(12) PATENT
(19) AUSTRALIAN PATENT OFFICE

(11) Application No. AU 199519979 B2
(10) Patent No. 706132

(54) Title
System and method for communication with a remote network device

(51)[6] International Patent Classification(s)
G06F 015/16

(21) Application No: 199519979

(22) Application Date: 1995.03.15

(87) WIPO No: WO95/25311

(30) Priority Data

(31) Number	(32) Date	(33) Country
08/213197	1994.03.15	US

(43) Publication Date : 1995.10.03
(43) Publication Journal Date : 1995.11.16
(44) Accepted Journal Date : 1999.06.10

(71) Applicant(s)
Digi International, Inc.

(72) Inventor(s)
Gene H Olson

(74) Agent/Attorney
SPRUSON and FERGUSON,GPO Box 3898,SYDNEY NSW 2001

(56) Related Art
US 5265239
US 4972368

OPI DATE 03/10/95 APPLN. ID 19979/95
AOJP DATE 16/11/95 PCT NUMBER PCT/US95/03183
INT
AU9519979

(51) International Patent Classification 6 :
G06F 15/16

A1

(11) International Publication Number: WO 95/25311

(43) International Publication Date: 21 September 1995 (21.09.95)

(21) International Application Number: PCT/US95/03183

(22) International Filing Date: 15 March 1995 (15.03.95)

(30) Priority Data:
08/213,197 15 March 1994 (15.03.94) US

(71) Applicant: DIGI INTERNATIONAL INC. [US/US]; 6400 Flying Cloud Drive, Eden Prairie, MN 55344 (US).

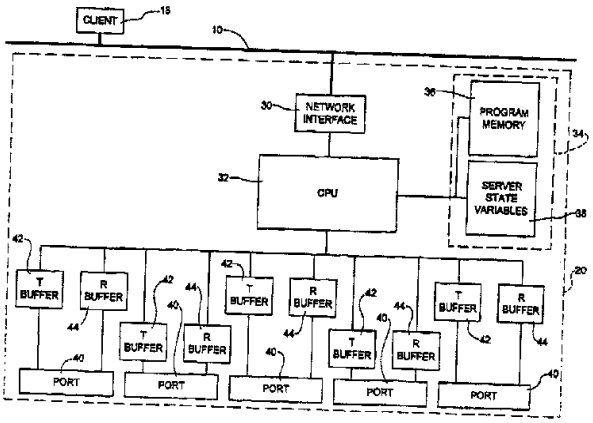
(72) Inventor: OLSON, Gene, H.; 5131 Aldrich Avenue South, Minneapolis, MN 55419 (US).

(74) Agents: TYSVER, Daniel, A. et al.; Faegre & Benson, 2200 Norwest Center, 90 South Seventh Street, Minneapolis, MN 55402 (US).

(81) Designated States: AM, AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, JP, KE, KG, KP, KR, KZ, LK, LR, LT, LU, LV, MD, MG, MN, MW, MX, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SI, SK, TJ, TT, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG), ARIPO patent (KE, MW, SD, SZ, UG).

Published
With international search report.

(54) Title: SYSTEM AND METHOD FOR COMMUNICATION WITH A REMOTE NETWORK DEVICE



(57) Abstract

A terminal server (20) for providing communication between a host computer and a synchronous, asynchronous or parallel communications port across a general purpose network (10) is provided utilizing a unique device driver interface and multiplexing communication protocol. The server (20) communicates data and control commands for multiple ports (40) on a single connection, thereby reducing demands on the network (10). In addition, genuine TTY devices are made available across the general purpose network which have all the characteristics of local communication ports. The present invention also makes possible access to the ports (40) on the server (20) to multiple host computers located on the network (10), allowing fair access to shared resources such as modems and printers. Finally, the invention can also be implemented in hardware, further increasing compatibility with existing host computer software and further reducing host computer overhead.

SYSTEM AND METHOD FOR COMMUNICATION WITH A REMOTE NETWORK DEVICE

Description

Technical Field

This invention relates to the field of communication between digital devices such as computers and computer related peripherals across a general purpose network.

Background Art

The need for digital computers to communicate to each other and to remotely located computer peripherals has long been felt. Numerous technologies have been designed to meet this need, including the use of modems for establishing long distance links over the analog phone network and the creation of networking technologies which allow computers and peripherals to communicate over a data transmission medium.

Modems are used in pairs to establish communication between two computers, with each modem being attached to the serial port of its local computer as well as connected to the analog telephone network. The serial port is a standard interface port on the computer which allows the computer to pass a serial bit stream of information to a peripheral such a modem or a printer. On UNIX based computers and other multi-user computer systems, this same serial port is used for communication with the terminals which allow users to interact with the computer. As a result, it is a relatively simple matter to utilize a modem to allow a remotely located terminal to communicate with a multi-user computer system over the phone lines.

Because modern multi-user computers are capable of supporting numerous simultaneous users, it is desirable to have a number of serial ports through which terminals can communicate with the computer.

Unfortunately, mounting all of the serial ports directly onto the computer can be logistically difficult, since multi-user computers can often support more than fifty terminal connections. As a result, multiplexors have been utilized to combined serial traffic from numerous, internally referenced serial ports into a single communication channel. In this way, only one communication cable needs to be attached to the computer. On the other end of the communication cable is another multiplexing device, which separates the combined data traffic into data for individual serial ports, and then provides the serial ports through which this data can be accessed. This type of multiplexing device, called a ports concentrator, can be used to provide multiple serial ports for terminals in a location remote from the computer. It is even possible to use multiplexing in a system which allows the data on the main communication cable to be transported across phone lines using modems. This allows multiple serial ports to be accessible at great distances from the computer.

Networking technology can also be utilized to allow remote communication between a computer and another computer or peripheral. Networks which connect computers and peripherals within a relatively small area are referred to as Local Area Networks, or LANs. A general purpose network is a communication system utilizing standard hardware and standard communication protocols, and operating in a multi-vendor environment. General purpose network hardware includes LAN technologies like Ethernet and Token Ring. Standard Network protocols include TCP/IP and SPX/IPX.

LANs are generally formed by connecting computers and peripherals together through a transmission medium, and then communicating over the medium by following standardized communication protocols. These protocols set forth such requirements as to how data

should be formatted and how data is designated for a particular device. Data cannot be sent over the network without conforming to these protocols.

Application programs on a host operating system generally depend strongly on the Application Program Interfaces (API) for local communication devices provided by the host operating system. This is especially important on systems like UNIX, whose traditional user interface is terminal command line operation.

General purpose networks most often support two types of communication, a connectionless unreliable datagram service and a reliable bi-directional bytesteam connection. The API to both of these services is very different from the serial port API, and in general programs written to run on serial devices (known as TTYs) require an adaptation layer to run correctly.

In the UNIX environment this is generally done with the network protocols Telnet and Rlogin using a virtual TTY device known as a pseudo-tty or PTTY device. PTTY devices are software entities, not hardware entities, that emulate UNIX TTYs well enough that most command line programs will run with them.

It is important to distinguish these PTTYs from genuine TTYs associated with and controlling actual serial port hardware. A PTTY has a slave side and a master side. The slave side of the PTTY is the TTY emulation device, and is used by programs that require the serial port API to function. The master side interfaces to a network program, such as Telnet or Rlogin, that can send data across the network.

The slave side of the PTTY accepts requests to change user options like line editing characters, tab expansion, and other functions done by the line discipline of the local operating system. However it ignores requests to change baud rate, character size,

action on received errors and BREAK signals, and the like.

The master side of the PTTY accepts reads, writes, and a very limited set of input output controls (ioctls) to change the way data is passed back and forth between the slave and master devices. Data written to the master side of the PTTY appears as received serial data on the slave side of the PTTY, as though it had been received on a serial port, and is processed as input serial data by the line discipline of the operating system. The data then appears as received data when read from the slave side of the PTTY. Data written to the slave PTTY is likewise processed as serial output data by the UNIX line discipline, after which it can be read on the master side.

Ioctl operations made to the slave side of the PTTY are invisible to the master side, hence it is not possible for the program accessing the master side of the PTTY to detect that these calls have been made. The interface is just not rich enough to support any operations that cannot be done in the line discipline of the local operating system.

It should also be noted that in most implementations, there is not a one-to-one correspondence between physical devices and PTTYs. When a connection is made into a UNIX system by Telnet or Rlogin, these program simply allocate the next available PTTY for the session. Hence from session to session a network user is likely to get different PTTY devices, in no obvious pattern. This makes it difficult or impossible to enforce system security or user options based on pseudo-tty devices. To overcome this to some extent, ad-hoc programs are commonly written by terminal server vendors that can be attached to the master side of a particular PTTY, so that the slave side of that PTTY remains attached to that program and is not periodically

reallocated to another use. Such programs can be used for incoming connects and running login programs, but are most often used to simulate dial-out modems, printers, or direct connect serial devices. This use expands the PTTY to other uses than Telnet and Rlogin, but because of their limited emulation of a genuine TTY device, they require varied workarounds and cause continual maintenance and compatibility problems.

Telnet also uses TCP/IP flow control for port data flow control and sends commands in sequence with the data so that commands cannot be processed until the data ahead of them in the data stream have also been processed. As a result, Telnet requires the use of the expedited data feature of TCP/IP to send certain commands such as user requested interrupts. The implementation of this requires that data ahead of the user interrupt be discarded before the interrupt can be processed.

Unfortunately, this procedure can place a tremendous strain on the host computer as well as the network itself. For instance, when typing on a remote terminal connected through the network, each keystroke is processed, sent over the network to the host computer, and then echoed back over the network to the remote terminal. On an Ethernet network, a transmitted character must be transmitted on a 64 byte minimum size message packet and the acknowledge also requires a 64 byte message packet. Thus, two 64-byte packets must be transmitted for each character typed.

In addition, the receiving of the character by the host computer requires a task switch to run the PTTY control software to receive the character, and a task switch back to the PTTY control software to transmit the echo character.

Devices called terminal servers make remote logins, utilizing the TELNET and RLOGIN commands, available to users through a serial port without

requiring the users to be logged onto a computer on the network. A user connected to a serial port on the terminal server can use RLOGIN or TELNET to connect to a multi-user computer on the network. A second user on the terminal server could also connect to the same multi-user computer, and thereby establish a second communications link with its own PTTY assigned to it. Connection made by the terminal server through TELNET, RLOGIN and similar procedures of course embody the same disadvantages as connection made using those procedures through a different computer on the network.

Summary of the Invention

In accordance with one aspect of the present invention there is provided a system comprising:

a server having a plurality of communication ports; and

a host computer having a driver communicatively coupling the host computer to the server via a network connection, wherein the driver emulates the communication ports of the server by defining a corresponding local communication port for each of the communication ports of the server, and further wherein the driver includes an application programming interface (API) by which an application program executing on the host computer is granted full control of one of the communication ports of the server, including hardware and software flow control, as if the communication ports of the server were local to the host computer.

In accordance with another aspect of the invention, there is provided a hardware device for a host computer, wherein the hardware device includes a driver that emulates a plurality of communication ports of a remote server that is communicatively coupled to the host computer via a network connection, wherein the driver defines a corresponding local communication port for each communication port of the server and includes an application programming interface (API) by which an

application program executing on the host computer is granted full control of one of the communication ports of the server, including hardware and software flow control, as if the communication ports of the server were local to the host computer.

The present invention includes a communication system which ameliorates the disadvantages of the prior art by allowing a host computer full access to the asynchronous ports located on a remote terminal server across a general purpose network. By utilizing a unique device driver interface and multiplexing communication protocol, the server is able to communicate data and control commands for multiple ports on a single connection, thereby reducing demands on the network. In addition, genuine TTY devices can be provided across the network which have all the characteristics of local communication ports. It is also possible to provide access to the ports on the server to multiple host computers located on the network, allowing fair access to shared resources such as modems and printers. Finally, the invention can also be embodied in hardware, further increasing compatibility with existing host computer software and further reducing host computer overhead.

Each of the embodiments of the present invention operates over the reliable communication link provided by a general purpose network between a client (usually a host computer) who wishes to access or provide access to remote

communication ports and a server that incorporates such ports.

Each of the embodiments of the present invention differs from traditional PTTY implementations in that it provides genuine TTY devices across a general purpose network. Unlike the PTTY implementations, the TTYs of these embodiments of the current invention have a one-to-one correspondence with actual hardware, and allow the host operating system to control that hardware as if the serial ports were directly connected to the host operating system and dedicated to this use.

Where it does not interfere with the assumptions made by application programs, each of the embodiments of the current invention then loosens these restrictions, and extends the functionality of these ports to allow sharing with other hosts on the network, and with dynamic port allocation with PBX-style hunt groups. The present invention as embodied allows each operating system to maintain a one-to-one correspondence with genuine TTYs, but can optionally allow those TTYs to be accessed for other purposes when they are not currently in use by the current operating system.

Each client that wishes to use one or more ports on the server opens a single connection to that server utilizing the protocol provided by embodiments of the present invention (the "Communication Protocol"). All communication between the client and server which uses the Communication Protocol then proceeds using this single connection, no matter how many ports are used by the client. Additional connections between the client and server may be opened under prior art protocols such as Telnet and Rlogin, however the existence of these prior art protocol connections does not affect the ports under the control of the embodiments of the present invention's Communication Protocol. When the connection between the client and server is broken for any reason, all the ports

under Communication Protocol control are closed, and all resources claimed by that client are returned.

The existence of a Communication Protocol connection does not automatically grant the client access to all port resources on the server. Before the client accesses a port, the client must first successfully OPEN the port, after which time the client is granted exclusive access to the port until the client CLOSES the port.

Once a port is opened, the client may send requests to change baud rate, control modem settings, and the like. The client may send data out the port, and the server automatically sends the client all data received in the port. Under the Communication Protocol, the client may request status updates from the server. The updates can be made immediately upon request, or the client may request the server inform the client of any changes to various status conditions (such as modem input signals) as they change.

Brief Description of the Drawings

Figure 1 illustrates a general purpose computer networking linking a variety of host computers and a server of an embodiment of the present invention.

Figure 2 illustrates a client computer attached across a general purpose network to a server of an embodiment of the present invention, including the critical components of the server.

Figure 3 illustrates a wrap-around sequence space with a single sequence number pointing toward a location in the sequence space.

Figure 4 illustrates a wrap-around sequence space with two sequence numbers pointing toward locations in the sequence space

Figure 5 illustrates the process by which a server controls the flow of data transmitted from a client through a server to a port.

Figure 6 illustrates the process by which a client controls the flow of data transmitted from a client through a server to a port.

Figure 7 illustrates the process by which a server controls the flow of data transmitted from a port through a server to a client.

Figure 8 illustrates a client computer attached across a general purpose network to a server of an embodiment of the present invention, including the critical components of the client driver.

Detailed Disclosure of the Invention

FIG. 1 shows a general purpose network 10 interconnecting a variety of computers including a UNIX computer 12, a Digital Equipment Corporation's VAX computer 14 and a Novell Network of PCs at a particular location 16. The network 10 can utilize any of a variety of standard networking communication protocols, such as TCP/IP, SPX/IPX and X.25, as long as the protocol provides for error free byte stream data transmission. The communication protocol operates on a specific logical topology such as Ethernet or Token Ring.

A typical network connecting the computers and networks shown in FIG. 1 would utilize the TCP/IP protocol running on top of Ethernet. The TCP portion of the protocol takes byte stream data and converts it to packets. The IP portion takes the packets and forms datagrams, while Ethernet takes the datagrams and forms frames. This typical configuration will be utilized for the purpose of explaining the present invention, although the invention itself is independent of the network protocol utilized.

A server 20 operates to connect various ports (not shown) to the network 10. Communication lines 22 can be connected to the ports on the server 20. Although only five communication lines 22 are shown in FIG. 1, the server 20 typically has sixteen or more ports which can be connected to the network 10.

FIG. 2 shows the primary elements of the communication between one or more clients 18 and the server 20 across a general purpose network 10. The client 18 can be a host computer such as a UNIX computer 12 or a VAX computer 14, or even a local area network 16. The server 20 is connected to the network 10 through the server's network interface 30. The network interface 30 provides the industry standard hardware and software control necessary for the server 20 to communicate over the network 10 in the same way as any other device on the network 10. As with all industry standard network interfaces, the network interface 30 is capable of receiving data frames from the network 10, determining whether the data is addressed to server 20, and, if so, make the data available in a byte stream form by removing the network formatting and reassembling the frames, datagrams and packets. In other words, the network interface 30 contains the functionality of the Transport, Network, Link and Physical layers of the International Standards Organization Open Systems Interconnection (ISO/OSI) model.

The byte stream data provided by the network interface 30 is then supplied to a central processing unit (CPU) 32, which coordinates the implementation of the Communication Protocol. The program modules necessary for implementation of the Communication Protocol modules necessary for implementation of the Communication Protocol are stored in the program memory portion 36 of the server memory 34.

After the CPU 32 has analyzed the data received from the network interface 30, raw data is passed through to the appropriate port 40 on server 20. The ports 40 on the server 20 may be asynchronous, synchronous or parallel ports. Attached to the ports 40 may be a variety of devices, such as terminals, printers or modems.

Data that is sent from the client 18 for transmission through the ports 40 is stored in an output or transmit buffer 42 unit the port 40 is ready to transmit the data. Data that is sent into the server 20 through the ports 40 is handled similarly. The data coming from port 40 for transmission to the client 18 is stored in input or receive buffer 44 until it is ready to be transmitted over the network 10 to the client 18. The Communication Protocol is executed by the CPU 32 and the step of formatting the byte stream transmission into the frames, datagrams and packets required on the network 10 is handled by the network interface 30.

The server memory 34 also contains server state variables 38, which contain the status information necessary to fully implement the Communication Protocol. The types of server state variables 38 stored in the preferred embodiment of the present invention are shown in Table 1.

These variable types represent a variety of data structures, and are used in various ways to control the workings of the server 20 and to facilitate communication between the client 18 and the server 20. To clarify the purpose of these variables, they are discussed below in detail.

The Client Connections list maintains a list of clients 18 that currently have a connection with the server 20 open under the Communication Protocol. In the

preferred embodiment, the server 20 can support eight or more simultaneous client connections.

The **Message Build Interval** controls the polling interval to determine how often the server 20 internally polls its state to determine whether information needs to be constructed according to the Communication Protocol and transmitted to the client 18. In the preferred embodiment, this polling takes place approximately fifty times per second. This results in a worst case delay time of twenty milliseconds, with an average of ten milliseconds. This is adequate for users at terminals, and for most sliding window protocols.

By transmitting information only at the message build interval, the server 20 is able to accumulate information during the interval. Thus, data is sent in larger packets than if data were sent a character at a time, which limits the demands on both the network 10 and the client 18.

The **Port Open State** variable is a data structure which tracks which clients 18 have a connection open to a port 40, and which clients 18 are waiting to open that port 40. In other words, for each open port 40 this data structure tracks which client 18 currently holds the port 40 and which clients 18 are on the Open wait list. Further details on opening a port 40 are set forth below.

The **Port Baud Rate** variable stores the baud rate of each port 40 in such a way that the baud rate of the port 40 is 1,843,200 divided by the **Port Baud Rate** variable. This representation requires only sixteen bits to represent all standard baud rates in the range 50-115K Baud.

The **Processing Flags** variables control, character size, hardware state, and input/output character processing for each port 40. There are four sets of **processing flags** that are stored in the server

state variables: CFLAGS, IFLAGS, OFLAGS AND XFLAGS. CFLAGS, IFLAGS and OFLAGS are standard flags provided by in the *termio* interface of most standard UNIX operating systems, and are used in the present invention as is standard in the industry. (Unix Programmers Manual, Termio).

CFLAGS has settings which control UART hardware settings on the port 40, including character size, stop bits, and parity. The actual CFLAGS settings that it makes sense to implement in the server 20 include CBAUD, CSIZE, CSTOPB, CREAD, PARENB, PARODD, and HUPCL. Similarly, IFLAGS has settings which control the processing of data at the server 20 received from the port 40. IFLAGS settings include control over the interpretation of Break conditions, error handling and software flow control settings. The IFLAGS implemented on the server 20 are IGNBRK, IGNPAR, PARMRK, INPCK, ISTRIP, IXON, IXOFF, IXANY, DOSMODE. OFLAGS controls the server 20 processing of data intended to be outputted over the ports 40. OFLAGS settings control such things as CR and NL settings and tab delay, and are defined by *termio* to include OPOST, OLCUC, ONLCR, OCRNL, ONOCR, ONLRET, OFILL, OFDEL, NLDLY, CRDLY, TABDLY, BSDLY, VTDLY, FFDLY.

In the preferred embodiment, various flags settings which are no longer used with modern communications equipment are not supported. These flags settings include OFILL, OFDEL, NLDLY, CRDLY, BSDLY, VTDLY and FFDLY, all from OFLAGS. Each of these settings could be supported with only slight modifications to the preferred embodiment, but the decision whether or not to support these setting is irrelevant to the present invention.

XFLAGS is a set of flags taken from *termio* LFLAGS, plus some additional functions. The XFLAGS settings are set forth in Table 2.

The XFLAGS variables (except for XCASE which is a UNIX line discipline or LFLAG variable) are extensions to the UNIX feature set either to expand the interface to multiple host operating systems or to provide value-added features. These XFLAGS settings interact with the settings of the CFLAGS, IFLAGS and OFLAGS settings. The details of the interactions are as follows.

If the XPAR setting of XFLAGS is reset or if the underlying hardware does not support mark/space parity, then the PARODD setting of CFLAGS selects odd parity if set and even parity if reset. Otherwise PARODD selects space parity if set and mark parity if reset.

If PARMRK of IFLAGS and XMODEM of XFLAGS are set, any change in one or more modem signals is encoded in the input stream as FF 80 MM, where MM is the updated modem signal value.

If XCASE of XFLAGS is set, certain non-alphabetic printing characters are converted to two-character equivalents on output according to UNIX specifications.

XTOSS of XFLAGS controls whether a character that resumes output during flowing control is ignored or not. When IXANY of IFLAGS is reset, output is restarted only when a matching XON or XXON character is received. However, when IXANY of IFLAGS is set, any received character resumes output. Thus, if XTOSS is set, the character that resumes output is discarded. If XTOSS is reset, any non-flow control character resuming output is placed in the data stream.

When XIXON of XFLAGS is set, an extra set of output flow control characters is enabled. Receipt of XXOFF stops output while XXON resumes it.

The **Flow Control Characters** variables determine what the flow control characters will be. Two sets of output flow control characters are supported and one set of input flow control characters is supported. In

addition, a flow control escape character is supported to allow input of flow control characters without interpretation. The **Flow Control Characters** include 1) the standard software flow control start character (XON), 2) the standard software flow control stop character (XOFF), 3) the extra software flow control start character (XXON), 4) the extra software flow control stop character (XXOFF) and 5) the flow Control Escape Character (LNEXT).

After receipt of the LNEXT character, the next character received is not recognized as a flow control character, and both characters are placed into the input data stream.

Returning to the Server State Variables 38, the provision of full modem control facilities to the client 18 are provided through eight **Modem Control** variables. Each of these variables use masks to represent the operating state of a modem. The seven modem variables are set forth as follows in Table 3.

All variables shown in Table 3 represent sets of modem signals, using the bit assignments shown in Table 4.

The use of the different **Modem Control** variables are as follows. The MOUT **Modem Control** Variable contains the values of RTS and DTR last requested by the client 18. These values are changed by issuing a request to the server 20. When the RTS/DTR bits in MFLOW are set, the corresponding values set in MOUT are temporarily ignored, and those signals indicate whether the server 20 is ready to receive data. When receive is paused, these signals are driven low. When receive is restarted, the signals are driven high.

MFLOW is used to inhibit the "in-band" output of data. Data, commands, and responses that are handled in the same sequence as the normal flow of data in and out of data ports is called in-band data. Data,

commands, and responses that are transmitted ahead of normal data, and are processed immediately upon receipt are called out-of-band data. In-band output is inhibited when any of CTS/DSR/DCD/RI are set in MFLOW and at least one of the corresponding hardware input signals is low.

Flow control characters and transmit immediate characters are not governed by software flow control or MFLOW. They are transmitted according to the CTS/DSR/DCD/RI flow control constraints given by MCTRL. The server 20 automatically resets any bit in MCTRL that is not set in MFLOW.

MSTAT contains the current state of the input and output modem signals.

MLAST contains the last modem status reported to the client 18.

MTRAN contains the modem signals that have changed since they were last reported to the client 18. It is possible for MSTAT to equal MLAST and still have bits set in MTRAN. This condition indicates that some signals in MSTAT made at least one transition, and then returned to the value last reported to the client 18.

When any of the six modem signals are set in MINT, corresponding changes in those modem signals are reported to the client 18.

The server 20 maintains per-port statistical counters in the server state variables 38 to keep track of the number of receive character errors occurring on that port 40. These counters are referred to as the **Line Error Counters**. A different counter is utilized to track each of the following counts: the number of UART overrun errors, the number of buffer overflow errors, the number of framing errors, the number of parity errors and the number of breaks received. Each of these variables are 16-bit wrap-around counters that keep track of the number of

line errors that have occurred in each category for that port 40.

An additional **Line Error Counter** variable tracks the report time in milliseconds. If this variable is zero, line errors are not automatically reported to the client 18. Otherwise the server 20 inspects the counters at each specified interval, and sends a complete report to the client 18 if any have changed since the last report.

The **Send Break** and **Send Immediate** variables allow a client 18 to request the server 20 to immediately send either a Break or another character to a port 40. The **Send Immediate** variable is a one character buffer which contains the character that the client 18 wishes to send. The **Send Break** variable contains the length in milliseconds of the requested break. Two special cases exist in the contents of the **Send Break** variables. When an FFFF value is sent to **Send Break**, this denotes an infinite break time. A 0000 sent to **Send Break** cancels any break in progress. Any other value sent to the server 20 is simply added to **Send Break** time, and the break proceeds for the combined duration of the two requests.

Since the current invention does not utilize TCP/IP flow control for port flow control, data received by the client 18 or server 20 can always be immediately processed. As a result, it is easy to send out-of-band commands and data without disrupting the flow of in-band data.

The server 20 also maintains a list of status conditions in the Server State Variables 38. These status conditions are stored in the **Event Reporting** variables, and represent conditions that are of general interest to the client 18. The client 18 can poll these, or request that the server 20 automatically send the appropriate event information whenever the conditions

change. The **Event Reporting** variables allow the server 20 to track the sending of this event information. Table 5 sets forth the various **Event Reporting** variables.

EINT can be set by the client 18 to create a predetermined condition which will cause event information to be sent by the server 20 to the client 18. Event information is sent to the client 18 whenever (ETRAN & EINT) is non-zero.

Note that it is possible for ESTAT to equal ELAST and still have bits set in ETRAN. This condition indicates that some conditions in ESTAT made at least one transition, and then returned to their ELAST values before they could be reported to the client 18.

The events that are recorded in these variables are set forth in Table 6, along with the masks which show how the events are indicated. The events listed in Table 6 are communication events well understood in the context of this invention.

The last two types of variables stored in the server state variables 38 are the **Input Sequence and Receive Buffer Parameters** variables and the related **Output Sequence and Transmit Buffer Parameters.** These variables represent the state of the transmit buffers 42 and the receive buffers 44 which are associated with each port 40.

To communicate the flow of data in the buffers 42, 44, the preferred embodiment of the server 20 uses sixteen bit wrap-around sequence numbers. Two sequence numbers are associated with each buffer 42, 44. The first sequence number communicates the sequence number of the next byte to be placed into the buffer. This first sequence number is labeled RIN for the receive buffer 44 and TIN for the transmit buffer 42. The second sequence number communicates the sequence number of the next byte to be removed from the buffer. This second sequence number is labeled ROUT for the receive buffer 44 and TOUT

for the transmit buffer 42. These sequence numbers begin at zero and advance by one for each byte of data transmitted or received. When the sequence numbers reach FFFF, they wrap back to zero and continue when the next byte of data is transmitted or received. It should be noted that the TIN, TOUT, RIN, and ROUT variables are sequence number variables used to communicate between the client 18 and the server 20. They do not specify the physical positions of data in the transmit and receive buffers 42, 44, which is done through techniques well known in the art and not described herein.

These sequence numbers, and the other variables which make up the **Input Sequence and Receive Buffer Parameters** and the **Output Sequence and Transmit Buffer Parameters** variable types are shown in Tables 7 and 8.

Wrap-around sequence numbers like TIN, TOUT, RIN and ROUT have some very helpful mathematical properties. However, sequence numbers can be confusing, especially when doing addition and subtraction, since special definitions of these functions must be used as a result of their wrap-around nature.

To clarify their use, an example embodiment of a transmit buffer 42 will be discussed, as illustrated in FIGs. 3 and 4. In the embodiment of FIG. 3, the TOUT sequence pointer (pointing to the next byte of data to be transmitted out of the buffer 42 to a port 40 and represented in FIG. 3 by arrow 50) is set to byte S. When n bytes of data are transmitted, the transmitted bytes are numbered S though S + (n - 1). By convention, we say the buffer has transmitted bytes between S and (S + n). However, since we are adding an ordinary number to a sequence number that wraps at 0xFFFF, we must compute the expression S + n using the following formula:

$$(S + n) \ \& \ 0xFFFF$$

with & indicating a logical AND operation. Thus, where S is 0xFFF8 and n is 5, the formula for computing S + n give us 0xFFFC. Similarly, where S is 0xFFF8 and n is 9, S + n is 2.

After transmitting n bytes, the TOUT sequence pointer must be advanced to the new location of the next byte to be transmitted. This is done by replacing the old value of TOUT with the computed result S + n.

To determine the number of bytes that exist between two sequence number, subtraction is used. However, a special rule must be used to determine the result of the subtraction of one sequence number from another. Thus, to determine the number of bytes between sequence number s1 and s2, the following formula is used:

```
(s2 - s1) & 0xffff
```

This formula works even if sequence s1 is numerically greater than sequence number s2. For example, if s1 = 0xFFF0 and s2 = 3, then (s2 - s1) & 0xFFFF is 0x13, the correct answer.

In fact the concept of distance between sequence numbers is so important that it is useful to define it as:

```
DIST(from, to) = (((to) - (from)) & 0xffff)
```

Given a pair of sequence numbers s1 and s2, it is true that s3 is the sequence location of a byte between s1 and s2 only if:

```
DIST(s1, s3) < DIST(s1, s2)
```

This formula can be read as: "s3 is between s1 and s2 if and only if the distance from s1 to s3 is less than the distance from s1 to s2."

Sequence numbers can communicate the amount of data, or the amount of free space in a remote buffer. Assume that in the transmit buffer 42 for a particular port 40, as shown in FIG. 4, the TOUT sequence number 50 is set to s1, while the TIN sequence number 52, representing the next byte to be transmitted through the port 40, is set to S2. In FIG. 4, both TOUT and TIN are represented by arrows pointing to locations in transmit buffer 42. It is possible to determine that the number of bytes in the transmit buffer 42 for that port is:

DIST(s1, s2)

In addition, if the transmit buffer 42 for that port is of fixed size SIZE, it is clear that the number of free bytes in the buffer is:

SIZE - DIST(s1, s2)

Both the client 18 and the server 20 make use of the special characteristics of the wrap-around sequence numbers to control the in-band flow of information between them. Both client 18 and server 20 send in-band data to the other only when pre-authorized to do so. This pre-authorization is accomplished by communicating the amount of data that the other is allowed to transmit to it. When the sender has sent enough data to fill this space, the sender pauses in transmission until the receiver removes some of the data and informs the sender that additional space is available.

To control the flow from client 18 to server 20, the server 20 provides the client 18 with the size of its transmit buffer 42, and the sequence of the first byte currently in the transmit buffer 42 (TOUT). The client 18 keeps track of the data that has previously

been sent to the server 20 by keeping its own client 18 send sequence number. By using this send sequence number, along with the information provided by the server 20, the client 18 can compute the amount empty space left in the server's transmit buffer 42. The client 18 therefore will not send any more data than will fit in that empty space.

This process is shown in detail by the flow chart of FIG. 5. The indication to start transmitting activity is shown in box 100 on the chart. The first step is to determine whether data has been received from the client 18 through the network interface 30 of the server 20. If so, as indicated in box 102, it is necessary to receive the data and place the data into the transmit buffer 42 for the appropriate port 40. Whenever data is placed in the transmit buffer 42, TIN is incremented, as is shown in box 104.

Whether or not data was received from the client 18, it is possible that data exists in the transmit buffer 42 that can be sent out through the port 40, as shown at query box 106. If data does exist, the data in the buffer 42 should be sent out over the port 40, box 108, and TOUT should be incremented, box 110.

At this point, the server 20 must determine whether it is necessary to report TOUT to the client 18. Three tests, indicated by query boxes 112, 114 and 116, are utilized in the preferred embodiment. The first test 112 compares TOUT, which indicates the next output byte to be transmitted, with TPOS. TPOS is one of the **Output Sequence and Transmit Buffer Parameter** variable stored in the server state variables 38. TPOS indicates the last value of TOUT that was reported to the client 18. If the difference between TOUT and TMAX, computed using the sequence number subtraction formula described above, is greater that TMAX, the TOUT will be reported. TMAX is also stored in the server state variables 38, and is set

by the client 18 to determine the transmission intervals at which TOUT will be reported. Note that whenever TOUT is reported to the client 18, at box 120, TPOS is then set to the value of TOUT at box 122.

The second test for reporting TOUT, at Box 114, determines whether the data was in the buffer 42 at least TTIME milliseconds ago, and since that time TOUT has not been reported, with TTIME being a server state variable 38 which can be set by the client 18. If so, TOUT will be reported to client 18.

The final test, 116, compares the value of TOUT with the value of TREQ. TREQ, stored as one of the server state variables 38, is set by the client 18 so that when TOUT passes TREQ, TOUT is reported to the client 18. Mathematically, to compare the wrap-around sequence number TOUT with TREQ, it is necessary to test their relationship by comparing whether (TIN - TREQ) is greater than or equal to (TIN - TOUT), using the sequence number subtraction formula described above. If TOUT has passed TREQ, then TREQ becomes invalidated, box 118, and TOUT is reported to the client 18, box 120.

To allow for flexibility in the client 18 driver software, the client 18 may send data to the server 20 whenever it is convenient to do so, so long as the client 18 does not overflow the server's transmit buffer 42. The procedure for making sure that the transmit buffer 42 does not overflow is shown in FIG. 6.

When a client 18 wishes to transmit data to a port 40, the driver on the client 18, which is handling the Communication Protocol for the client 18, must request that the server 20 open the port 40 for it, as shown in box 130. The details of opening a port 40 are explained below. After a successful open, the client 18 assumes that TSIZE, the size of the transmit buffer 42 connected to the open port 40, and TOUT are zero. In addition, the client 18 resets its own Send Sequence

Number, which is a wrap around sequence number variable stored and maintained on the client 18. The client 18 then requests TSIZE from the server as shown in box 132. Note that the client cannot actually send data, as shown in box 140, until TSIZE is received from the server 20 as explained below.

The client 18 next determines if data has been received from the host computer, at box 134. If data has been received, the data must be prepared for transmission, box 135. The preparation for transmission is later discussed in detail.

The client 18 then tests if there is any data to be sent at box 136. If data is ready to be sent, the client's driver must compute the maximum number of bytes that can be accepted by the server 20 without overflowing the transmit buffer 42, box 138. This is accomplished by comparing the client's Send Sequence Number with the most recently reported value of TOUT, and subtracting this difference from TSIZE:

```
Max # Bytes to Send =
     TSIZE - (Send Sequence # - TOUT))
```

Of course, the subtraction of the sequence number TOUT from the client's Send Sequence Number must be done in compliance with the formula for subtracting sequence numbers.

If the calculated maximum number of bytes to be sent is greater than zero, box 140, then up to that number of bytes can be sent to the server 20, as shown in box 144. The client's Send Sequence Number must then be incremented by the number of bytes sent, box 146.

Regardless of whether data has been sent, the client 18 must next determine whether the server 20 has transmitted the value for TOUT or TSIZE, as shown in box

148. If so, the value of TOUT or TSIZE as stored at the client 18 is updated, box 149.

To avoid falsely reporting that the transmitter is idle when it is not, the server 20 recognizes a very special case. When the transmit buffer 42 is empty, but the UART is still busy sending in-band data, the server 20 reports (TOUT - 1) instead of TOUT. This convention allows the client 18 to assume all data has been successfully sent when the client receives TOUT equal to the Send Sequence Number of the last data sent to the server 20.

The handling of data received from the port 40 and transmitted from the server 20 to the client 18 is shown in FIG. 7. The server 20 is not authorized to transmit data to the client 18 until the server 20 receives a value for RWIN, as seen in box 150. RWIN is an **Input Sequence and Receive Buffer** variable which is set by the client 18 to indicate the highest sequence number the client 18 wishes to accept.

When the initial RWIN is received from the client 18, the server 20 examines the port 40 to see if incoming data is available to be placed into the receive buffer 44, shown at box 152. If data has been received, it is placed in the receive buffer 44 and the RIN sequence number is incremented accordingly. This is shown in box 154 and box 156.

Because it is not known by the server 20 how quickly the client 18 will be able to receive data from the server 20, the server 20 has the ability to implement input flow control on the port 40. Flow control is initiated when the amount of data remaining in the receive buffer 44 (calculated as RIN - ROUT) exceeds the value of RHIGH, another **Input Sequence and Receive Buffer**. Thus, after data has been placed in the receive buffer 44 and RIN has been incremented, the server 20 determines whether input control shall be invoked, boxes

162 and 164. Flow control is released when the amount of data in the receive buffer 44 drops below **Input Sequence and Receive Buffer** RLOW, which is tested after data is sent to the client 18 as shown in boxes 158 and 160.

If the review of the port 40 in step 152 did not find new data on the port 40, the server 20 then determines whether data is available to be sent in the receive buffer 44. If not, as indicated in FIG. 7 at box 166, the server 20 again waits for data at the port 40.

If data is available on the receive buffer 44, then the server 20 must compute the maximum number of bytes that can be sent to the client 18, at box 168. The number is determined by the simple formula (RWIN - ROUT), calculated according to the sequence number subtraction formula. If the maximum number of bytes to send is zero, then the server 20 must wait for RWIN to be updated, shown as at query box 170. If the calculated number is greater than zero, then the server 20 must determine whether it is appropriate to send the data bytes to the client 18 at this time. Two tests are used to trigger the sending of data. The first test, at box 172, compares the number of bytes in the receive buffer 44 (RIN - ROUT) with RMAX, an **Input Sequence and Receive Buffer** which can be set by the client 18. If the bytes in the receive buffer 44 exceeds RMAX, then up to the maximum number of bytes allowed is sent to the client 18, box 176, and ROUT is incremented accordingly, box 178.

If the number of bytes in the receive buffer 44 did not exceed RMAX, then the server 20 determines if data has been in the receive buffer 44 at least RTIME milliseconds ago and since that time data has not been sent to the client 18. If so, then, as shown in query box 174, data is sent to the client 18.

If data is not sent to the client 18, the data remains in the receive buffer 44 until one of the two conditions for sending data, 172 or 174, becomes true.

After a port 40 is opened, RMAX is initialized to one, RTIME to zero, RLOW to 1/4 RSIZE, RHIGH to 3/4 RSIZE and all the sequence numbers are zeroed. These variables however can be altered as desired.

Generally, the server 20 sends all available data for each port 40 when it sends any data for that port 40, unless it is restricted by the calculated maximum number of bytes that can be sent. This convention reduces the number of packets both server 20 and client 18 must process, and generally improves client 18 efficiency.

On the client side, in the host computer, the operation can vary greatly depending upon the operating system of the host computer. For the purposes of describing the operation on a client 18, the implementation on a UNIX host system will be outlined, as shown on in Figure 8. There is provided access to the ports 40 on the server 20 through the use of a client driver 200 which implements the API of the host operating system by emulating a driver for a set of locally connected serial ports 204. To do this, the client driver 200 maintains for each remote server port 204, an input or receive buffer 206, an output or transmit buffer 208, and information concerning the state of the remote server 20 as well as the interface to the host computer, which are stored in status memory 210.

The driver 200 also interfaces to a standard network interface 212 which is connected to the general purpose network 10. All communication from the client driver 200 to the server 20 is transmitted over the network 10. The network interface 212 maintains only one connection over the network 10 for each server 20, regardless of the number of ports 40 that the driver 200 has open under the Communication Protocol.

When the system is booted, the operating system start-up procedure initializes the driver 200, and for

each server 20 starts up a user mode daemon 214 which opens a STREAMS connection 218 to the server 20. The daemon 214 then opens a STREAMS connection 220 to a control device 202 associated with the driver 200. The Daemon 214 next, using the STREAMS interface, requests that the operating system link the stream 218 below the control device 202 so that the driver 200 can directly send and receive data on the network connection to the server 20 w/o further interaction from the daemon 214. By doing so, the driver 200 is able to both transmit TCP/IP data to and receive TCP/IP data from the server 20 in the STREAMS queue service routine of the control device. This eliminates the task switch overhead required by the prior art.

The connection made is a bytestream connection, allowing the driver 200 to reliably send and received sequenced data to the server 20. The driver 200 relies completely upon the reliability of this connection and has no provision to retry for lost packets and the like. Should the driver 200 ever detect an error in the data received from the server 20, the driver 200 passes a hangup signal up to the daemon 214, which then closes and attempts to reopen the connection to the server 20. Such an error cannot occur during normal operation. The error can only occur due to a failure of the remote server software, the networking software, the driver software, or some sort of computer failure. The standard TCP/IP software in the host computer effectively insures this sort of error cannot occur due to lost or garbled packets as normally happens on a general purpose computer network 10.

Once the driver 200 has access to the network connection, the driver requests the server 20 to return information as to the number of serial ports 40 on the server 20, and other hardware and software characteristics of the server 20.

When a user mode task 216 attempts to open a TTY serial port 204, the host operating system 215 makes an open call to the driver 200. If the port 204 is not already open by another user mode task 216, the driver sends an open request to the server 20 to gain exclusive access to the port 40 on the server 20 associated with the TTY port 204 of the driver 200. If the port 40 is available, the server 20 responds, granting exclusive access to the client 18 until the client 18 closes the port 40, or the connection to the client 18 is lost.

If the port 40 is not available, the action taken depends on the type of open request originally made by the user mode task 216. A user mode task 216 may request that the open fail if it cannot be done immediately, or it may request that the request wait until the port 40 becomes available. In the former case, the driver 200 requests an immediate open to the server 20, which the server 20 rejects if the port 40 is busy, and the driver 200 then returns an error to the user mode task 216. In the latter case, the driver 200 issues a waiting open to the server 20, asking to be put on a queue of waiting clients until the port 40 eventually becomes available. The server 20 then returns an indication that the request is queued, and the driver 200 puts the user mode task 216 to sleep until the server 20 notifies the driver 200 that the port 40 is available.

Once the driver 200 has successfully opened a server port 40, the driver 200 sends inquiry packets to the server 20 to learn the hardware and software characteristics of the port 40, including the hertz value of the baud rate generator, and whether the port 40 can support mark and space parity. The driver 200 also sends the server 20 the size of the receive buffer 206 for port 204 so that the server 20 knows how much data can be received by the client 18 without further authorization. The server 20 responds to the inquiries, sending the

characteristics of the port 40, and including the size of transmit and receive buffers 42, 44 of the port 40.

When the driver 200 has received the reply from the server 20, the driver 200 is ready to send and receive data to the port 40, and make control and status inquiries on behalf of a user mode task 216, when requested to do so by the host operating system open / close / ioctl (input/output control) interface. Thereafter, all data received on the port 40 of the server 20 is automatically sent to the client 18, where it is stored in a receive buffer 206. The server 20 is initially authorized to send as much input port data as will fit in this buffer 206, but no more, so there is no possibility of a buffer overrun.

When a user mode task 216 requests to read data, the host operating system calls the read routine of the driver 200. If sufficient data has been received from the server 20, according to the parameters of the read request, the driver 200 returns that data to the host operating system 215 which in turn passes the data to the user mode task 216. If sufficient data is not available, the driver 200 puts the user mode task 216 to sleep until the data arrives, or until the read request times out or is interrupted according to the API of the host operating system 215.

After the driver 200 has removed data from the receive buffer 206, the driver 200 informs the server 20 that this data has been removed by incrementing a sequence number (RWIN) by the number of bytes removed, as described above. The server 20 is then authorized to send additional data until that data reaches the sequence number RWIN.

When a user mode task 216 requests to write data, the host operating system 215 calls the write routine of the driver 200. If the driver 200 then copies as much of the user data as will fit into the transmit

buffer 208. If all of the data could be placed in the transmit buffer 208, the server 20 completes the request, and the user mode task 216 is allowed to continue. If not all of the data fit in the transmit buffer 208, the driver 200 puts the user mode task 216 to sleep until data can be sent to the server 20, freeing up enough space in the transmit buffer 208 so the remaining data can be placed in the buffer 208.

When a user mode task 216 makes a control or inquiry request (known as an ioctl request), the host operating system 215 calls the ioctl routine in the driver 200. If the request is to change the hardware or software characteristics of the port 204, the driver 200 notes the change in the status memory 210, and sends a command to the server 20 to make the necessary changes. If the request is an inquiry request which the driver 200 can respond to without consulting the server 20, the driver 200 will do so. If the request requires data in the server 20, the driver 200 sends an inquiry request to the server 20 for this information and uses the information returned to satisfy the request of the user mode task 216. The types of requests that can be made by user mode tasks 216 are very diverse, and the action takes varies widely. In some cases the request can be satisfied immediately, and in some cases the user mode task 216 must be put to sleep until the request can be completed.

When a user mode task 216 makes the last close to the TTY port 204, the host operating system calls the close routine of the driver 200. This causes the driver 200 to put the user mode task 216 to sleep, to wait until all the data in the transmit buffer 208 has been transmitted and to wait until any pending request to the server 20 are complete. The driver 200 then sends a close request to the server 20. Upon receipt of the close request, the server 20 de-allocates the port 40 and

possibly reallocates it to another client 18. The driver 200 then completes the close, and the user mode task 216 continues.

To optimize the efficiency of the operations of the driver 200, and minimize the load on the host operating system 215, driver 200 communications the server 20 are made at periodic intervals about fifty times/second. At each such interval, the driver 200 inspects each active port 204 that is open to that server 20, and places requests and data for all such ports 204, 40 into a common message, and sends that message to the server 20 as a single operation. This action minimizes the total number of TCP/IP messages (or other general purpose network messages) that must be sent to the server 20, and so enhances the operation of the system.

It should be recognized that embodiments of the current invention provides a protocol which can be used to simulate serial port hardware across a network 10. When used in this manner, a board or other hardware is installed in a host computer in such a way that is provides a serial port interface to the host computer similar to a local serial port board. In this embodiment, the board contains a general-purpose network protocol stack, in addition to other software that provides the serial port interface and the functionality described above in connection with the client driver 200. Using this board, drivers can be used on the host operating system that are aware or only a serial Port interface, and are unaware of the existence of the network 10.

Such a hardware implementation has several advantages. First, it can be used to emulate a hardware or software interface for which drivers are already available (or can easily be adapted) on the host operating system. Second, the board can be installed in a computer where networking hardware or software is not

readily available, thereby providing network services to the host computer. Third, the board can include an additional processor and hardware which off-loads much of the serial port overhead onto the card. Finally, such a card could be multi-functional in nature by containing, for example, a small number of physical devices that could either be reserved for the local computer or be network accessible.

All information, including data and commands, exchanged between the server 20 and the client 18 is sent over the byte stream provided by the network protocol in small units called packets. The format of these packets makes up the Communication Protocol.

The Communication Protocol packets are similar in format to CRT terminal escape sequences, in that the first few bytes of the sequence determine the format and length of the data that follows.

The creation of the packets for communication from the server 20 to the client 18 takes place in the CPU 32 of the server 20. Packets that are sent from the client 18 to the server 20 are created by the special driver operating on the client 18. The packets themselves have a variable format and variable length, depending on the type of information they contain. The format of each packet is determined by processing it sequentially. The value of earlier bytes completely determines the format and content of the bytes that follow.

Table 9 shows the different types of packets which can be sent between a server 20 and a client 18. The table shows how the first byte breaks down a packet into major subtypes. Values denoted illegal were not supported in the preferred embodiment of the present invention.

Data packets are used to send all in-band data between the client 18 and server 20. When received by

the server 20, the data in the data packets are transmitted through to the appropriate port 40 as described above.

Window sequence packets inform the receiver about the state of the sender's sequence numbers. The transfer of this information is essential for the working of the data flow control, as outlined above. Window sequence packets are used by the server 20 to send TOUT to the client 18, and are used by the client 18 to send RWIN to the server 20.

Command packets have different formats and different lengths depending on the value of the command type field in byte one. Commands are sent to set and to query about the values of the server state variables 38. In addition, command packets are utilized to open a port 40, allowing the client 18 to guarantee synchronization with the server 20, to ascertain the size of the server transmit and receive buffers 42, 44, to ascertain the capabilities of a particular port 40, to flush the buffers 42, 44 of the server 20, to send a break or an immediate character and to pause or restart input or output. The command packets implemented in the preferred embodiment are set forth in Table 10.

Two of the most important command packets involve communication between the client 18 and the server 20 in connection with opening a port 40. The Open Request command for opening a port 40 is sent by a client 18 who wishes to obtain exclusive access to a port 40 on a server 20. When the server 20 receives an Open Request command, the server 20 attempts to open the port 40 for the client 18 and then responds with an Open Response packet which informs the client 18 of the results of the open attempt.

The format of the Open Request Command Packet and the Open Response Command Packet are set forth in Tables 11 and 12.

The command packet to open a port 40 can take different forms, depending on which type of open command is desired by the requesting client 18. The possible types of open commands are an immediate open, a persistent open, an incoming open.

A request for an immediate open succeeds and the port 40 is assigned to the requesting client 18 only if the port 40 is valid and immediately available. If the port 40 is not immediately available, the request fails.

When a persistent open request is made, the request will succeed and the port 40 will be assigned to the client 18 if the port 40 is available. If the port 40 is busy, the server 20 returns an Open Response packet to the client 18 which indicates the port 40 is busy, and then places the client 18 on a waiting list for that port 40. When the port 40 subsequently becomes available, the port 40 is opened, and the client 18 is notified with a second Open Response indicating success.

The third type of open request is a request for an incoming open. This request succeeds immediately if the port 40 is available, and a carrier is present on the port 40. If the port 40 is busy, or if a carrier is not present, the server 20 returns the corresponding Open Response code and places the client 18 on a waiting list for that port 40. When the port 40 subsequently becomes available with carrier present, the port 40 is opened, and the client 18 is notified with a second Open Response indicating success.

A client 18 may also send a cancel waiting type of Open Request for a particular port 40. When this command is received from a client 18, the server 20 checks to see if the client 18 is on the waiting list for that port 40. If the client 18 is found on the list, the client 18 is removed, and the server 20 returns an Open Response indicating success.

It is possible to organize ports 40 on a server 20 into a port hunt group. When this is done, the server 20 provides one or more logical ports designated as hunt group ports so that any attempt to open one of these ports in fact opens a physical port 40 configured as a member of the hunt group. Thus, if a client 18 requests a port 40 which is part of a port hunt group, the actual port 40 opened will be different from the port 40 requested. The client 18 must inspect the PNUM portion of the Open Response after a successful open, and use that port number for all subsequent references to the port 40.

The Synchronization Request command causes the recipient to respond with a Synchronization Response. It has no other effect. The primary purpose of this command is to allow the client to guarantee synchronization with the server. For example, a client 18 may wish to confirm a requested baud rate change before returning to the calling program. The client 18 achieves this by first sending a baud rate change to the server 20, and then following it with a Synchronization Request. Since the server 20 processes commands sequentially, when the client 18 receives the matching the Synchronization Response, the client 18 can be confident the baud rate change has been accomplished.

The Sequence Request and Response Packets are provided so the client 18 can accurately track the flow of data in and out of the server 20. The server 20 responds to this packet by providing RIN and TOUT to the client 18.

The client 18 may send a Status Request Packet to discern the current state of ESTAT and MSTAT in the server 20. The server 20 returns the requested data in a Status Response Packet, and takes no other action. This command is one of two ways the client 18 may learn the status of server 20. The client 18 may also use the

Select Event Conditions Packet to have the server automatically report selected status changes as they occur.

The client 18 sends the Line Error Request Packet to poll the line error counters, to configure periodic reporting of line error counters, and to clear the counters. The server 20 sends a Line Error Response in response to a poll for the error counters, and also at periodic intervals as configured with the variable LTIME. All Line Error Request packets may set the server variable LTIME. An LTIME value of 0 disables automatic reporting. An LTIME value of FFFF leaves LTIME unchanged.

After the client 18 has set LTIME non-zero, the server 20 inspects the state of the error counters every LTIME milliseconds (to an accuracy of approximately 20 milliseconds) and sends an automatic Line Error Response packet if they have changed since the last report.

The Buffer Request Packet is sent by the client 18 to learn the size of the server transmit buffers 42 and receive buffers 44. The server 20 responds with the Buffer Response Packet. After a successful open, and before sending data to the server 20, the client 18 needs to ascertain the size of the server's buffers 42, 44. The size of the transmit buffer 42 is needed so the client 18 can know how much data the client 18 is allowed to send. The size of the receive buffer 44 is needed so the server 20 can set the flow control high and low water marks accordingly.

The Port capability request packet is generally sent by the client 18 to learn the capabilities of the hardware and software of a port 40. Unlike other port-level commands, the port 40 need not be open when this command is issued.

The Set Baud Rate command is used by the client 18 to set the port baud rate, CFLAGS, IFLAGS, OFLAGS and XFLAGS of a port 40.

The client 18 sends the Select Event Conditions command to specify which ESTAT and MSTAT conditions cause the server 20 to send an Event Packet. The client may also poll for ESTAT and MSTAT using the Status Request Packet.

The Set Window Trigger command allows the client 18 to set TREQ, one of the server status variable 38. TREQ is the client 18 requested level at which a Window Sequence Packet should be sent. If TREQ is outside the range of data currently in the output buffer, the server 20 immediately responds with a Window Sequence Packet.

The Set Modem Outputs and Flow Control packet is sent by the client 18 to set modem outputs, and select modem-signal hardware flow control. Note that when RTS and DTR modem flow control are selected, the values of RTS and DTR in MOUT are ignored.

The client 18 sends the Set Receive High/Low Water Marks packet to set receive flow control high and low water marks. When this packet is processed by the server 20, if takes effect immediately. If the number of characters in the input buffer exceeds RHIGH, flow control is immediately invoked. If less than RLOW, flow control is immediately released.

The client 18 sends the Set Flow Control Characters packet to set server 20 software flow control characters. Similarly, the client sends the Set RMAX and RTIME command to set the parameters RMAX and RTIME and the Set TMAX and TTIME command to set TMAX and TTIME.

The client sends the Send Character Immediate packet to send a character ahead of in-band data. In sending immediate data, software flow control is ignored, and hardware flow control uses the variable MCTRL instead

of MFLOW. The client 18 utilizes the Send Break Immediate packet to send a hardware break signal. The Break signal is also sent ahead of all in-band data, and regardless of software or hardware flow control.

The client sends Flush Buffers packet to flush either or both the server input and output buffers. The Pause Input/Output packet pauses any set of software flow control conditions in the server. It is ineffective to attempt to pause input if the number of characters in the receive buffer is less than RLOW. Finally, the Unpause Input/Output packet is used to unpause any set of software flow control conditions in the server. It is ineffective to attempt to unpause input if the number of characters in the receive buffer is greater than RHIGH.

Event packets are sent by the server 20 to inform the client 18 of changes in ESTAT and MSTAT conditions. The server 20 sends an event packet for each port 40 whenever: (ETRAN & EINT) or (MTRAN & MINT) are non-zero.

To assure this request does not cause transitions to be lost that would otherwise be reported in event packets, the server 20 first sets ELAST and MLAST as follows, and then reports ELAST and MLAST to the client 18.

```
ELAST ^= (ESTAT ^ ELAST) | (ETRAN & EINT)
ETRAN = ESTAT ^ ELAST

MLAST ^= (MSTAT ^ MLAST) | (MTRAN & MINT)
MTRAN = MSTAT ^ MLAST
```

Module Select Packets are used to select a particular grouping of ports 40. To reduce communication bandwidth, ports 40 are logically grouped into modules, with each module containing only sixteen ports 40 (ports 0-15). There need be no relationship between a logical

module grouping and the physical layout of the ports 40 on the server(s). By grouping ports 40 into modules, the addressing in all commands and responses references need to only refer to which port (0-15) is desired within the current module. Module 0 is selected by default. Modules 0-7 can be selected with a one byte module select packet. Modules 8-255 require a two-byte packet. It is possible to convert between a module and port pair and a physical port number. When module n is selected, port K in all subsequent packets refers to physical port 16*n+K.

An ID request packet is sent by either the client 18 or server 20 to get identity or configuration information from the remote. The remote responds with a corresponding ID response packet.

The debugging packet types supports access to ports 40 of the server 20 for debugging purposes, such as transmitting debugging data between the client 18 and server 20. These packets could also be used to allow the client 18 access to the server program memory 36, either to review the contents of program memory 36 or to change the programming of the server 20 as stored in program memory 36.

The reset packet type is sent by a client 18 or server 20 to report a protocol error to the remote, prior to shutting down the connection. The packet contains an explanation of the reason for the reset in an ASCII format. The Reset Packet should be the last packet sent in a conversation.

Although this discussion has not specifically covered operating systems other than UNIX, it will be obvious to one skilled in the art that there are features documented in XFLAGS, the error counters and the event flags that are required to satisfy the application programming interfaces of Novell, Windows, Windows NT, OS/2 and DOS. As a result, the implementation of the

invention in any of these operating systems is obvious to one skilled in the art.

The invention is not to be taken as limited to all of the details thereof as modifications and variations thereof may be made without departing from the spirit or scope of the invention.

Table 1: Server State Variables

Type of Server State Variables	Frequency
Client Connections	per server
Message Build Interval	per server
Port Open State	per port
Port Baud Rate	per port
Processing Flags	per port
Flow Control Characters	per port
Modem Control	per port
Line Error Counters	per port
Send Break and Send Immediate	per port
Event Reporting	per port
Input Sequence and Receive Buffer Parameters	per port
Output Sequence and Transmit Buffer Parameters	per port

Table 2: XFLAGS Settings

Mask	Name	Description
0001	XPAR	Enable Mark/Space parity.
0002	XMODEM	Enable in-band modem signaling.
0004	XCASE	Convert special characters to multiple-character sequences on output.
0040	XTOSS	Toss IXANY characters.
2000	XIXON	Enable a second set of Output Software Flow Control Characters.

Table 3: Modem Variables

Name	Description
MOUT	Client specified modem output values.
MFLOW	Modem flow control for in-band data.
MCTRL	Modem flow control for out-of-band data.
MSTAT	Current modem status.
MLAST	Last modem status sent to the client.
MTRAN	Set of modem signals which have changed, but those changes have not yet been sent to the client.
MINT	Modem signal changes to be reported to the client.

Table 4: Modem Signals in Modem Variables

Mask	Name	Description
01	DTR	Data Terminal Ready.
02	RTS	Request To Send.
10	CTS	Clear to Send.
20	DSR	Data Set Ready.
40	RI	Ring Indicator.
80	DCD	Data Carrier Detect.

Table 5: Event Reporting Variables

Name	Description
ESTAT	Current state of event conditions.
ELAST	Last state of the event conditions sent to the client.
ETRAN	Set of event conditions which have seen transitions, but those transitions have not yet been reported to the client.
EINT	Set of event conditions which cause an event to be generated.

Table 6: Recorded Events

Mask	Name	Description
0001	OPU	Output paused unconditionally by client.
0002	OPS	Output paused by regular software flow control.
0004	OPX	Output paused by extra software flow control characters.
0008	OPH	Output paused by hardware flow control.
0010	IPU	Input paused unconditionally by client.
0020	IPS	Input paused by high/low water marks.
0040	TXB	Transmit break pending.
0080	TXI	Transmit immediate pending.
0100	TXF	Transmit flow control character pending.
0200	RXB	Break received.

Table 7: Input Sequence and Receive Buffer Variables

Name	Description
RIN	Sequence number of the next byte of data to be read from the UART and placed in the local receive buffer.
ROUT	Sequence number of the next byte to be transmitted from the receive buffer to the client.
RWIN	Highest Sequence number the client wishes to accept.
RMAX	Receive trigger maximum.
RTIME	Receive trigger time.
RSIZE	Size of the local receive buffer.
RLOW	Software flow control low water mark.
RHIGH	Software flow control high water mark.

Table 8: Output Sequence and Transmit Buffer Parameters

Name	Description
TSIZE	Size of the transmit buffer in bytes.
TIN	Sequence of the next byte to be placed in the transmit buffer.
TOUT	Sequence of the next byte to be sent from the transmit buffer out the port.
TPOS	Value of TOUT last reported to the client.
TMAX	TOUT report maximum.
TTIME	TOUT report time (milliseconds).
TREQ	Client requested sequence number.

Table 9: Packet Types

Nibble 0	Nibble 1	Description
0	port 0-15	Data Packet. 1 byte of data follows for the specified port.
1	port 0-15	Data Packet. 2 bytes of data follow for the specified port.
2	port 0-15	Data Packet. 3 bytes of data follow for the specified port.
3	port 0-15	Data Packet. 4 bytes of data follow for the specified port.
4	port 0-15	Data Packet. 5 bytes of data follow for the specified port.
5	port 0-15	Data Packet. 6 bytes of data follow for the specified port.
6	port 0-15	Data Packet. 7 bytes of data follow for the specified port.
7	port 0-15	Data Packet. 8 bytes of data follow for the specified port.
8	port 0-15	Data Packet. Byte 1 of the packet specifies 1-255 bytes of data following for the specified port.
9	port 0-15	Data Packet. Bytes 1-2 of the packet specify 1-65535 bytes of data following for the specified port.
10	port 0-15	Window Sequence Packet. Bytes 1-2 specify a window sequence number.
11	port 0-15	Command Packet. Byte 1 specifies a command number which determines the length and format of the data that follows.

Nibble 0	Nibble 1	Description
12	port 0-15	Event Packet. 3 bytes of status follow.
13	*	Illegal.
14	*	Illegal.
15	module 0-7	Module select Packet. Nibble 1 contains the module number to be selected.
15	8	Module select Packet. Byte 1 specifies a module select code in the range 0-255.
15	9-10	Illegal.
15	11	ID Request packet. Requests an ID response.
15	12	ID Response packet. Byte 1 determines the size of the data which follows.
15	13	Debugging and Control Packet.
15	14	Debug Packet. Specifies a number of following bytes to be utterly ignored. Used for stress testing.
15	15	Reset Packet. Followed by a null-terminated ASCII string describing the reason for the reset.

Table 10: Command Type Assignments & Lengths

Command Type (byte 1 value)	Command Length (bytes)	Command Name
10	3	Open Request
11	6	Open Response
12	3	Synchronize Request
13	3	Synchronize Response
14	2	Sequence Request
15	6	Sequence Response
16	2	Status Request
17	5	Status Response
18	6	Line Error Request
19	14	Line Error Response
20	2	Buffer Request
21	6	Buffer Response
22	2	Port Capability Request
23	32	Port Capability Response
40	12	Set Baud Rate, CFLAGS, IFLAGS, OFLAGS and XFLAGS
42	5	Select Event Conditions
43	4	Set Window Trigger
44	5	Set Modem outputs and Flow Control
45	6	Set High/Low Water marks
46	7	Set Flow Control Characters

Command Type (byte 1 value)	Command Length (bytes)	Command Name
47	6	Set RMAX and RTIME
48	6	Set TMAX and TTIME
60	3	Send Character Immediate
61	4	Send Break Immediate
62	3	Flush input/output
63	3	Pause input/output
64	3	Unpause input/output

Table 11: Open Request Packet

Position	Name	Description
Nibble 0	MTYPE	Message type 11: Command.
Nibble 1	PORT	Port number 0-15.
Byte 1	CTYPE	Command type 10: Open Request.
Byte 2	REQ	Request Type: 0=Immediate open 1=Persistent open. 2=Incoming open. 3=Cancel/Close immediate open. 4=Cancel/Close persistent or incoming open.

Table 12: Open Response Packet

Position	Name	Description
Nibble 0	MTYPE	Message type 11: Command.
Nibble 1	PORT	Port number 0-15.
Byte 1	CTYPE	Command type 11: Open Response.
Byte 2	REQ	Request Type: 0=Immediate open 1=Persistent open. 2=Incoming open. 3=Cancel/Close immediate open. 4=Cancel/Close persistent or incoming open.
Byte 3	RESP	Request code: 0=Success. 1=Port busy. 2=No carrier. 3=Resource allocation problem, try again. 4=Request not supported on this device.
Bytes 4-5	PNUM	Actual port number (16*module+port) opened by this request.

The claims defining the invention are as follows:

1. A system comprising:

a server having a plurality of communication ports; and

a host computer having a driver communicatively coupling the host computer to the server via a network connection, wherein the driver emulates the communication ports of the server by defining a corresponding local communication port for each of the communication ports of the server, and further wherein the driver includes an application programming interface (API) by which an application program executing on the host computer is granted full control of one of the communication ports of the server, including hardware and software flow control, as if the communication ports of the server were local to the host computer.

2. The system as claimed in claim 1, wherein the driver maintains a single network connection from the host computer to the server as the application program requests additional local communication ports from the driver.

3. The system as claimed in claim 1, wherein the driver defines a TTY device as the local communication port.

4. The system as claimed in claim 1, wherein the driver receives input/output (I/O) settings from the application program via the application programming interface, and further wherein the driver communicates the I/O settings to the server for configuring hardware characteristics of the granted server communication port.

5. The system as claimed in claim 1, wherein the server communication ports are serial ports.

6. The system as claimed in claim 1, wherein the network connection is a TCP connection.

7. The system as claimed in claim 1, further comprising a UNIX daemon executing on the host computer for establishing the network connection as a reliable bi-directional bytestream connection over a network, opening a control device associated with the driver, and utilizing a STREAMS interface to link the reliable bytestream network connection to the driver.

8. A hardware device for a host computer, wherein the hardware device includes a driver that emulates a plurality of communication ports of a remote server that is communicatively coupled to the host computer via a network connection, wherein the driver defines a corresponding local communication port for each communication port of the server and includes an application programming interface (API) by which an application program executing on the host computer is granted full control of one of the communication ports of the server, including hardware and software flow control, as if the communication ports of the server were local to the host computer.

9. The hardware device as claimed in claim 8, wherein the driver maintains a single network connection from the host computer to the server as the application program requests additional local communication ports.

10. The hardware device as claimed in claim 8, wherein the driver defines a TTY device as the local communication port.

11. The hardware device as claimed in claimed in claim 8, wherein the driver receives input/output (I/O) settings from the application program via the application programming interface, and further wherein the driver communicates the

I/O settings to the terminal server for configuring hardware characteristics of the terminal server communication port.

12. The hardware device as claimed in claim 8, wherein the terminal server communication ports are serial ports.

13. The hardware device as claimed in claim 8, wherein the network connection is a TCP connection.

14. The hardware device as claimed in claim 8, further comprising a UNIX daemon executing on the host computer for establishing the network connection as a reliable bi-directional bytestream connection over a network, opening a control device associated with the driver, and utilizing a STREAMS interface to link the reliable bytestream network connection to the driver.

15. A terminal server substantially as hereinbefore described with reference to the accompanying drawings.

16. A method for transmitting data and commands, said method being substantially as hereinbefore described with reference to the accompanying drawings.

17. A system for communicating digital data said system being substantially as hereinbefore described with reference to the accompanying drawings.

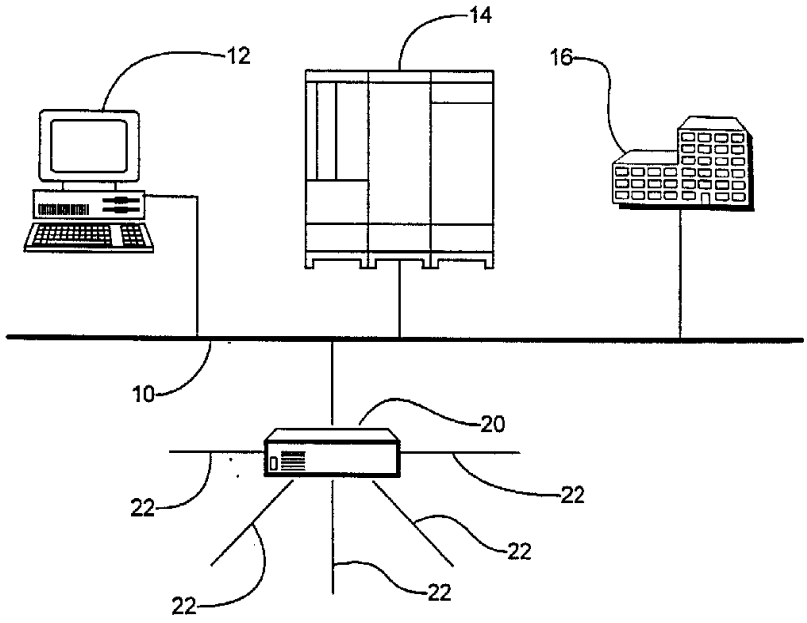
DATED this Twenty-ninth Day of March 1999

Digi International Inc.

Patent Attorneys for the Applicant

SPRUSON & FERGUSON

Fig. 1



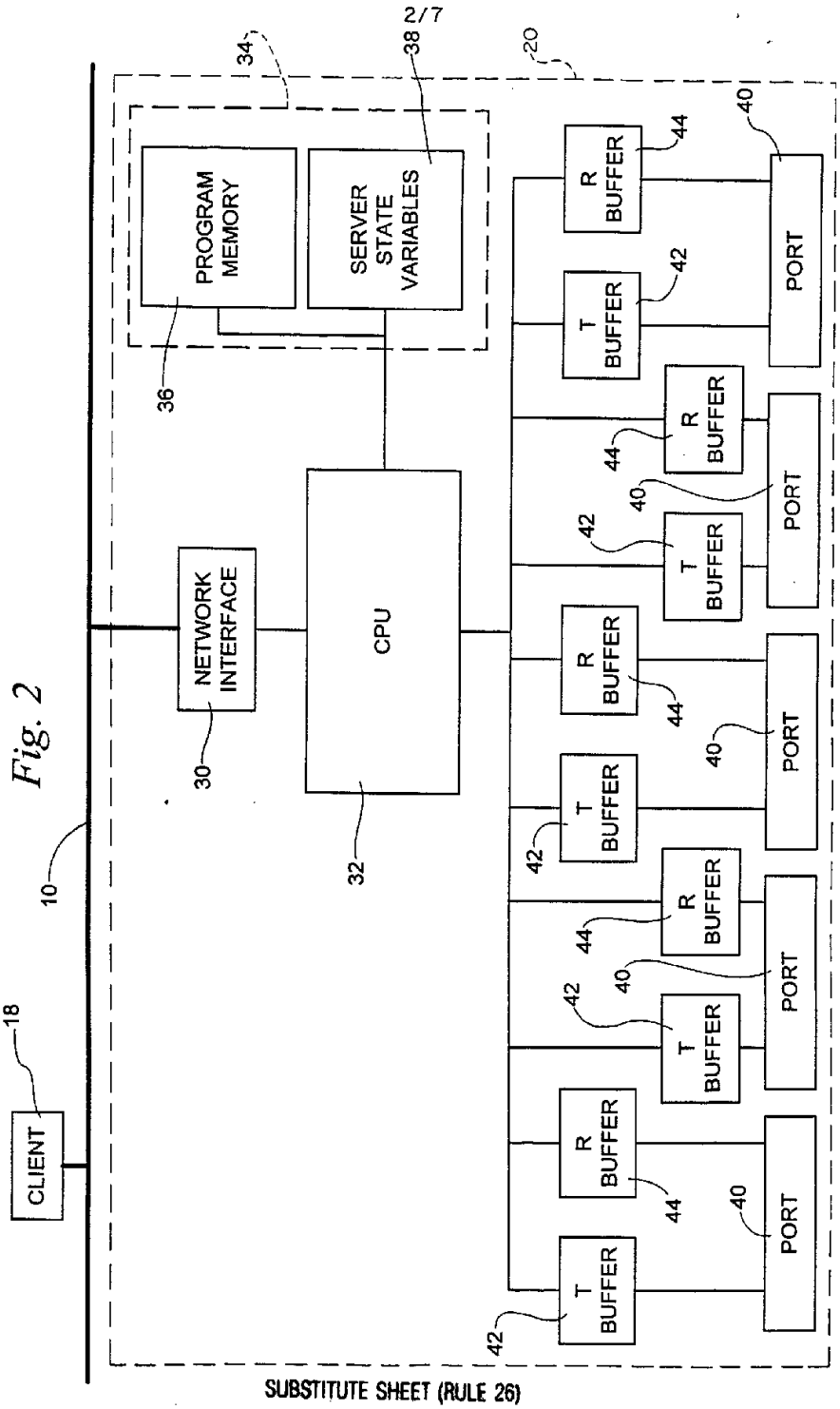

CLIENT
18
10
NETWORK INTERFACE
30
CPU
32
PROGRAM MEMORY
36
SERVER STATE VARIABLES
38
34
20
R BUFFER
T BUFFER
PORT
40
42
44

Fig. 2

Fig. 4

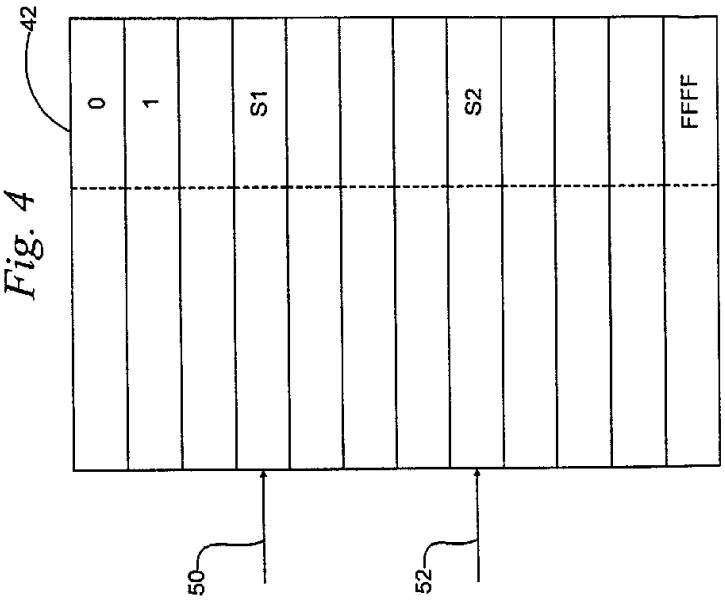


Fig. 3

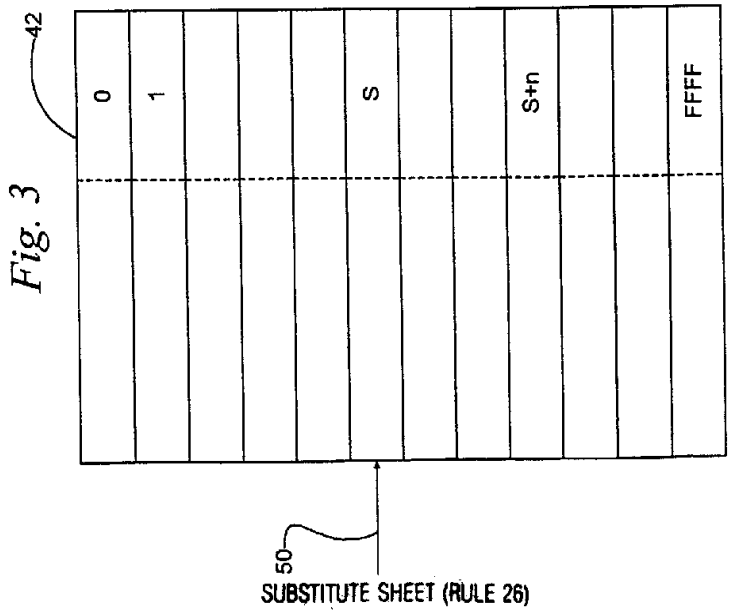
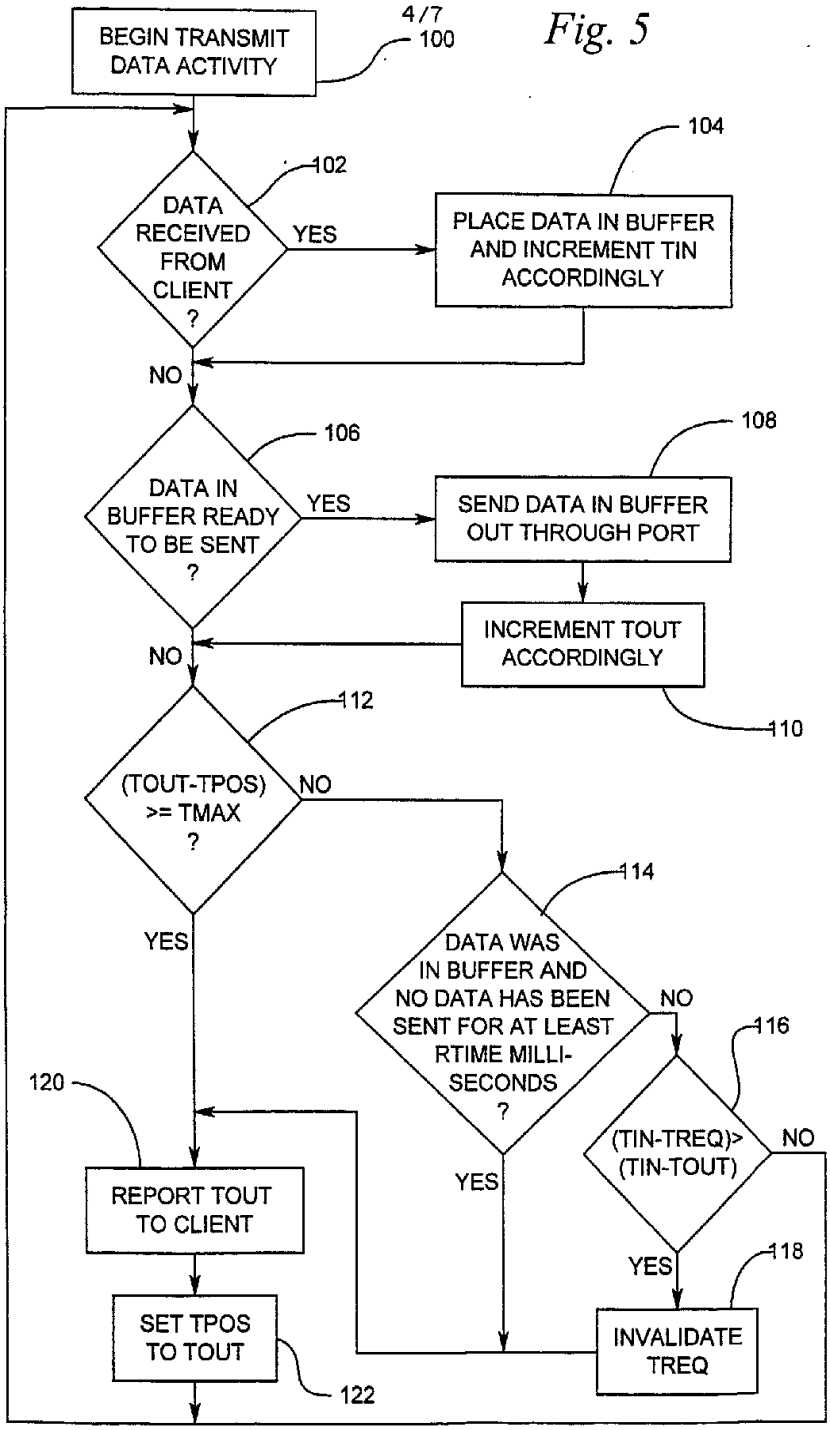


Fig. 5


BEGIN TRANSMIT DATA ACTIVITY
100
DATA RECEIVED FROM CLIENT ?
102
YES
PLACE DATA IN BUFFER AND INCREMENT TIN ACCORDINGLY
104
NO
DATA IN BUFFER READY TO BE SENT ?
106
YES
SEND DATA IN BUFFER OUT THROUGH PORT
108
INCREMENT TOUT ACCORDINGLY
110
NO
(TOUT-TPOS) >= TMAX ?
112
NO
YES
DATA WAS IN BUFFER AND NO DATA HAS BEEN SENT FOR AT LEAST RTIME MILLI-SECONDS ?
114
NO
YES
(TIN-TREQ)> (TIN-TOUT)
116
NO
YES
REPORT TOUT TO CLIENT
120
SET TPOS TO TOUT
122
INVALIDATE TREQ
118

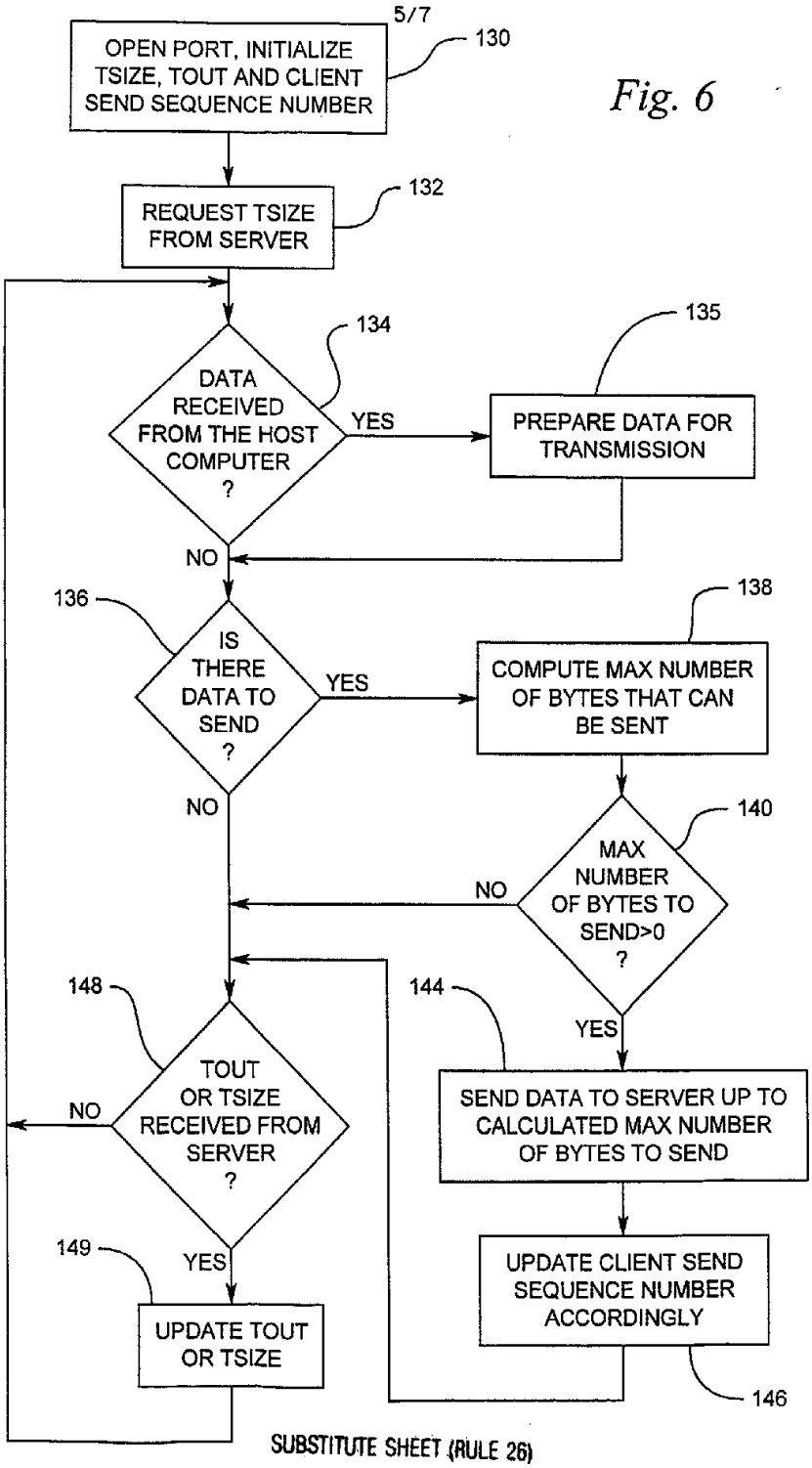

OPEN PORT, INITIALIZE TSIZE, TOUT AND CLIENT SEND SEQUENCE NUMBER
130
REQUEST TSIZE FROM SERVER
132
DATA RECEIVED FROM THE HOST COMPUTER ?
134
YES
PREPARE DATA FOR TRANSMISSION
135
NO
IS THERE DATA TO SEND ?
136
YES
COMPUTE MAX NUMBER OF BYTES THAT CAN BE SENT
138
NO
MAX NUMBER OF BYTES TO SEND>0 ?
140
NO
YES
TOUT OR TSIZE RECEIVED FROM SERVER ?
148
NO
YES
UPDATE TOUT OR TSIZE
149
SEND DATA TO SERVER UP TO CALCULATED MAX NUMBER OF BYTES TO SEND
144
UPDATE CLIENT SEND SEQUENCE NUMBER ACCORDINGLY
146

Fig. 6

Fig. 7

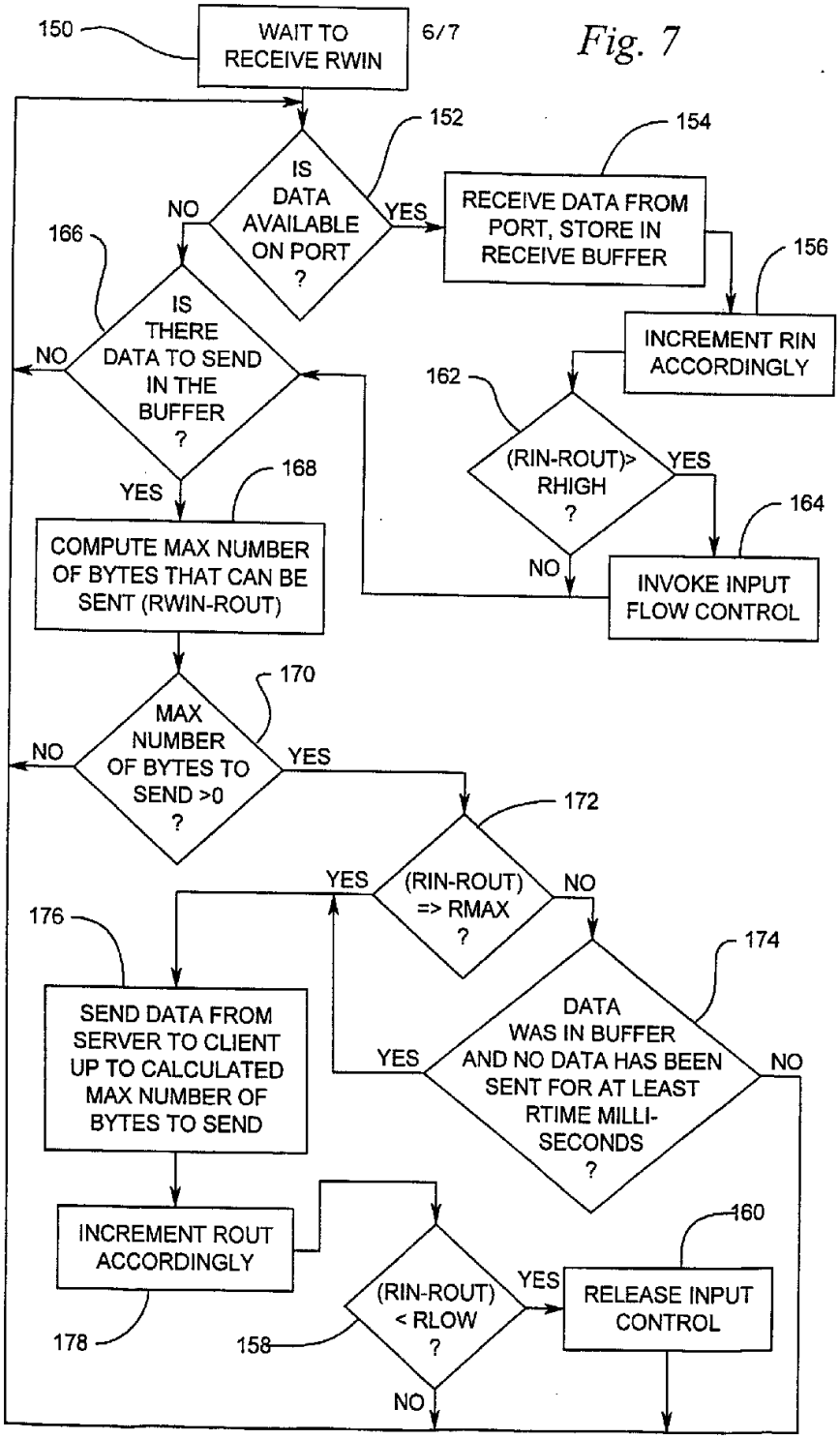
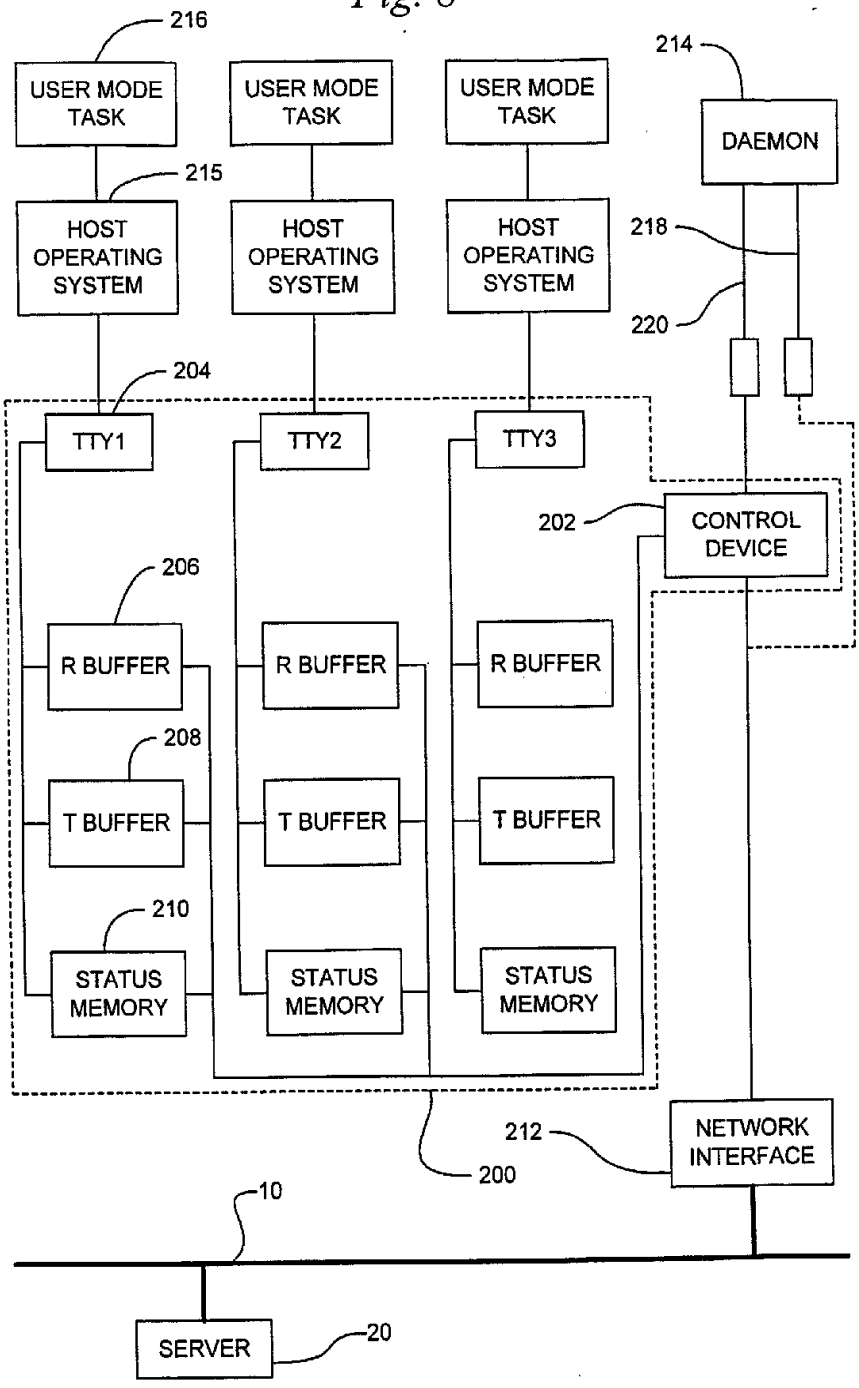

150
WAIT TO RECEIVE RWIN
152
IS DATA AVAILABLE ON PORT ?
NO
YES
154
RECEIVE DATA FROM PORT, STORE IN RECEIVE BUFFER
156
INCREMENT RIN ACCORDINGLY
166
IS THERE DATA TO SEND IN THE BUFFER ?
NO
YES
162
(RIN-ROUT)> RHIGH ?
YES
NO
164
INVOKE INPUT FLOW CONTROL
168
COMPUTE MAX NUMBER OF BYTES THAT CAN BE SENT (RWIN-ROUT)
170
MAX NUMBER OF BYTES TO SEND >0 ?
NO
YES
172
(RIN-ROUT) => RMAX ?
YES
NO
176
SEND DATA FROM SERVER TO CLIENT UP TO CALCULATED MAX NUMBER OF BYTES TO SEND
174
DATA WAS IN BUFFER AND NO DATA HAS BEEN SENT FOR AT LEAST RTIME MILLI-SECONDS ?
YES
NO
INCREMENT ROUT ACCORDINGLY
178
158
(RIN-ROUT) < RLOW ?
YES
NO
160
RELEASE INPUT CONTROL

Fig. 8


216
USER MODE TASK
USER MODE TASK
USER MODE TASK
214
DAEMON
215
HOST OPERATING SYSTEM
HOST OPERATING SYSTEM
HOST OPERATING SYSTEM
218
220
204
TTY1
TTY2
TTY3
202
CONTROL DEVICE
206
R BUFFER
R BUFFER
R BUFFER
208
T BUFFER
T BUFFER
T BUFFER
210
STATUS MEMORY
STATUS MEMORY
STATUS MEMORY
212
NETWORK INTERFACE
200
10
SERVER
20