Release 2.0 of the UNIX Virtual Protocol Machine

P. F. Long
C. Mee, III

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This memorandum describes the second release of the UNIX† Virtual Protocol Machine (VPM). VPM is a general-purpose synchronous UNIX communications interface that allows link-level protocols such as BISYNC and HDLC to be implemented on the KMC11-B (a DEC microcomputer) in a high-level language. The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs.

The first release of VPM supports a class of byte-oriented half-duplex protocols collectively known as BISYNC. The present release adds support for bit-oriented, full-duplex protocols such as the international standard High-Level Data Link Control (HDLC). Other features of Release 2.0 include:

1. An increase in the number of buffers that the interpreter can accept at one time.
2. Additional debugging facilities.
3. Provisions for interprocess communication between the protocol script and a UNIX driver or a user process.
4. A cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

The procedures for adding VPM Release 2.0 to a UNIX 3.0 system and testing it to ensure proper operation are given.

Introduction

This memorandum describes the second release of the UNIX Virtual Protocol Machine (VPM). The first release was described in a previous memorandum [1], which should be read as background for this memorandum. See also the *UNIX User's Manual* [4] entry for *vpm*(4).

VPM is a general-purpose UNIX interface for synchronous communications lines. VPM allows link-level protocols such as BISYNC and HDLC to be implemented on the DEC KMC11-B microcomputer in a high-level language. The hardware required to support VPM is a PDP-11/70, /45, or /34, or a VAX-11/780 host computer, a KMC11-B microcomputer, and a DMC11-DA, -FA, or -FD synchronous communications interface. All of the above items are manufactured by DEC. The use of the KMC microcomputer allows the VPM to perform direct-memory-access (DMA) transfers to and from main memory. The link-level communications protocol is executed by the VPM interpreter running in the KMC microcomputer. This implementation technique leads to a portable protocol representation and efficient protocol execution.

The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs. The compiler, which executes in the host computer, translates a protocol described in a high-level language into a load module for the KMC. The load module contains the VPM interpreter and a compiled representation of the protocol. The

† UNIX is a trademark of Bell Laboratories.

interpreter executes the protocol, communicates with the UNIX driver in the host computer, and controls the communications line interface.

The first release of VPM supported a large class of protocols collectively known as BISYNC. These protocols are distinguished by the use of control characters to provide framing and transparency. At the frame level, these protocols operate in a half-duplex manner, although they sometimes use full-duplex communications facilities to reduce the time required to reverse the direction of transmission.

Release 2.0 of VPM adds support for bit-oriented, full-duplex protocols. This class of protocols includes IBM's Synchronous Data Link Control (SDLC) and the international standard High-Level Data Link Control (HDLC). LAPB, a subset of HDLC which is the link-level protocol specified in the BX.25 Bell System Standard, has been implemented using VPM and is available with this release [2,3]. The interpreter used for bit-oriented protocols is different from that used for character-oriented (BISYNC) protocols. The appropriate interpreter is selected by means of a compiler option.

Other features of Release 2.0 include:

1. An increase in the number of transmit and receive buffers that the interpreter can accept at one time.
2. Additional debugging facilities.
3. provisions for interprocess communication between the protocol script and a UNIX driver or a user process.
4. A cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

Support for Bit-Oriented Protocols

The capability to use bit-oriented protocols such as HDLC is provided by a new set of communications primitives. These primitives are frame-oriented and non-blocking, whereas the BISYNC primitives are character-oriented and blocking. The new primitives are fully described in the manual entry for *vpmc*(1C). An overview of these primitives follows.

The VPM interpreter maintains a set of queues for transmit buffers and another set of queues for receive buffers. When a transmit buffer is passed to the KMC by the UNIX driver, the buffer is appended to the unopened-transmit-buffer queue. The protocol script in the KMC obtains a transmit buffer from the unopened-transmit-buffer queue by means of the *getxfrm* primitive; the buffer is then said to be *open*. In order to get (open) a transmit buffer, the script must provide a transmit-sequence number. This sequence number must be in the range 0-7 and must be distinct from the sequence number currently assigned to every other currently-open transmit buffer. This sequence number is used to identify the buffer for subsequent calls to the *xmtfrm* and *rtnxfrm* primitives. The *xmtfrm* primitive initiates transmission of the specified buffer, using the control information specified by a previous *setctl* primitive. Transmission proceeds asynchronously. The script can test for completion of an output transfer by means of the *xmtbusy* primitive. Open transmit buffers can be transmitted any number of times. When the script decides that a buffer has successfully been received at the destination, it notifies the interpreter by means of the *rtnxfrm* primitive. This causes the buffer to be placed on the transmit-buffer-return queue; the buffer is then no longer considered to be open and the sequence number can be reused. The driver is notified as soon as possible that the buffer has been closed. The buffer is then removed from the transmit-buffer-return queue.

When a receive buffer is passed to the KMC by the driver, the buffer is placed on the empty-receive-buffer queue. When the first byte of a new frame arrives, an empty receive buffer is obtained from the empty-receive-buffer queue and the incoming characters are placed into the buffer as they arrive. An incoming frame will be discarded if the frame is too short (less than four bytes including CRC), if the frame is too long to fit in the receive buffer, or if the CRC is incorrect. If a frame is received successfully, the buffer is placed on the completed-receive-frame queue, otherwise the buffer is returned to the empty-receive-buffer queue. When the script executes a *rcvfrm* primitive, the buffer at the head of the completed-receive-frame queue is removed from that queue and becomes the current receive buffer. If the script subsequently executes a *rtnrfrm* primitive before executing another *rcvfrm* primitive, the current receive buffer is placed on the receive-buffer-return queue. If the script executes a *rcvfrm* primitive before executing a *rtnrfrm* primitive, the current receive buffer, if any, is returned to the empty-receive-frame queue. Buffers on the receive-buffer-return queue are returned to the driver at the first opportunity.

If the empty-receive-buffer queue is empty when the first byte of a new frame is received, the first five bytes of the frame are retained in a staging area and the remainder of the frame is discarded. This allows a protocol script to receive a control frame (up to seven bytes including CRC) when no data buffer is available. When the next *rcvfrm* primitive is executed, the script will receive the information in the staging area along with an indication that the remainder of the frame has been discarded. If another frame arrives while the staging area is thus occupied, the new frame is discarded entirely.

A count is kept of the number of frames discarded for each reason. These counters may be read and reset from the host computer.

The VPM Split Driver

Because the VPM interpreter and a protocol script generally use most of the memory of the KMC, any higher levels of protocol that are required must be executed by the host CPU. The purpose of the VPM split driver is to provide a framework in which higher-level protocols can be implemented conveniently using low-level routines in the VPM driver to communicate with the interpreter in the KMC.

A set of functions has been written that provides a general-purpose interface to the link-level protocol being executed by the interpreter in the KMC. Their capabilities include a means to queue transmit and empty receive buffers for use by the protocol script in the KMC, to start and stop the script, and to send commands to and receive reports from the script. A means of getting a copy of and resetting the VPM interpreter's error counters is also provided. These functions will be referred to as interface functions or collectively as the interface module. Appendix 1 contains a description of each of these routines.

To implement higher levels of a protocol as a UNIX device driver, a set of routines must be written to implement the standard UNIX system calls: *open*, *close*, *read*, *write*, and *ioctl* as well as the required protocol. These routines will be referred to as protocol functions or collectively as a protocol module. The standard VPM driver does not implement a higher-level protocol but instead provides a transparent user interface that can be used by applications that supply their own higher levels of protocol. This driver can be used as an example for those interested in writing a different protocol module. Appendix 2 contains a description of these routines.

At least two other protocol modules have been written thus far. They are the Synchronous Terminal Interface (see *st*(4)) and the BANCS THP Interface.

Release 2.0 of VPM allows up to four different VPM protocol modules to be executing simultaneously. One KMC and one interface-module minor device[1] are required for each protocol being executed. Any number of protocol modules may be implemented, but no more than four can be in use at any one time because no more than four KMCs are supported. In general, each

1. Strictly speaking, the interface module is not a driver and therefore does not have minor devices; however, the minor device number in this case selects an element of the data-structure array associated with the interface module in the same way that the minor device number associated with a driver selects an element of a data-structure array.

protocol module can have up to 256 minor devices. The VPM Release 2.0 protocol module, however, can have at most 16 minor devices; this restriction is due to the fact that the minor device number of the VPM protocol module is used not only to specify the VPM minor device but also to specify the interface-module minor device and the KMC minor device. The low-order four bits of the protocol-module minor device number determine the protocol-module minor device; the next two bits determine the interface-module minor device; the next two bits determine the KMC minor device.

Transmit buffers and receive buffers are passed between the VPM interpreter, the interface module, and the protocol module by means of pointers to data structures known as *buffer descriptors*. The buffer-descriptor structure is defined as follows:

```
struct vpmbd {
        short   c_ct;           /* Buffer size */
        short   d_adres;        /* Low-order 16 bits of buffer address */
        char    d_hbits;        /* High-order 2 bits of buffer address */
        char    d_type;         /* Protocol-dependent */
        char    d_sta;          /* Protocol-dependent */
        char    d_dev;          /* Protocol-dependent */
        struct buf *d_buf;      /* Pointer to system buffer descriptor */
        int     d_bos;          /* Index of next byte in buffer */
        int     d_vpmtdev;      /* Minor device number */
}
```

For empty receive buffers, *c_ct* must be equal to the buffer size in bytes; for transmit buffers, *c_ct* must be equal to the number of bytes to be transmitted. When a receive buffer is returned to the protocol module, *c_ct* is equal to the number of data bytes in the buffer. *D_adres* and *d_hbits* must contain an 18-bit UNIBUS-mapped buffer address; the low-order 16 bits must be in *d_adres* and the high-order two bits must be in the low-order two bits of *d_hbits*. *D_type, d_sta*, and *d_dev* are protocol-dependent; when using the BISYNC interpreter these three bytes may be read and modified by the protocol script. See the discussion of *getxbuf, getrbuf, rtnxbuf,* and *rtnrbuf* in the manual entry for *vpmc*(1C). *D_buf* contains a pointer to a system buffer descriptor; this is used to return the buffer to the system buffer pool. *D_bos* is the index of the first byte in the buffer not yet returned to the user. *D_vpmdev* is the minor device number of the protocol-module minor device to which the buffer is allocated.

The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver and the protocol script to aid in debugging new protocol modules and protocol scripts. It may also be used to debug other drivers or system code not related to the VPM driver. This driver can be configured to have a number of minor devices. Each minor device provides a means by which a user program can read data generated by functions within the operating system. This data is recorded by calls to *trsave* as described in Appendix 3. Each call to *trsave* generates a unit of data known as an *event record* which consists of a channel number (one byte), a count (one byte) and *count* bytes of data. The channel number can be used to multiplex up to 16 data streams on each minor device.

Associated with each minor device of the trace driver is a *clist* queue which is used to save event records provided a user program has that minor device open and has enabled the channel to which the event records were written. Channels may be enabled in any combination, using the *ioctl* command VPMTRCO. See the manual entry for *trace*(4). While a minor device read queue is full, event records for that minor device are discarded. Appendix 3 contains a description of each trace-driver routine.

Minor device 0 of the trace driver is used by the VPM driver to record a variety of debugging information generated within the VPM driver and also to record the data generated by the *trace* primitive in a protocol script. Minor device 1 of the trace driver is used to record the information generated by the *snap* primitive in a protocol script. The *vpmtrace* and *vpmsnap* commands are available for reading and formatting the data passed via these two minor devices. These two commands are described in the manual entry for *vpmstart*(1C). Appendix 4 contains a description of the VPM driver event trace.

Miscellaneous Improvements

Two new primitives have been added to the protocol language to allow communication between the link-level protocol script in the KMC and a higher-level protocol implemented in a user program or a VPM protocol module. The *getcmd* primitive allows the script to receive a four-byte command from a user program or a protocol module. The standard VPM protocol module allows a user program to pass a command to the script via an *ioctl* system call. Other VPM protocol modules can pass a command to the script by calling the *vpmcmd* routine in the VPM interface module. The *rtnrpt* primitive allows the script in the KMC to send a four-byte report to a protocol module or to a user program. The standard VPM protocol module allows a user program to receive a script report by means of an *ioctl* system call. A protocol module can receive reports from the interface module by calling the *vpmrpt* routine of the VPM interface module.

The *trace* primitive of the protocol language has been augmented to allow two arguments. The form with one argument is still supported; if only one argument is given, the second argument is assumed to be zero. A *snap* primitive has been added. This primitive causes four bytes of data from the script followed by a four-byte time stamp to be placed on the read queue for trace driver minor device 1.

The *timeout* primitive provided in Release 1.0 has been supplemented by a new *timer* primitive that allows a script to initialize a timer or test its current value. If the argument to *timer* is non-zero, the timer is initialized with the value of the argument. The timer is decremented ten times a second until the timer reaches zero. If the *timer* primitive is called with an argument of zero, it returns the current value of the timer. This value is zero if the timer has expired, otherwise non-zero.

In release 1.0 of VPM, the interpreter would accept at most one transmit buffer and one receive buffer at any given time. In Release 2.0 the interpreter will accept up to four transmit buffers and four receive buffers at a time. This applies to both the character-oriented (BISYNC) interpreter and the bit-oriented (HDLC) interpreter.

For applications with requirements for monitoring the integrity of the computer hardware and software, a form of cross-checking between the UNIX driver and the KMC has been implemented. Every three seconds the VPM interpreter in the KMC sends an "I'm-OK" report to the host; the host responds by sending an "I'm-OK" command to the KMC. If either the host or the KMC does not receive the "I'm-OK" signal within a reasonable time period, an error termination occurs.

Appendix 5 contains detailed instructions for adding VPM Release 2.0 to a UNIX 3.0 system. Appendix 6 describes a number of test programs and procedures that may be used to check the VPM hardware and software and to gain familiarity with the system.

Acknowledgements

We would like to thank our supervision, especially R. C. Haight and G. W. R. Luderer, for their support of the Virtual Protocol Machine. L. A. Wehr provided the initial idea of interpreting protocol descriptions with the KMC and helped us with debugging and useful advice from time to time. R. V. Baron of Department 9362 suggested a number of new features that became part of this release. R. M. Ermann of Department 5251 wrote the protocol script for the LAPB protocol and suggested several improvements in the HDLC primitives.

References

[1] Long, P. F. and Mee, C., III. *Release 1.0 of the UNIX Virtual Protocol Machine*, Bell Laboratories.

[2] Ermann, R. M. *Formal Specification of X.25 Compatible Link Protocol*, Bell Laboratories.

[3] Ermann, R. M. *Portable Implementation of BX.25 Level 2*, Bell Laboratories.

[4] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories.

Appendix 1: The VPM Interface Module

The VPM interface functions provide a general-purpose interface between a higher-level protocol implemented in a VPM protocol module and the link-level protocol script executed by the VPM interpreter in the KMC. The KMC driver is used by the interface functions to pass commands to and receive reports from the VPM interpreter. When reports are received by the interface module that must be passed on to the protocol module, the protocol module's receive-interrupt routine (*vpmtrint* in the case of the standard VPM protocol module) is called.

This appendix describes each interface function. *Dev* is an argument to many of the interface functions and has the same meaning for all but two of them: the low-order four bits of the *dev* argument are *not* used by the interface functions; the next two bits determine the interface module minor device number; the next two bits determine the KMC minor device. Although *dev* is declared as an *int*, only the low-order eight bits are meaningful at this time. In calls to the *vpmtrace* and *vpmsnap* routines, *dev* need not be a minor device number because it is just saved as part of the event record. The definition of *dev* will not be repeated for each function.

vpmcmd (dev, cmd)
int dev;
char *cmd;

This function passes a command to the script. *Cmd* is the address of a four-byte array. The four bytes are passed to the VPM interpreter, which saves them until the protocol script executes a *getcmd* primitive. Only the most recent four bytes passed by a *vpmcmd* call are saved by the VPM interpreter.

struct vpmbd *vpmdeq (clp)
struct clist *clp;

This function removes the buffer-descriptor pointer at the head of the queue pointed to by *clp* and returns it to the caller. If the queue is empty, a null pointer is returned.

vpmemptq (dev, bdp)
int dev;
struct vpmbd *bdp;

This function is used to pass an empty receive buffer for use by the interpreter in the KMC. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer descriptor is appended to the empty-receive-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another empty receive buffer, the buffer at the head of the queue is removed and passed to the KMC. The sum of the number of buffers on the empty-receive-buffer queue and the number of receive buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

vpmxmtq (dev, bdp)
int dev;
struct vpmbd *bdp;

This function is used to pass a transmit buffer to the interpreter in the KMC. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer descriptor is appended to the transmit-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another transmit buffer, the buffer at the head of the queue is removed and passed to the KMC. The sum of the number of buffers on the transmit-buffer queue and the number of transmit buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

vpmenq (bdp, clp)
struct vpmbd *bdp;
struct clist *clp;

If *bdp* is a null pointer, the number of buffer-descriptor pointers on the *clist* queue pointed to by *clp* is returned. If *bdp* is not a null pointer, the buffer descriptor pointed to by *bdp* is appended to the *clist* queue pointed to by *clp* and the number of pointers currently on that queue is passed as the return value.

char *vpmerrs (dev, n)
int dev, n;

This function is used to read and reset the error counters in the VPM interpreter. The function passes a GETECMD command to the VPM interpreter and blocks until the interpreter responds; this command causes the interpreter to copy its error counters to an array in the interface module and send a completion report to the driver. After the copy operation is completed, a pointer to the error-count array is passed to the caller as the return value. The second argument is not currently used.

char *vpmrpt (dev)
int dev;

This function is used to receive a script report from the KMC. When the protocol script executes a *rtnrpt* primitive, four bytes of data are passed to the interface module. If a *rtnrpt* has been executed by the protocol script since the last call to *vpmrpt*, a pointer to the four bytes passed by the most recent *rtnrpt* primitive is returned; otherwise zero is returned.

vpmsave (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function creates an event record with the following structure:

```
struct {
        short   c_seqn;         /* Sequence number */
        char    c_type;         /* Argument type */
        char    c_dev;          /* Argument dev */
        short   c_word1;        /* Argument word1 */
        short   c_word2;        /* Argument word2 */
}
```

This event record is passed to the *trace* driver using *trsave*.

vpmsnap (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function is similar to *vpmsave*. The only difference is that a time stamp *(long s_lbolt)* is added to the event record after *word2*. A protocol script may generate a time-stamped event record by executing the *snap* primitive.

```
vpmstart (dev, type, rint)
int dev, type;
int (*rint)( );
```

This function must be called on the first open of the protocol-module minor device associated with the interface-module minor device and KMC identified by *dev*. *Type* is a number that identifies the program running in the KMC and must agree with the value specified when the KMC load module was loaded into the KMC. For VPM interpreters, *type* is conventionally 6. *Rint* is the name of a protocol-module routine to be called by the interface module when it needs to return a transmit buffer, a receive buffer, a script report, or an error-termination code. See the description of *vpmtrint* in Appendix 2 for an example of such a routine. *Vpmstart* sends a RUN command to the VPM interpreter which causes it to begin execution of the protocol script. If the interface module identified by *dev* is not configured, ENXIO is returned. If the module is already running, i.e., *vpmstart* has been called and *vpmstop* has not been called, or if the KMC is not running or was loaded using a different magic number, EACCES is returned. A return value of zero indicates a normal completion.

```
vpmstop (dev)
int dev;
```

This routine is called to halt the execution of the protocol script by the interpreter. The routine waits until the last transmit buffer has been returned by the protocol *script*, or until five seconds have elapsed, and then sends a HALT command to the VPM interpreter which causes the interpreter to stop executing the protocol script. When the interpreter acknowledges the HALT command, or after five seconds, any transmit or receive buffers still enqueued on the interface module's transmit- and empty-buffer queues are returned to the protocol module. This does not include buffers contained in the interpreter's queues. Generally, when the protocol script is halted normally, the interpreter will have one or more empty receive buffers. If the interpreter or protocol script terminates in error, some transmit buffers may also remain unaccounted for. The upshot of this is that a protocol module must keep a record of all buffers in use for each particular minor device, so that these buffers can be returned to the pool of available buffers when that minor device is closed.

Appendix 2: The VPM Protocol Module

This appendix gives a detailed description of the functions that make up the standard VPM protocol module. The description may be useful as a guide in writing other VPM protocol modules. The *dev* argument to the following routines is declared as an *int*; however, only the low-order eight bits are meaningful at this time. The low-order four bits are used to determine the minor device of the protocol module; the next two bits determine the minor device of the interface module; the next two bits determine the KMC minor device.

vpmopen (dev, flag)
int dev, flag;

This function opens the protocol-module minor device specified by the low-order four bits of *dev*. *Flag* contains the option bits specified on the *open* system call. Exclusive or non-exclusive *opens* are permitted. If the driver is opened for both reading-and-writing, the *open* is exclusive, i.e., no further *opens* are permitted. If the device is opened for reading only or for writing only, the *open* is non-exclusive and subsequent *opens* for reading only or writing only are permitted. If this device is not open when this function is called, it obtains a number of non-addressable system buffers to be used as receive buffers and passes them to the VPM interpreter using the interface routine *vpmemptq*. *Vpmopen* also calls the interface routine *vpmstart* if the minor device was not already open.

vpmclose (dev)
int dev;

This function closes the minor device specified by the low-order four bits of *dev*. It calls the interface routine *vpmstop*, flushes the receive queue for the specified minor device, releases its buffers, and reinitializes its data structure.

vpmwrite (dev)
int dev;

This function implements the *write* system call. If the transmit queue is not full, the function obtains a non-addressable system buffer, copies up to 512 bytes of the user's *write* data into it, and enqueues the buffer on the level 2 transmit queue using the interface function *vpmxmtq*. These steps are repeated until all of the user's *write* data has been copied. If the transmit queue is full when this function is called or if it becomes full while the function is executing, the calling process is blocked until there is room in the queue for more transmit buffers.

vpmread (dev)
int dev;

This function implements the *read* system call. When it is called, the calling process is blocked until the receive queue is non-empty. As data is received by the VPM interpreter, it is placed into an empty receive buffer. When the protocol script decides that the data contained in a particular buffer is valid, it executes a *rtnrbuf* (BISYNC) or *rtnrfrm* (HDLC) primitive which causes the buffer descriptor pointer to be passed to the interface modules interrupt routine. The interface module then passes the buffer descriptor pointer to the protocol module by calling the protocol module's interrupt routine. The protocol module enqueues the buffer descriptor pointer on the receive queue and wakes up (unblocks) the reader(s). The number of bytes requested, or the data in one buffer, whichever is less, is copied to the user process; the number of bytes copied is passed as the return value. Any bytes remaining in a buffer are used to satisfy subsequent *read* requests.

vpmioctl (dev, cmd, arg, mode)
int dev, cmd, mode;
char *arg;

This function implements the *ioctl* system call. *Cmd* determines the function to be performed as follows:

> VPMCMD — Pass a command to the protocol script. The first four bytes of the array pointed to by *arg* are passed to the VPM interpreter which saves them and passes them to the protocol script the next time it executes a *getcmd* primitive.
>
> VPMERRS — Get and reset the VPM interpreter's error counters. The eight-byte array containing the VPM interpreter's error counters is copied to the user array pointed to by *arg*. The interpreter's copy of the error counters is then set to zero.
>
> VPMRPT — Get a report from the protocol script. If the protocol script has executed a *rtnrpt* primitive since the last time this *ioctl* command was issued, the script report (four bytes) is copied to the user array pointed to by *arg* and *one* is passed as the return value; otherwise, *zero* is passed as the returned value.

The *mode* argument is not used. The values for VPMCMD, VPMERRS, and VPMRPT are defined in file */usr/include/sys/vpm.h*.

vpmtrint (dev, code, bdp)
int dev, code;
struct vpmbd *bdp;

The address of this function is passed to the protocol module using the *vpmstart* function described in Appendix 1. This routine is called from the interface module to return transmit buffers, receive buffers, script reports, or error termination codes. It is usually called at interrupt priority and therefore must not sleep or do unnecessary work. *Code* identifies the purpose of the call and determines the meaning of *bdp* as follows:

> RRTNXBUF — *Bdp* is a pointer to the buffer descriptor for a transmit buffer. This call is made when the protocol script executes a *rtnxbuf* (BISYNC) or a *rtnxfrm* (HDLC).
>
> RRTNRBUF — *Bdp* is a pointer to the buffer descriptor for a receive buffer. This call is made when the protocol script executes a *rtnrbuf* (BISYNC) or a *rtnrfrm* (HDLC).
>
> RRTNEBUF — *Bdp* is a pointer to the buffer descriptor for an empty receive buffer. This call is used to return empty receive buffers when the interface module is stopped by calling *vpmstop*.
>
> ERRTERM — *Bdp* is the error-termination code passed to the interface module by the VPM interpreter when it halts the protocol script because of an error condition. The meaning of these error codes is given in the manual entry for *vpm*(4).

The values for RRTNXBUF, RRTNRBUF, RRTNEBUF, and ERRTERM are defined in file */usr/include/sys/vpm.h*.

Appendix 3: The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver, a protocol script, or some other driver. See the manual entry for *trace*(4).

A description of each routine of the trace driver follows.

tropen (dev)
int dev;

This function opens the minor device specified by *dev* exclusively.

trclose (dev)
int dev;

This function closes the minor device specified by *dev*. It discards any data on the read queue and initializes the data structure associated with the minor device.

trread (dev)
int dev;

This function implements the *read* system call; it sleeps until at least one event record is available on the read queue associated with *dev*. It then copies records to the user until the user's read count is less than the number of bytes in the next event record or until the read queue is empty. The number of bytes copied is passed as the return value.

trioctl (dev, cmd, arg, mode)
int dev, cmd, arg, mode;

This function implements the *ioctl* system call. *Cmd* indicates the operation to be performed. The driver has one command:

> VPMTRCO — Enable a trace channel. In order for data to be saved on the read queue for minor device *dev*, the device must be open and the channel to which it is written must be enabled. This command enables channel *arg*, which must be in the range 0 to 15. Any combination of channels may be enabled by repeatedly calling this function with different values of *arg*. All channels are disabled when the minor device is closed.

trsave (dev, chno, buf, ct)
char dev, chno, *buf, ct;

If minor device *dev* of the trace driver is open and channel *chno* of that minor device is currently enabled then *chno* and *ct*, followed by *ct* bytes starting at address *buf*, are copied onto the read queue associated with *dev*, provided the read queue for that device has room for the complete event record. If there is not room for the complete event record, the record is discarded.

Appendix 4: The VPM Event Trace

Calls to the interface routine *vpmsave* have been placed strategically throughout the standard VPM protocol module (*vpmt.c*) and the VPM interface module (*vpmb.c*) to provide an event trace for debugging new protocol modules and/or protocol scripts. A protocol script may generate an event record by executing a *trace* primitive. All such event records are discarded unless some user program has opened minor device 0 of the trace driver and enabled channel 0 of that minor device. The command *vpmtrace*(1C) opens this device and enables channel 0, then reads event records and prints them on the standard output as they are received. Each kind of event record that is generated by the VPM driver will be described by giving the *vpmsave* function call as it appears in *vpmt.c* or *vpmb.c*, followed by an example of the line printed by *vpmtrace* as a result of this call. Following this, the context of the *vpmsave* call and the definition of the parameters passed will be given. The definition of a parameter that appears in more than one call will not be repeated. The first five calls to *vpmsave* occur in the source file *vpmt.c*; the remaining calls occur in *vpmb.c*.

vpmsave ('p', dev, ec, 0)

```
243 p 100 15 0
```

Called if *vpmstart* returns an error code. The first field of the printed record contain a sequence number assigned by *vpmsave*. The remaining four fields contain the four remaining arguments to *vpmsave* in the same order as they appear in the call to *vpmsave*. The first argument to *vpmsave*, in this case a 'p', identifies the record type. *Dev* is the minor device number as defined earlier; *ec* is the value returned by *vpmstart*.

vpmsave ('o', dev, vp−>vt_state, 0)

```
244 o 100 1 0
```

Called just before the normal return point of *vpmopen*. The variable, *vp−>vt_state*, contains the state bits for the protocol module. Refer to the source file, *vpmt.c*, for the definitions of the state bits.

vpmsave ('c', dev, vp−>vt_state, 0)

```
245 c 100 13 0
```

Called from *vpmclose* just before the state bits are initialized.

vpmsave ('w', dev, ct, dp)

```
246 w 100 1000
```

Called just before putting a buffer-descriptor pointer on the transmit queue in *vpmwrite*. *Ct* is the number of bytes in the buffer. When executing on a PDP11, *dp* is the pointer to the buffer descriptor; *dp* is not meaningful when executing on a VAX because pointers are four bytes on a VAX and the argument corresponding to *dp* is declared as a *short*.

vpmsave ('r', dev, cnt, dp−>d_bos)

```
247 r 100 500 500
```

Called from *vpmread* just after *cnt* bytes have been moved to the user's read buffer. The parameter *dp−>d_bos* is the number of bytes remaining in the current receive buffer.

vpmsave ('s', dev, vp−>vb_state, 0)

```
248 s 100 401 0
```

Called just before the normal return from *vpmstart*. The parameter *vp−>vb_state* contains the state bits for the interface module. For the definitions of the state bits, refer to the source file *vpmb.c*.

vpmsave ('t', dev, vp−>vb_state, vp−>vb_xbkmc)

249 t 100 0 0

Called just before the normal return from *vpmstop*. The parameter *vp−>vb_xbkmc* is the number of transmit buffers currently held by the VPM interpreter. It can be non-zero if the protocol script or interpreter terminates in error.

vpmsave ('X', dev, vp−>vb_xbkmc, 0)

250 X 100 1 0

Called from *vpmbrint*, the interface module's receive-interrupt routine, each time the VPM interpreter returns a transmit buffer.

vpmsave ('R', dev, vp−>vb_vrkmc, 0)

251 R 100 1 0

Called from *vpmbrint* each time the VPM interpreter returns a receive buffer. The parameter *vp−>vb_rbkmc* contains the number of receive buffers currently held by the interpreter.

vpmsave ('T', dev, sel4, sel6)

252 T 100 370 21 34

Called from *vpmbrint* when a *trace* report is received from the interpreter. This occurs when the protocol script executes a *trace* primitive. *Sel4* contains the value of the script location counter (plus two) at the time the *trace* primitive was executed. By referring to the assembly-language listing of the protocol script generated by the *−l* option of *vpmc*, the point in the protocol script at which the *trace* was executed can be determined. The value of the location counter is two greater than the location of the *trace* instruction as shown in the assembly-language listing. *Sel6* contains the byte or bytes passed by the *trace* primitive. *Vpmtrace* prints these two bytes in separate fields.

vpmsave ('E', dev, sel4, sel6)

253 E 244 21

Called from *vpmbrint* when an error-termination report is received from the interpreter. *Sel4* contains the script location counter at the time execution of the script was terminated. *Sel6* contains the termination code. For an explanation of these codes see the manual entry for *vpm*(4).

vpmsave ('P', dev, sel4, sel6)

254 P 100 2105 1055

Called from *vpmbrint* when a script report is received from the interpreter. This occurs when the protocol script executes a *rtnrpt* primitive. *Sel4* and *sel6* contain the four bytes transferred by this primitive.

vpmsave ('F', dev, sel4, sel6)

255 F 100 3 0

Called from *vpmbrint* when an error-count report is received from the interpreter. *Sel4* and *sel6* do not contain any meaningful data for this event type.

vpmsave ('S', dev, sel4, sel6)

256 S 100 401 0

Called from *vpmbrint* when a start-up report is received from the interpreter. The low-order eight bits of *sel4* contain a parameter defining the maximum number of transmit buffers the interpreter can accept; the high-order eight bits contain a parameter defining the maximum number of receive buffers. *Sel6* contains the options supported by the interpreter.

vpmsave ('C', dev, vp−>vb_state, bp−>vb_xbkmc)

257 C 100 1 0

Called from *vpmclean* just before the data structure associated with *dev* is initialized.

vpmsave ('O', dev, vp−>vb_state, 0)

258 0 100 1 0

Called from *vpmok* if the interpreter should fail to indicate its sanity by issuing an "I'm-OK" report within the prescribed time limit.

Appendix 5: Adding VPM to a UNIX Release 3.0 System

The UNIX Release 3.0 distribution tapes contain VPM Release 2.0. This includes the compiler, drivers, interpreters, utility commands, protocol scripts, and test programs.

The *makefile vpm.mk* found in */usr/src/cmd/vpm* may be used to make and install all VPM commands.

To add the VPM and trace drivers to a UNIX 3.0 system, do the following:

1. Make sure that the following two lines appear in the file */etc/master*:

```
vpm      0   37   206   vpm  0   0    15    16    5
trace    0   35   206   tr   0   0    16    4     1
```

2. Add the following line to the file */usr/src/uts/*/cf/cfigpa* (or its equivalent):

```
vpm      0   0    0     n
```

   where *n* is the number of minor devices required. The * represents either *pdp11* or *vax*.

3. To the same file add the following line for each trace minor device:

```
trace    0   0    0     n
```

   where *n* is the number of minor devices required. Minor device 0 is used by the *vpmtrace* command and minor device 1 is used by *vpmsnap*.

4. If KMCs are being added to the system, add the following line to the same file for each KMC:

```
kmc11     vector     address     priority
```

   where *vector* is the interrupt vector location (octal), *address* is the device address (octal), and *priority* is the bus request level (normally 5).

A special file must be created in */dev* for each KMC, VPM, and trace device. To make these special files, use *mknod*(1M) as follows:

For KMCs:

```
/etc/mknod /dev/kmc? c X ?
```

where *X* is the major device number of the KMC driver as printed by **config −t** (see the manual entry for *config*(1M)*[4]*) and *?* is the minor device number that must be in the range 0 to 3.

For VPMs:

```
/etc/mknod /dev/vpm c Y Z
```

where *vpm* is a unique device name; *Y* is the major device number of the VPM driver; and *Z* is a decimal or octal number whose binary representation is defined as follows: the low-order four bits specify one of up to 16 minor devices of the standard VPM protocol module; the next two bits specify one of up to four VPM interface-module minor devices; the next two bits specify the minor device number of the KMC to be used for this special file.

For trace devices:

```
/etc/mknod /dev/trace c Y 0
/etc/mknod /dev/snap c Y 1
```

where *Y* is the major device number of the trace driver.

Hardware Installation and Switch Settings

The KMC11-B microprocessor and DMC11-DA, -FA, or -MD line unit must be installed in adjacent slots of a PDP-11 or VAX-11/780 backplane. Care should be taken not to exceed the DC power capacity of the cabinet in which the items are installed. The microprocessor and line unit are interconnected by a one-foot ribbon cable. The DMC11-DA or -FA line unit is connected to a suitable synchronous modem by a DEC-supplied modem cable. If the HDLC interpreter is used, the modem must be optioned for full-duplex (four-wire) operation; at speeds above 1200 bits per second this will normally require a private line. The DMC11-DA has an RS-232 interface that is suitable for connection to data sets such as the 208 and 209. The DMC11-FA has a CCITT V35 interface. The DMC11-MD has an integral 56 KB modem; this unit must be connected by a pair of coaxial cables to another DMC11-MD. The device address and interrupt vector address switches on the KMC should be set for the selected addresses. The KMC should also be wired for the selected bus priority (normally 5). All switches and jumpers on the DMC line unit should be in the normal configuration prescribed by the relevant DEC maintenance manual, but with one exception: the NO CRC switch (switch S2 in switch pack number 1) should be in the ON position. The purpose of this switch setting is to inhibit hardware CRC generation. Hardware CRC generation is not used with the VPM software for this device.

If the KMC is a Revision E, a DEC field change (ECO number NU007) is required before it can be used with the VPM or DZ/KMC software. If the change has already been installed, the capacitor that controls the KMC internal clock (capacitor C40, located four IC's over from the right edge of the KMC hex board—component side facing you, fingers down) will have a value of 4700 pF.

Appendix 6: Testing VPM

During the course of developing and testing VPM, a number of programs and test procedures have evolved which may prove useful to those adding VPM to a system or using VPM for the first time. These programs and procedures will help to check the correct installation and operation of the hardware and software as well as help a new user of VPM to gain familiarity with the package. These programs may be found in */usr/src/cmd/vpm/demo* and */usr/src/cmd/vpm/scripts*.

Decbin

Decbin is a simple KMC program that exercises enough of the KMC memory and instruction set so that a correct result provides reasonable assurance that the KMC is functioning properly. It does *not* exercise the interface between the KMC and the DMC11 line unit.

To run this test, you must compile file *decbin.k* in directory */usr/src/cmd/vpm/demo*. This can be done as follows:

```
/lib/cpp /usr/src/cmd/vpm/demo/decbin.k | kasb
```

You must then load and run the resulting *a.out* and then dump the KMC and its registers. The following sequence of commands will accomplish this:

```
kasb −d /dev/kmc?
.reset
.load
.run
.reset
.dump
.regs
```

The *.regs* command to *kasb* will produce a register dump similar to the following:

```
csr:    377   0   0   0   0   0   0  20
lur:      0  20   0 101   0 377 377  53
reg:      0 326  42  64   0 276   0  46
reg:    142  73 321  71 156  61 116 356
 io:    377 377 377 377 377 371 377 377
npr:      0  20   0 brg: 356   0 mem:  61
```

If the value of r5 (the sixth number in line three of the register dump) is not 276, something is wrong with the KMC hardware or the software used to load and execute programs in it.

Tset

Tset is a C program that opens a particular *vpm* device (*/dev/vpm0*) and writes a string of characters to it. It then reads the same device and compares the string of characters received to the string sent. If the two strings match, the program prints the string followed by the message "It worked!!!!!." This program will work only when a loop-back script such as *loop.r* has been loaded into the KMC. To run this test:

1. Compile *tset.c*:

   ```
   cc −o tset tset.c
   ```

2. Compile *loop.r*:

   ```
   vpmc −o loop.o loop.r
   ```

3. Load *loop.o* into the KMC:

   ```
   /etc/vpmstart /dev/kmc? 6 loop.o
   ```

4. If testing the VPM event-tracing capability, execute *vpmtrace*:

   ```
   /etc/vpmtrace > t&
   ```

5. Execute *tset*:

   ```
   tset
   ```

6. Print *t*:

   ```
   cat t
   ```

Sr

Sr opens */dev/vpm0* and forks to create a *send* process and a *receive* process. The *send* process reads up to 512 bytes at a time from its standard input and writes them to */dev/vpm0*. The receive process reads */dev/vpm0* and writes the received data to its standard output. This program may be used with the protocol script *loop.r*. The procedure for running *sr* is similar to that used with *tset*. Steps 2, 3, and 4 need not be repeated if the interpreter and *vpmtrace* are still running.

To execute *sr*:

```
sr < infile > outfile
```

The *send* process exits after it has read and transmitted the last data block of the file. The *receive* process goes into a loop that sets an alarm and reads */dev/vpm0*. If the alarm goes off before the *read* completes, the process exits.

Tcmd

Tcmd.c when used with the protocol script *tcmd.r* tests several new features of Release 2.0 of VPM: communications between a user program or a protocol module and the protocol script, reading and resetting the interpreter's error counters, and the time-stamped tracing capability. To execute *tcmd*, follow the procedures given for the first test using *tcmd.c* and *tcmd.r* in place of *tset.c* and *loop.r*. Execute *vpmsnap* instead of or in addition to *vpmtrace*.

Lapb.r

Lapb.r is the protocol script for BX.25 Level 2. To install this script in a particular KMC, proceed as follows:

```
cp /usr/src/cmd/vpm/scripts/lapb.r .
cp /usr/src/cmd/vpm/scripts/const .
cp /usr/src/cmd/vpm/scripts/tconst .
vpmc −mi hdlc −o lapb.o lapb.r
/etc/vpmstart /dev/kmc? 6 lapb.o
```

Testing this script requires two KMCs, which may be on different host computers. The KMCs must be connected by a pair of full-duplex synchronous modems or by a full-duplex synchronous null modem.[2] *Sr* should be executed simultaneously on both machines to read and write the VPM device associated with each KMC. If both KMCs are on the same host machine, it will be necessary to edit and compile a copy of *sr.c* so that it opens */dev/vpm1* instead of */dev/vpm0*. The original and modified versions of *sr* can then be executed simultaneously to exercise the two KMCs.

2. A suitable null modem is the Avanti 300, which is manufactured by Avanti Communications Corporation, Newport, RI.

To obtain maximum efficiency from this script, it may be necessary to modify the values of some of the parameters in the *const* file. The appropriate values for these parameters depend on the link speed and maximum frame size. Guidelines for adjusting these parameters are given in [3].

Lapbt.r

This script is identical to *lapb.r* except for some additional *trace* statements. It may be tested in the same manner as *lapb.r*. *Vpmtrace* may be used to display the trace information.

Itr.r

Itr.r is a simplified version of *lapb.r*. Unlike *lapb.r* and *lapbt.r*, this script can be exercised in a loop-back mode. To run a loop-back test, attach a DEC H-325 test connector to the end of the modem cable for the DMC11-DA line unit that is connected to the KMC11-B to be used for the test. Then compile *itr.r* and load the resulting *a.out* into the KMC using the procedure described above for *lapb.r*, substituting *itr* for *lapb*. A loop-back test can then be run using *tset* or *sr*.

January 1981