for the networking of TINI for real-time weather data logging and the deployment of the acquired data over Ethernet and hard-wired serial communication links. An electronic instrumentation system for real-time measurement of three weather parameters (namely: temperature, relative humidity and barometric pressure) is designed, constructed and calibrated. The weather parameters are deployed via TINI TBM390 over Ethernet and hard-wired serial links. A novel technique on how to use point-to-point-protocol (PPP) networking with TINI runtime environment for dial-up networking including detailed configuration is also demonstrated for both Linux and Microsoft Windows®XP clients. Several critical issues on the design, implementation and deployment of the proposed weather datalogger system are also presented. The comparisons of the measured and actual (true) values demonstrate the efficiency of the proposed strategy presented in this paper. Finally, all the Java programs developed to implement and demonstrate the techniques proposed in this paper are presented.

Keywords: *1-Wire networking, distributed networking, iButton®, Java communications API, point-to-point-protocol (PPP), real-time embedded networked system, TINI TBM390, weather datalogger.*

1. INTRODUCTION

An embedded system is a special purpose computer system built into a larger device. It is a combination of computer hardware and software and perhaps additional mechanical or other parts, designed to perform a dedicated function [1]. The proliferation of programmable processors in embedded systems has happened largely because of the availability of powerful, inexpensive processors and high density, low cost memory. However, three factors are accelerating this trend. First is the emergence of standard run-time platforms such as Windows® computer environment and Java that simplifies systems programming and foster interoperability especially at the device level [2]. Second is the emergence of integrated software development environments such as Green Hills software and Wind River's Tornado, which simplifies applications development. Third is the integration of embedded systems with the Internet, which simplifies the development, networking and management of distributed embedded systems. An embedded platform that is gaining popularity in the embedded market is Java, an operating system independent platform originally developed for set-top boxes. The java platform has two main components. One is the Java communications Application Programming Interface (Java Comm API) which provides basic knowledge, utility, input/output network and applet services. The second is the java virtual machine which separates java application from the details of the underlying browser, operating system or processor.

The proliferation of the Internet and Web Access has occurred at a phenomenal rate to permeate the societies within which it operates. It has allowed the sharing of ideas, collaboration and the exchange of services and information. Furthermore, the proliferation of the Internet has encouraged the interfacing of engineering applications to the Internet. Nowadays many devices do not have this interface implemented and even the suitability of making this interface internal could be discussed in terms of cost, security and efficiency. Hence, it is very interesting to have a small, low cost, external component to increase the functionality for a wide range of control devices.

The Tiny InterNet Interface (TINI) board model on an E10 socket developed by Dallas semiconductors [3] appears as a suitable platform that satisfies all the above requirements discussed until now. The TINI card was designed to give a voice on the network to many types of devices: from small sensors and actuators to factory automation equipment and legacy hardware. TINI has a built-in Java runtime environment. Java was primarily designed and introduced to facilitate real-time embedded systems development and not the Internet itself [4]. However, Java was not commonly used for embedded systems [5], but today the relative cost of programming rises compared with hardware costs. Obviously, it is cheaper to deploy a hardware/software programming board with full support for Java such as the TINI than writing a program for separate interface. This situation fits perfectly with the Java philosophy [4]. One of the novelties provided by TINI is the support for Point-to-Point Protocol (PPP)

communication [6], [7], [8]. The Point-to-Point Protocol (PPP) provides IP packet transport over a serial link.

The main challenge in setting up the platform for embedded systems development using TINI is tied to installing and setting up the Java Comm API. Once the Java Comm API is not installed and configured correctly, the TINI will not work. However, a systematic way of installing the Java Communication API correctly on Windows® and Linux systems machines together with the initialization and configuration of TINI and Java Comm API for real-time embedded networked application have been provided in [9] and have been adopted in this work.

Although, the TINI board model (TBM) has been developed over a decade ago but its deployment for weather data acquisition is still at the infant stage [6], [10]. This paper attempts to provide an efficient networking of the TINI for real-time online datalogging of three weather parameters namely: relative humidity, temperature and pressure over Ethernet, hard-wired serial and dial-up using point-to-point protocol communication interfaces. The paper is organized as follows. Section 2 presents TINI hardware, implementation and network connectivity. The design, implementation and testing of the real-time embedded weather datalogger is presented in Section 3. Conclusion and recommendation for future directions are given in Section 4. All the programs required to demonstrate the techniques presented in this paper are provided in the appendix.

2. TINI HARDWARE, IMPLEMENTATION AND NETWORK CONNECTIVITY

Tiny internet Interface (TINI) is a platform developed by Dallas semiconductors [3] to provide system designers and software developers with a simple, flexible, and cost-effective means to design a wide variety of hardware devices that can connect directly to corporate and home networks. The TINI platform used in this work is the TINI Board Model 390 (TBM 390) [9]. The TBM 390 is a combination of a small but powerful chip-set and a java programmable runtime environment. The chip-set provides processing, control, device-level communication and networking capabilities. TINI's networking capability extends the connectivity of any attached device by allowing interaction with remote systems and users through standard network applications.

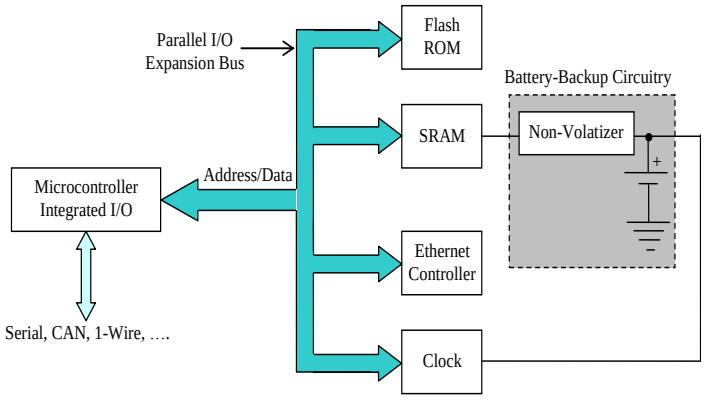

Fig. 1: A full-featured TINI hardware implementation

This configuration, shown in Fig. 1, extends the reach of embedded devices to Ethernet networks. It also provides an accurate time reference for time-stamping purposes. Without the clock, commonly used Java methods such as *java.lang.System.currentTimeMillis* and *java.util.Date* methods that use *currentTimeMillis* return constant, and therefore useless. Peripheral devices such as the Ethernet controller and clock are included into the system's memory map. Another addition that is shown in Fig. 1 is the battery-back circuitry. The battery is a very small, single-cell lithium battery. Both the SRAM and clock used in TINI designs have very low stand-by power requirements, which mean that an appropriately chosen lithium cell will keep the clock running and the SRAM data persistent for over 10 years.

2.1 Protocol Conversion via TINI

TINI is designed to meet the functional requirements for commercial and industrial embedded network applications. However, because of its low-cost hardware and the availability of free software development tools, it is beginning to find a home in the educational and hobbyist arenas as well. TINI can be used for traditional stand-alone embedded tasks such as monitoring and controlling a local device or system, but the majority of applications utilize TINI's networking capabilities. A few applications of the technology include the following: *1). Industrial controls*: TINI's integrated Controller Area Network (CAN) support is instrumental in implementing factory automation equipment, networked switches, and actuators; *2). Web-based equipment monitoring and control*: It can be used for communication with equipment to provide remote diagnostics and data collection for purposes such as monitoring device utilization; *3). Protocol Conversion*: TINI-based systems can be used to connect legacy devices to Ethernet networks. Depending on the I/O capabilities of the legacy system, this may be a job that can be done with a PC or workstation as well. However, TINI can do the job at a fraction of the cost and size; *4). Environmental monitors*: Using TINI's built-in support for 1-Wire networking, an application can query sensors and report the results to remote hosts.

The schematic of Fig. 2 shows a model in which TINI is employed as a protocol converter (or link) between a legacy embedded device and an Ethernet network [6]. The legacy device may communicate with the outside world using an RS232 serial port, Controller Area Network (CAN), or perhaps some type of parallel interface. The Java application running on TINI performs the task of communicating with the attached device in its native language (using a device-specific communication protocol) and presents the results to remote systems reachable via a TCP/IP network. The link provided by TINI is bidirectional, allowing a remote system to control as well as monitor a device.

Furthermore, Fig. 2 focuses on an embedded system that controls and provides network connectivity to a single device. However, TINI can also serve to interconnect various types of networks by bridging the gap between smaller, localized networks of inexpensive and lightweight devices and a "big world" TCP/IP network such as the Internet.

In general, TINI applications interface to other equipment and networks as opposed to humans. Due to the embedded control and I/O-centric nature of most embedded network applications, there is no built-in hardware or API support for a human interface. TINI-based systems often provide a remote display by implementing a network server, such as an HTTP server, allowing the user to interact with the system using a network client such as a Web browser. Local display and data entry can be obtained by interfacing to a PDA over a wireless link such as infrared (IR) or a hard-wired serial link. TINI systems requiring dedicated human interfacing capability can be implemented using liquid crystal displays (LCDs) and keypads.

2.2 1-Wire Chips and iButton: Device and Network Configuration

1-Wire chips provide network connectivity to otherwise mute entities. A 1-Wire network is a collection of one or more uniquely addressable devices that share a single conductor for communication and power. The single conductor is often referred to as a bus. The 1-Wire devices attached to the bus are always slaves. This implies the existence of a master that initiates all communication with the devices. The extremely simple hardware configuration of a 1-Wire network is shown in Fig. 3.

Although; an in-depth treatment of 1-Wire networking is beyond the scope of this work; a thorough treatment of 1-Wire networking can be found in [3], [11]–[13]. This work presents only brief description of the 1-Wire API and examines adapters and containers, the classes that represent them, and how they are used to monitor and control devices on a 1-Wire network.

A 1-Wire Network is a complex arrangement of devices, wire and connections. The radius of a 1-wire network is the distance from the master end to the furthest slave device in meters. The weight of the network is the total amount of connected 1-wire in the network in meters. In general, the weight of the network limits the rise time on the cable, while the radius establishes the timing of the slowest signal reflections. As a rule, no 1-wire network may ever have radius greater than 75m. At this distance, the port will fail due to the time delay of the cable. In practice, however other factors usually limit the radius to smaller values than this [11].

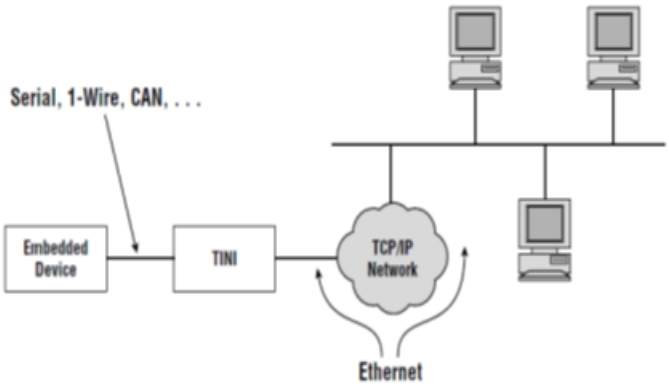

Fig. 2: Protocol conversion via the TINI

2.2.1 1-Wire Network Topologies

At least, three major topologies exist for networking 1-wire devices, namely [12]:

1. Linear topology: The 1-wire bus is a single pair, starting at the master and extending farthest. Slate devices other slaves devices are attached to the pair along its length without significant (> 3m) branches or "stubs";
2. Stubbed Topology: The 1-wire bus is a single main line starting at the master and extends main line starting at the master and extending to the farthest slave device. Other slave devices are attached to the main line through branches stubs 3m or more in length; and
3. Star Topology: The 1-Wire bus is spelt at or near the master end and extends multiple branches of varying lengths, presumably with slave device along or at the ends of the branches. When different topologies are intermixed it becomes much more difficult to determine the effective limitations of the network.

To allow networks to grow in complexity, without growing in weights and radius methods have been devised wherein the network is divided into section that are electronically switched ON one at a time as in switched networks described in [12]. Using 1-Wire switching devices like the DS2009, the network may physically resemble one topology but may electrically resemble another. However, the linear topology scheme is implemented in this work [12].

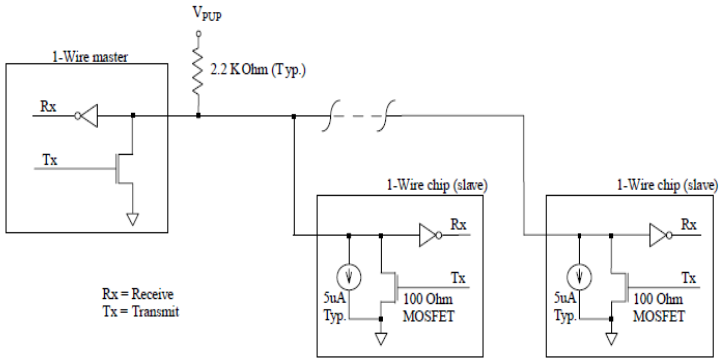

Fig. 3: 1-Wire network hardware configuration

2.2.2 Limitations of 1-Wire Networking

Several factors determine the maximum radius and weight of a network [3], [11]–[13]. Some of these factors can be controlled and some cannot. The master-end interface has a great deal of influence on the allowable size of a 1-Wire network. The interface must provide sufficient drive current to overcome the weight of the cable of the slave devices. It must also perform the 1-Wire waveform with timing that are within specification and are outsized for the change and discharge times of the network and if must provide importance match to the network so that signals are reflected back to the line to interface with other network devices.

When the network is small, very single master-end interfaced are acceptable. Capacitance is low, related energies arrive too soon to pose a problem and cable losses are at a minimum,

but when line lengths becomes longer and more and more devices are connected complex forces come into play and the master-end interface must be able to handle them. Network radius is limited by the timing of wave form reflection and the time delay produced by the cable as well as by resistance of the cable and degradation of signal levels at about 75m, the delay in getting a response from slave at the far end of the cable back to the master is simply beyond the limits of the protocol [11]. Network weight is limited by the ability of the cable to be changed and discharged quickly enough to satisfy the 1-Wire Protocol.

2.2.3 Master-End Interface Device

A number of master-end interfaces for personal computers have been developed over the years and a wide variety of methods have been employed to interface 1-wire networks to microcontrollers, but there has been little consistency in that each master was designed with a different intended use and was not always reliable when pressed into alternative service [14]. The master-end hardware in most common use today includes [14]: DS9097 PC serial port adaptor, DS9097U PC serial port adaptor (DS2480.B-based), DS1410E PC parallel port adaptor, DS9097U-E25 serial port adaptor w/EPROM programming, Microcontroller with slew-rate limited FET and 1kΩ resistor, microcontroller with advanced bus interface, and Microcontroller with DS 2480B bus interface.

2.3 Adapters

The term Port Adapter or simply adapter is used to refer to a 1-wire master. Each 1-wire network has exactly one master that is responsible for initiating all network communication as well as delivering the power and programming pulses required for certain device families. The term adapter is used because 1-wire masters, typically attach to another physical port such as serial, parallel, or USB port and perform a translation between the host port and the 1-wire network it controls.

The *com.dalsem.onewire* package is the root of the 1-wire API. This package contains only two classe: *OneWireAccessProvider* and *OnewireExceptions*. *OneWireAccessProvider* provides the static method *enumeratedAllAdapters*, which returns an Enumeration of all adapters registered with the operating system. The *FindAdapter* program of *Program 1* finds all of the adapters and displays their names. Running the *FindAdapters* program on TINI produced the following adapters:

TINIExternalAdapter
TINIInternalAdapter

Both *TINIExternalAdapter* and *TINIInternalAdapter* are subclasses of *DSPortAdapter*.

Often if an application knows the target adapter's name, then the adapter instance can be created directly using the *CreateAdapters* given in *Program 2* and this approach is preferable when compared to the *FindAdapters* of *Program 1*. Running the *CreateAdapters* (i.e. *Program 2*) on TINI produces the same result as *FindAdapters* of *Program 1*. The internal adapter is so named because its physical interface is simply one of TINI's microcontroller's port pins and therefore can never be omitted from any hardware implementation. This adapter is used by the operating system during the host process to read Ethernet address stored in the EPROM of the on-board 1-Wire Chip.

The external adapter uses a serial to 1-Wire converter that is attached auxiliary serial port (*Serial 1*) of TINI's microcontroller. All TINI hardware designs include the external adapter chip. The external is a full-featured port adapter capable of controlling 1-wire networks that cover a large area and potentially have many attached devices. The two 1-wire adapters supported by TINI can be encapsulated in an adapter instance and can therefore be determined programmatically i.e. determining an adapter capabilities as shown *AdapterFeatures* in *Program 3*.

The Adapter features program creates instances of *TINIInternalAdapter* and *TINIExternalAdapter* and queries both for their capabilities. Running this program on TINI produces the following results:

Internal Adapter:
Supports Overdrive Speeds – true
Supports Flexible timing – false
External Adapter
Supports overdrive speeds – true
Supports flexible timing –true

2.3.1 Searching for 1-Wire Devices

One of the major roles served by an adapter is managing the address discovery (or search) process by which the address of every device attached to the network is discovered. The super class all adapters, *DSPortAdapter*, contains several methods used to configure and execute the discovery process. The method *getAllOnWire* Devices returns an enumeration of *OneWireContainer* objects.

Since different adapters provide different interfaces to control the search process, *FindFirstDevice* and *FindNextDevice* are abstract, forcing the super classes that encapsulate real adapters to implement the search process. To discover the adapters o all devices attached to the network, an application invokes *FindFirstDevice* and then invokes *FindNextDevice* repeatedly until *FindNextDevice* returns false. The *FastCensus* program of *Program 4* discovers and displays all of the devices attached to the default adapter including Thermocron ibutton.

Execution of fastCensus on TINI generates the following results without the overhead of creating an Enumeration of *OneWireContainer* Objects:

1 – Wire net address:
F300000018A4BC12
8F00000018A37A12
AD00000018A51612
D600000018A37912
3D34C00000609F21

It is important to realize that invoking *FindFirstDevice* does not typically return the address of the 1-Wire device that is physically nearest to the adapter. The ordering of device discovery is logical, not physical.

2.4 Containers

The container classes provide high-level access to the services offered by specific families of devices shielding the programmer from the low-level details of possibly complicated communication protocols. The *OneWireContainer* class, defined in the *com.dalsemi.onewire.container* package, is the super class of all devices specific containers and implements default functionally that is shared by all 1-Wire devices, specifically the address. It also provides methods for identifying and describing the devices in a textual form.

All 1-Wire device families are represented by a subclass of *OneWireContainer* and also exist in *com.dalsem.onewire.container* package. The name of the container is simply a hexadecimal string representation of the device family identity (id) appended to *OneWireContainer's* fully qualified class name.

An instance of *OneWireContainer* (or a subclass) maintains a reference to its parent adapter that is used for all communication with the device. Typically, one an application has used the adapter's search capability to obtain containers of the devices in which it is interested; all further communication with the device goes through the container, not to the adapter. However, while containers provide a very useful abstraction from the low-level device details, there are at times when it is better to avoid the overhead of containers and communicate directly with devices. Several methods in DSPortAdapter class provide the support necessary to communicate with any 1-Wire chip.

OneWireContainer objects are created by invoking any of the methods on an instance of a subclass of *DSPortAdapter*. The *DSPortAdapter* methods that create container objects use *forName* and *newInstance* of Class. The *FindContainers* of *Program 5* is essentially the same as the *FastCensus* of *Program 4* except it uses *getfirstDevice* and *getNextDevice* to obtain the device containers.

Running *FindContainers* on the same 1-Wire network as *Program 4* outputs each chip's address as well as a short description and format part name.

DS2406, Dual Addressable Switch at address F300000018A4BC12
DS2406, Dual Addressable Switch at address 8F00000018A37A12
DS2406, Dual Addressable Switch at address AD00000018A51612
DS2406, Dual Addressable Switch at address D600000018A37912
DS1921, Thermocron at address 3D34C00000609F21

Note that on TINI, device specific container classes are not available by default as part of the API. They must be included with the application during the conversion process.

2.5 Transmission Control/Internet Protocol (TCP/IP) Networking

TINI's main objective is to provide a powerful a platform for developing embedded applications that connect non–networked devices to the network. TINI's broad networking is its most compelling feature, and Java suitability for writing networked applications is one of the primary reasons TINI provides a Java runtime environment. This project does not cover a general treatment of TCP/IP networking or writing network applications in Java rather it focuses primarily on programming for TINI's networking environment. Six application layer protocols are supported in the TINI API as shown in Fig. 4.

All of the application layer protocols except *ping* are implemented using the socket classes in the java.net packages as the network transport mechanism. Ping is not really a protocol but it is and application wrapper over a subject of ICMP. The ping class directly, invokes native methods that are exposed in the network stack;s ICMP module. Support for most of the application layer protocols is provided by the sub-packages of *com.dalsemi.tininet* which includes *com.dalsemi.tninet.http, com.dalsemi,tininet.icmp, com.dalsemi.tininet.dhcp, com.dalsemi.tininet.dus*. The FTP and Telnet protocols are implemented the *com.dalsemi.shell.cerver* ftp and *com.dalsemi.shell.server.telnet* respectively. Both FTP and Telnet are implemented as servers and are typically used by system shells such as slush support for using FTP as a client is of course available using the URL classes in the java.net package. The protocols in Fig. 4 are those for which the TINI networking API provides supports. Other networking protocols can be written in Java and can run on TINI with little or no changes to the codes.

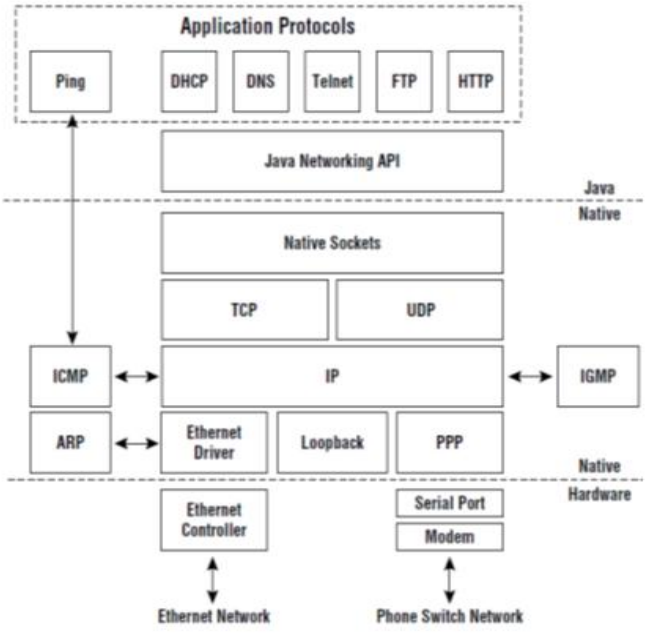

Fig. 4: Network protocol stack

2.5.1 The Network Interface

TINI supports three distinct network interface types i.e. Ethernet, PPP over a serial link and a loopback. There are a maximum of four network stack for the sake of viewing the configuration of individual interfaces, they have been assigned number starting with 0. Interface 0 is always the Ethernet interface (eth0). Interface 1 is always the loopback (L_o) interface, and interfaces 2 and 3 are available for use by up to simultaneous PPP (Point-to-Point Protocol) connections. The state of all the network interfaces can be queried using the slush command "*ipconfig –x*". The *SimpleEthernetAddressReader* of *Program 6* reads the Ethernet address directly from its storage in the on-board 1-Wire chips and produces the result in a trivial fashion.

Program 6: SimpleEthernetAddressReader

```
import com.dalsemi.tininet.TININet;
class SimpleEthernetAddressReader {
        public static void main(String[] args) {

        System.out.println(TININet.getEthernetAddress());
        }
}
```

Executing *Program 7* command gives the following results:

```
TINI /> java SimpleEthernetAddressReader.tini
0:60:35:0:55:27
```

Just like the Ethernet driver, the point-to-point protocol (PPP) module a packet-oriented interface to the IP layer. PPP is actually a very broad and flexible protocol that allows for the transmission of various types of network packets over various types of physical links such as serial, parallel, and even Ethernet. However, on TINI, PPP is used strictly as a mechanism to transmit IP datagrams over a serial link. PPP uses one of the serial port drivers for all data transmission. Usually a modem is attached to the serial port to support dial up networking applications.

The loopback interface (L_o) allows a server and (potentially multiple) clients to communicate on the same host possibly different process. The class network id 127 is reserved for use by the loopback interface with a subnet mask of 255.0.0.0 so that all of the IP address between 127.0.0.1 and 127.255.255.255 are valid IP addresses and use the loopback interface. No IP traffic destined to a 127 address is transmitted to any physical network interface. TINI follows the convention of binding the name "*localhost*" with the IP address 127.0.0.1. Using the slush "ping" command to ping local host produces the following output.

```
TINI/> ping localhost
Got a reply from mode localhost/127.0.0.1
Sent 1 request(s), got 1 reply (s)
```

The loopback interface is generally used by the network stack implementers for taking purposes, but it can also be used by multiple applications running on the same host as a mechanism for IPC. (Inter-Process Communication).

2.5.2 Setting Network Parameters

On non-embedded hosts, networking applications does not need to worry about configuring basic network parameters such as the IP address and subtract mask. This task is performed by a network administrator using the operating system (OS). During application development this is also true for TINI. The network settings are established using the slush *ipconfig* command. Ultimately, however, if an application is to control the entire system, it will replace slush as the Java application that is lunched automatically when the systems boots. In this case the application must be able to query and configure network parameters.

The following are brief description of networking parameters configured using *TININet* of the *TININet* class in the configurable using *TININet* of the *TININet* class in the *com.dalsemi.tininet* package.

IP address: A 32- bit integer that encodes the host's network identification as well as the identification of the host on that network. Every host on an Internet has a unique IP address. Each network interface has its own IP address.

Subnet Mask: A 32 – bit integer used by the TCP/IP protocol stack as a bit mask to separate the network and host portions of the IP address.

Gateway (Router) IP Address: The IP address of the default router. A router is (usually) dedicated machine connected to at least two networks that forwards IP datagrams between the various networks.

Primary DNS Address: The IP address of the preferred DNS (Domain Name System) server. A DNS server resolves IP address to human readable host names and vice-versa.

Secondary DNS Address: the IP address of an alternate DNS server. If a request to a primary DNS server is unanswered, the DNS client implementation will send the request to the secondary DNS server.

DNS time – out value: The amount of time (in milliseconds) that the DNS client will wait for a response from DNS server before timing out and possibly retransmitting the request.

Domain Name: A string representing the domain name (for example, "*dalsemi.com*").

DHCP Server IP address: The IP address of the DHCP (Dynamic Host Configuration Protocol) server. A DHCP client can obtain several network parameters without knowing anything other its own Ethernet address in advance. One of the biggest improvements with DHCP is that IP Addresses are assigned dynamically from a predetermined pool of available addresses. *DHCPClient* does not begin negotiating for an IP address until the *start* method, inherited from Thread, is invoked. The datagram requests the following parameters from any DHCP server listening on the network. An IP address, the subnet mask, the default gateway (router) IP address, primary and secondary DNS server IP addresses, and the mailhost IP address.

Hostname: A string representing the local host's (not localhost) name. The host's name is not necessarily the same as its DNS name.

Mailhost: The IP address of the machine running a SMTP (Simple Mail Transfer Protocol) server. This must be set to use the mail to protocol supported by the Java URL (Uniform Resource Locator) classes.

HTTP proxy server: The IP address of a machine that forwards HTTP (HyperText Transfer Protocol) requests as desire. If the TINI is behind a firewall, using a proxy server may be required to satisfy HTTP requests of host outside of the Local network. The class *HTTPServer* in the *com.dalsemi.tininet.http* package implements a very simple HTTP server. It supports only HTTP GET requests and serves up static information contained within files at or below a specified root directory. It is not intended to be a dedicated web server application but rather provide HTTP serving capability that an application can launch and forget about. Because *HTTPServer* is very small and simple by design, it doesn't meet every application requirements as a general purpose web server except as a Web server for this project. Applications that need access to information provided by web servers use the familiar URL classes in the java.net package. There is one additional configuration parameter that can be set by an application using the URL classes: a proxy server. Often corporate networks are protected behind a firewall and the only way HTTP requests can be reach the Internet is through a proxy server. A proxy server is simply a machine that receives requests from a client and forwards them to another server. The proxy server has special privileges to communicate with hosts outside the firewall.

HTTP proxy Port: A 16 – bit integer that specifies the port number on which the HTTP proxy server expects to receive HTTP requests.

Next, an "*nslookup*" application is created using just the *InetAdress* class. The *getAllByName* method is used to generate an array of all DNS entries for the given input using the *DNSTest* application program given in Program 7. They are displayed using the *InetAddress.tostring* method, which generates a string containing both the host name and IP Address. Before running this program on TINI, the primary DNS server is set and following output shows the IP Address of the *iButton* Web server.

TINI /> javaDNSTest.tini.www.ibutton.com
www.ibutton.com/198.3.123.121

If the *DNSTest* program is executed again on TINI but with a host name as "*bogus*", *getAllByName* and throw an unknown *HostException*.

TINI /> java DNSTest.tini bogus.ainthere.com

Lookup error: Could not find an entry for bogus.airthere.com

The *SetProxyServer* method takes the server name as a *string* representing either an IP Address encoded in dotted – decimal notation or a DNS name. The *setProxyPort* method takes an integer value specifying the 16–bit port number on which the proxy server receives HTTP requests. Both the server and port are persistent across system reboots. If *setProxyServer* is invoked with an empty *string*, it will disable the use of a proxy server. By default, the URL protocol handling classes do not use a proxy.

2.5.3 Developing the Mini-Browser

The *MiniBrowser* program given in *Program 8* reads the contents of a URL through a proxy server Mini-Browser requires that the URL, proxy server name (or IP Address), and proxy port be specified on the command line. After setting both proxy server and proxy port, it opens a connection to the specified URL. In this test configuration one TINI is running *FluidPressureSensor* HTTP server application and another TINI (behind a firewall) is running *MiniBrowser*. The only way for HTTP requests to escape the firewall is through the proxy server named Wally on port 576.

TINI />Java MiniBrowser.tini http://198.3.123.182/index.html Wally.dalsemi.com 576

```
<Html>
(Head>
<Title> FluidPressureSensor</Title>
</Head>
<Body>FluidPressureSensor Result </h1>
<h1>
</Body>
</Html>
```

The Internet control Message Protocol (ICMP) is the mechanism used by nodes on a TCP/IP network to transfer error and control information. Even though ICMP messages often provide error information regarding IP datagrams, they travel encapsulated within IP datagrams. ICMP is used by routers to transmit error messages and by hosts, like TINI, to determine the reachability of a remote destination.

The *pingNode* method effectively provide one bit of information: if the remote node was reachable it can be used to programmatically determine whether just a particular service running on the remote has died or whether the host machine itself has become unreachable on the network, providing for more precise error reporting.

The *Pinger* shown in *Program 9* uses a *pingNode* to create a ping type of application that is much more useful than the slush ping command. It requires the remote node, number of ICMP echo requests to be transmitted, and the TTL for the echo requests to be specified on the command line.

If *pingNode* returns a negative value, then the specified remote node is unreachable. Any non-negative return value indicates that some type of message was received in response to the outbound ICMP echo request message and the response array contains an IP datagram carrying the response. *Pinger*

computes the length of the IP header and uses that value as an offset to extract the ICMP type byte.

3. DESIGN, IMPLEMENTATION AND TESTING OF THE REAL-TIME EMBEDDED WEATHER DATALOGGER

This section presents the hardware and software development process that captures and logs data implemented on a TCP/IP network server, making the data available to remote clients. Ultimately, the server accepts connections over both Ethernet and the PSTN (Public Switched Telephone Network) using PPP to manage dial–up connections. Support for dial–up networking is primarily what makes the datalogger truly remote; this allows access to any client computer anywhere in the world with Internet access without requiring the presence of an Ethernet network at the data collection site. It assumes nothing more than a serial modem and a connection to the public phone network. It is important to note that these data will be collected over Ethernet at the collection site before being retransmitted to the Internet for any client in the world. Any interested client can access the web site over both Ethernet and PSTN why PPP to manage the dial–up connections.

The actual data collected by the application is not terribly important. The main point here is some sensors or other physical devices (or possibly multiple devices) and uploaded it to any interested client over a TCP/IP network. For this reason, the software framework used for data collection will be relatively general purpose and reusable to allow for collecting data from other types of devices.

The data logging application consist of several classes. The class contains the *"main method"* is in a class called *Datalogger*. The entire application (project) will be referred to as "*DataLogger*" as this is the name of the binary that will be executed on TINI. At this stage in the project development, the *DataLogger* combines three major concepts described previously which includes TCP/IP networking, serial communications and 1-Wire networking.

3.1 The Design and Construction of the Relative Humidity, Temperature and Pressure Datalogger System

Although, there are over 20 *containers*, one for each device family; but here we consider the *container* from one of the more interesting 1-Wire device families, the DS2438 A/D (Analog to Digital) converter [15]. The DS2438 includes an A/D converter, a temperature sensor, an elapsed time meter, and 40 bytes of nonvolatile memory. The practical uses for a device that can measure analog voltages and currents as well as sense temperature are nearly unlimited.

We would expect a *container* designed to encapsulate the DS2438's behavior to provide simple methods for accessing the memory, reading the current temperature, and returning the voltage read on its A/D pin. For our purposes we'll need to be able to read the temperature and the input voltage on the V_{ad} pin as well as the supply voltage (V_{dd}). *OneWireContainer26* (the DS2438 has a family *id* of 0x26) provides the following methods to serve these purposes:

public void doADConvert(int channel, byte[] state)
throws OneWireIOException, OneWireException
public double getADVoltage(int channel, byte[] state)
throws OneWireIOException, OneWireException
public void doTemperatureConvert(byte[] state)
throws OneWireIOException, OneWireException
public double getTemperature(byte[] state)

Both the voltage and temperature measurements are split into two phases: performing a conversion and reading the result. So, for example, to read an analog voltage from the DS2438, an application invokes *doADConvert*, followed by *getADVoltage*. The channel parameter of the *doADConvert* method allows the caller to specify which analog voltage is desired. Note that here we need to read both V_{ad} and V_{dd}.

This work uses a DS2438 and its associated container to create a 1-Wire network for relative humidity, temperature and barometric pressure measurements. The circuit diagram for the humidity sensor is shown in Fig. 5. This circuit uses a core Honeywell's HIH-4602 humidity sensor [16] that outputs an analog voltage that can be used in conjunction with the supply voltage and temperature to calculate the relative humidity using Eq. (1) and Eq. (2). The DS2438 provides a 1-Wire communication interface for the composite sensor as well as the analog-to-digital conversion and the temperature measurement. The schottky diode, D_2, is used to protect the circuit from negative voltages greater than about 400 millivolts in magnitude. Schottky diode D_1 and capacitor C_2 are used to build a parasite power supply that "steals" energy from the bus during high periods. Finally, R_3 and C_3 serve as a low-pass filter.

The output of the humidity sensor is an analog voltage proportional to the supply voltage. From the HIH-4602 datasheet [16], the relative humidity at 25˚C can be computed with respect to the supply voltage (V_{dd}) using Eq. (1).

$$RH_{sensor} = \frac{\frac{V_{ad}}{V_{dd}} - 0.16}{0.0062} \quad (1)$$

Of course, what we're really interested in is the true relative humidity without any dependence on the supply voltage or a fixed temperature. Equation (2) provides the means to compute the true relative humidity.

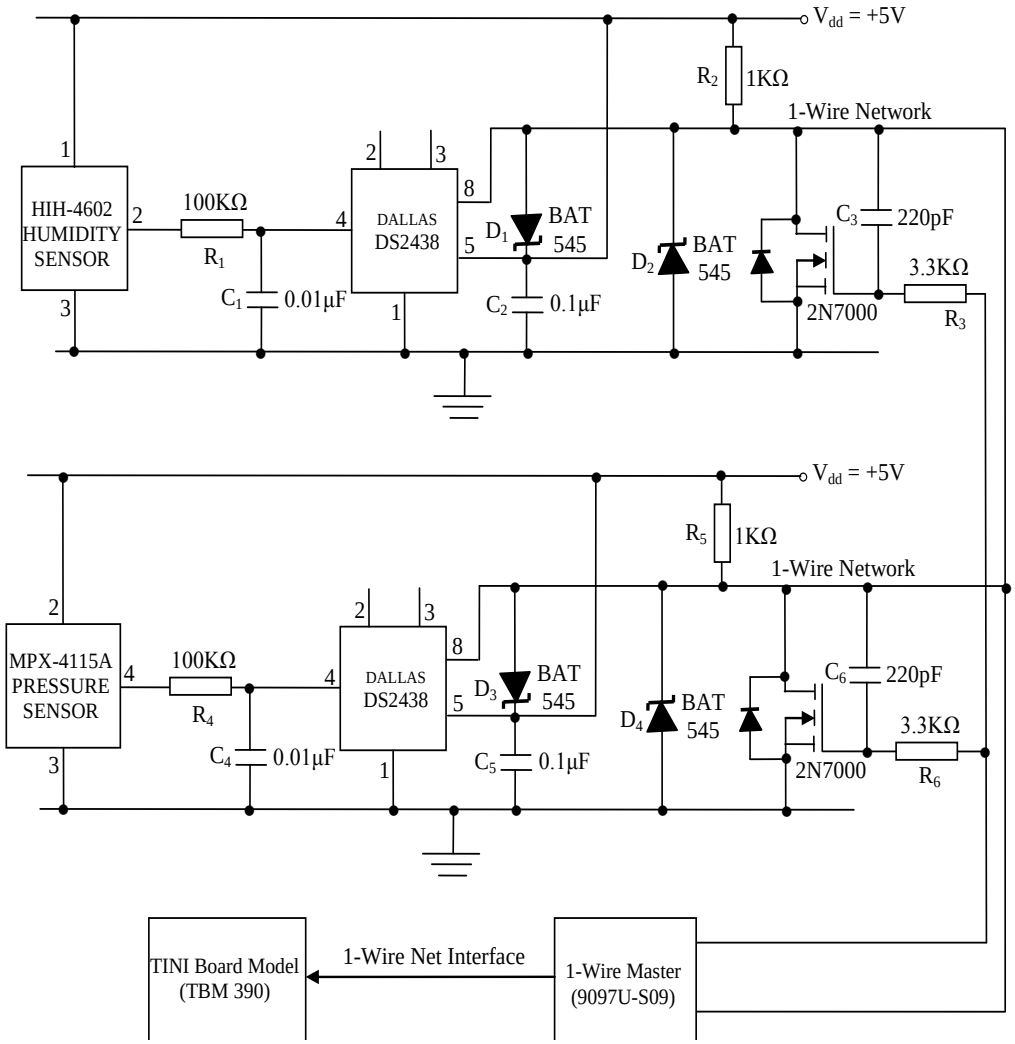

Fig. 5: Block diagram of the electronic instrumentation system for real-time online relative humidity, temperature and barometric pressure for measurements

$$RH_{true} = \frac{RH_{sensor}}{1.0546 - 0.00216 \times T} \quad (2)$$

where T is the temperature measured in degrees Celsius (˚C). From these equations we can see that we need three measurements to compute a value for RH_{true}, V_{ad}, V_{dd}, and T. Fortunately, the internal A-to-D converter inside DS2438 can measure both V_{ad} and V_{dd}.

The pressure measurement is achieved using the Freescale's MPX4115A which is an integrated silicon pressure sensor altimeter/barometer pressure sensor on-chip signal conditioned, temperature compensated and calibrated [17]. The connection diagram as implemented in this work is shown in Fig. 5. The detailed architecture for the MPX4115A can be found in the datasheet [17]. The schottky diode, D_4, is used to protect the circuit from negative voltages greater than about 400 millivolts in magnitude. Schottky diode D_3 and capacitor C_5 are used to build a parasite power supply that "steals" energy from the bus during high periods. Finally, R_6 and C_6 serve as a low-pass filter.

Again, the output of the pressure sensor is an analog voltage proportional to the supply voltage. From the MPX4115A datasheet [17], the relative humidity at 25˚C can be computed with respect to the supply voltage (V_{dd}) using Eq. (3).

$$P_{sensor} = \frac{\frac{V_{ad}}{V_{dd}} + 384.62}{0.094} \quad (3)$$

Of course, what we're really interested in is the true barometric pressure without any dependence on the supply voltage or a fixed temperature. Equation (4) provides the means to compute the true relative humidity.

$$P_{true} = \frac{P_{sensor}}{6.691 - 0.2 * T} \quad (4)$$

where T is the temperature measured in degrees Celsius (˚C). From these equations we can see that we need three measurements to compute a value for P_{true}, V_{ad}, V_{dd}, and T. Fortunately, the internal A-to-D converter inside DS2438 can measure both V_{ad} and V_{dd}.

Please, note that each DS2438 has unique *id* and as such the V_{ad} measure for relative humidity is different from that measured for the pressure sensor. Now that we have all of the

information required to write the complete *WeatherSensorProg* program shown in *Program 16* which reads the temperature, relative humidity and pressure using *OneWireContainer26*; next we present the implementation of the temperature, relative humidity and pressure datalogger.

3.2 Implementation of the Temperature, Relative Humidity and Pressure DataLogger

DataLogger implements the network (TCP/IP) server and accepts and manages in bound connections from remote clients. A complete *DataLogger* class including its constructor is shown in *Program 10*. During construction of *DataLogger* an instance of *DataLogger* is created specifying the sample count and delay time in seconds between samples. *DataLogger* requires the number of data readings to be maintained and the delay in seconds between each reading to be specified on the command line. The skeleton of the *DataLogger main's method* is shown in *Program 11*. After extracting the samples and count values from the command line, the *main method* creates a new *DataLogger* object, which also creates a new thread of execution. After constructing the new instance of *DataLogger*, the start method is invoked to kick off the server.

The server spends eternity in the *run method*, processing network connections. *DataLogger's run method* along with the inner class *LogWorker* is shown in *Program 12*. It starts by creating a *ServerSocket* object to listen for inbound connections from remote clients. As implemented in this work, the *DataLogger* uses a port number of 5588 but the application can easily be modified to use a port number specified on the command line. After the *ServerSocket* object is created, the *run method* enters an infinite loop that accepts and process inbound clients connections. To keep the logging classes reasonably general purpose and reusable, an abstract class will be created called LoggingDaemon1 to drive the data collection process.

Subclasses of *LoggingDaemon* implement the *CaptureSample* method to handle the details of collecting a single data sample. The *LoggingDaemon* thread is set to a daemon thread and when the last non–daemon exists, *LoggingDaemon* will exist, along with any other daemon threads, allowing the process to terminate. This is because; there is no point in continuing logging data if there is no server running to allow clients to download it. The *LoggingDaemon constructor* is shown in *Program 13* which requires the maximum number of samples to be held in the samples vector along with the delay between consecutive samples.

LoggingDaemon's run method is given in *Program 14*. As long as the *stopLogging* method is not invoked, the *run method* spins in an infinite loop collecting data samples at the specified interval. If the *CaptureSample method* returns null, there is no change in samples. The *run method* simply goes to sleep until it is time to try another sample. *LoggingDaemon's Writelog* method in *Program 15* is invoked by the server when a client establishes a connection with the server, requesting a log of the recent data samples.

The complete *WeatherSensorProg* program shown in *Program 16* captures and read the relative humidity, temperature and pressure from the circuit of Fig. 5. However, since there is need to encapsulate the individual data readings within an object *WeatherSensorSample* class shown in *Program 17* is created. This class is just a thin wrapper on the sample data that provides public "get" methods for the individual fields. Human Sample's constructor takes the readings attained using the *WeatherSensorProg* class and stores them in the temperature and fluid pressure fields

Now that the simple framework for collecting, maintaining, and outputting a group of samples have been created, then the class that performs the actual work of collecting individual samples can also be created. The class *WeatherSensorLogger* shown in *Program 18* extends *LoggingDaemon* and provides implementations for the captured sample and *WriteLog* entry methods.

WeatherSensorLogger creates a new instance of the class *WeatherSensorProg*, which is used to perform the relative humidity, pressure and temperature measurements. Here, we develop a datalogging application capable of capturing data and serving it up to any client over an Ethernet network connections. Before the test running the datalogger, a small client application is also developed that will connect to the server and download its current log.

The name of the server, the TINI running the *DataLogger* application, is extracted from the first argument on the command line. The *DataLoggerClient* shown in *Program 19* uses the server name to establish a connection to the server. After connection has been established, the *getInputStream* method is invoked on the socket instance to get a stream that can be used for uploading the log information from the server. The input stream is buffered, and the result is wrapped in a *DataInputStream*. At this point, the client is ready to read the data in the same format in which it is written by the server.

The temperature, relative humidity and pressure measurements are simply read as *doubles* and displayed. Both the client and server programs are ready to be executed. The server is launched on TINI using the following command.

TINI /> java DataLogger.tini 48 1800 &
starting DataLogger…

A number of 48 samples were specified with 1800-second delay between each sample. After running the server for about the hour to allow it to acquire a few samples the client is launched with the following command.

java DataLoggerClient 192.168.0.15
Total readings = 2
Entry 0: Saturday March 16 14:20:46 GMT 2013, TEMP = 23.531 °C RH = 27.733 % BP = 3872.106 mb
Entry 1: Saturday March 16 14:50:42 GMT 2013, TEMP = 23.438 °C RH = 28.067 % BP = 3865.362 mb

3.3 Implementation of the PPP Daemon and Managing the PPP Data Link

The point here is to provide support for managing a PPP interface. The top-level network server is what was implemented in the *DataLogger's* class of *Program 10*. It blocks on accept, creating for a connection to be established over an Ethernet network interface. It does not really care if the connection is established over an Ethernet network or a serial line using PPP. The server to be implemented now is a "*dial–up*" server that allows clients to establish TCP/IP connections to TINI using a PPP interface. The dial–up server will implement in a class called *PPPDaemon*. The *PPPDaemon* is shown in *Program 20* and it implements two interfaces: *PPPEventListener* to receive PPP event notification and *DataLinkListener* to receive notification about errors that occur with the physical data link. The *PPPDaemonListener* interface is given in *Program 21*.

After initializing the *Listener* and *maxRetries* fields, *PPPDaemon's* constructor creates either a *PPPSerialLink* or a *PPPModemLink* object, depending on the *ModemLink* Boolean passed to the constructor. Next, a new PPP object is created and IP address for both the local interface and the remote peer are set.

After creating a new PPP object, *PPPDaemon* is in the INIT state. At this point, there is no PPP traffic travelling across the physical data link. Transistor to the STARTING State, the owner of the *PPPDaemon* object invokes the *StartDaemon* method shown in *Program 22*. The *StartDaemon* adds its own object (*this*) as a listener for PPP events and invokes the open method on its PPP object.

It is easier to understand the operation of *PPPDaemon* as a finite state Machine (FSM). The state diagram of the FSM implemented by the *PPPDaemon* is shown in Fig. 6 [6]. The solid lines represent state transitions caused *PPPDaemon* invoking methods on its PPP object. The dashed lines represent transition errors detected by the native PPP implementation. At this point, PPP waits for a client to begin LCP (Line Control Protocol) negotiation. Once a client successfully completes the line negotiation, PPP requests login information and remote peer replies with a user name and password. Now the communication link is fully established and ready for IP traffic, then down on the PPP object will be invoked and remove the network interface that was added during the UP state processing.

The state machine as implemented by the *PPPEvent* in *Program 23* is designed to run continuously, retrying if transient errors occurs. Every time a connection is successfully established (the UP state is reached), the error count is reset to 0. The *PPPDaemon* already implemented maintains a reference to an instance of a class that implemented the *PPPDataLink* interface. This reference is used by the server to control the data link. The *PPPDataLink* program is shown in *Program 24*.

The *InitializeLink* method is used to perform any specific required to use that data link. After the link has been successfully initialized, the *PPPDaemon* invokes *getPort* to acquire a reference to the Link's serial port. The interface *DataLinkLIstener* shown in *Program 25* defines the method *DataLInkError* that will be invoked by the object controlling the data link upon detection of an unrecoverable error. When the Listener's *DataLinkError* of *Program 26* is invoked, it typically set some internal state and calls the close method on the PPP object.

The PPP server maintains a retry count and put an upper limit on the number of retries that can be caused by a persistent error in either the data link or the underlying PPP object. The retry count is reset to 0 after every successful transition to the UP state. All PPP traffic flows over a serial port. The serial port may not have a Modem attached. The *PPPSerialLink* of *Program 27* class will be created to provide functionality that is common to both hard-wired serial and modem configurations

Most practical uses of TINI require the use if an external serial modem [5], [6], [7]. Since all communication with the modem will be over a serial port, a class is created to manage the modem communication as a subclass of *PPPSerialLink*. Upon Construction, the *PPPserialLink* of *Program 28* invokes its *superclass's* constructor to acquire and initialize the serial port. It also creates a *ModemCommand* object to manage sending commands to and receiving responses from the modem.

PPPModemLink implements the *SerialPortEventListener* interface. Initializing the modem link involves the following three steps: *1)* Reset the modem, *2)* wait for a ring, and *3)* answer the phone. Both the *InitializeLink* and *ResetModem* methods are shown in *Program 29*. The modem reset is initiated by dropping the DTR line low, delaying for a couple of seconds, and then raising DTR back high. If the expected response is received, *ResetModem* returns normally. If the response is not received within the specified time-out value (6 seconds in this case), a *DataLinkException* is thrown by the *secondCommand* method of the *ModemCommand* class.

The *ModemCommand* Class, partially shown in *Program 30* is a utility class used by *PPPModemLink* to handle the details of serial communication with the Modem. The *WaitforMatch* method of *Program 31* takes a string used for the desired pattern match. It uses both serial port receive time-outs and thresholds to control the reading of data and manage a Line. Ultimately, an implementation of *PPPDaemon* and the supporting data link classes have been generated so that the *DataLogger* class can be enhanced to accept network connections over both PPP and Ethernet interfaces. The *PPPEthernetDataLogger* program shown in *Program 32* shows the additions and modifications made to the *DataLogger* class for the sole purpose of adding PPP daemon support 18].

3.4 Testing the Fluid Pressure and Temperature DataLogger Using Ethernet and PPP Dial-Up Connection.

The remote *DataLogger* develop up to this point will accept TCP connections (sockets) from multiple network interfaces

– specifically the PPP and Ethernet interfaces. The *DataLogger* was tested over Ethernet only using the *DataLoggerClient* developed in *Program 19*.

Testing the new PPP functionally is relatively a little more tasking. However, the *DataLoggerClient* can be used without modification to test both interfaces simultaneously. The setup shown in Fig. 7 tests the full dial–up using point-to-point protocol networking capabilities provided by *PPPDaemon* using analog modems (build around TINI) and a phone line simulator or a PSTN [18]. The TINI's network interfaces IP addresses are 192.168.0.15 and 192.168.1.1 for Ethernet and PPP respectively.

The *DataLogger's* PPP support is also tested using a hard-wired serial connection directly to a Windows XP Client as shown in Fig. 8. The TINI's network interface IP address is still retained as 192.168.0.15 and 192.168.1.1 for Ethernet and Windows XP Client.

Adding a couple of debugging statements as shown in *Program 33* to the *DataLogger's* run method of *Program 12*, displays connection information including both the remote Client's IP address and TINI's local interface IP address.

Program 33: The Debug Statements

```
Public void run () {
    ……
    While (true) {
    ……
    s = ss.accept () ;
    …….
    System.out.println ("New client:" + s.toString ());
    System.out.println ("Local interface:" + s.getLocalAddress());
    ……
    }
}
```

The *DataLogger* is hunched by supplying the sample count, sample rate, and client authentication information as command line parameters.

TINI /> java DataLogger.tini 48 1800 duct kid
Starting DataLogger…

To test PPP interface, a new dial-up connection is created and either manually connected to the TINI or optionally connected depending on whatever dial–up capability is provided on the client's operating system.

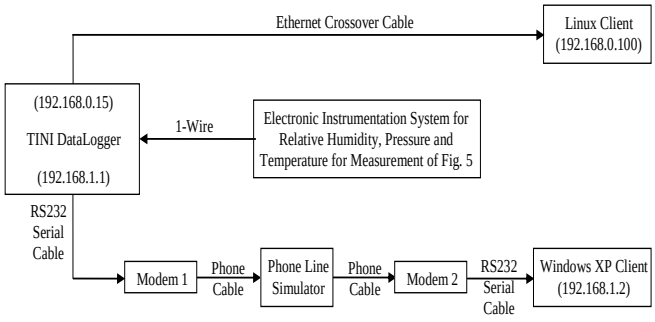

Fig. 7: Ethernet and PPP dial-up modem test configuration

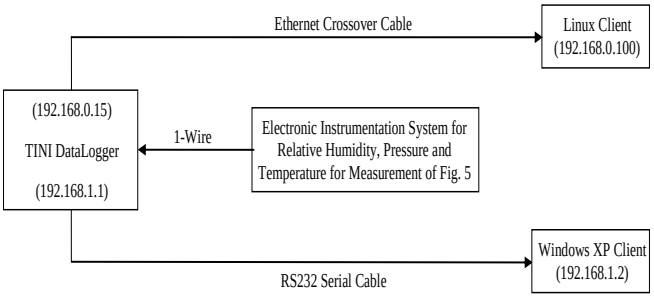

Fig. 8: Ethernet and hard–wired serial test configuration

Regardless of the mode of connection, once the connection is initiated, the following sequence of events occurs: client modem dials TINI's Modem, TINI's modem answers the incoming call, PPP option negotiation begins, authentication information is transmitted from the remote peer to TINI, and finally the IP addresses of TINI and remote peer are established. Executing the "*ipconfig –x*" command at the slush prompt produces the following output.

```
…
Interface 2 is active
Name            : PPP0
Type            : Point -to -point Protocol
IP Address      : 192.168.1.1
Subnet Mask     :255.255.255.0
Gateway         : 0.0.0.0
…
```

A new network interface has been added to the system as a result of the *PPPDaemon* invoking *addInterface* on its PPP object after the modem link was established. The local address is set to and the interface name is the same as supplied by *addInterface*. The "*PPP0*" interface will remain in the system until *removeInterface* is invoked in response to a PPP Closed event.

At this point, both the Ethernet (*eth0*) and PPP (*PPP0*) interfaces are active and the server is connected over both using the *DataLoggerClient*. Output from launching the Linux (Ethernet) client gives the following results:

java DataLoggerClient 192.168.0.15
Total readings = 2
Entry 0: Saturday March 16 15:10:25 GMT 2013, TEMP = 28.457 °C RH = 29.522 % BP = 3886.512 mb
Entry 1: Saturday March 16 15:40:26 GMT 2013, TEMP = 28.562 °C RH = 28.128 % BP = 3889.731 mb
Output from launching WinXp (PPP) Client (same for Hard-wired Serial)
Java DataLoggerClient 192.168.1.1
Total Readings = 2
Entry 0: Saturday March 16 15:10:25 GMT 2013, TEMP = 28.457 °C RH = 29.522 % FP = 3886.512 mb
Entry 1: Saturday March 16 15:40:26 GMT 2013, TEMP = 28.562 °C RH = 28.128 % FP = 3889.731 mb
DataLogger (TINI) Output:
New Client: Socket [adder = 192.168.0.100/192.168.0.100, Port = 1056, localport = 5588]
Local interface: 192.168.0.15/192.168.0.15

New Client: Socket [adder = 192. 168.1.2/192.168.1.2, Port = 1949, localport = 5588]
Local interface: 192.168.1.1/192.168.1.1

When the Linux Client establishes its connection; *DataLogger* displays the remote client's IP address (192.168.0.100) and the IP address of its own local Ethernet; interface (192.168.015). When the Windows XP client connects the client (192.168.1.2) and server (192.168.1.1), the IP addresses displayed are those selected in *PPPDaemon's* constructor during initialization of the PPP object. In the Ethernet case, both IP addresses were statistically configuration outside of the program control. In the PPP case, however the IP addresses were set programmatically by *PPPDaemon*. The local port value displayed in the TINI output is *DataLogger's* SERVER_PORT Number (5588) for both connections.

4. CONCLUSION

An efficient technique for the networking of TINI TBM390 for real-time online datalogging applications have been designed, constructed, implemented and tested over Ethernet, hard-wired serial and point-to-point (PPP) communication links. An electronic instrumentation system for real-time measurements of three weather parameters namely: temperature, relative humidity and barometric pressure have also been is designed, constructed, calibrated and implemented. The weather parameters were deployed via TINI TBM390 over Ethernet, hard-wired serial and point-to-point communication links. A novel technique on how to use point-to-point-protocol (PPP) networking with TINI runtime environment for dial-up networking including detailed initialization and configuration have also been demonstrated for both Linux and Microsoft Windows® clients with several critical issues on the design, implementation and deployment of the proposed weather datalogger system are also are presented.

The comparisons of the measured and actual (true) values of the three weather parameters demonstrated the efficiency of the proposed strategy. The paper also presented all the Java programs required to develop, implement and demonstrate the techniques proposed in this work. As a future work, the techniques proposed and demonstrated in this paper forms the foundation for the development of real-time embedded system, built around TINI, for online pressure and temperature measurements, monitoring and control of fluid flow in a pipe to track pipeline vandalization which is a common phenomenon in the Nigeria.

Furthermore, the concepts and techniques proposed in this work can be adapted to develop a complete meteorological weather station that can convert physical quantities like relative humidity, temperature, solar irradiance, (sun intensity), atmospheric pressure, wind (i.e., wind speed and direction), rainfall (precipitation), and cloud (cloud thickness and height from ground) to electrical voltages in a form that will be accepted and uploaded by TINI to clients over web servers.

REFERENCES

[1] B. Margolin, "Embedded System", 1997, Available [Online]: http://ww.netbsding/misc/embed.hthml.

[2] J. Bean, "SOA and web services interface design: Principles, techniques, and standards", Elsevier, 2010, USA.

[3] Dallas Semiconductors inc. (Maxim Integrated Products), "Tiny Internet Interface (TINI) and IBUTTONS", Available [Online]: http://www.ibutton.com.

[4] G. Roussel and E. Duris, "Java et Internet Concepts et Programmation" Vuibert, 2000.

[5] Sun. "Java Web Site – useful for updated documentation". Available [Online]: http://java.sun.com.

[6] D. Loomis "The TINI TM Specification and Developers Guide", Dallas Semiconductor Corporation, Addison Wesley, Upper Saddle River, New Jersey, U.S.A, 2001.

[7] S. S. Shillewar, "Using TINI point-to-point protocol (PPP)", Oriental Journal of Computer Science & Technology, vol. 4, no. 1, 2011, pp. 153 – 158.

[8] J. J. Elosua and J. C. Burguillo, "www-Based remote control using TINI cards and Brazil", 15th IFAC World Congress, Barcelona, Spain, 2002, pp. 167 – 172.

[9] V. A. Akpan, R. O. A. Osakwe and A. Amaku, "Initialization and configuration of TINI & Java communications API for real-time embedded networked applications", International Journal of Communication, Network and System Sciences (IJCNS), Paper ID: 9701789, 2013 (Accepted).

[10] M. A. Joshi, M. R. Jathar and S. C. Mehrotra, "Distributed system for weather data collection through TINI microcontroller", International Journal of Environmental Science and Development, vol. 2, no. 1, Feb., 2011, pp. 70 – 72.

[11] Dallas Semiconductor Corporation, "1-Wire Networking", Available [Online]: http://www.ibutton.com/ibuttons/standard.pdf.

[12] Maxim Integrated Products, Guidelines for reliable long line 1-Wire® networks, Tutorial 148, Sept. 22, 2008. Available [Online]: www.maximintegrated.com/app-notes/index.mvp/id/148 and http://www.maximintegrated.com/an148.

[13] Maxim Integrated Products, "Advanced 1-Wire network driver, AN 244 Reference schematic". Available [Online]: http://www.maximintegrated.com/an244.

[14] Maxim Integrated Products, DS9097U-DS9097U-S09 Universal 1-Wire Com Port Adapter, http://www.maximintegrated.com/datasheet/index.mvp/id/2983.

[15] Maxim Integrated Products, DS 2438 Battery Monitor, http://pdfserv.maximintegrated.com/en/ds/DS2438.pdf.

[16] Honeywell, "HIH-4602-A/C relative humidity sensor datasheet", 2007. Available [Online]: http://sensing.honeywell.com/index.php/ci_id/51480/la_id/1/document/1/re_id/0.

[17] Freescale Semiconductor, "MPX4115A: Integrated silicon pressure sensor altimeter/barometer pressure sensor on-chip signal conditioned, temperature compensated and calibrated", Technical Data, Document Number: MPX4115, Rev 5, August 2006. Available [Online]: http://www.freescale.com/files/sensors/doc/data_sheet/MPX4115.pdf.

[18] Maxim Integrated Products, "Application Note 704: Asynchronous serial-to-Ethernet device servers, MaximApp704-1". Available [Online]: http://www.maximintegrated.com/

About Authors

Vincent A. Akpan obtained the B.Sc. degree in Physics Electronics from the Delta State University (DELSU), Abraka, Nigeria in 1997; a Master of Technology (M.Tech.) degree in Instrumentation from The Federal University of Technology, Akure (FUTA), Nigeria in 2003; and a Ph.D. degree in Electrical & Computer Engineering from the Aristotle University of Thessaloniki (AUTH), Thessaloniki, Greece in 2011.

Between 1998 and 2004, he was a Graduate Assistant with the Department of Physics, DELSU. Since 2004 he has been with the Department of Physics Electronics, FUTA where he is currently Lecturer II. Between October 2011 and July 2012, he was a Visiting Lecturer in Computational Intelligence with the Department of Computer Science, College of Information & Communication Technology (ICT), Salem University, Lokoja, Nigeria. During the same period, he was also a Visiting Lecturer in Digital Signal Processing (DSP), Computer Systems Architecture & Embedded Systems Engineering with the Department of Physics Electronics & Computer Science, Wesley University of Science & Technology, Ondo, Nigeria. Between October 2012 and July 2013, he was an Adjunct Lecturer in Intelligent Control & Embedded Systems Engineering with the Department of Mechatronics Engineering, College of Engineering, Afe Babalola University, Ado-Ekiti, Nigeria. His integrated research interests include: intelligent adaptive predictive control, computational intelligence (neural networks, fuzzy logic and genetic algorithms), instrumentation, real-time embedded systems, signal processing & machine vision, system identification, and Robotics & Automation. He is the co-author of a book and has authored or co-authored more than 30 articles in refereed journals and conference proceedings. He is a regular reviewer in more than 10 international scientific/academic journals and several IEEE sponsored conferences. He is the managing editor of the Nigerian Journal of Pure and Applied Physics (NJPAP) as well as an editorial board member for the American Journal of Intelligent Systems (AJIS).

Dr. Akpan is a member of the IEEE, USA and The IET, UK. He was one among the two Nigerian recipients of the 2005/2006 Greek State Scholarship (IKY) for a Ph.D. programme tenable in Greece.

Reginald A. O. Osakwe received his B.Sc. degree in Physics from the University of Lagos, Nigeria in 1979; his M.Sc. in solid state Physics from the Purdue University, West Lafayette, Indiana, U.S.A. in 1984; and his Ph.D in electronics from the University of Ibadan, Nigeria in 1995.

Between 1991 and 2007, he was a lecturer with the Department of Physics, Delta State University, Abraka, Nigeria. Since 2007 he has been with the Department of Physics, Federal University of Petroleum Resources, Effurun, Delta State, Nigeria where he is currently a Senior Lecturer. During the periods 1998 – 2009 he has been a visiting lecturer to the Shell Intensive Training Programme (SITP), Warri, Delta State under the Shell Petroleum Development Company (SPDC) of Nigeria. His research interests are on microprocessor architecture, embedded computer systems, chaotic dynamics, and image and signal processing systems.

Amaku Amaku received his B.Sc. degree in Electronics and Computer Technology from University of Calabar, Cross River State, Nigeria in 2006; and his M.Sc. degree in Information Technology from the National Open University of Nigeria (NOUN), Nigeria in 2013.Since 2011 he has been with the College of Information and Communication Technology (CITC), Salem University, Lokoja, Kogi State, Nigeria where he is currently Technologist II. He has co-authored four (4) articles in refereed conference proceedings. His research interest is in computer systems architecture, real-time embedded systems and network systems engineering.
Mr. Amaku is a graduate member of the Nigerian Society of Engineers.

APPENDIX A: Efficient networking of TINI for real-time weather data logging & deployment over Ethernet and serial communication links

Program 1: FindAdapters

```
import java.util.Enumeration;
import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.adapter.DSPortAdapter;

  class FindAdapters {
        public static void main(String[] args) {
                Enumeration e =
OneWireAccessProvider.enumerateAllAdapters();
                while (e.hasMoreElements()) {
                        System.out.println(
                                ((DSPortAdapter)
e.nextElement()).getAdapterName());
                }
        }
  }
```

Program 2: CreateAdapters

```
import com.dalsemi.onewire.adapter.TINIInternalAdapter;
import com.dalsemi.onewire.adapter.TINIExternalAdapter;

  class CreateAdapters {
        public static void main(String[] args) {
                TINIExternalAdapter external = new
TINIExternalAdapter();
                TINIInternalAdapter internal = new
TINIInternalAdapter();
                System.out.println(external.getAdapterName());
                System.out.println(internal.getAdapterName());
        }
  }
```

Program 3: Adapterfeatures

```
import com.dalsemi.onewire.adapter.TINIExternalAdapter;
import com.dalsemi.onewire.adapter.TINIInternalAdapter;
import com.dalsemi.onewire.OneWireException;
```

```
class AdapterFeatures {
    public static void main(String[] args) {
        try {
            TINIInternalAdapter internal = new TINIInternalAdapter();
            System.out.println("Internal Adapter:");
            System.out.println(" Supports overdrive speeds - " + internal.canOverdrive());
            System.out.println(" Supports flexible timing - " + internal.canFlex());

            TINIExternalAdapter external = new TINIExternalAdapter();
            System.out.println("External Adapter:");
            System.out.println(" Supports overdrive speeds - " + external.canOverdrive());
            System.out.println(" Supports flexible timing - " + external.canFlex());
        } catch (OneWireException owe) {
            System.out.println(owe.getMessage());
        }
    }
}
```

Program 4: FastCensus

```
import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.OneWireException;
import com.dalsemi.onewire.adapter.DSPortAdapter;

class FastCensus {
    public static void main(String[] args) {
        DSPortAdapter adapter = null;
        try {
            adapter = OneWireAccessProvider.getDefaultAdapter();
            adapter.beginExclusive(true);

    adapter.setSpeed(adapter.SPEED_REGULAR);
            if (adapter.findFirstDevice()) {

    System.out.println(adapter.getAddressAsString());
                while (adapter.findNextDevice()) {

    System.out.println(adapter.getAddressAsString());
                }
            }
        } catch (OneWireException owe) {
            System.out.println(owe.getMessage());
        } finally {
            adapter.endExclusive();
        }
    }
}
```

Program 5: FindContainers

```
import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.OneWireException;
import com.dalsemi.onewire.container.OneWireContainer;
import com.dalsemi.onewire.adapter.DSPortAdapter;

class FindContainers {
    public static void main(String[] args) {
        DSPortAdapter adapter = null;
        try {
            adapter = OneWireAccessProvider.getDefaultAdapter();
            adapter.beginExclusive(true);

    adapter.setSpeed(adapter.SPEED_REGULAR);
            OneWireContainer owc = null;
            if ((owc = adapter.getFirstDeviceContainer()) != null) {

    System.out.println(owc.getName()+","+owc.getAlternateNames() +

                " at address "+owc.getAddressAsString());

                while ((owc = adapter.getNextDeviceContainer()) != null) {

    System.out.println(owc.getName()+","+owc.getAlternateNames() +

                " at address "+owc.getAddressAsString());
                }
            }
        } catch (OneWireException owe) {
            System.out.println(owe.getMessage());
        } finally {
            adapter.endExclusive();
        }
    }
}
```

Program 6: SimpleEthernetAddressReader

```
import com.dalsemi.tininet.TININet;
class SimpleEthernetAddressReader {
    public static void main(String[] args) {

    System.out.println(TININet.getEthernetAddress());
    }
}
```

Program 7: DNSTest

```
import java.net.InetAddress;
import java.net.UnknownHostException;

class DNSTest {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java DNSTest.tini name");
```

```
                System.exit(1);
        }
        try {
                InetAddress[] names = InetAddress.getAllByName(args[0]);
                for (int i = 0; i < names.length; i++) {
                        System.out.println(names[i].toString());
                }
        } catch (UnknownHostException uhe) {
                System.out.println("Lookup error:" + uhe.getMessage());
        }
    }
}
```

Program 8: MiniBrowser

```
import java.net.*;
import java.io.*;
import com.dalsemi.tininet.TININet;

class MiniBrowser {
    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: MiniBrowser URL proxy_server proxy_port");
            System.exit(1);
        }

        //TININet.setProxyServer(args[1]);

        //TININet.setProxyPort(Integer.parseInt(args[2]));

        try {
            URL u = new URL(args[0]);
            InputStream in = u.openConnection().getInputStream();

            byte[] content = new byte[512];
            int count = 0;
            do {
                count = in.read(content);
                System.out.write(content, 0, count);
            } while (count != -1);

        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Program 9: Pinger

```
import java.net.*;
import com.dalsemi.tininet.icmp.Ping;

class Pinger {
    static final int ICMP_ECHO_REPLY = 0;
    static final int ICMP_TIME_EXCEEDED = 11;

    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: java Pinger.tini node count max_hops");
            System.exit(1);
        }
        try {
            InetAddress addr = InetAddress.getByName(args[0]);
            int count = Integer.parseInt(args[1]);
            int ttl = Integer.parseInt(args[2]);
            byte[] response = new byte[256];
            for (int i = 0; i < count; i++) {
                long rtt = Ping.pingNode(addr, (byte) ttl, response);
                if (rtt == -1) {
                    System.out.println("No response from host:"+args[0]);
                } else {
                    // Compute length of IP header
                    int ipHdrLength = (response[0] & 0x0f)<<2;
                    int type = response[ipHdrLength];
                    switch (type) {
                        case ICMP_ECHO_REPLY:
                            int sequence = ((response[ipHdrLength+6] & 0xFF) << 8) +
                                (response[ipHdrLength+7] & 0xFF);
                            System.out.println("Reply from:"+addr.toString()+" in "+rtt+"ms"+
                                " ttl="+(response[8]&0xff)+" sequence="+sequence);
                            break;
                        case ICMP_TIME_EXCEEDED:
                            // Hack out the source ip address and convert it to a String.
                            StringBuffer sb = new StringBuffer(15);
                            for (int j = 0; j < 4; j++) {
                                sb.append(Integer.toString(response[12+j] & 0xff));
                                if (j < 3) {
                                    sb.append('.');
                                }
```

```
            }

            System.out.println("Time exceeded message from:"+

InetAddress.getByName(sb.toString())+" in "+rtt+"ms");

            break;
                                                default:

            System.out.println("Unexpected ICMP message type: " +
type);

            break;
                                                    }
                                            }
                                            try {

            Thread.sleep(1000);
                                            } catch
(InterruptedException ie) { }
                                    }
                            }
                            catch (UnknownHostException uhe) {
                                    uhe.printStackTrace();
                            }
            }
    }
```

Program 10: DataLogger

```
import java.io.*;
import java.net.*;
import com.dalsemi.tininet.ppp.PPPException;

    class DataLogger extends Thread implements
PPPDaemonListener {
            // Port number for inbound connections, well above well
known ports
            static final int SERVER_PORT = 5588;

            PPPDaemon ppps;
            PressureLogger logger;
            TemperatureLogger logger;
            private String name;
            private String password;

            DataLogger(int samples, int delay, String name, String
password)
                    throws PPPException, LoggingException {

                    // Set authentication information
                    this.name = name;
                    this.password = password;
                    // Create and start the logging daemon
                    logger = new HumidityLogger(samples, delay);
                    logger.start();
                    // Create a server to manage PPP dial-up
requests
                    PPPDaemon ppps = new PPPDaemon(this,
"serial0", 19200);
                    ppps.startServer();
            }

            public void serverError(String error) {
                    System.err.println("Error in PPP
server:"+error);
                    ppps.stopServer();
            }

            public boolean isValidUser(String name, String password)
{
                    return (this.name.equals(name) &&
this.password.equals(password));
            }

            public void run() {
                    ServerSocket ss = null;
                    try {
                            ss = new
ServerSocket(SERVER_PORT);
                    } catch (Exception e) {
                            e.printStackTrace();
                            // Abort if we can't create ServerSocket
instance
                            return;
                    }

                    while (true) {
                            Socket s = null;
                            try {
                                    // Wait for client connections
over PPP or Ethernet
                                    s = ss.accept();
                            } catch (IOException ioe) {
                                    // Shut down the logging
daemon
                                    logger.stopLogging();
                            System.out.println("Fatal problem
with server socket, exiting");
                                    ioe.printStackTrace();
                                    // Fall out of run method
                                    break;
                            }

                            System.out.println("New client:" +
s.toString());
                            System.out.println("Local interface:"
+ s.getLocalAddress());
                            // Create a new thread to handle this
connection
                            (new LogWorker(s)).start();
                    }
            }

            private class LogWorker extends Thread {
                    private Socket s;

                    private LogWorker(Socket s) {
                            this.s = s;
```

```
        }

        public void run() {
            DataOutputStream dout = null;
            try {
                dout = new DataOutputStream(new
BufferedOutputStream(s.getOutputStream()));
                logger.writeLog(dout);
                dout.flush();
            } catch (IOException ioe) {
                System.out.println("I/O error writing log data");
                ioe.printStackTrace();
            } finally {
                try {
                    s.close();
                    dout.close();
                } catch (IOException e) {}
            }
        }

        public static void main(String[] args) {
            System.out.println("Starting DataLogger ...");
            if (args.length != 4) {
            System.out.println("Usage: java DataLogger samples delay username password");
                System.exit(1);
            }

            // Total number of samples to maintain
            int samples = Integer.parseInt(args[0]);
            // Delay between each sample in minutes
            int delay = Integer.parseInt(args[1]);

            try {
                (new DataLogger(samples, delay, args[2], args[3])).start();
            } catch (Exception e) {
                System.out.println("Error creating data logger");
                e.printStackTrace();
            }
        }
    }
```

Program 11: DataLogger's Main Method

```
    ……..
        public static void main(String[] args) {
            System.out.println("Starting DataLogger ...");
            if (args.length != 4) {
            System.out.println("Usage: java DataLogger samples delay username password");
                System.exit(1);
            }

            // Total number of samples to maintain
            int samples = Integer.parseInt(args[0]);
            // Delay between each sample in minutes
            int delay = Integer.parseInt(args[1]);

            try {
                (new DataLogger(samples, delay, args[2], args[3])).start();
            } catch (Exception e) {
                System.out.println("Error creating data logger");
                e.printStackTrace();
            }
        }
    }
```

Program 12: DataLogger's run method

```
    ……..
        public void run() {
            ServerSocket ss = null;
            try {
                ss = new ServerSocket(SERVER_PORT);
            } catch (Exception e) {
                e.printStackTrace();
                // Abort if we can't create ServerSocket instance
                return;
            }

            while (true) {
                Socket s = null;
                try {
                    // Wait for client connections over PPP or Ethernet
                    s = ss.accept();
                } catch (IOException ioe) {
                    // Shut down the logging daemon
                    logger.stopLogging();
                    System.out.println("Fatal problem with server socket, exiting");
                    ioe.printStackTrace();
                    // Fall out of run method
                    break;
                }

                System.out.println("New client:" + s.toString());
                System.out.println("Local interface:" + s.getLocalAddress());
                // Create a new thread to handle this connection
                (new LogWorker(s)).start();
            }
        }

        private class LogWorker extends Thread {
            private Socket s;

            private LogWorker(Socket s) {
                this.s = s;
```

```
                }

                public void run() {
                        DataOutputStream dout = null;
                        try {
                                dout = new
DataOutputStream(new
        BufferedOutputStream(s.getOutputStream()));
                                logger.writeLog(dout);
                                dout.flush();
                        } catch (IOException ioe) {
                                System.out.println("I/O
error writing log data");
                                ioe.printStackTrace();
                        } finally {
                                try {
                                        s.close();
                                        dout.close();
                                } catch (IOException e) {}
                        }
                }
        }
```

Program 13: LoggingDaemon's Constructor

```
import java.io.*;
import java.util.*;

public abstract class LoggingDaemon extends Thread {
        private int maxSamples;
        private int delay;
        // Circular buffer to hold the 'maxSamples' recent
measurements
        private Vector samples;
        private int index = 0;
        private boolean logEm = true;

        public LoggingDaemon(int maxSamples, int delay) throws
LoggingException {
                this.maxSamples = maxSamples;
                // Convert delay from seconds to milliseconds
                this.delay = delay * 1000;
                samples = new Vector(maxSamples);
                this.setDaemon(true);
        }

        public void run() {
                while (logEm) {
                        Object smp = captureSample();
                        if (smp != null) {
                                synchronized (samples) {
                                        if (samples.size()
== maxSamples) {
                                                //
Remove the oldest entry

        samples.removeElementAt(0);
                                                }

        samples.addElement(smp);
                                }

                                try {
                                        Thread.sleep(delay);
                                } catch (InterruptedException ie) {}
                        }
                }

                public void stopLogging() {
                        logEm = false;
                }

                public void writeLog(DataOutputStream dout) throws
IOException {
                        Vector sc = (Vector) samples.clone();
                        dout.writeInt(sc.size());
                        for (Enumeration e = sc.elements();
e.hasMoreElements(); ) {
                                writeLogEntry(e.nextElement(), dout);
                        }
                }

                public abstract Object captureSample();

                public abstract void writeLogEntry(Object sample,
DataOutputStream dout)
                        throws IOException;
        }
```

Program 14: LoggingDaemon's run method

```
……….
        public void run() {
                while (logEm) {
                        Object smp = captureSample();
                        if (smp != null) {
                                synchronized (samples) {
                                        if (samples.size()
== maxSamples) {
                                                //
Remove the oldest entry

        samples.removeElementAt(0);
                                                }

        samples.addElement(smp);
                                }

                                try {
                                        Thread.sleep(delay);
                                } catch (InterruptedException ie) {}
                        }
                }
```

Program 15: LoggingDaemon's Writelog method

```
……..
        public void writeLog(DataOutputStream dout) throws
IOException {
```

```
                Vector sc = (Vector) samples.clone();
                dout.writeInt(sc.size());
                for (Enumeration e = sc.elements();
e.hasMoreElements(); ) {
                        writeLogEntry(e.nextElement(), dout);
                }
        }
```

program 16: WeatherSensorProg

```
import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.adapter.DSPortAdapter;
import com.dalsemi.onewire.OneWireException;
import com.dalsemi.onewire.container.OneWireContainer;
import com.dalsemi.onewire.container.OneWireContainer26;

  public classWeatherSensorProg {
        DSPortAdapter adapter;
        OneWireContainer26 owc;
        byte[] state;

        WeatherSensorProg(DSPortAdapter adapter) throws
OneWireException {
                this.adapter = adapter;
                // Only find DS2438 family devices
                adapter.targetFamily(0x26);
      adapter.setSpeed(adapter.SPEED_REGULAR);
                if (!adapter.findFirstDevice()) {
                        throw new OneWireException("No
DS2438 A to D chip found");
                }

                owc = (OneWireContainer26)
adapter.getDeviceContainer();
                state = owc.readDevice();
        }

        public double getTemperature() throws OneWireException
{
                owc.doTemperatureConvert(state);
        state = owc.readDevice();

                return owc.getTemperature(state);
        }

        public double getSensorRH( ) throws OneWireException {
                // Read Vad

        owc.doADConvert(OneWireContainer26.CHANNEL_VAD
, state);
                double Vad =
owc.getADVoltage(OneWireContainer26.CHANNEL_VAD, state);
                // Read Vdd

        owc.doADConvert(OneWireContainer26.CHANNEL_VD
D, state);
                double Vdd =
owc.getADVoltage(OneWireContainer26.CHANNEL_VDD, state);

                return (vad/vdd – 0.16)/0.0062;
        }

        public double getTrueRH() throws OneWireException {
                return getSensorRH()/(1.0546-
0.00216*getTemperature());
        }

  public double getSensorP() throws OneWireException {
                // Read Vad

        owc.doADConvert(OneWireContainer26.CHANNEL_VAD
, state);
                double Vad =
owc.getADVoltage(OneWireContainer26.CHANNEL_VAD, state);
                // Read Vdd

        owc.doADConvert(OneWireContainer26.CHANNEL_VD
D, state);
                double Vdd =
owc.getADVoltage(OneWireContainer26.CHANNEL_VDD, state);

                return (vad/vdd +384.62)/0.094;
        }

        public double getTrueP( ) throws OneWireException {
                return getSensorP/(6.691-0.2*Temperature);
        }

        void displayData() {
                try {
        adapter.beginExclusive(true);
                        System.out.println("Temperature =
"+getTemperature()+" °C");
   System.out.println("Relative Humidity = "+getSensorRH()+"
%");
                        System.out.println("Pressure =
"+getSensorP ( )+"mb");
                } catch (OneWireException owe) {
                        System.out.println(owe.getMessage());
                } finally {
        adapter.endExclusive();
      }
   }

        public static void main(String[] args) {
                try {
                 WeatherSensorProg Pressure =

        newWeatherSensorProg(OneWireAccessProvider.getDefa
ultAdapter());
                        Pressure.displayData();
                } catch (OneWireException owe) {
                        System.out.println(owe.getMessage());
                }
        }
  }
```

Program 17: WeatherSensorSample

```
  public class WeatherSensorSample {
        private double temperature;
  private double relativehumidity;
```

```
        private double pressure;
        private long   timeStamp;

        public FluidpressureSample(double Fluidpressure, double
temperature) {
                this.temperature = temperature;
   this.relativehumidity = relativehumidity;
                this.pressure = pressure;
                timeStamp = System.currentTimeMillis();
        }

        public long getTimeStamp() {
                return timeStamp;
        }

        public double getTemperature() {
                return temperature;
        }

        public double getRelativeHumidity() {
                return relativehumidity;
        }
   public double getPressure() {
                return pressure;
        }

   }
```

Program 18: WeatherSensorLogger

```
import java.io.IOException;
import java.io.DataOutputStream;
import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.adapter.DSPortAdapter;
import com.dalsemi.onewire.OneWireException;

   public class WeatherSensorLogger extends LoggingDaemon {
        private WeatherSensor sensor;
        private DSPortAdapter adapter;

        public WeatherLogger(int maxSamples, int delay) throws
LoggingException {
                super(maxSamples, delay);
                try {
                        adapter                         =
OneWireAccessProvider.getDefaultAdapter();
                        sensor          =          new
WeatherSensor(adapter);
                } catch (OneWireException owe) {
                        throw new LoggingException("Error
creating Environmental Sensor:" + owe.getMessage());
                }
        }

        public Object captureSample() {
                try {
                        adapter.beginExclusive(true);
                        double          temp          =
sensor.getTemperature();
   double relhumid = sensor.getTrueRH();
                        double pressure = sensor.getTrueP();
                        return                          new
WeatherSensorSample(temp, relhumid, pressure);
                } catch (OneWireException owe) {
                        System.out.println("Error reading
sensor");
                        owe.printStackTrace();
                        // No need to terminate app because of
a failed reading
                        return null;
                } finally {
                        adapter.endExclusive();
                }
        }

        public void writeLogEntry(Object sample,
DataOutputStream dout) throws IOException {

        dout.writeLong(((WeatherSensorsample)sample).getTimeS
tamp());

        dout.writeDouble(((WeatherSensorsample)sample).getTem
perature());
   dout.writeDouble(((WeatherSensorsample)sample).getRelativeHu
midity());
   dout.writeDouble(((WeatherSensorsample)sample).getPressure())
;
        }
   }
```

Program 19: DataLoggerClient

```
import java.io.*;
import java.net.*;
import java.util.Date;

   class DataLoggerClient {
        static final int PORT = 5588;

        public static void main(String[] args) {
                if (args.length != 1) {
                        System.out.println("Usage:          java
DataLoggerClient server");
                        System.exit(1);
                }

                Socket s = null;
                DataInputStream din = null;
                try {
                        s                       =                       new
Socket(InetAddress.getByName(args[0]), PORT);
                        din = new DataInputStream(new
BufferedInputStream(s.getInputStream()));
                        // Read number of data entries coming
our way
                        int entries = din.readInt();
                        System.out.println("Total
readings="+entries);
                        for (int i = 0; i < entries; i++) {
                                System.out.print("Entry " + i
+ ":" +new Date(din.readLong()));
```

```
                                System.out.print(", TEMP =
" + din.readDouble());
                                System.out.println(", RH = "
+ din.readDouble());
   System.out.print(", BP = " + din.readDouble());
                        }
                } catch (IOException ioe) {
                        System.out.println("Error
downloading readings:"+ioe.getMessage());
                        ioe.printStackTrace();
                } finally {
                        try {
                                s.close();
                                din.close();
                        } catch (IOException _) {}
                }
        }
   }
```

Program 20: PPPDaemon

```
import java.io.*;
import javax.comm.SerialPort;
import com.dalsemi.tininet.ppp.*;

   public class PPPDaemon implements PPPEventListener,
DataLinkListener {
        private PPP ppp;
        private PPPDataLink dataLink;
        private int maxRetries;
        private PPPDaemonListener listener;

        public PPPDaemon(PPPDaemonListener listener) throws
PPPException {
                // Default to using 'internal' serial port @19200
bps
                // connected to a modem
                this(listener, "serial0", 19200, 3, true);
        }

        public PPPDaemon(PPPDaemonListener listener, String
portName, int speed)
                throws PPPException {

                this(listener, portName, speed, 3, true);
        }

        public PPPDaemon(PPPDaemonListener listener, String
portName,
                                        int speed, int
maxRetries, boolean modemLink) throws PPPException {

                this.listener  = listener;
                this.maxRetries = maxRetries;
                try {
                        if (modemLink) {
                                dataLink = new
PPPModemLink(portName, speed, this);
                        } else {
                                dataLink = new
PPPSerialLink(portName, speed, this);
                        }
                } catch (DataLinkException dle) {
                        throw new PPPException("Unable to
initialize PPPDaemon:" + dle.getMessage());
                }

                // Create and initialize a new PPP object to
control
                // our data link.
                ppp = new PPP();
                // Local address 192.168.1.1
                ppp.setLocalAddress(new byte[] {(byte) 192,
(byte) 168, 1, 1});
                // Vend address 192.168.1.2 to client
                ppp.setRemoteAddress(new byte[] {(byte) 192,
(byte) 168, 1, 2});
                // We're a PPP server
                ppp.setPassive(true);
                ppp.setAuthenticate(true);
        }

        private int retryCount;
        private boolean isUp;

        public void pppEvent(PPPEvent ev) {
                switch (ev.getEventType()) {
                        case PPPEvent.STARTING:
                                try {
                                        // Now we need to
bring up the physical link

        dataLink.initializeLink();

        ppp.up((SerialPort) dataLink.getPort());
                                } catch (DataLinkException
dle) {
                                        listener.serverError("Data
link error:"+dle.getMessage());
                                        ppp.close();
                                }
                                break;

                        case
PPPEvent.AUTHENTICATION_REQUEST:
                ppp.authenticate(listener.isValidUser(ppp.getPeerID(),
ppp.getPeerPassword()));
                                break;

                        case PPPEvent.UP:
                                // Reset error count after
successfully bring up connection
                                retryCount = 0;
                                ppp.addInterface("ppp0");
                                isUp = true;
                                break;

                        case PPPEvent.STOPPED:
                                ppp.close();
                                if (++retryCount <
maxRetries) {
```

```
                                        ppp.close();

                                    } else {

        listener.serverError("Unable to establis PPP connection");
                                    }
                                    break;

                                case PPPEvent.CLOSED:
                                    if (isUp) {

        ppp.removeInterface("ppp0");
                                        ppp.down();
                                        isUp = false;
                                    }
                                    try {
                                        // Sleep before recycling ppp connection

        Thread.sleep(1000);
                                    } catch (InterruptedException ie) {}
                                    ppp.open();
                                    break;

                                default:
                                    break;
                            }
                        }

        public void startDaemon() throws PPPException {
                            retryCount = 0;
                            try {
                                // Add PPP event listener to driver state machine
                                ppp.addEventListener(this);
                            } catch (java.util.TooManyListenersException le) {
                                throw new PPPException("Unable to add event listener");
                            }
                            ppp.open();
                        }

        public void stopDaemon() {
                            // Don't receive any more PPP events
                            ppp.removeEventListener(this);
                            ppp.close();
                        }

        public void dataLinkError(String error) {
                            System.err.println("Error in data link:"+error);
                            ++linkErrors;
                            ppp.close();
                        }
    }
```

Program 21: PPPDaemonListener interface

```
    ……..
    public interface PPPDaemonListener {
        public void serverError(String error);
        public boolean isValidUser(String name, String password);
    }
```

Program 22: Start Daemon

```
    ……..
        public void startDaemon() throws PPPException {
            retryCount = 0;
            try {
                // Add PPP event listener to driver state machine
                ppp.addEventListener(this);
            } catch (java.util.TooManyListenersException le) {
                throw new PPPException("Unable to add event listener");
            }
            ppp.open();
        }

        public void stopDaemon() {
            // Don't receive any more PPP events
            ppp.removeEventListener(this);
            ppp.close();
        }
```

Program 23: PPPEvent

```
    ……..
    public void pppEvent(PPPEvent ev) {
            switch (ev.getEventType()) {
                case PPPEvent.STARTING:
                    try {
                        // Now we need to bring up the physical link

        dataLink.initializeLink();

        ppp.up((SerialPort) dataLink.getPort());
                    } catch (DataLinkException dle) {

        listener.serverError("Data link error:"+dle.getMessage());
                        ppp.close();
                    }
                    break;

                case PPPEvent.AUTHENTICATION_REQUEST:

        ppp.authenticate(listener.isValidUser(ppp.getPeerID(), ppp.getPeerPassword()));
                    break;

                case PPPEvent.UP:
```

```
                        // Reset error count after successfully bring up connection
                        retryCount = 0;
                        ppp.addInterface("ppp0");
                        isUp = true;
                        break;

                    case PPPEvent.STOPPED:
                        ppp.close();
                        if (++retryCount < maxRetries) {
                            ppp.close();

                        } else {

        listener.serverError("Unable to establis PPP connection");
                        }
                        break;

                    case PPPEvent.CLOSED:
                        if (isUp) {

        ppp.removeInterface("ppp0");
                            ppp.down();
                            isUp = false;
                        }
                        try {
                            // Sleep before recycling ppp connection

        Thread.sleep(1000);
                        } catch (InterruptedException ie) {}
                        ppp.open();
                        break;

                    default:
                        break;
                }
            }
```

Program 24: PPPDataLink

```
import javax.comm.SerialPort;

public interface PPPDataLink {
    public SerialPort getPort();
    public void initializeLink() throws DataLinkException;
}
```

Program 25: DataLinkListener

```
……..
public interface DataLinkListener {
    public void dataLinkError(String error);
}
```

Program 26: DataLinkError

```
……..
public void  DataLinkError (String error)
    System.out.println("Error in data limk:" + error);
    ++ linkErrors;
    ppp.close();
……..
}
```

Program 27: PPPSerialLink

```
import javax.comm.*;
import java.io.*;
import com.dalsemi.system.TINIOS;

public class PPPSerialLink implements PPPDataLink {
    protected DataLinkListener listener;
    protected SerialPort sp;
    protected InputStream in;
    protected OutputStream out;

    public PPPSerialLink(String portName, int speed, DataLinkListener listener)
            throws DataLinkException {

            this.listener = listener;
            try {
                // Create and initialize serial port
                sp = (SerialPort) CommPortIdentifier.getPortIdentifier(portName).open("PPPDataLink", 5000);
    sp.setSerialPortParams(speed, SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);


    TINIOS.setRTSCTSFlowControlEnable(0, true);

    sp.setFlowControlMode(SerialPort.FLOWCONTROL_RTSCTS_IN |

                SerialPort.FLOWCONTROL_RTSCTS_OUT);

                    in = sp.getInputStream();
                    out = sp.getOutputStream();
                } catch (Exception e) {
                    throw new DataLinkException("Error configuring serial port" + e.getMessage());
                }
    }

    public void initializeLink() throws DataLinkException {
    }

    public SerialPort getPort() {
            return sp;
    }
}
```

Program 28: PPPModemLink

```
import javax.comm.*;
import java.io.*;
import java.util.TooManyListenersException;

  public class PPPModemLink extends PPPSerialLink implements
SerialPortEventListener {
        private ModemCommand mc;

        public PPPModemLink(String portName, int speed,
DataLinkListener listener)
                throws DataLinkException {

                super(portName, speed, listener);
                try {
                        sp.addEventListener(this);
                } catch (TooManyListenersException tmle) {
                        throw new
DataLinkException("Unable to register for serial events");
                }
                mc = new ModemCommand(sp, in, out);
        }

        public void initializeLink() throws DataLinkException {
                resetModem();
                mc.receiveMatch("RING", null, 0);
                mc.sendCommand("ATA\r", "CONNECT", 25);
        }

        private void resetModem() throws DataLinkException {
                // Clear RTS and DTR
      sp.setDTR(false);
      sp.setRTS(false);

      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {}

                // Set RTS and DTR
      sp.setDTR(true);
      sp.setRTS(true);

      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {}

      // Sync modem to serial port baud rate
      mc.sendCommand("AT\r", "OK", 6);
        }

        public void serialEvent(SerialPortEvent ev) {
                if ((ev.getEventType() == SerialPortEvent.CD)
&& !ev.getNewValue()) {
                        listener.dataLinkError("Lost carrier
detect");
                }
        }
  }
```

Program 29: InitializeLink and ResetModem

```
        public void initializeLink() throws DataLinkException {
                resetModem();
                mc.receiveMatch("RING", null, 0);
                mc.sendCommand("ATA\r", "CONNECT", 25);
        }

        private void resetModem() throws DataLinkException {
                // Clear RTS and DTR
      sp.setDTR(false);
      sp.setRTS(false);

      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {}

                // Set RTS and DTR
      sp.setDTR(true);
      sp.setRTS(true);

      try {
        Thread.sleep(2000);
      } catch (InterruptedException ie) {}

      // Sync modem to serial port baud rate
      mc.sendCommand("AT\r", "OK", 6);
        }
```

Program 30: ModemCommand

```
import javax.comm.*;
import java.io.*;

  public class ModemCommand {
        private SerialPort sp;
        private InputStream in;
        private OutputStream out;

        public ModemCommand(SerialPort sp, InputStream in,
OutputStream out) {
                this.sp = sp;
                this.in = in;
                this.out = out;
        }

        public void sendCommand(String command, String
response, int timeout)
                throws DataLinkException {

                try {
                        // Transmit the command
                        out.write(command.getBytes());
                } catch (IOException ioe) {
                        ioe.printStackTrace();
                        throw new DataLinkException("Error
sending command to modem");
                }

                waitForMatch(response, timeout);
        }
```

```
public void receiveMatch(String match, String response, int timeout) throws DataLinkException {
        try {
                waitForMatch(match, timeout);
                if ((response != null) && (response.length() > 0)) {

out.write(response.getBytes());
                }
        } catch (IOException ioe) {
                ioe.printStackTrace();
                throw new DataLinkException("IO Error receiving a match to:"+match);
        }
}

private void waitForMatch(String match, int timeout) throws DataLinkException {
        try {
                sp.enableReceiveTimeout(100);

sp.enableReceiveThreshold(match.length());

                byte[] mb = new byte[match.length()];
                long timer = 0;
                if (timeout > 0) {
                        // Time out when timer > currentTimeMillis
                        timer = timeout*1000+System.currentTimeMillis();
                }

                StringBuffer modemSpew = new StringBuffer();
                while ((timer == 0) || (System.currentTimeMillis() < timer)) {
                        int count = in.read(mb);
                        if (count > 0) {
                modemSpew.append((new String(mb, 0, count)).toUpperCase());
                                if (modemSpew.toString().indexOf(match.toUpperCase()) >= 0) {
                                        return;
                                }
                        }
                }

                throw new DataLinkException("Timed out waiting for match:"+match);
        } catch (Exception e) {
                e.printStackTrace();
                throw new DataLinkException("IO Error receiving a match to:"+match);
        }
}
```

Program 31: waitforMatch

```
……
        private void waitForMatch(String match, int timeout) throws DataLinkException {
                try {
                        sp.enableReceiveTimeout(100);

        sp.enableReceiveThreshold(match.length());

                        byte[] mb = new byte[match.length()];
                        long timer = 0;
                        if (timeout > 0) {
                                // Time out when timer > currentTimeMillis
                                timer = timeout*1000+System.currentTimeMillis();
                        }

                        StringBuffer modemSpew = new StringBuffer();
                        while ((timer == 0) || (System.currentTimeMillis() < timer)) {
                                int count = in.read(mb);
                                if (count > 0) {
                        modemSpew.append((new String(mb, 0, count)).toUpperCase());
                                        if (modemSpew.toString().indexOf(match.toUpperCase()) >= 0) {
                                                return;
                                        }
                                }
                        }

                throw new DataLinkException("Timed out waiting for match:"+match);
                } catch (Exception e) {
                        e.printStackTrace();
                        throw new DataLinkException("IO Error receiving a match to:"+match);
                }
        }
}
```

Program 32: PPPEthernetDataLogger

```
import java.io.*;
import java.net.*;
import com.dalsemi.tininet.ppp.PPPException;

class DataLogger extends Thread implements PPPDaemonListener {
        // Port number for inbound connections, well above well known ports
        static final int SERVER_PORT = 5588;

        PPPDaemon pppd;
        FluidPressureLogger logger;
        private String name;
        private String password;
```

```
        DataLogger(int samples, int delay, String name, String password)
                throws PPPException, LoggingException {

                // Set authentication information
                this.name = name;
                this.password = password;
                // Create and start the logging daemon
                logger = new FluidPressureLogger(samples, delay);
                logger.start();
                // Create a server to manage PPP dial-up requests
                PPPDaemon pppd = new PPPDaemon(this, "serial0", 19200);
                pppd.startDaemon();
        }

        public void daemonError(String error) {
                System.err.println("Error in PPP server:"+error);
                pppd.stopDaemon();
        }

        public boolean isValidUser(String name, String password) {
                return (this.name.equals(name) && this.password.equals(password));
        }

        public void run() {
                ServerSocket ss = null;
                try {
                        ss = new ServerSocket(SERVER_PORT);
                } catch (Exception e) {
                        e.printStackTrace();
                        // Abort if we can't create ServerSocket instance
                        return;
                }

                while (true) {
                        Socket s = null;
                        try {
                                // Wait for client connections over PPP or Ethernet
                                s = ss.accept();
                        } catch (IOException ioe) {
                                // Shut down the logging daemon
                                logger.stopLogging();
                                System.out.println("Fatal problem with server socket, exiting");
                                ioe.printStackTrace();
                                // Fall out of run method
                                break;
                        }

                        System.out.println("New client:" + s.toString());
                        System.out.println("Local interface:" + s.getLocalAddress());
                        // Create a new thread to handle this connection
                        (new LogWorker(s)).start();
                }
        }

        private class LogWorker extends Thread {
                private Socket s;

                private LogWorker(Socket s) {
                        this.s = s;
                }

                public void run() {
                        DataOutputStream dout = null;
                        try {
                                dout = new DataOutputStream(new
        BufferedOutputStream(s.getOutputStream()));
                                logger.writeLog(dout);
                                dout.flush();
                        } catch (IOException ioe) {
                                System.out.println("I/O error writing log data");
                                ioe.printStackTrace();
                        } finally {
                                try {
                                        s.close();
                                        dout.close();
                                } catch (IOException e) {}
                        }
                }
        }

        public static void main(String[] args) {
                System.out.println("Starting DataLogger ...");
                if (args.length != 4) {
                        System.out.println("Usage: java DataLogger samples delay username password");
                        System.exit(1);
                }

                // Total number of samples to maintain
                int samples = Integer.parseInt(args[0]);
                // Delay between each sample in minutes
                int delay = Integer.parseInt(args[1]);

                try {
                        (new DataLogger(samples, delay, args[2], args[3])).start();
                } catch (Exception e) {
                        System.out.println("Error creating data logger");
                        e.printStackTrace();
                }
        }
}
```