AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

The Virtual Protocol Machine

By M. J. FITTON,* C. J. HARKNESS,* K. A. KELLEMAN,*
P. F. LONG,* and C. MEE III*

(Manuscript received August 12, 1983)

The *UNIX*™ operating system Virtual Protocol Machine (VPM) is a package of software tools that allows a wide variety of link-level data communications protocols to be implemented efficiently in a high-level language. The resulting protocol implementations are independent of the particular communications hardware, the host machine architecture, and the host operating system, and therefore can be ported easily from one hardware/software environment to another. An extension to VPM, the Common Synchronous Interface (CSI), provides similar benefits for the higher-level protocol software that runs in the *UNIX* system host. The implementations of VPM use Programmable Communications Devices (PCDs) to off load the link-level communications processing from the host CPU. A high-level language protocol description is translated by a protocol compiler that runs on the host machine. The resulting module is then loaded into the PCD and executed. The other components of VPM are a transparent protocol driver that allows user processes to interact directly with a link-level protocol implementation, a real-time trace capability to facilitate debugging, and several utility programs. VPM has been implemented on several different PCDs and several types of host computers. VPM-based protocol implementations can be ported with little or no change from one VPM implementation to another. VPM and CSI greatly reduce host system overhead while producing maximum communica-

* AT&T Bell Laboratories.

tions throughput. A number of different higher-level protocols and their link-level counterparts have been implemented in the *UNIX* system using CSI and VPM; among them are X.25, 3270 emulation, a synchronous terminal interface, and a facility for remote job entry to IBM hosts.

I. INTRODUCTION

Data communications protocols have evolved in response to a need for reliable, efficient, and high-speed communication between host computers and their terminals and, more recently, between pairs of host computers.[1] The functions provided by these protocols include:

1. Framing to determine which bits constitute a character and which characters make up a message.
2. Error control using cyclic redundancy checks to detect errors and retransmission to correct them.
3. Flow control to prevent data from piling up at the receiving end faster than they can be processed.
4. Multiplexing to allow several independent data streams to be transmitted concurrently over one physical link.
5. Call establishment and clearing procedures to allow use of switched networks.

Modern communications protocols are organized into layers, or levels, to manage complexity and provide flexibility of implementation. Each higher layer uses the facilities provided by the next lower level and augments them with additional functionality. Level 1, the lowest level, is usually defined in terms of the electrical interfaces at either end; it provides a basic data transfer facility with no error control or flow control. Level 1 is used directly, for example, by simple asynchronous terminals. Level 2, frequently referred to as the link level, provides reliable transmission across a single physical link; it includes procedures for error control, flow control, and call establishment and clearing. An example of a level-2 protocol is IBM's Binary Synchronous Communications procedure, also known as BSC or BISYNC. Level 3, if used, typically provides multiplexing of independent data streams. Still higher levels have also been defined.

The use of Programmable Communications Devices (PCDs) is an effective and economical way to implement link-level protocols. Lower-level protocol functions typically involve byte or bit operations allowing the use of inexpensive processors that are matched to these tasks. Protocol execution can proceed asynchronously using Direct Memory Access (DMA) methods and interrupts to interact with the host when necessary. Moving protocol execution to PCDs improves protocol throughput and allows more effective use of the host computer.

The Virtual Protocol Machine (VPM) is a software package that provides a set of tools for the writing, executing, and debugging of link-level protocol programs. These programs, which are referred to as protocol scripts, are portable across a wide range of PCDs that are used in conjunction with the various *UNIX* operating system hosts.

To implement VPM on a PCD, the PCD should have certain minimal functionality. It should have a means of direct access to the host's memory and be able to interrupt the host in order to notify it of completed operations or problems that are detected. It must, of course, support one or more serial communications lines and have sufficient random access memory to hold the PCD control program and at least one protocol script. It is important that the PCD handle byte operations efficiently. Interrupt-driven communication lines are not necessary but can be useful with some PCDs.

The VPM software package consists of:

1. A protocol compiler that executes on the *UNIX* system host and translates a protocol script into a form suitable for execution on a particular PCD.
2. PCD control programs that are specific to each supported PCD that implement the VPM primitives, manage communication with the host computer, and provide an environment for executing a protocol script in the PCD.
3. A transparent *protocol driver* that allows a user process to interact directly with a level-2 protocol program executing in a PCD; it provides no protocol features except basic packetization and simple flow control. [A protocol driver is a character pseudo device driver that uses the Common Synchronous Interface (CSI)].
4. A trace driver that provides a mechanism for tracing the execution of a link-level protocol executing in a PCD, as well as a higher-level protocol executing in the host.
5. A CSI that provides a general interface between level-3 protocols executing in a *UNIX* system host and their level-2 counterparts executing in a PCD; it allows implementations of higher-level protocols to be portable between the various *UNIX* system host computers regardless of the particular PCDs that are used to implement VPM on those hosts.
6. Miscellaneous utility programs to save and format trace output, load compiled protocol scripts into PCDs, and connect protocol driver minor devices with particular communications lines.

Figure 1 shows the relation between the various components of VPM.

A typical application of VPM and CSI includes a level-3 protocol implemented as a *UNIX* system character device driver, communicating through CSI with a level-2 protocol implemented in a PCD. When a higher-level protocol is not required or is being implemented at user

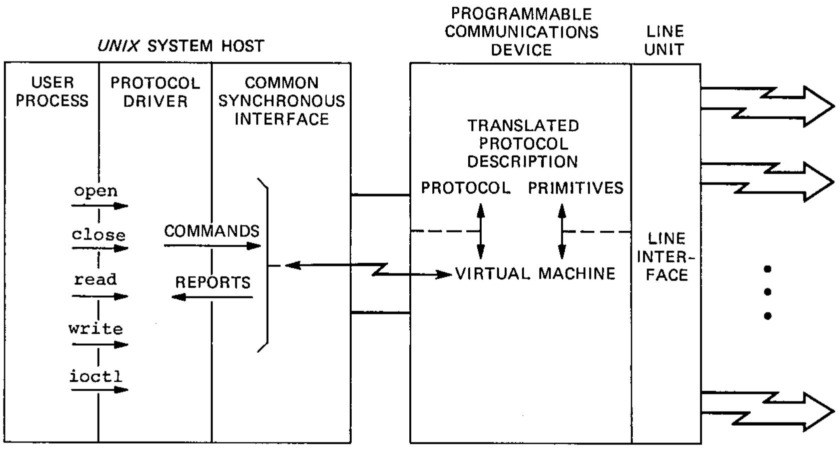

Fig. 1—VPM execution environment.

level, the user process can access the level-2 protocol through the transparent protocol driver.

Applications that have been developed using VPM and CSI include: (1) remote job entry to IBM systems, (2) support for synchronous terminals, (3) emulation of IBM 3270 cluster controllers, (4) levels 2 and 3 of the international standard X.25 data communications protocol, (5) support of asynchronous terminals through the standard terminal subsystem, and (6) support of the *Teletype*® 5620 Dot Mapped Display (DMD) terminal.

VPM and CSI have been implemented on the AT&T 3B20, AT&T 3B5, VAX-11*, and PDP-11* computers.

II. THE VIRTUAL MACHINE

The essential component of VPM is a set of communications primitives embedded in a high-level langauge. C was chosen as the host language for VPM because of its good bit-manipulation and control-statement facilities and for its familiarity to the expected user community.[2]

The communication primitives were designed with two goals in mind. The first was to allow each protocol description to be coded in a manner that is convenient, readable, and makes visible the details of the protocol. The second goal was to hide the details of the particular hardware on which VPM is implemented.

There are three sets of primitives corresponding to three different

* Trademark of Digital Equipment Corporation.

classes of protocols. One set supports half-duplex character-oriented protocols such as IBM's Binary Synchronous Communications (BISYNC) protocol. Another set supports bit-oriented full-duplex protocols such as the international standard High-Level Data Link Control (HDLC) procedure. A third set of primitives supports full-duplex asynchronous terminals such as those commonly used as login terminals with the *UNIX* system. The primitives for bit-oriented protocols are available only on the DEC* computers; the primitives for asynchronous communication are available only on the 3B5 computer.

The primitives for character-oriented protocols allow the protocol script to interact with the line interface on a character-by-character basis. Each incoming character is obtained by the script using an `rcv` primitive and is examined so that appropriate action can be taken. Similarly, each outgoing character, including all control characters, is generated explicitly by the protocol script and passed to the line interface using a `xmt` primitive. Reflecting the half-duplex nature of these protocols, the `xmt` and `rcv` primitives block if an incoming character is not immediately available or if the outgoing character cannot be accepted immediately by the line interface (a few characters are buffered in hardware or software; the number depends on the implementation). Other primitives provide for opening and closing transmit buffers and receive buffers, fetching characters one at a time from transmit buffers, storing characters one at a time into receive buffers, and initializing and updating a 16-bit Cyclic Redundancy Checksum (CRC) calculation. The protocol script is responsible for determining which incoming and outgoing characters should be incorporated into the checksum calculation, if any. Figure 2 shows a program fragment that transmits a block in transparent BISYNC.

The primitives for communication with asynchronous terminals are also character-oriented, and in many ways are similar to those just described. As an aid to performance, some of these primitives manipulate buffers as well as characters. The protocol script normally operates on a character-by-character basis but has the option of transmitting blocks of characters as well. These primitives are full-duplex and nonblocking, and include timer facilities as well as character-transmission routines. In several cases, the functional definition of the primitive is similar for synchronous and asynchronous processing, but the details of the implementation are different, so a different name is used.

The primitives for bit-oriented protocols are nonblocking and allow the protocol script to interact with the VPM control program on a complete-frame basis. Incoming and outgoing characters are processed

* Trademark of Digital Equipment Corporation.

```
#define DLE     0x10
#define ETB     0x26
#define PAD     0xff
#define STX     0x02
#define SYNC    0x32

unsigned char crc[2];
unsigned char byte;
/*
 * Transmit a block in transparent BISYNC
 */
xmtblk()
{

        /*
         * Initialize CRC calculation and send start-of-block character
         */
        crcloc(crc);
        xsom(SYNC);
        xmt(DLE);
        xmt(STX);
        /*
         * Get bytes from the transmit buffer and transmit them
         * adding DLE characters as required; update the CRC
         * calculation
         */
         while (get(byte) == 0) {
                 if (byte == DLE)
                         xmt(DLE);
                 xmt(byte);
                 crc16(byte);

         }
         /*
          * Transmit end-of-block characters and CRC
          */
         xmt(DLE);
         xmt(ETB);
         crc16(ETB);
         xmt(crc[0]);
         xmt(crc[1]);
         xeom(PAD);
```

Fig. 2—Example use of character-oriented primitives.

by the VPM control program without intervention by the protocol script. The script polls the control program via a `rcvfrm` primitive to determine if a completed receive frame is available. The control program assumes that up to five characters at the beginning of each incoming frame are control information that may be processed later by the protocol script. These characters are stored temporarily and passed to the protocol script via the `rcvfrm` primitive. All characters after the first two are placed into a receive buffer, if one is available; otherwise the characters are discarded. All characters are included in the CRC calculation. If an incoming frame is a data frame and the protocol script accepts it as correct, the script passes it to the host protocol driver using the `rtnrfrm` primitive. Other primitives transmit a control frame or data frame with specified control information in the first few bytes, determine whether a transmission is currently in progress, and manage queues of transmit and receive buffers. Figure 3 shows a program fragment that transmits a data frame in the Link Access Procedure B (LAPB) subset of HDLC.

```
#define T1FLAG  01
#define INCMOD8(X)          {if(++X == 8) X = 0;}
#define ADDMOD8(X,Y,Z)      {Z = X + Y; if(Z >= 8) Z = Z - 8;}
#define START_T1            {tstatus = T1FLAG; t1 = T1;}
/*
 *      Transmit a data frame if one is available.
 */
xmtdata() {
        /*
         *      If this is not a retry, get a new frame if available.
         */
        if (VS == unopened) {
                if (getxfrm(VS))
                        return(FALSE);
                INCMOD8(unopened);
        }
        /*
         *      Set up address and control bytes.
         */
        con = (VS&07)<<1;
        con |= (VR&07)<<5;
        ac[1] = con;
        ac[0] = faraddr;
        /*
         *      Start T1 timer if not currently running.
         */
        if (!(tstatus & T1FLAG)) {
                START_T1;
        }
        /*
         *      Set up control information and transmit frame.
         */
        setctl(ac,2);
        xmtfrm(VS);
        INCMOD8(VS);
        return(TRUE);
```

Fig. 3—Example use of bit-oriented primitives.

Several primitives are available for use with all three classes of protocols. Among these are facilities for receiving commands from and sending reports to a *UNIX* system driver or user process, generating trace event records, and starting and resetting software timers.

For a detailed description of the VPM primitives, see the entry for vpmc (1M) in the *UNIX* System Administrator's Manual.[3]

III. COMMON SYNCHRONOUS INTERFACE

The *UNIX* operating systems's Common Synchronous Interface (CSI) is a device-independent interface between a level-3 protocol executing as a part of the system and a level-2 protocol executing in a PCD. CSI allows level-3 protocol drivers to be independent of the host computers on which they run and the PCDs used to implement their level-2 protocol. Figure 1 illustrates the interaction of the level-3 protocol driver and the level-2 protocol through CSI.

The interface consists of a set of functions used by level 3 and a set of reports that are generated by level 2. The two classes of functions are service functions and command functions. Service functions are used for buffer administration. Command functions are used to set up and communicate with the level-2 protocol. The level-3 driver receives reports from the level-2 protocol and the PCD device driver via an interrupt routine. The more important functions and reports are described below. Some nonessential functions and reports have been omitted for clarity.

Service functions provide standard buffer queue management for level-3 protocol drivers. A standard CSI buffer structure is used to maintain buffers, allowing machine-independent buffering. Each buffer structure has buffer descriptors associated with it for maintaining buffer addresses, sizes, and any machine-dependent information. The service functions include:

1. `csialloc`—Allocate a buffer area for use by the level-3 driver. This function is typically called once during initialization to allocate buffer space for use by level 3.
2. `csifree`—Free the buffer area allocated for level 3.
3. `csibget`—Get a buffer descriptor and a buffer from the buffer area. This function is used by the level-3 protocol driver to obtain data buffers as needed.
4. `csibrtn`—Return a buffer descriptor and its associated buffer. This function is used when a buffer will no longer be needed by the level-3 protocol driver.
5. `csicopy`—Copy buffers to or from user space. This function provides a machine-independent way to copy data between system and user space.

Command functions are used to manage the communications link and communicate with the level-2 protocol script. The command functions include:

1. `csiattach`—Make a logical connection between a protocol driver and a synchronous line. This function is called before starting the level-2 protocol.
2. `csidetach`—Disconnect a protocol driver from a synchronous line.
3. `csistart`—Start the level-2 protocol. After a logical connection has been made, this function is used to start operation of the line (e.g., when a user requests a service).
4. `csistop`—Stop the level-2 protocol. This function is used to halt operation of the line.
5. `csixmtq`—Queue a transmit (full) buffer for level 2. This function is typically used by the level-3 protocol driver to transfer data on the line.
6. `csiemptq`—Queue a receive (empty) buffer for level 2. This function is used to provide level 2 with buffers for incoming data.
7. `csiscmd`—Send a command to the level-2 protocol. This function is typically used to communicate control information to level 2.

Reports are passed to a level-3 driver routine that is indicated when the logical connection is established. The level-3 driver receives two types of reports. Reports received as a result of a function call are referred to as solicited reports. Reports that are not issued as a result of a function call are referred to as unsolicited reports. Solicited reports indicate the disposition of the corresponding function. The solicited reports include:

1. CSISTART—Issued in response to a start command from the `csistart` routine. The report indicates if the line was started or if any errors occurred.

2. CSISTOP—Issued in response to a stop command from the `csistop` routine. The report indicates that the level-2 protocol has been halted.

3. CSIRXBUF—Issued when the level-2 protocol program returns a transmit buffer to the level-3 protocol driver. This report typically indicates that the data have been transmitted.

4. CSIRRBUF—Issued when the level-2 protocol program returns a receive buffer to the level-3 protocol driver. The report typically indicates that data have been received.

5. CSICMDACK—Issued when the level-2 protocol receives a command from the `csicmd` routine.

Unsolicited reports indicate random events from the level-2 protocol script. The unsolicited reports include:

1. CSITERM—Occurs when the protocol terminates abnormally. The report contains an indication of the reason for termination.

2. CSISRPT—Occurs when the level-2 protocol passes information to the protocol driver.

IV. TRACE DRIVER

The trace driver provides a means by which a user program can receive trace information generated by a VPM protocol driver and script to aid in debugging. It can also be used to debug other drivers or operating-system code that is not related to a VPM protocol driver or script. This driver can be configured to have a number of minor devices. Each trace-driver minor device provides a means by which a user program can read data that are generated by functions within the operating system. These data are recorded by issuing calls to the `trsave` function. Each call to `trsave` generates a unit of data known as an *event record,* which consists of a channel number, a count, and *count* bytes of data. The channel number can be used to multiplex up to 16 data streams on each minor device. Each channel can be enabled or disabled by an `ioctl` system call.

Event records that are generated for a minor device that is not currently open, or for a channel that is not currently enabled, are discarded. This allows a user program to control the activation and deactivation of tracing.

Minor device 0 of the trace driver is used by the VPM transparent driver and CSI to record a variety of debugging information generated within these modules and also to record the data generated by `trace` primitives in the protocol script. Two commands, `vpmsave` and `vpmfmt`, are available for reading and formatting data passed via the minor devices of the trace driver. Trace information can be displayed in real time if appropriate.

V. IMPLEMENTATIONS

5.1 DEC computers

The implementation of VPM on DEC computers (VAX-11, PDP-11) uses a programmable communications device known as a KMC11-B. The KMC11-B is a small (12K bytes), fast (200-ns instruction time), single-board computer that attaches to the UNIBUS* of a VAX* or PDP* computer. The KMC11-B can become bus master to perform DMA transfers to and from the host computer's main memory. The KMC11-B can be fitted with any of several types of communications interfaces. One type interfaces a single synchronous line at speeds of up to 56 kb/s. Another type interfaces up to eight synchronous lines at speeds up to 19.2 kb/s. The actual speed at which the interfaces can be used depends on the protocol.

Because of the small memory size of the KMC11-B, the VPM compiler for the DEC computers translates a protocol script into an intermediate language that is interpreted by a control program in the KMC11-B. This intermediate language consists of binary instructions for a hypothetical computer with a simple one-address instruction set. The VPM primitives are implemented as single instructions for this virtual machine.

The VPM compiler for the DEC machines does not support the full C language. While essentially all of the control structures and operators of C are admitted, there is only one data type: unsigned characters. All variables are global.

Besides interpreting the compiled protocol script, the VPM control program is responsible for: (1) communicating with the host computer (via eight bytes of shared memory) in order to receive commands from the host and send reports to the host, (2) servicing the synchronous line interface(s), (3) monitoring modem status, (4) maintaining a series of software timers, and (5) maintaining queues of transmit buffers and receive buffers.

The VPM control program for the eight-line interface uses an

* Trademark of Digital Equipment Corporation.

efficient real-time scheduling algorithm to meet the needs of communications processing: once the virtual process for a given line gets control of the processor, that process is allowed to run until it blocks. A process can block voluntarily by executing a `pause` primitive. Once a process blocks, it is not rescheduled until the occurrence of some event that could change the state of the protocol for that line. Such events are:

1. Arrival of an incoming character or completion of an outgoing character for a character-oriented protocol; completion of an incoming or outgoing frame for a bit-oriented protocol.
2. Notification by the host of the availability of a transmit buffer or a receive buffer or a command from the host.
3. Expiration of a timer previously started by the process.

As processes become unblocked, they are placed on the end of a ready-to-run queue and scheduled in a First-In First-Out (FIFO) manner.

Because of the limited memory space in the KMC11-B, the implementation for the eight-line interface requires that all eight lines share a single copy of the compiled protocol script; this implies that all eight lines must be running the same level-2 protocol. Each line has a 256-byte data area that is used to hold the local variables for that line and as a save area on a context switch. Memory protection is provided by the interpreter.

5.2 AT&T 3B20 computer

The 3B20 is a 32-bit general-purpose minicomputer manufactured by AT&T. It supports three different PCDs. One PCD supports character-oriented protocols using the VPM primitives; the other two PCDs support X.25 LAPB and are not user-programmable but are controlled by CSI. The remainder of this section describes the character-oriented PCD.

The 3B20 implementation of VPM differs from that on the DEC machines. Protocol programs are not interpreted, but are compiled into machine language and executed directly. The PCD consists of a microcomputer system with four RS-232/449 ports. One of the ports also supports a CCITT V.35 interface for communication at speeds up to 56K bits per second. The major software components are a full C language compilation system, a library of VPM primitives, a small operating system to oversee execution of protocol programs, and a *UNIX* system driver to interface to CSI.

A C compiler-based VPM implementation was chosen because C language support existed for the hardware before the VPM was implemented, and the PCD has ample memory. Supporting the full C language allows protocol programs to be as sophisticated as the application requires and real-time constraints permit.

Protocol programs run under the control of a small VPM operating system. It supports five independent processes: four protocol programs and one control program. All processes and the operating system reside in the same address space. The memory and address space not used by the system is partitioned statically into four pieces, one partition for each port. There is no hardware memory protection and processes are expected to be cooperative.

VPM primitives such as `rcv` and `xmt` are implemented using a lower-level set of primitives that are defined by the operating system. The intent was to provide a system that could be extended beyond VPM if desired. These subprimitives provided facilities for scheduling, transferring messages to and from the driver, doing DMA to the host memory, and copying data and accessing peripheral device registers.

Processes are scheduled in a round-robin fashion using a one-tenth of a second time slice. A process will run until it either gives up the CPU, or is preempted after running for one-tenth of a second. A process is always runnable unless it has been stopped or exited. The `pause` primitive gives up the CPU until all the other processes have had a chance to run. `Rcv` is implemented as:

```
while (receive queue is empty) {
        check modem status
        pause();
}
return next character from the queue
```

Characters are placed into the receive queue by the operating system through interrupts. The `xmt` primitive is similar. It puts characters into a queue, and the characters are actually transmitted at interrupt level.

The VPM operating system is brought into service by down loading it through a standard “device on-line” command. After being down loaded, the control process runs and waits for work to do. The control process has three functions: (1) down load, stop, and start protocol programs; (2) respond to audits or “sanity checks” from the driver; and (3) respond to “set Universal Synchronous/Asynchronous Receiver/Transmitter (USART) options” commands from the driver.

A protocol program is created in two steps. First, the C source is compiled and linked with the VPM primitive library, with loader relocation information left intact. The output of this step is a generic object program that can be run on any port of any PCD. The next step is to relocate the program to the memory partition that is appropriate for the particular port being used, and then down load it into the PCD.

5.3 AT&T 3B5 computer

The AT&T 3B5 is also a 32-bit general-purpose minicomputer. It is somewhat smaller than its predecessor, the 3B20, but it is software compatible with it. VPM forms the software structure used to support most data-linking capabilities on the 3B5.

The 3B5 VPM implementation is based on that for the 3B20, with C programs compiled into machine language and executed directly. In fact, the CSI, trace driver, protocol scripts, transparent driver, and many utility programs were simply ported from the 3B20 and recompiled. Because the PCD hardware is much different, the VPM operating system was redesigned, but it maintains the same interfaces as that on the 3B20. Thus, protocols that run on the 3B20 will, in general, run on the 3B5 with just a recompilation.

The PCD hardware consists of an intelligent peripheral controller, which runs the scripts, plus a collection of boards containing line interfaces for the various protocol classes. Several of these boards may be serviced simultaneously by the controller, with many different protocols running simultaneously.

The major software components have already been described in connection with the 3B20. On the 3B5, however, memory availability is the only limit on the number of processes supported by the VPM operating system, and a limited degree of protection between protocol programs exists. Memory allocation is dynamic, done when the scripts are loaded into the peripheral controller or by request of the running script via a primitive. Multiple instances of the same protocol may share the same copy of their program, using separate stacks and data areas.

The controller operating system and the primitives reside in Erasable Programmable Read-Only Memory (EPROM), but much of the code may be selectively replaced by down loading new versions when the system is initialized. Scheduling, event handling, and the rest of the program creation and down-load process are as described for the 3B20. In addition to the standard trace facility, routines exist that allow a script to output directly to an optional debugging port on the PCD rather than back to the host.

While VPM was originally intended to support only synchronous interfaces, on the 3B5 computer it has been extended to include asynchronous communication as well. This involved, besides providing the necessary hardware, the addition of the small collection of asynchronous primitives that were outlined in a previous section. These primitives are used to support a standard *UNIX* system terminal interface using either RS-232C or *Teletype* Standard Serial Interface.

VI. APPLICATIONS

VPM has been used by *UNIX* system developers and customers to implement a variety of protocols supporting various networking applications. Some of the more widely used protocols and applications have been developed for official *UNIX* system distribution; these are briefly described below. Many other protocols and applications have been developed by our customers; some of these are listed in the miscellaneous section below.

6.1 Remote job entry

The Remote Job Entry (RJE) system connects *UNIX* systems to IBM 360/370 computers by simulating a remote work station. The basic facility provided by RJE is the remote execution of jobs created on the *UNIX* system.

The IBM and *UNIX* systems communicate using a character-oriented protocol known as Houston Automatic Spooling Priority (HASP) multileaving. Three processes are used to implement the multileaving protocol: a PCD program and two user processes. The protocol program implements level 2. It performs header consistency and CRC-16 checks on received blocks, and it generates the CRC-16 data for transmitted blocks. It also performs Extended Binary-Coded Decimal Interchange Code (EBCDIC) to American National Standard Code Information Interchange (ASCII) translation on print data. The two user processes multiplex and demultiplex multiple job streams to and from a single data link.

6.2 Synchronous terminals

Two applications of VPM support IBM 3277-compatible display station (terminal) clusters. The Synchronous Terminal (ST) system allows terminal clusters to be connected to a *UNIX* system host, while the 3270 Emulation (EM) system allows applications to connect to hosts that support terminal clusters. Both of these packages have been implemented using VPM CSI.

Synchronous terminals communicate with the host through a single cluster controller using the BISYNC line protocol. Message traffic is regulated by using a polling and selecting scheme. The host polls the cluster for available input data and selects specific terminals for output.

The ST system software consists of a level-2 protocol script and a level-3 driver. The script implements the polling and selecting functions of the line protocol. The driver provides two different user interfaces: (1) In application mode, the controlling user process completely manages the display terminal screen. (2) In line mode, the driver provides enough basic screen management to make the device usable as a login terminal for most of the standard *UNIX* system commands.

The EM system software consists of a level-2 protocol script and a level-3 driver interface. The script implements the BISYNC line protocol of a display station controller. The driver interface is in two parts: a controller interface driver that handles link administration and controller functions, and a terminal interface driver that supplies the user-level interface.

6.3 X.25 interface

X.25 is an international standard layered data communications protocol that allows several virtual channels to be multiplexed over a single physical link. Each channel has its own flow control and error control.

The current version of X.25 in the *UNIX* system consists of three levels. On DEC computers, level 2 is implemented as a VPM protocol script. On AT&T computers, level 2 is implemented on PCDs that do not support VPM. Level 3 of X.25 is implemented using CSI, which makes it portable across all *UNIX* system hosts that support CSI.

6.4 5620 DMD support

The *Teletype* 5620 Dot Mapped Display (DMD) terminal is an intelligent peripheral containing a keyboard and display, an electronic "mouse" for cursor pointing, and an RS-232 output port for a dot matrix printer. The driver that supports it utilizes VPM. Through application code, options in the VPM-based driver, and software running on the DMD, multiple windows are supported on the terminal display.

This driver is based on the asynchronous terminal package with the addition of multiple communications channels and knowledge of the communications protocol used by the code running on the DMD. This involves dynamically replacing the line discipline used in standard terminal mode with one that multiplexes and demultiplexes packets intended for a virtual terminal, and it ensures that all packets are properly ordered. Flow control is provided to ensure that packets are not sent more quickly than they can be received.

6.5 Miscellaneous

Some customer-developed applications of VPM include:

- LEAP—A package similar to the 3270 emulation package that is used to load-test IBM host applications that use 3270-compatible terminals.

- Bell Administrative Network Communications Systems (BANCS)—A message-switching network for business communications. The internal protocols are based on BISYNC. A *UNIX* system interface to control the BANCS switches has been developed using VPM.
- BLN—An AT&T Bell Laboratories Network that connects hosts from different vendors typically running different operating systems. An interface to BLN for *UNIX* system hosts was developed using VPM.
- WANG—A protocol script was developed to allow *UNIX* systems to interface to a WANG word processer.

VII. CONCLUSION

VPM was developed in response to a need to implement several different character-oriented protocols on DEC's KMC11-B microprocessor. We did not have the resources or the inclination to develop and support assembly-language implementations of these protocols plus an unpredictable number of future requirements. We therefore were led to develop a general-purpose package for implementing level-2 protocols rather than several different assembly-language implementations of specific protocols.

As this effort unfolded, new requirements led us to expand VPM to include bit-stuffing protocols as well. When the *UNIX* system was ported to new computers with different PCDs, VPM became the means of porting level-2 protocol implementations to the different PCDs involved. Since VPM allowed the representation of a level-2 protocol to be hardware independent, it could be ported to other environments with little or no change. In a few cases, protocol implementations that were developed using VPM have been ported to environments unrelated to the *UNIX* system.

As VPM was extended to new *UNIX* system hosts, and higher-level protocols such as X.25 were implemented as *UNIX* system drivers, it became necessary to provide a means that would ensure the portability of these drivers. This led to the definition of the Common Synchronous Interface (CSI), which provides a device-independent interface between level-2 and level-3 protocols.

The clear success of VPM as a *UNIX* system facility is gratifying to all of us who had a part in developing it. The goal of opening up data communications programming to applications programmers has been met; customers really are writing their own communications applications. The ability to program link-level protocols in a high-level language has been valuable in debugging implementations of complex protocols such as X.25. The ability to port protocol implementations between computers, although not considered in the original goals, has become perhaps the most important feature.

VIII. ACKNOWLEDGMENTS

In addition to the authors of this article, a large number of people have contributed directly or indirectly to the development of VPM; among them are R. V. Baron, C. A. Bishop, R. J. Butera, T. A. Dolotta, J. A. Dziadosz, R. M. Ermann, R. C. Haight, C. B. Hergenhan, D. E. Jimenez-Puttress, G. W. R. Luderer, G. J. McGrath, B. Nohejl, V. H. Rosenthal, R. M. Sabrio, A. L. Sabsevitz, L. J. Schroeder, D. R. Shuman, B. A. Tague, B. E. Todd, and L. A. Wehr.

Finally, the utility of the *UNIX* system architecture, philosophy, and tools as a basis for the development of VPM is gratefully acknowledged.

REFERENCES

1. D. W. Davies, D. L. A. Barber, W. L. Price, and C. M. Solomonides, *Computer Networks and Their Protocols*, New York: Wiley, 1979.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
3. *UNIX System Administrator's Manual—Release 5.0*, Western Electric, June 1982.

AUTHORS

Michael J. Fitton, B.S. (Computer Science), 1981, Rutgers University; M.S. (Computer Science), 1983, Stevens Institute of Technology; AT&T Bell Laboratories, 1977—. Mr. Fitton has been involved in software development for the *UNIX* operating system. His initial work involved the design and implementation of tools for the Programmer's Workbench version of the *UNIX* system. He has also worked on data communications software and operating system development. He is currently working on the development of a multiprocessor version of the *UNIX* operating system.

Carol J. Harkness, B.A. (Mathematics), 1969, University of Wisconsin; M.S. (Computer Science), 1970, Purdue University; AT&T Bell Laboratories, 1969—. Ms. Harkness has designed editors, compilers, assemblers, and test tools for a variety of *ESS*™ projects. She became involved with microprocessors through debugging tool development, then went on to developing peripheral software and firmware for the AT&T 3B5 computer. She is currently the Supervisor of the Network Design group, developing the AT&T 3B Net local area network for the 3B computer line. Member, ACM, IEEE, Phi Beta Kappa, Phi Kappa Phi, Sigma Epsilon Sigma.

Keith A. Kelleman, B.S. (Electrical Engineering), 1979, Lafayette College; M.S. (Computer Science), 1981, Stevens Institute of Technology; AT&T Bell Laboratories, 1979—. At AT&T Bell Laboratories, Mr. Kelleman has been involved with *UNIX* operating system development. He is currently working on the development of a demand paged kernel for the *UNIX* system. His

previous assignments were to develop a *UNIX* system for the AT&T 3B20 computer and to convert the RJE system to VPM.

Paul F. Long, B.S. (Engineering Mathematics), 1960, M.S. (Applied Mathematics), 1963, North Carolina State University; Bellcomm, Inc., 1965–1972; AT&T Bell Laboratories, 1972—. Mr. Long has worked on various applications and systems programming projects and supervised similar activities over the last 18 years. In addition to working on VPM, he participated in the *UNIX* operating system BX.25 implementation as well as other *UNIX* system networking projects. Member, Tau Beta Pi, Pi Mu Epsilon, Sigma Xi.

Carl Mee III, B.S. (Mathematics), 1957, The University of the South; M.A. (Mathematics), 1964, The University of Virginia; U.S. Air Force, 1958–1962; Bellcomm, Inc., 1964–1966, 1968–1972; Informatics, Inc., 1966–1968; AT&T Bell Laboratories, 1972—. Mr. Mee has worked on a variety of applications and systems programming projects. From 1978 to 1983 he worked on the development of communications facilities and other software for the *UNIX* operating system. He is currently working on the development of video-based interactive information systems.