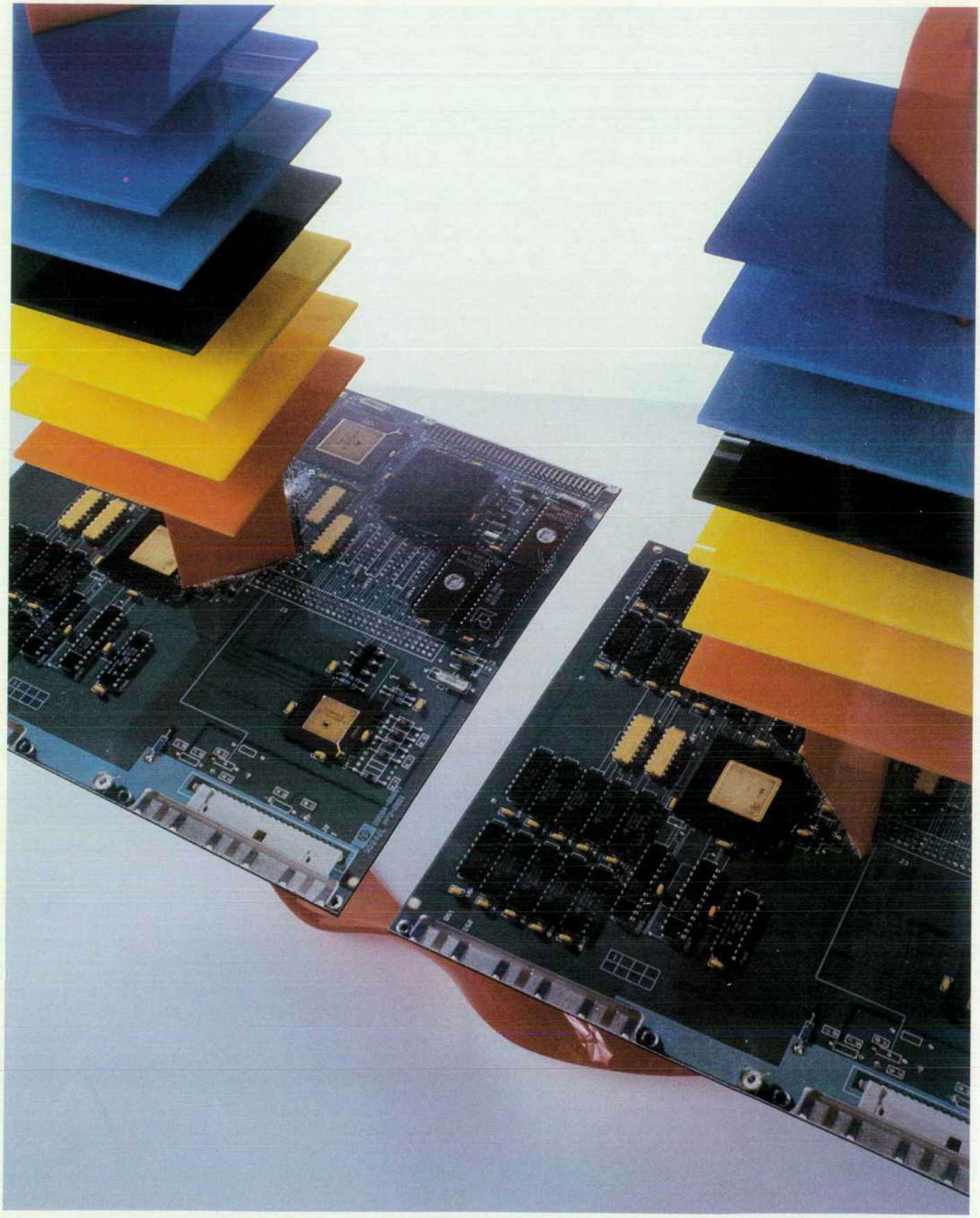
HEWLETT-PACKARD
JOURNAL

FEBRUARY 1990



HEWLETT
PACKARD

HEWLETT-PACKARD JOURNAL

February 1990 Volume 41 • Number 1

Articles

Departments

The **Hewlett-Packard Journal** is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company makes no warranties, express or implied, as to the accuracy or reliability of such information. The Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design, and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

In this Issue

Open systems are systems that communicate with the outside world using standard protocols. Because they communicate using standard protocols, open systems can be interconnected and will work together regardless of what company manufactured them. For a user who is configuring a computer system or network, the benefit of open systems is the freedom to choose the best component for each function from the offerings of all manufacturers. In 1983, the International Organization for Standardization (ISO) published its Open Systems Interconnection (OSI) Reference Model to serve as a master plan for coordinating all open systems activities. The OSI model starts with a framework that organizes all intersystem communications functions into seven layers. Specific protocols perform the functions of each layer. Any organization can have an open system by implementing these standard protocols. The movement towards this model as the global standard for open systems has steadily gained momentum. Hewlett-Packard, along with many other manufacturers and the governments of many countries, is committed to the development of standards and products based on the OSI model.

The HP OSI Express card implements the OSI model for HP 9000 Series 800 computers. The hardware and firmware on the card off-load most of the processing for the seven-layer OSI stack from the host computer. This not only gets the job done faster and improves throughput, but also leaves more time for the host processor to service user applications. Although it's only a single Series 800 I/O card, the HP OSI Express card implements many complex ideas and required a major design effort that claims most of this issue. You'll find an overview of its design on page 6. The interface between the card driver (which is part of the host software) and the operating system on the card is a set of firmware routines called the backplane handler; it's described in the article on page 8. The card's architecture and most of its operating system are determined by an HP concept called the common OSI networking environment, or CONE (see page 18). CONE defines how the protocol firmware modules interact and provides system functions to support the protocol modules. The top three layers of the OSI Express card protocol stack—the application, presentation, and session layer modules—are described in the article on page 28. These three layers share the same architecture and are implemented using tables. In the protocol module for the fourth OSI layer—the transport layer—are the important functions of error detection and recovery, multiplexing, addressing, and flow control, including congestion avoidance (see page 36). The bottom three OSI layers are the network, data link, and physical layers. The bottom of the OSI stack on the OSI Express card is covered in the article on page 45. Because of the number of layers in the OSI stack, data throughput is an important consideration in the design of any OSI implementation. Performance analysis of the HP OSI Express card began in the early design stages and helped identify critical bottlenecks that needed to be eliminated (see page 51). As a result, throughput as high as 600,000 bytes per second has been measured. Troubleshooting in a multivendor environment is also an important concern because of the need to avoid situations in which vendors blame each other's products for a problem. Logging, tracing, and other support features of the OSI Express card are discussed in the article on page 67. Finally, debugging and final testing of the card's firmware are the subjects of the articles on pages 59 and 72, respectively.

Both the use and the performance of fiber optic voice and data communications systems continue to increase and we continue to see new forms of measuring instrumentation adapted to the needs of fiber optic system design, test, and maintenance. The article on page 80 presents the design and applications of the HP 71400A lightwave signal analyzer. This instrument is designed to measure signal strength and distortion, modulation depth and bandwidth, intensity noise, and susceptibility to reflected light of high-performance optical systems and components such as semiconductor lasers and broadband photodetectors. Unlike an optical spectrum analyzer, it does not provide information about the frequency of the carrier. Rather, it acts as a spectrum analyzer for the modulation on a lightwave carrier. It complements the lightwave component analyzer described in our June 1989 issue, which can be thought of as a network analyzer for lightwave components. Using high-frequency photodiodes and a broadband amplifier consisting of four advanced microwave monolithic distributed amplifier stages, the lightwave signal analyzer can measure lightwave modulation up to 22 GHz. (This seems like a huge bandwidth until one realizes that the carrier frequency in a fiber optic system is 200,000 GHz or more!) A companion instrument, the HP 11980A fiber optic interferometer (page 92), can be used with the analyzer to measure the linewidth, chirp, and frequency modulation characteristics of single-frequency lasers. The interferometer acts as a frequency discriminator, converting optical phase or frequency variations into intensity variations, which are then detected by the analyzer. Chirp and frequency modulation measurements with the interferometer use a new measurement technique called a gated self-homodyne technique.

R.P. Dolan
Editor

Cover

This is an artist's rendition of the seven layers of the International Organization for Standardization's OSI Reference Model on the HP OSI Express card, and the communication path between two end systems over a network.

What's Ahead

The April 1990 issue will feature the design of the HP 1050 modular liquid chromatograph and the HP OpenView network management software.

An Overview of the HP OSI Express Card

The OSI Express card provides on an I/O card the networking services defined by the ISO OSI (Open Systems Interconnection) Reference Model, resulting in off-loading much of the network overhead from the host computer. This and other features set the OSI Express card apart from other network implementations in existence today.

by William R. Johnson

THE DAYS WHEN A VENDOR used a proprietary network to "lock in" customer commitment are over. Today, customers demand multivendor network connectivity providing standardized application services. HP's commitment to OSI-based networks provides a path to fill this customer requirement.

The ISO (International Organization for Standardization) OSI (Open Systems Interconnection) Reference Model was developed to facilitate the development of protocol specifications for the implementation of vendor independent networks. HP has been committed to implementation of OSI-based standards since the early 1980s. Now that the standards have become stable, OSI-based products are becoming available. One of HP's contributions to this arena is the OSI Express card for HP 9000 Series 800 computers.

The OSI Express card provides a platform for the protocol stack used by OSI applications. Unlike other networking implementations, the common OSI protocol stack resides on the card. Thus, much of the network overhead is offloaded from the host, leaving CPU bandwidth available for processing user applications. This common protocol stack consists of elements that implement layers 1 through 6 of the OSI Reference Model and the Association Control Service Element (ACSE), which is the protocol for the seventh layer of the OSI stack. Most of the application layer functionality is performed outside the card environment since applications are more intimately tied to specific user functions (e.g., mail service or file systems). An architectural view of the OSI Express card is given in Fig. 1, and Fig. 2 shows the association between the OSI Express stack and the OSI Reference Model.

The series of articles in this issue associated with the OSI Express card provides some insight into how the project team at HP's Roseville Networks Division implemented the card and what sets it apart from many other implementations currently in existence today. This article gives an overview of the topics covered in the other articles and the components shown in Fig. 1.

OSI Express Stack

The protocol layers on the OSI Express card stack provide the following services:

Media Access Control (MAC) Hardware. The MAC hardware is responsible for reading data from the LAN interface into the card buffers as specified by the link/MAC interface software module. All normal data packets destined for a particular node's address are forwarded by the logical link control (LLC) to the network layer.

Network Layer. The network layer on the OSI Express card uses the connectionless network service (CLNS). The OSI Express card's CLNS implementation supports the end-system-to-intermediate-system protocol, which facilitates dynamic network routing capabilities. As new nodes are brought up on the LAN, they announce themselves using this subset of the network protocol. The service provided by CLNS is not reliable and dictates the use of the transport layer to provide a reliable data transfer service. Both the transport layer and CLNS can provide segmentation and reassembly capabilities when warranted.

Transport Layer Class 4. In addition to ensuring a reliable data transfer service, the OSI Express transport is also responsible for monitoring card congestion and providing flow control.

Session Layer. The OSI Express card's implementation of the session layer protocol facilitates the management of an application's dialogue by passing parameters, policing state transitions, and providing an extensive set of service primitives for applications.

Presentation Layer and Association Control Service Element (ASCE). The OSI Express card's presentation layer extracts protocol parameters and negotiates the syntax rules for transmitting information across the current association. Both ACSE and the presentation layer use a flexible method of protocol encoding called ASN.1 (Abstract Syntax Notation One). ASN.1 allows arbitrary encodings of the protocol header, posing special challenges to the decoder. ASCE is used in the transmission of parameters used in the establishment and release of the association.

OSI Express Protocols

The OSI protocols are implemented within the common OSI networking environment (CONE). CONE is basically a network-specific operating system for the OSI Express card. The utilities provided by CONE include buffer management, timer management, connection management, queue management, nodal management, and network management. CONE defines a standard protocol interface that provides module isolation. This feature ensures portability of networking software across various hardware and/or software platforms. The basic operating system used in the OSI Express card is not part of CONE and consists of a simple interrupt system and a rudimentary memory manager.

Service requests are transmitted to CONE using the backplane message interface (BMI) protocol. Application requests are communicated through the OSI user interface to the CONE interface adapter (CIA). The interface adapter bundles the request into the BMI format and hands it off to the system driver. The BMI converts message-based requests, which are asynchronous, from the interface adapter into corresponding CONE procedure calls, which are synchronous.

The backplane handler controls the hardware that moves messages between the host computer and the OSI Express card. A special chip, called the HP Precision bus interface chip, is used by the backplane handler to gain control of the HP Precision bus and perform DMA between the OSI Express card and the host memory space. Another special chip, called the midplane memory controller, is used by the backplane handler to take care of OSI Express card midplane bus arbitration and card-resident memory. The backplane handler conceals the interactions of these two chips from CONE and the driver.

Diagnostics and Maintenance

The OSI Express card uses three utilities to aid in fault detection and isolation. The hardware diagnostics and maintenance program uses the ROM-resident code on the card to perform initial configuration of the MAC hardware. After configuration, the program is used to access the ROM-based test code that exercises both local and remote networking hardware. The same utility is also used to download the OSI Express card software into RAM. The host-based network and nodal management tool contains the tracing (event recording) and logging (error reporting) facilities. The network and nodal management tool can be used to report network management events and statistics as well. However, it is primarily used to resolve protocol networking problems causing abnormal application behavior (e.g., receipt of nonconforming protocol header information). The software diagnostic program, which is the third fault detection program on the OSI Express card, was developed to aid in the identification of defects encountered during the software development of the card. This program uses the software diagnostic module on the card to read and write data, set software breakpoints, and indicate when breakpoints have been encountered. The interface to the software diagnostic program provides access to data structures and breakpoint locations through the use of symbolic names. It also searches CONE-defined data structures with little user intervention.

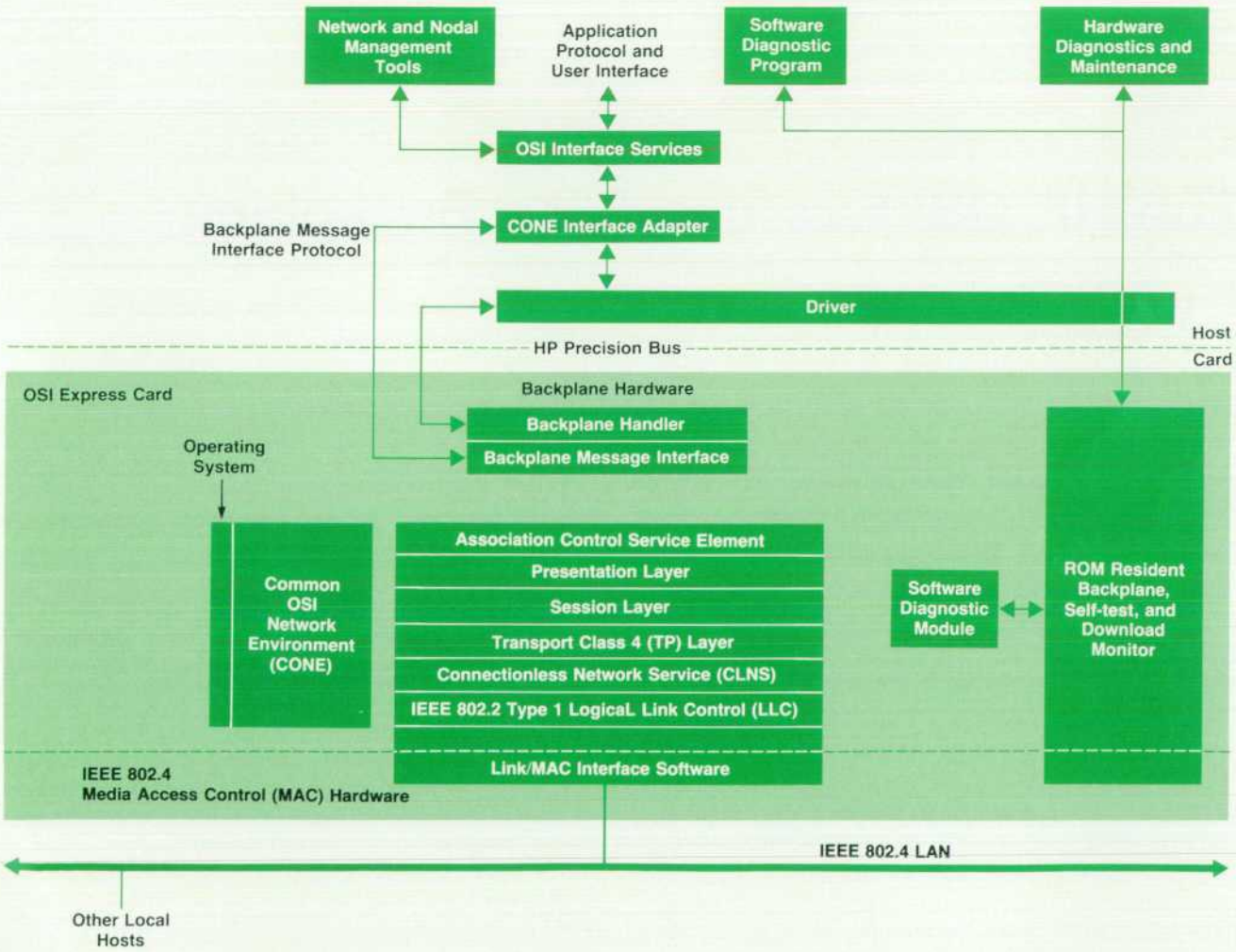


Fig. 1. *HP OSI Express card overview.*

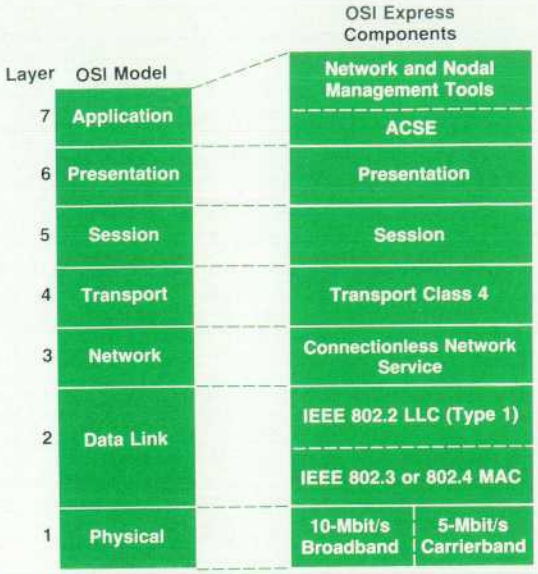


Fig. 2. *Comparison between the OSI Express components and the OSI Reference Model.*

In a multivendor environment it is crucial that networking problems be readily diagnosable. The OSI Express diagnostics provide ample data (headers and state information) to resolve the problem at hand quickly.

An implementation of the OSI protocols is not inherently doomed to poor performance. In fact, file transfer throughput using the OSI Express card in some cases is similar to that of existing networking products based on the TCP/IP protocol stack. Performance is important to HP's customers, and special attention to performance was an integral part of the development of the OSI Express card. Special focus on critical code paths for the OSI Express card resulted in throughputs in excess of 600,000 bytes per second. Intelligent use of card memory and creative congestion control allow the card to support up to 100 open connections.

Acknowledgments

Much credit needs to be given to our section manager Doug Boliere and his staff—Todd Alleckson, Diana Bare, Mary Ryan, Lloyd Serra, Randi Swisley, and Gary Wermuth—for putting this effort together and following it through. Gratitude goes to the hardware engineers who gave the networking software a home, as well as those who developed the host software at Information Networks Division and Colorado Networks Division.

The HP OSI Express Card Backplane Handler

The backplane on the HP OSI Express card is handled by a pair of VLSI chips and a set of firmware routines. These components provide the interface between the HP OSI Express card driver on the host machine and the common OSI networking environment, or CONE, on the OSI Express card.

by Glenn F. Talbott

THE HP OSI EXPRESS CARD BACKPLANE handler is a set of firmware routines that provide an interface between the common OSI networking environment (CONE) software and the host-resident driver. CONE provides network-specific operating system functions and other facilities for the OSI Express card (see the article on page 18). The handler accomplishes its tasks by controlling the hardware that moves messages between the host computer and the OSI Express card. The backplane handler design is compatible with the I/O architecture defined for HP Precision Architecture systems,[1] and it makes use of the features of this architecture to provide the communication paths between CONE and the host-resident driver (see Fig 1). The HP Precision I/O Architecture defines the types of modules that can be connected to an HP Precision bus (including processors, memory, and I/O). The OSI Express card is classified as an I/O module.

The OSI Express card connects to an HP 9000 Series 800 system via the HP Precision bus (HP-PB), which is a 32-bit-wide high-performance memory and I/O bus. The HP-PB allows all modules connected to it to be either masters or slaves in bus transactions. Bus transactions are initiated

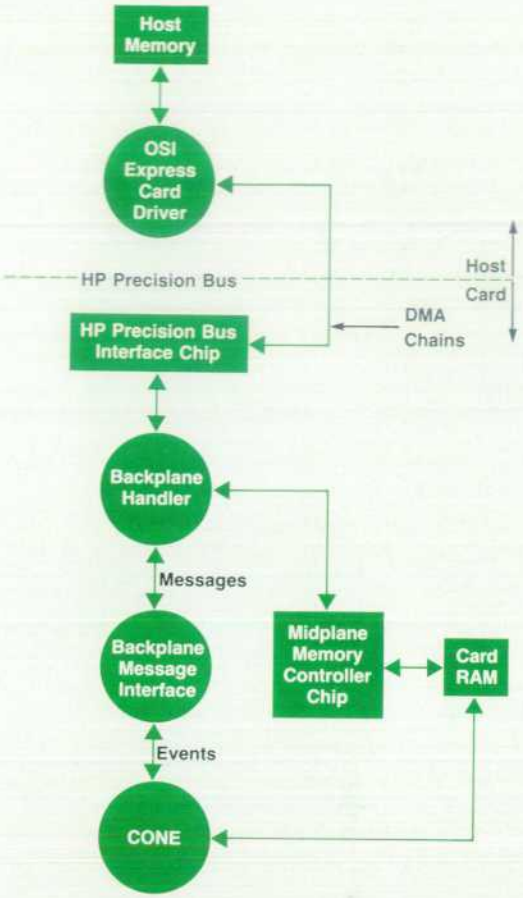


Fig. 1. *The data flow relationships between the OSI Express card driver on the host computer and the major hardware and software components on the card.*

by a master and responses are invoked from one or more slaves. For a read transaction, data is transferred from the slave to the master, and for a write transaction, data is transferred from the master to the slave. Each module that can act as a master in bus transactions is capable of being a DMA controller. Bus transactions include reading or writing 4, 16, or 32 bytes and atomically reading and clearing 16 bytes for semaphore operations.

The OSI Express card uses a pair of custom VLSI chips to perform DMA between its own resident memory and the host memory. The first chip is the HP-PB interface chip, which acts as the master in the appropriate HP Precision bus transactions to perform DMA between the OSI Express card and the host system memory space. The second chip is the midplane memory controller, which controls the DMA between the HP-PB interface chip and the OSI Express card resident memory. The memory controller chip also performs midplane bus arbitration and functions as a dynamic RAM memory controller and an interrupt controller. See the box on page 15 for more information about the HP-IB interface chip and the midplane memory controller chip. The backplane handler hides all the programming required for these chips from the host computer OSI Express driver and CONE.

Host Interface

The HP Precision I/O Architecture views an I/O module as a continuously addressable portion of the overall HP Precision Architecture address space. I/O modules are assigned starting addresses and sizes in this space at system initialization time. The HP Precision I/O Architecture further divides this I/O module address space (called soft physical address space, or SPA) into uniform, independent register sets consisting of 16 32-bit registers each.

The OSI Express backplane handler is designed to support up to 2048 of these register sets. (The HP-PB interface chip maps HP-PB accesses to these register sets into the Express card's resident memory.) With one exception, for the backplane handler each register set is independent of all the other register sets, and the register sets are organized in inbound-outbound pairs to form full-duplex paths or connections. The one register set that is the exception (RS 1) is used to notify the host system driver of asynchronous events on the OSI Express card, and the driver is always expected to keep a read transaction pending on this register because it is set to receive notification of these events.

Register Sets

The registers are numbered zero through 15 within a given register set. The registers within each set that are used by the backplane handler as they are defined by the HP Precision I/O Architecture are listed below. The registers not included in the list are used by the backplane handler to maintain internal state information about the register set.

Number	Name	Function
4	IO_DMA_LINK	Pointer to DMA control structure
5	IO_DMA_COMMAND	Current DMA chain command
6	IO_DMA_ADDRESS	DMA buffer address
7	IO_DMA_COUNT	DMA buffer size (bytes)
12	IO_COMMAND	Register set I/O command
13	IO_STATUS	Register set status

The OSI Express card functions as a DMA controller and uses DMA chaining to transfer data to and from the card. DMA chaining consists of the DMA controller's autonomously following chains of DMA commands written in memory by the host processor. HP Precision I/O Architecture defines DMA chaining methods and commands for HP Precision Architecture systems. A DMA chain consists of a linked list of DMA control structures known as quads. Fig. 2 shows a portion of a DMA chain and the names of the entries in each quad.

Data Quad. The data quad is used to maintain reference to and information about the data that is being transferred. The fields in the data quad have the following meaning and use.

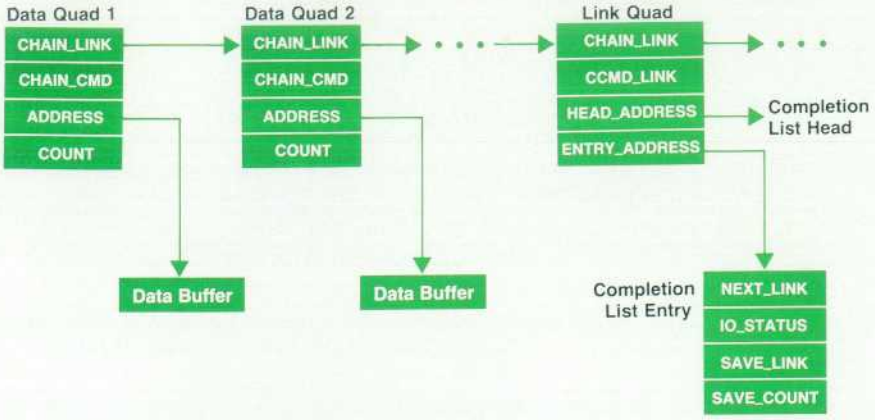


Fig. 2. *A portion of a DMA chain.*

Field	Meaning and Use
CHAIN_LINK	Pointer to the next quad in the chain
CHAIN_CMD	DMA chaining command plus application-specific fields
ADDRESS	Memory address of the data buffer
COUNT	Length of the data buffer in bytes

Bits in the application-specific fields of CHAIN_CMD control the generation of asynchronous events to CONE and the acknowledgment of asynchronous event indications to the host.

Link Quad. A link quad is created by the driver to indicate the end of a DMA transaction (note: not the end of a DMA chain). When a link quad is encountered in the chain, a completion list is filled in and linked into a completion list. If the CHAIN_LINK field does not contain an END_OF_CHAIN, DMA transfers continue. The fields in the link quad have the following meaning and use.

Field	Meaning and Use
CHAIN_LINK	Pointer to the next quad in the chain, or END_OF_CHAIN value
CCMD_LINK	Causes a completion list entry to be created and may specify whether the host should be interrupted
HEAD_ADDR	Address of the completion list
ENTRY_ADDR	Address of the completion list entry to be used to report completion status

Completion List Entry. A completion list entry is used to indicate the completion status of a DMA transaction. One is filled in when a link quad is encountered in the DMA chain. The fields in the completion list have the following meanings and use:

Field	Meaning and Use
NEXT_LINK	Pointer used to link the entry into the completion list
IO_STATUS	Completion status field, a copy of the IO_STATUS register
SAVE_LINK	Pointer to the quad where an error occurred, or to the link quad in the case of no error
SAVE_COUNT	Residue count of bytes remaining in the buffer associated with the quad pointed to by SAVE_LINK, or zero if no error

The completion list head contains a semaphore that allows a completion list to be shared by multiple I/O modules, and a pointer to the first entry in the completion list.

DMA Chaining

DMA chaining is started by the host system driver when the address of the first quad in a DMA chain is written into the IO_DMA_LINK register of a register set. To tell the OSI Express card to start chaining, the driver writes the chain command CMD_CHAIN into the register set's IO_COMMAND register. This causes an interrupt and the Express card's backplane handler is entered. From this point until a completion list entry is made, the DMA chain belongs to the OSI Express card's register set, and DMA chaining is under control of the backplane handler through the register set. Fig. 3 shows the flow of activities for DMA chaining in the driver and in the backplane handler.

Once control is transferred to the backplane handler, the first thing the handler does is queue the register set for service. When the register set reaches the head of the queue, the backplane handler fetches the quad pointed to by IO_DMA_LINK and copies the quad into the registers IO_DMA_LINK, IO_DMA_COMMAND, IO_DMA_ADDRESS, and IO_DMA_COUNT. The backplane handler then interprets the chain command in register IO_DMA_COMMAND, executes the indicated DMA operation, and fetches and copies the next quad pointed to by IO_DMA_LINK. This fetch, interpret, and execute process is repeated until the value in IO_DMA_LINK is END_OF_CHAIN. When END_OF_CHAIN is reached, the backplane handler indicates that the register set is ready for a new I/O command by posting the status of the DMA transaction in the IO_STATUS register.

The DMA operation executed by the backplane handler is determined by the chain command in the IO_DMA_COMMAND register. For quads associated with data buffers, this chain command is CCMD_IN or CCMD_OUT for inbound or outbound buffers, respectively. In this case the backplane handler transfers the number of bytes of data specified in the IO_DMA_COUNT register to or from the buffer at the host memory location in the IO_DMA_ADDRESS register. The IO_DMA_ADDRESS and IO_DMA_

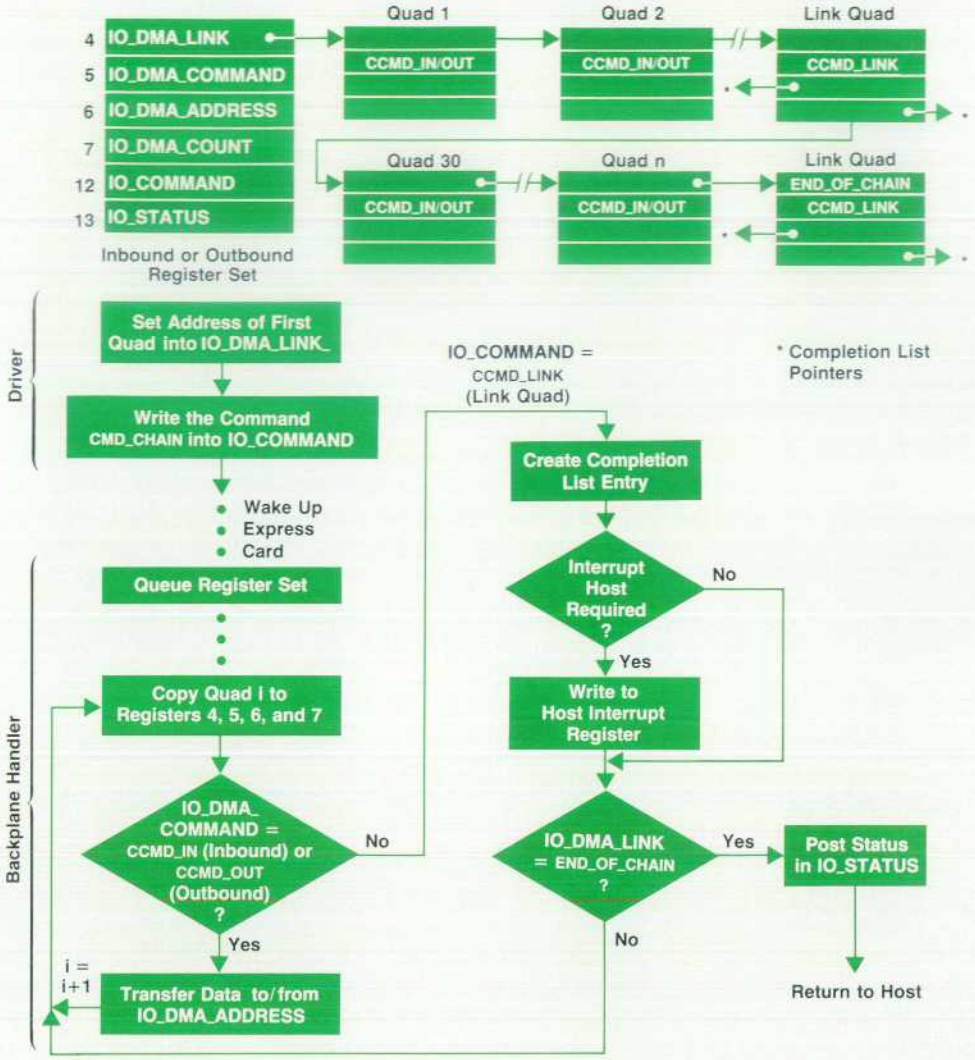


Fig. 3. *Flow of activities involved in a DMA chaining operation.*

COUNT registers are incremented and decremented as the data is transferred.

The link quads containing a CCMD_LINK chain command cause the backplane handler to report the status of the previous DMA transfers and continue chaining if the register containing the CHAIN_LINK field does not indicate END_OF_CHAIN. The CCMD_LINK can also cause the backplane handler to generate an interrupt to the host processor which indicates to the driver that a completion list entry is ready to be read.

Completion List Entry

When a link quad containing the CCMD_LINK chain command is encountered, a completion list entry is created. Creating a completion list is a three-or-four-step process. First, the backplane handler acquires the semaphore in the completion list head at the address in HEAD_ADDR (see Fig. 4a). This is accomplished by repeatedly mastering (gaining control of the bus) a read-and-clear bus transaction until a nonzero value is returned. When a nonzero value has been read, the OSI Express card owns the semaphore and can proceed to the next step. The second step is to fill the four fields of the completion list entry indicated by the pointer ENTRY_ADDR in the link quad. The third step is to write a nonzero value into the semaphore field of the completion list head, thus releasing the semaphore, and insert the new completion list entry into the completion list (see Fig. 4b). These three steps are done automatically by the HP Precision bus interface chip on command from the backplane handler.

The optional fourth step of the completion list insertion process is to generate an interrupt to the host processor. If the CCMD_LINK specifies, the address of the host processor and the value written in the processor's external interrupt register are packed into the chain command word containing the CCMD_LINK. The backplane handler uses these values to master a write to the host processor and cause an interrupt.

When the OSI Express driver has built a DMA chain and started the OSI Express card traversing the chain, subsequent DMA chains can be appended to the existing chain without interrupting the card. To do this the driver simply writes the address of the first quad in the new chain into the CHAIN_LINK word of the last quad of the old chain. Since the driver does not know whether an append is successful (the card may have already fetched the last quad in the old chain), there is a mechanism to verify the success of an

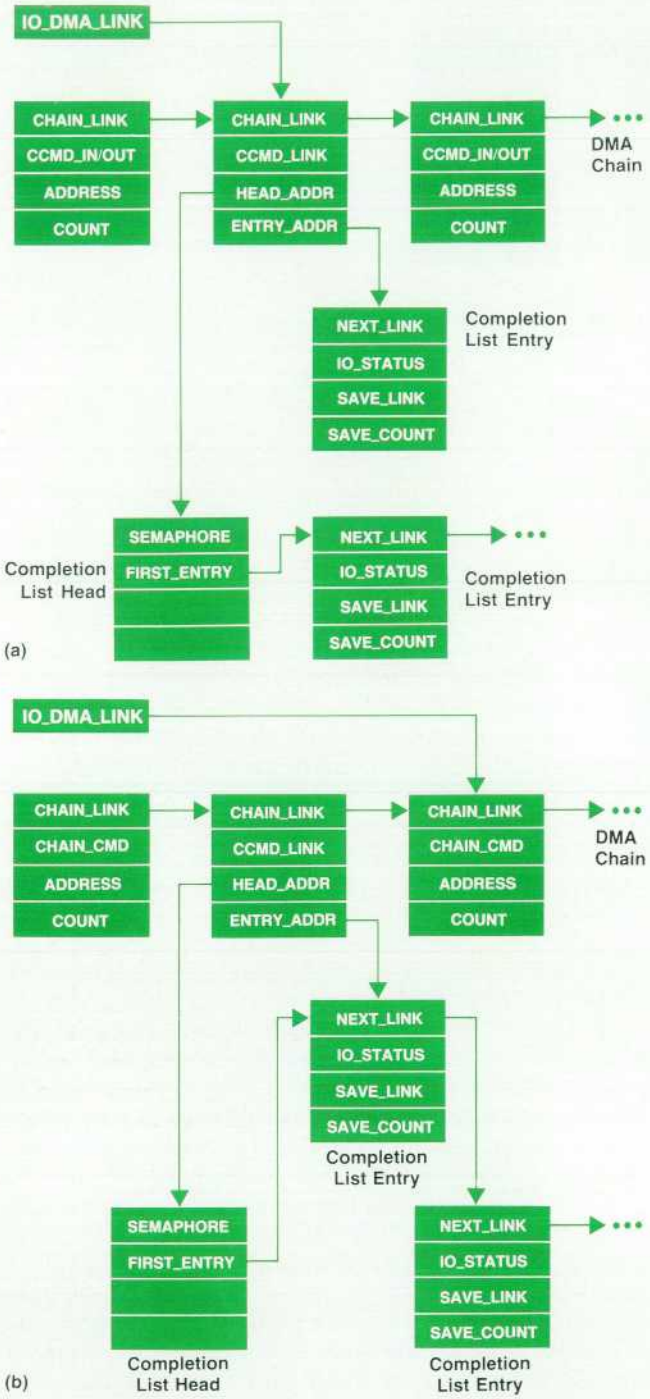


Fig. 4. *(a) Completion list before executing a* CCMD_LINK *chain command. (b) Completion list after executing a* CCMD_LINK *chain command.*

append. When the driver reads the completion list entry for the old chain, a bit in the IO_STATUS word indicates whether or not the OSI Express card found END_OF_CHAIN in the last quad. If this bit is set (END_OF_CHAIN found) the append is not successful and the driver must start the new chain by writing the address of the first quad of the new chain to the register set's IO_DMA_LINK register and a CMD_CHAIN to the IO_COMMAND register. Using the append mechanism, the OSI Express card can run more efficiently when the driver can stay ahead of the card in posting DMA chains. This way the driver only starts one chain (generating an interrupt on the Express card) on each register set being used.

Procedure Call Interface

Data transfers between the host computer and the OSI Express card are via DMA. DMA chains containing data and control information are created by the host driver, and the backplane handler uses the HP-PB register sets to transfer the data to and from the OSI Express card. On the OSI Express card the data is moved to and from the protocol layers. Access to the protocol layers is provided by the common OSI network environment, or CONE, and access to CONE is through the backplane message interface (BMI). Fig. 1 shows the main elements of this of this hierarchy, except the protocol layers. The backplane message interface is responsible for converting backplane message (asynchronous) requests into corresponding CONE (synchronous) procedure calls for outbound data transfers, and converting CONE procedure calls into backplane message requests for inbound data transfers. The reasons for this particular interface design are discussed in more detail on page 27.

Handshake Procedures

The backplane handler interface to CONE uses a set of procedures, which are written in C, to transfer messages to and from CONE. CONE makes initialization and data movement request calls to the backplane handler, and the backplane handler makes completion and asynchronous event procedure calls to CONE. The data movement requests are made by CONE executing at a normal interrupt level. The completion and event calls are made by the backplane handler at the backplane handler interrupt level (level three) to CONE. These completion and event procedures set flags for processing later by CONE at a normal interrupt level. The completion and event procedures are located in the backplane message interface module. Pointers to these routines are passed to the backplane handler at initialization time for each register set. Although these procedures are located in the BMI, CONE is responsible for initiation, interpretation, and action for messages to and from the backplane handler, and the BMI is the interprocess communication handler.

Initialization and Data Movement Procedures. These procedures, which are located in the backplane handler, are used by CONE to send messages to the backplane handler.

- BH_assoc_rs(). This procedure is used by CONE to enable an inbound and outbound register pair when a network connection is established. It is also used to disable the register pair when the connection is broken. The parameters passed when this procedure is called include:
  - The register set number.
  - An identifier that is meaningful to CONE and is used to identify subsequent asynchronous events.
  - The priority to be used in servicing the register set.
  - Pointers to the three completion and event procedures for this register set.
  - A pointer to a block of memory to be used by the back-

plane handler to queue asynchronous events.
- A pointer to a block of memory to be used by the backplane handler to copy event parameters from the host computer.
- The length of the event parameter memory block.

- BH_put_data() and BH_get_data(). These routines are used to start a data transfer request—BH_put_data for inbound transfers and BH_get_data for outbound transfers. They are also instrumental in determining the state transitions in the backplane handler's main interrupt service routine. The parameters passed when these procedures are called include:
  - The register set number.
  - An identifier that is returned with the BHI_put_data_done() or BHI_get_data_done() call to identify this particular request.
  - A pointer to a block of memory to be used by the backplane handler to queue this request. The block of memory ensures that the queue depth of requests held by the backplane handler is not limited by the resources directly available to the backplane handler.
  - A pointer to a structure of chained data buffers to be sent or filled. This structure is matched to the structures created by the CONE memory manager.
  - The total number of bytes requested for the transfer.
  - A status value passed to the host computer in the completion list entry.
  - A bit-field mode parameter that controls various aspects of the transfer, such as whether errors and acknowledgments of previous asynchronous events should be sent to the host computer.

Completion and Event Procedures. These procedures, which are located in the backplane message interface module, are used by the backplane handler to send messages to CONE.

- BHI_cmd_arrival(). This procedure is used to announce asynchronous events to CONE. There are two asynchronous events that cause BHI_cmd_arrival() to be called by the backplane handler. The first event is the posting of outbound data to a register set by the host driver. The first quad in the DMA chain associated with the register set has its transparent bit set and the quad's data buffer is set to contain information about how much outbound data is being sent. The transparent bit causes a call to BHI_cmd_arrival(), passing the buffer attached to the first quad. The second case in which BHI_cmd_arrival() is called is the resetting of a register set by the driver. CONE must acknowledge the receipt of a BHI_cmd_arrival() call with a BH_get_data() call. The backplane handler's internal logic prevents more than one BHI_cmd_arrival() per register set from being outstanding at any time. The parameters passed in a call to BHI_ cmd_arrival include:
  - The register set of the event.
  - An identifier that is meaningful to CONE (established with BH_assoc_rs()).
  - A code indicating the type of event.
  - The length of data in an event parameter block.
- BHI_put_data_done() and BHI_get_data_done(). These procedures are used to announce the completion and freeing of resources from prior data movement requests. The parameters passed with these procedures include:
  - An identifier that is meaningful to CONE (established by the BH_put_data() or BH__get_data() request).
  - A count of the number of bytes moved to or from the host computer.
  - A status value passed from the host computer.
  - An error value to indicate backplane handler errors.

Inbound and Outbound Requests

Fig. 5 illustrates how these routines are used to perform the handshakes for data transfers between the backplane handler and CONE. CONE starts off by calling BH_assoc_rs() to enable an inbound and outbound register set pair when a connection is established.

Outbound Requests. When BHI_cmd_arrival() is called to inform CONE that the host computer has posted outbound data to an outbound register set, CONE allocates the required buffer space and calls BH_get_data(), specifying an acknowledgment of the BHI_cmd_arrival() call. When the data has been transferred across the backplane, BHI_get_data_done() is called, triggering CONE to send the outbound data across the network.

Inbound Requests. When CONE receives inbound data, it calls BH_put_data() to send the data across the backplane, specifying that an asynchronous event must be sent to the host and giving the size of the data. After the host computer receives the asynchronous event, it posts reads to accept the data. After the data has been transferred, the backplane handler calls BHI_put_data_done(), triggering CONE to release the buffers used by the inbound data so they can be used to receive more data.

The send and receive data sequences are repeated as often as necessary to move data across the backplane. Note that as long as CONE has free buffers available, CONE does not have to wait for a preceding BHI_get_data_done() to allocate the next outbound buffer and call BH_get_data(). Also, as long as free buffers are available, CONE can receive data from the network and call BH_put_data() without waiting for the preceding BHI_put_data_done() calls to indicate that the host has taken previous data. When the connection is cut,

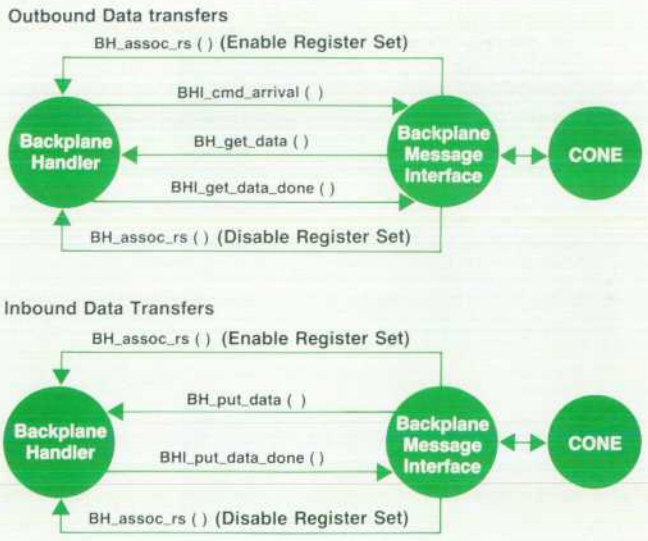


Fig. 5. *Handshake sequences between the backplane handler and CONE (via the backplane message interface).*

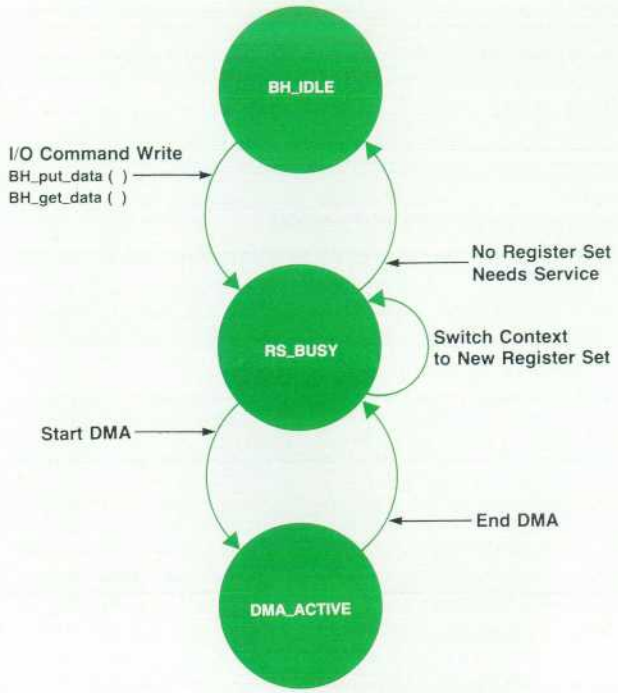


Fig. 6. *The backplane handler state diagram.*

CONE calls BH_assoc_rs() to disable the register sets used by the connection.

The Backplane Handler

The simplified state diagram shown in Fig. 6 shows the behavior of the backplane handler to inputs from the OSI Express card driver on the host computer and from CONE through the backplane message interface.

In the BH_IDLE state the backplane handler is typically not executing because the OSI Express card processor is either executing in the CONE protocol stack, or the processor is in an idle loop itself. There are two ways to get out of BH_IDLE. Either a new I/O command is written by the host driver into a register set's IO_COMMAND register causing an interrupt, or the backplane handler's main interrupt service routine is called from CONE via BH_put_data() or BH_get_data() to process a new request. In either case at least one register set will be queued for service, and the backplane handler will find the queued register set, switch context to that register set, and enter the RS_BUSY state.

In the RS_BUSY state the backplane handler does all the processing required to service one register set, moving the register set through the various register set states. If a long DMA transfer is started and the backplane handler must exit to await DMA completion, the backplane handler will enter the DMA_ACTIVE state. DMA_ACTIVE is a transitory state that ends when the DMA completes and the backplane handler returns to the RS_BUSY state. When one register set can progress no further through the register set states, the backplane handler switches to the next queued register set. When there are no more register sets, the backplane handler returns to the BH_IDLE state.

(continued on page 16)

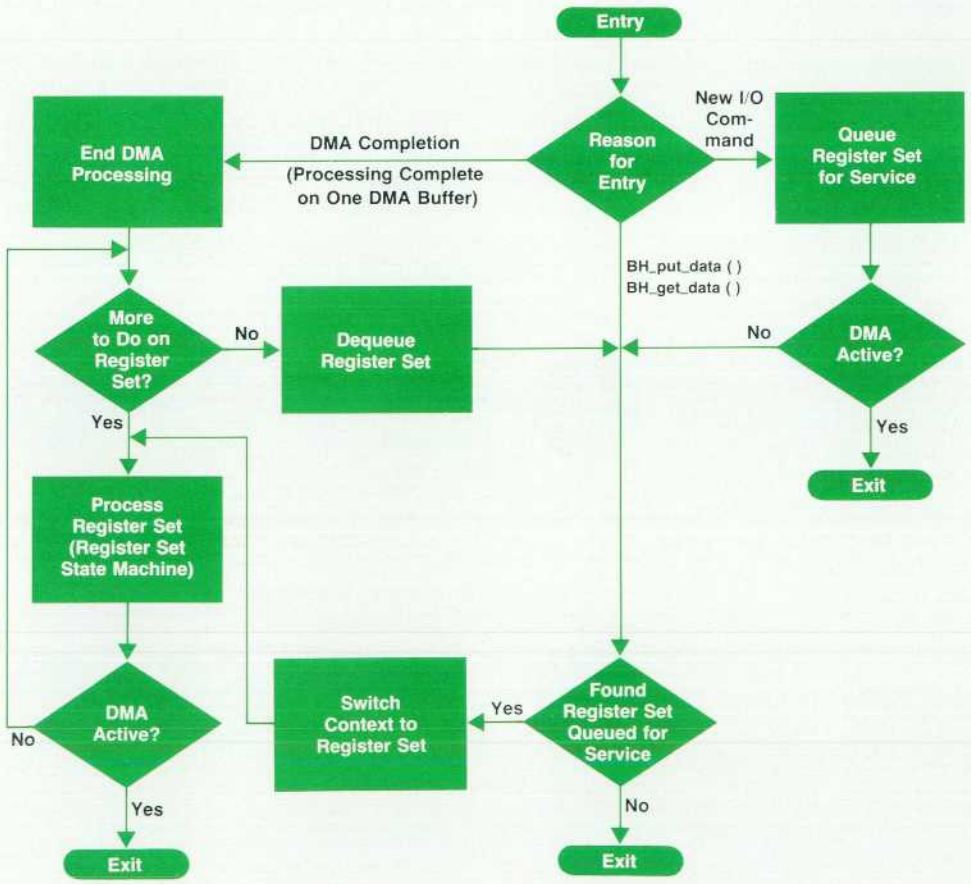


Fig. 7. *Flowchart for the backplane handler's main interrupt service routine.*

Custom VLSI chips for DMA

The OSI Express card uses a pair of custom VLSI circuits to perform DMA between the OSI Express card resident memory and the host system's memory. The first chip is the Hewlett-Packard Precision bus interface chip and the other is the midplane memory controller chip. The bus interface chip masters the appropriate HP Precision bus transactions to perform DMA between the OSI Express card and the host system memory space. The memory controller chip is responsible for controlling DMA between the bus interface chip and the OSI Express card resident memory, performing midplane bus arbitration, and functioning as a dynamic RAM memory controller and an interrupt controller.

The bus interface chip functions as a bus master when doing DMA on the HP Precision bus and as a bus slave when responding to direct I/O to and from the OSI Express card registers by the host processor. The memory controller chip serves as a DMA controller when the bus interface chip is doing DMA, performing DMA to or from card memory when the bus interface chip asserts a DMA request (DMAR). The memory controller chip also serves as a bus arbitrator when the bus interface chip responds to direct I/O from the host computer, granting the bus interface chip the bus when it asserts a bus request (BUSRQ).

Both chips are connected to a 68020 processor, dynamic RAM, and address and data buses as shown in Fig. 1. All RAM addresses on the address bus are translated by the memory controller chip into addresses that map into the physical RAM space.

DMA between the host system and the OSI Express card is a complex process, considering that:

- All HP Precision bus DMA data transfers are either 16 or 32 bytes and must be size-aligned.
- DMA bus transfers on the OSI Express card bus are 16 bits, and a one-byte shift is required if even-addressed OSI Express card bytes are transferred to odd-addressed host bytes.
- DMA transfers on the HP Precision bus side can be specified to start or end on arbitrary byte boundaries, with garbage data used to pad to 16-byte alignment and size.
- DMA transfers on the OSI Express card memory side can be specified to start or end on arbitrary byte boundaries with no extra data allowed.

The bus interface chip and the memory controller chip combine to make the task of doing DMA between OSI Express card memory and host memory almost as simple as programming addresses and counts. Fig. 2 shows some of the basic elements on both chips. The figure is drawn showing DMA from the OSI Express card to the host computer. To go the other way, reverse the direction of the data flow arrows.

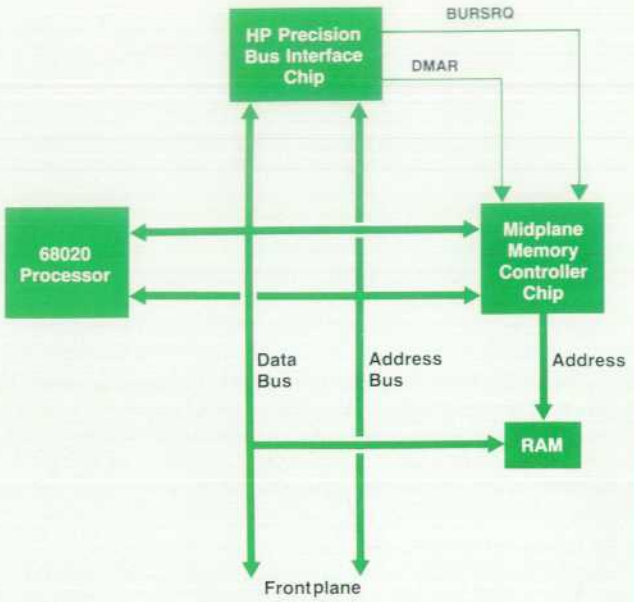


Fig. 1. *OSI express card data and address buses.*

The bus interface chip uses a pair of 32-byte swing buffers so that an HP-PB transaction can proceed in parallel with an OSI Express card midplane transaction. The bus interface chip PDMA_ADDRESS register is a pointer into host memory. It is initialized to the size-aligned boundary below the desired starting address and is incremented by the size of the transactions (16 or 32 bytes).

The bus interface chip N_COUNT and M_COUNT registers

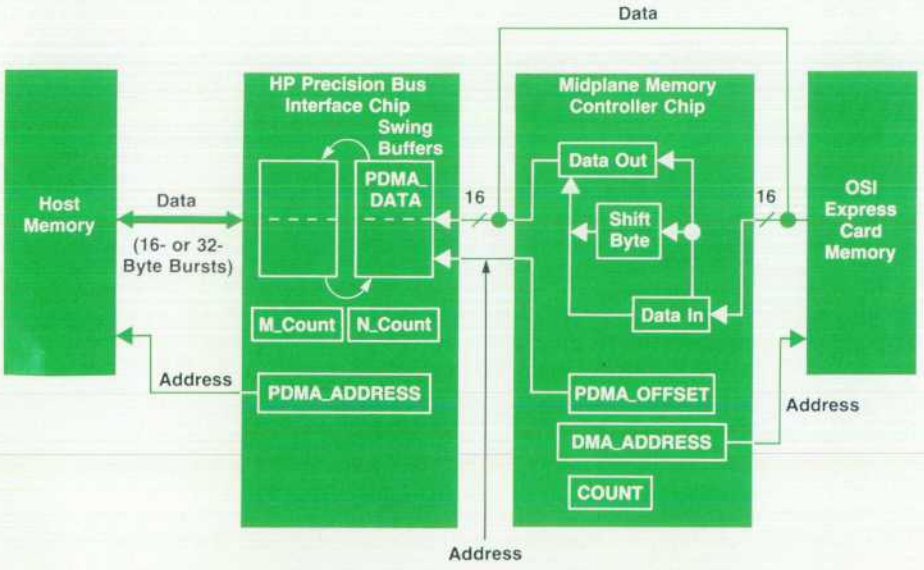


Fig. 2. *Basic elements of the HP Precision Bus interface chip and the midplane memory controller chip.*

count down as the DMA transfer progresses on the HP Precision bus side (N_COUNT) and the OSI Express card midplane side (M_COUNT). N_COUNT is decremented by the HP Precision bus transaction size (16 or 32 bytes) and M_COUNT is decremented by the midplane transaction size (2 bytes). Both registers are normally initialized to the desired size of the transfer. However, if the transfer is from the host system to the OSI Express card and the starting host address is not 16 (or 32) byte aligned, the amount of misalignment is added to N_COUNT to cause that number of bytes to be read and discarded. The bus interface chip will assert DMAR as long as both M_COUNT and N_COUNT are greater than zero and the swing buffer on the OSI Express card midplane side is not full (or not empty for host-to-OSI Express card transfers).

The memory controller chip has the task of aligning misaligned host computer and OSI Express card data. If data on the host computer starts on an odd byte and the OSI Express card data starts on an even byte, or vice versa, the data is passed through the memory controller chip using the shift byte register to provide the one-byte shift required for all data transfers between the OSI Express card memory and the bus interface chip. If the starting addresses match (odd - odd or even - even) then DMA data is transferred directly between the bus interface chip and the OSI Express card memory without passing through the memory controller chip. There is a two-clock-cycle penalty for each 16 bits transferred when byte shifting DMA data.

The memory controller chip DMA_ADDRESS register, which sources the OSI Express card memory address, is initialized to the starting address of the transfer and is incremented by two bytes as the data is transferred (one byte for first or last byte as required by misalignment and length). The COUNT register is initialized to the number of bytes required and is decremented as the DMA_ADDRESS register is incremented. The PDMA_OFFSET register is a five-bit rollover counter that is used to provide addressing into the bus interface chip swing buffers. PDMA_OFFSET is masked to four bits when 16-byte HP Precision bus transactions are being used so that it counts from 0 to 15 and rolls to zero. PDMA_OFFSET is initialized to an offset value depending on the size alignment of the desired host starting address (zero for size-aligned transfers). The memory controller chip will drive the DMA as long as the bus interface chip asserts DMAR and the memory controller chip COUNT register is greater than zero.

(continued from page 14)

Main Interrupt Service Routine

The backplane handler's main interrupt service routine is the component of the backplane handler that drives the backplane handler state machine. A flowchart of the backplane handler main interrupt service routine is shown in figure Fig. 7.

On entry to the main interrupt service routine, a three way decision is made based on the reason for entry.

- If the entry is from a call by BH_ put_data() or BH_get_data() the routine searches for a queued register set to service.
- If the entry is from a new command written to a register set, the register set is queued for service, and if the backplane handler state is DMA_ACTIVE, an exit is taken. Otherwise the interrupt service routine searches for a queued register set to service.
- If the entry is from a DMA completion, the backplane handler ends DMA processing and enters a loop for processing one register set. This loop consists of a test to see if there is further action that can be taken on the register set, register set processing (which drives the register set state machine) if the test is successful, and a test for DMA_ACTIVE. If the first test fails and there is nothing further that can be done on the current register set, that register set is removed from the queue of register sets requesting service and the interrupt service routine searches for a queued register set to service. If the second test shows that DMA is active, an immediate exit is taken. Note that there are no context switches to another register set before a particular register set being serviced reaches DMA completion. This is because on new command entries, if the backplane handler state is DMA_ACTIVE an exit is taken with no context switch. Also, BH_ put_data() and BH_get_data() will queue a register set for service but not call the main interrupt service routine if the backplane handler state is DMA_ACTIVE.

All paths through the main interrupt service routine that do not exit with DMA_ACTIVE eventually wind up searching for another queued register set to service. Register sets are queued for service in multiple priority queues. Each priority queue is serviced in a first in, first out fashion before stepping to the next-lower-priority queue. (Register set priorities are established at initialization.) When a register set is found requesting service, a context switch is made to that register set and the loop that processes register sets is entered. When there are no more register sets requesting service the main interrupt service routine exits.

Register Set State Machine

The backplane handler sends and receives multiple streams of data on register sets and maintains those register sets as independent state machines. Each register set is an instance of a register set state machine. Register set state changes are driven by the process register set block in the main interrupt service routine. A simplified register set state diagram is shown in Fig. 8.

A register set leaves the RS_IDLE state either when a new request is started (BH_put_data() or BH_get_data() queue a request and then queue the register set for service) or when a host data buffer becomes available (host driver posts a DMA chain, and a normal data quad is fetched). If a new request is started, the register set transitions to the REQ_PEND state. If a new host buffer becomes available the register set transitions to the DATA_PEND state. The register set may stay in either REQ_PEND or DATA_PEND for a long time waiting for driver action, resources to free up, or network data to be received to cause the transitions to REQ_DATA_PEND.

Once in the REQ_DATA_PEND state, DMA data will flow through a register set until either the end of the host data is encountered or the end of the local request data is encountered, or both. When one of these events is encountered, the register set will transition back to the appropriate REQ_PEND, DATA_PEND, or RS_IDLE state.

The ability of a register set to go between either the REQ_PEND or DATA_PEND state and the REQ_DATA_PEND state repeatedly allows the OSI Express card to use the backplane

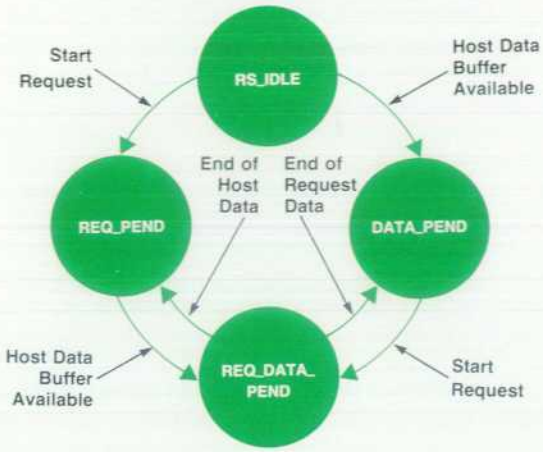


Fig. 8. *Register set state diagram.*

handler as a packet segmentation or reassembly point. When networking buffer memory on the OSI Express card is scarce and a large buffer of outbound data is posted by the driver, CONE can allocate one small buffer to send the data. The one buffer can be used over and over again by going through multiple iterations of passing it to the backplane handler in a BH_get_data() call and then transmitting it across the network. Each successive BH_ get_data() call reads successive blocks of data from the host computer's buffer. On the inbound side the process can be repeated using BH_put_data(). The backplane handler is also flexible enough to perform the same service for the host computer, using large buffers on the card and multiple small buffers on the host computer. The result is that because of the backplane handler's ability to move data spanning buffer boundaries on either the host computer or the OSI Express card, the driver and CONE need not worry about accurately matching buffers with each other.

Asynchronous Event Handling

For inbound and outbound data transfers the backplane handler must process asynchronous events to notify CONE and the host system of these data transfers. In the outbound direction the CONE modules must be notified when the host driver posts a buffer of outbound data so that CONE can allocate outbound buffers to transport the data to the network. CONE needs to be told how much data is outbound so that it can allocate resources before the data is read onto the OSI Express card. The same problem exists in the inbound direction. When a packet of data arrives at the backplane handler from the network, the host driver and networking code must be told of its arrival and size so that host networking memory can be efficiently allocated.

In the outbound direction, the driver prefixes each outbound message, which may be made up of multiple large physical buffers linked with DMA chaining quads, with a quad and a small buffer containing size and other information about the outbound message. A bit is set in the prefix quad indicating that it is a transparent command (transparent to the backplane handler), and the entire DMA chain is posted on a register set.

When the transparent command quad is fetched by the backplane handler, the small buffer associated with the quad is copied into the event parameter buffer for that register set. BHI_cmd_arrival() is then called and the transparent command and event parameters are passed on to CONE. The backplane handler will then suspend fetching quads on that register set until CONE has acknowledged the BHI_cmd_arrival() event with a BH_get_data() call on that register set. This prevents a subsequent transparent command from overwriting the original command in the the event parameter buffer until CONE has acknowledged the first transparent command. CONE allocates the resources needed to send part or all of the data across the network, and then calls BH_get_data() with the acknowledge bit set.

In the inbound direction, transparent indications provide event notification to the driver and host networking software. One register set (RS 1) is used as a high-priority transparent indication register set. This register set is serviced by the backplane handler at a priority higher than any other register set, and the driver always keeps a DMA chain of small buffers and completion list entries posted on the transparent indication register set.

When the first packet of an inbound message arrives from the network, the packet is placed in a line data buffer consisting of one or more physical buffers. A physical buffer containing the size and other information about the inbound message is prefixed to the line data buffer, and the prefixed line data buffer is posted to the backplane handler in a BH_put_data() call with the transparent indication bit set. When the request generated by the BH_put_data() call arrives at the head of the request queue on the register set, the request is then requeued onto the transparent indication register set. The data is then sent via DMA into one of the small host computer buffers posted there to receive the data, and then the backplane handler creates a completion list entry.

When the driver reads the completion list entry associated with the transparent indication register set, the transparent indication is passed on to host networking software, which allocates the resources necessary to receive the message. The driver then posts the allocated buffers on the correct register set (as indicated in the transparent indication) with an acknowledge bit set in the first quad's CHAIN_CMD word. The backplane handler then sends the data via DMA into the buffers on the host via the appropriate register set.

Conclusion

Four main benefits have resulted from the design of the OSI Express card backplane handler. The first three are all related in that they are derived from the flexibility of the register set state machine. These benefits include:

- The producer and consumer processes on the host and on the OSI Express card do not have to be time-synchronized. Data transfers may be started either by the host system or the OSI Express card register set being used. The host system can post buffers to start the transfer or CONE can start the transfer by calling procedures BH_put_data() or BH_get_data().
- Data buffers on the host system and the OSI Express card

do not need to match in size. Large buffers on the host can be filled (or emptied) from multiple small buffers on the card, and large buffers on the card can be filled (or emptied) from multiple small buffers on the host. Neither the host nor the CONE modules resident on the I/O module need to know about the buffer sizes on the other side of the backplane.

- The independence of buffer sizes has resulted in reduced overhead for packet assembly and disassembly (a normal operation for network software). The backplane handler allows the OSI Express card to combine packet assembly and disassembly with the data copy that is required to cross the backplane. This allows the OSI Express card networking software to accomplish packet assembly and disassembly without the added overhead of a data copy.
- The problem of one connection or data path blocking data flow on another path at the backplane interface is eliminated. The primary reason for the backplane handler's maintaining multiple independent register sets is to prevent one path from blocking another. If one of these paths becomes blocked because a consumer stops taking data, the remaining paths continue to carry data without the intervention of the networking application on the OSI Express card or the host system.

Acknowledgments

Special thanks to Jim Haagen-Smit who made significant contributions to the design and development of the backplane handler, and in reviewing this article. I would also like to acknowledge the efforts of the HP Precision bus interface chip design team, especially Vince Cavanna and Calvin Olsen, and the midplane memory controller chip design team, especially Mark Fidler and Alan Albrecht, for providing these remarkable integrated circuits and reviewing this article.

References

1. D.V. James, et al, "HP Precision Architecture: The Input/Output System," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pp. 23-30.

CONE: A Software Environment for Network Protocols

The common OSI network environment, or CONE, provides a network-specific operating system for the HP OSI Express card and an environment for implementing OSI protocols.

by Steven M. Dean, David A. Kumpf, and H. Michael Wenzel

IMPLEMENTING HIGH-PERFORMANCE and reliable network protocols is an expensive and time-consuming endeavor. Supporting products containing these protocols is also costly, considering changes in standards, hardware, and application emphasis. Because of these challenges, in the early 1980s HP began to develop a framework for providing portable protocol modules that could be used in a number of products to minimize incompatibility problems and development and support costs. Early network protocol portability concepts were used in networking products for the HP 9000 Series 500 computers,[1] the HP 9000 Series 300 computers, the HP Vectra personal computer, and the HP code for connecting Digital Equipment Corporation's VAX/VMS systems to HP AdvanceNet.[2] Other concepts in modularity and protocol flexibility were developed for products on HP 3000 computers[3] and HP 1000 computers.[4] In anticipation of new standards for ISO OSI (Open Systems Interconnection) protocols, an HP interdivisional task force was formed to define a networking environment for protocols that would incorporate the best ideas identified from current and previous network products, and provide protocols that were portable to a maximum number of machines. This environment is called CONE, or common OSI networking environment.

CONE is a system design for a set of cooperating protocol modules, a collection of functions that support these modules, and a comprehensive specification for module interfaces. A protocol module contains the code that implements the functions for a particular layer of the OSI stack. As shown in Fig. 1, the overall OSI Express card network system is structured as nested boxes. The more deeply nested boxes contain more portable code. The network protocol code contains the data structures and functions that implement the protocol layers. The execution environment defines all the interfaces to the network protocol modules, providing services that are tuned to support network protocols and ensure isolation from the embedding operating system. The embedding operating system includes the facilities provided by the operating system for the processor on the OSI Express card. These facilities include a simple interrupt system and a rudimentary memory manager. The system interface is composed of small, partially portable

modules that perform whatever actions are necessary to adapt the embedding operating system for network use. The services provided by the system interface include:

- Interfaces to interrupt service routines for card-to-host computer DMA
- LAN frontplane hardware and timer functions
- Message channels from the card to the host for error reporting
- Tracing and network management.

This article describes the CONE architecture and the features it provides to support the OSI model.

OSI Addressing

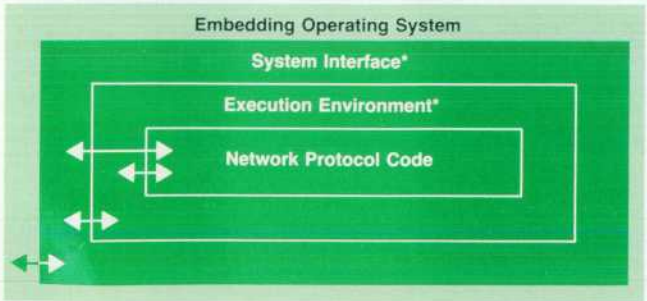
Service Access Points and Connections

Two concepts that are central to the OSI model are service access points (SAPs) and connections (see Fig. 2). These concepts apply at every OSI layer and represent the relationship between a protocol layer and a black box containing all the protocol layers below it.

An SAP is an addressable point at which protocol services are provided for a layer user. A layer user is the next-higher protocol layer (e.g., the layer user of the network layer is typically the transport layer). SAPs for higher-layer users are identified by address or SAP selector information carried by the protocol header. Protocol headers are discussed in the next section.

A connection represents an association between a pair of users for the exchange of information. In the CCITT X.25 standard, which defines protocols that correspond to the first three layers of the OSI model, connections are called virtual circuits. Each connection represents a separate communication path that is maintained by lower-layer protocols. If data stops moving on one connection (e.g., if an application stops receiving data), data can still be exchanged over other connections, since they are independent.

An analogy will serve to illustrate these concepts. A service access point is like a multiline telephone—the kind with the lighted buttons across the bottom, which is typically used by small businesses or departments. The telephone (SAP) is the point at which service is offered by the telephone company (lower-layer protocols). The telephone has a telephone number (address or SAP selector) which is used by the telephone company to identify it when placing calls (see Fig. 3). A connection is like an individual call from one telephone number to another. Just like the lighted buttons on the telephone, several connections may be alive simultaneously between two or more phone numbers. Each lighted button (connection endpoint identifier) can be viewed as the end of an imaginary wire which is used to represent that distinct instance of communication with a remote user. The same pair of telephones may even have more than one connection active between them at a time, each with its own lighted button on each telephone. The user can specify which connection will send or receive data by pressing the related button (connection endpoint identifier). If a remote user stops listening on a given connection, the local user is still free to talk on other connections whose remote users are more responsive.



*Functions and Data Structures Provided by CONE.

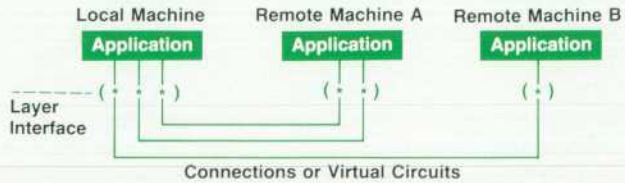
Fig. 1. *Layered architecture of the HP OSI Express card network system.*

Protocol Headers

Most networking protocols send data from a local to a remote layer user by adding protocol control information to the front of the layer user's data buffer. This prepended control information is called a protocol header. The concatenated result then becomes user data for the next-lower layer of protocol (see Fig. 4). This works much the same as envelopes within other envelopes, with the outermost envelopes corresponding to lower layers of protocol. Each protocol layer's header control information corresponds to handling instructions on each envelope. When a packet is received by a machine, each protocol layer examines and removes its handling instruction envelope (header) and delivers the contents to the next-higher protocol layer. One crucial piece of header information identifies which module is the next-higher layer. In the OSI model, this is called the SAP selector. Datagram protocols carry the SAP selector in each packet and treat each packet independently of all others. Connection-oriented protocols only exchange the (possibly large) SAP selectors during the connection establishment handshakes. Successive packet headers carry only a connection endpoint identifier, which is a dynamically allocated shorthand reference that is mapped by the receiving protocol to the specific connection between a pair of layer users.

Addressing Relationships

Every user application finds a remote application via some sort of application directory, which is analogous to a telephone directory. To communicate with an application on another machine, the directory maps the target application's name into an NSAP (network service access point) and an n-tuple vector of SAP selectors. The NSAP is the intermachine address for the machine, and the n-tuple vec-



* Connection Endpoint
( ) Service Access Point

Fig. 2. *Service access points (SAPs) and connections.*

tor contains an entry (intramachine address) for each OSI layer used to communicate with the application on the target machine. There are many schemes for assigning SAP selector values to each of the entries in the n-tuple vector. The ISO OSI standards offer little guidance as to which is the best scheme. However, the important thing is that the n-tuple vector combination be unique for application-to-application communication over a network.

Fig. 5 shows the addressing relationships between the top four layers of the protocol stack for one machine on a network. The intermachine address, or NSAP, for all the applications on this machine is X. The lines in Fig. 5 do not represent connections but addressing relationships, that is, they show which module is pointed at by an address and what are valid address combinations. For application A in Fig. 5, the n-tuple vector is P1, S1, T1 and for application B the n-tuple vector is P22, S1, T1. For these two applications the protocol stack uses the presentation layer SAP selector values P1 and P22 to tell these two applications apart. For application C, which has the n-tuple P44, S9, T1, the presentation layer SAP selector P44 would be redundant because no other application uses the subvector S9, T1. For application D the n-tuple is P77, S9, T2. Since application C and D have the same SAP selector for the session layer (S9), the SAP selectors are interpreted within the context of the transport layer SAP selectors T1 and T2, respectively.

Direct applications are applications that use the services of lower OSI layers and bypass some of the upper-layer protocols. To the rest of the OSI stack these applications look like alternative modules to the upper OSI layers. For example, applications E and F use the session layer directly. To the lower-layer protocols they look like alternative protocol modules of the presentation layer. The address vectors for applications E and F are S32, T2 and S99, T2, respectively. Applications G and H use the transport layer directly and they are addressed by the vectors T40 and T50, respectively.

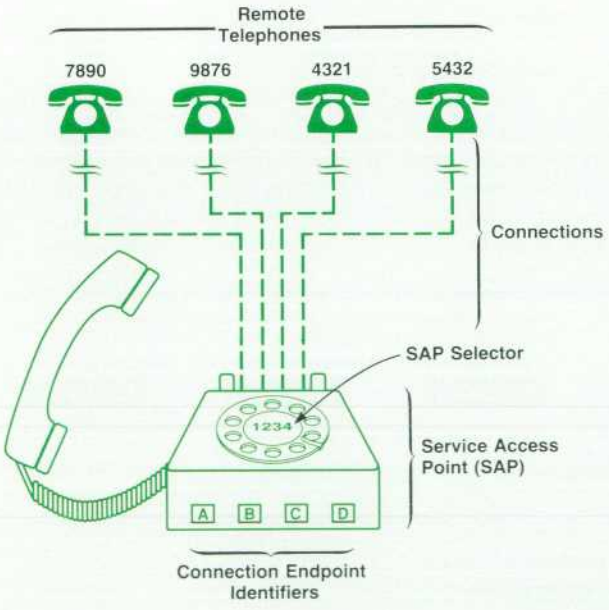


Fig. 3. *Telephone analogy illustrating SAPs and connections.*

Protocol Module Interfaces

In CONE, interfaces to protocol layers are procedure-based, as opposed to being message-based as in many previous network products. Procedure-based means that protocol modules call one another instead of sending messages to each other through the operating system. This minimizes the number of instructions because a data packet can pass through the protocol layers and be processed without being queued. When necessary, protocol interface procedure calls are converted to messages to cross a process boundary—for instance, when crossing the OSI Express card backplane into the host operating system. Within the OSI Express card protocol stack, higher-level protocol layers call lower-level protocols to process outbound packets, and lower-level protocol layers call higher-level protocols to process inbound packets. To avoid bugs that would be very hard for a protocol designer to anticipate, reentrance is not allowed, that is, a protocol module cannot call back into the protocol module that called it. This means that packets move in one direction at a time through the protocol stack before all the procedures return to the outermost CONE routine.

Protocol layer interrelationships and protocol module interfaces in CONE are represented by three central data structures: protocol entries, paths, and service access point (SAP) entries.

Protocol Entries

For the OSI Express card there is a protocol entry data structure for each protocol layer in the system. This includes protocols from physical layer 1 (IEEE 802.3 or 802.4 LAN) to application layer 7 which contains the Association Control Service Element (ACSE). Fig. 6 shows the configuration of these data structures after power-up. The protocol entry for each protocol layer contains a list of pointers to all of its standard procedure entry points and other information, such as protocol identifiers, statistics, and trace and log masks. Standard procedure entry points include separate calls for actions like establishing and destroying network connections, sending and receiving data, and special control commands. This list of entry points is used to bind modules dynamically in a way similar to the protocol switch table in the University of California, Berkeley UNIX

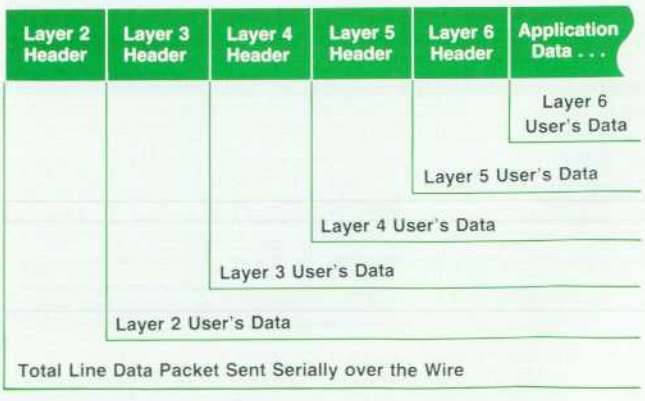


Fig. 4. *Nesting of protocol headers.*

networking implementation.[5]

Separate entry points exist for categorically different actions and for each direction of packet travel—for example, the entry points SP_Send_Down() for outbound packets and SP_Send_Up() for inbound packets, which appear in the session layer shown in Fig. 6. These separate entry points speed access to a protocol's action-handling routines and allow protocols to take advantage of implicit assumptions about the state of a path, thus reducing extraneous state checks and minimizing the number of instructions in the most common data-handling cases. All protocols handle the same parameter structure for each procedure call, allowing protocols to be used interchangeably as building blocks in different combinations as necessary to reach a given destination.

The SAP lookup tables are also set up for each protocol layer right after power-up and all are empty except the tables for the data link (layer 2) and internet protocol (layer 3) layers. The SAP lookup table contains the SAP selectors. Part of the system configuration at power-up is to set up the SAP lookup tables so that the data link protocol module (layer 2) can find the network module (layer 3) and layer 3 can find the transport module (layer 4). The remote network SAP (NSAP) table is also empty because there is no communication with remote nodes at the beginning. If a remote node did try to connect right after power-up and before any applications started to run, the internet protocol layer would create a destination entry to remember who is calling and then it would use its SAP entry to find the transport layer to give it the packet. The transport layer would send an error packet back to the remote node because no transport SAP selector values would be active—the transport layer would not know of any layer users above it yet.

UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

Path Data Structure

When an application begins to communicate with an application on another machine, several data structures are set up by CONE to handle the connection between the two applications. One of these data structures is the path data structure. The path data structure represents an individual connection and serves as a focal point to tie together the collection of all supporting information required to talk to a remote application. It also represents the intramachine route taken through the protocol layers by packets on a given connection from the user to the LAN interface. It consists of an ordered list of all the protocols involved in

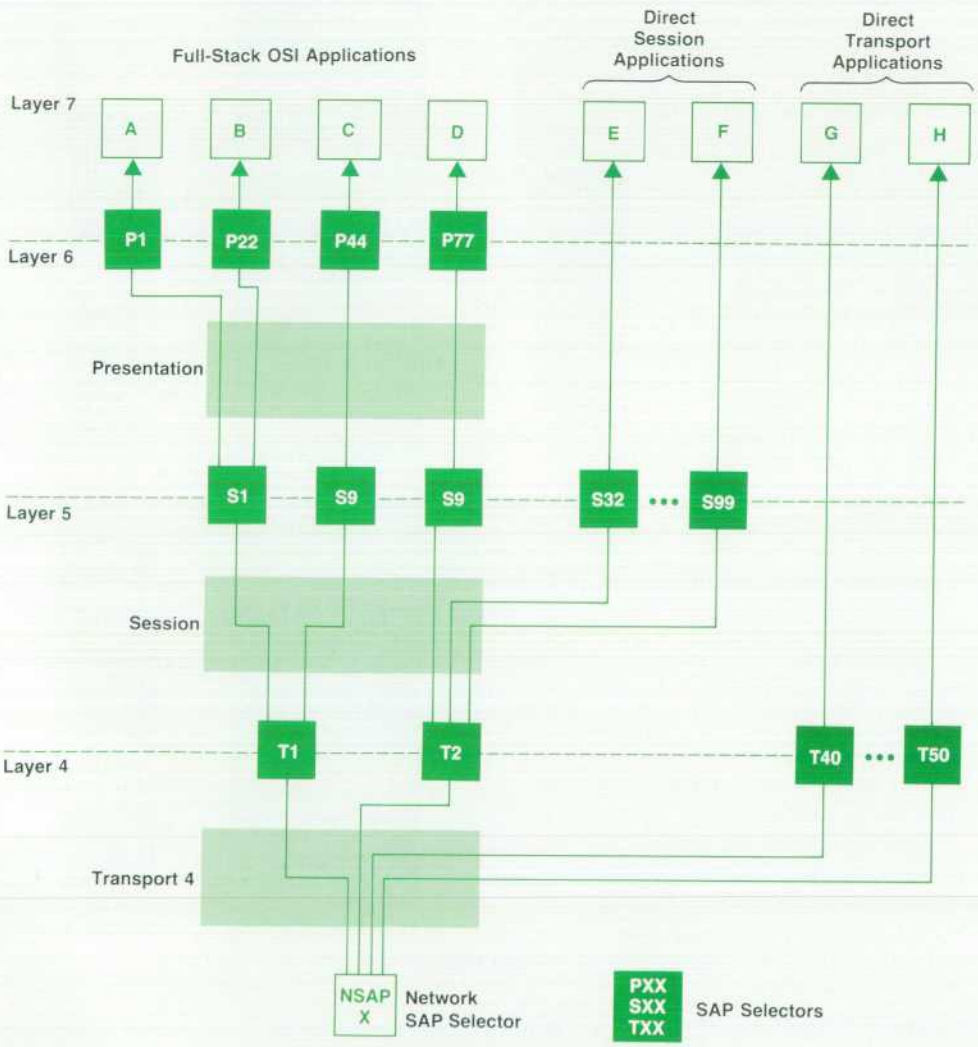


Fig. 5. *Addressing relationships on one machine in a network.*

the conversation, together with their connection state information for this connection (see Fig. 7). As each protocol module is called by CONE to process an event (① in Fig. 7), it is passed a pointer to its entry in this list. This pointer is represented by the PathEntry parameter shown in the interface call IP_Send_Down shown in Fig. 7. The other parameter, buf, points to the parameter block that points to the line data buffers containing the data packets. Parameter blocks and line data buffers are discussed later when the CONE memory manager is described. When each protocol is finished with its part of the overall processing, the PathEntry pointer is used to find the next protocol module to be called, either above or below the current one, depending on whether the packet is being received or transmitted (see the previous and next entries in the path data structure in Fig. 7). Different stacks of protocols can be used for different connections by changing the makeup of the path template. Paths are used by both datagram and connection-oriented protocols on a packet-by-packet basis.

SAP Entries

SAP entries are used by protocols to find each other when a path is first being created. A SAP entry contains the SAP selector value that represents the intramachine address of the next-higher layer user. This relationship is recorded in a standard data structure so that other subsystems like tracing, logging, network management, and dynamic debuggers can know which modules are involved with a given path or packet. Each path entry points to the SAP entry that represents the user on the local end of the connection (② in Fig. 7).

When an OSI application is first activated, it sets up the n-tuple vector of SAP selectors stored in the SAP lookup tables. Each cell in the n-tuple is handled by a separate protocol layer. When CONE calls a protocol module that serves a new user, it passes the user's SAP selector value, user dependent parameters, and a pointer to the related protocol global entry for the next-higher layer in the n-tuple. The called protocol layer adds the new SAP selector value to its SAP lookup table. The relationship of each new SAP selector value to other values and the network topology is protocol dependent because, besides the SAP selector value, information from the protocol header on an incoming packet is often used by the protocol layer as part of the key value to find a given SAP entry. The responsibility of managing these key values belongs to the protocol module. CONE supports the protocols in this function by providing address management utility routines that perform common functions like creating and destroying SAP entries and high-speed mapping of key values to SAP-entry pointers for a given SAP entry.

Besides SAP entries, there is another structure called the destination entry, which is used by the data link layer and the network layer to contain network intermachine addresses and other information about the remote node. In alignment with the functions defined in the OSI model, destination entries for the network layer represent the NSAP for a remote machine beyond the LAN, and destination entries for the data link layer represent machines that share the same link (e.g., a LAN) with the local machine. The destination entry is a standard data structure for all the information that needs to be remembered about a remote machine. Besides the NSAP, examples of other information that would be stored in a destination entry include route and remote dependent protocol parameters (e.g., packet size, options, version). This structure can be used to filter trace and log data for each destination to avoid overloading output files. Transient relationships can exist between the network and data link layer destination entries to represent routing information—for example, to forward a packet to the network layer for destination A, use the data link layer on destination B as the next stop. References to destination entries are counted to ensure that they are held in existence while they are pointed at by other structures.

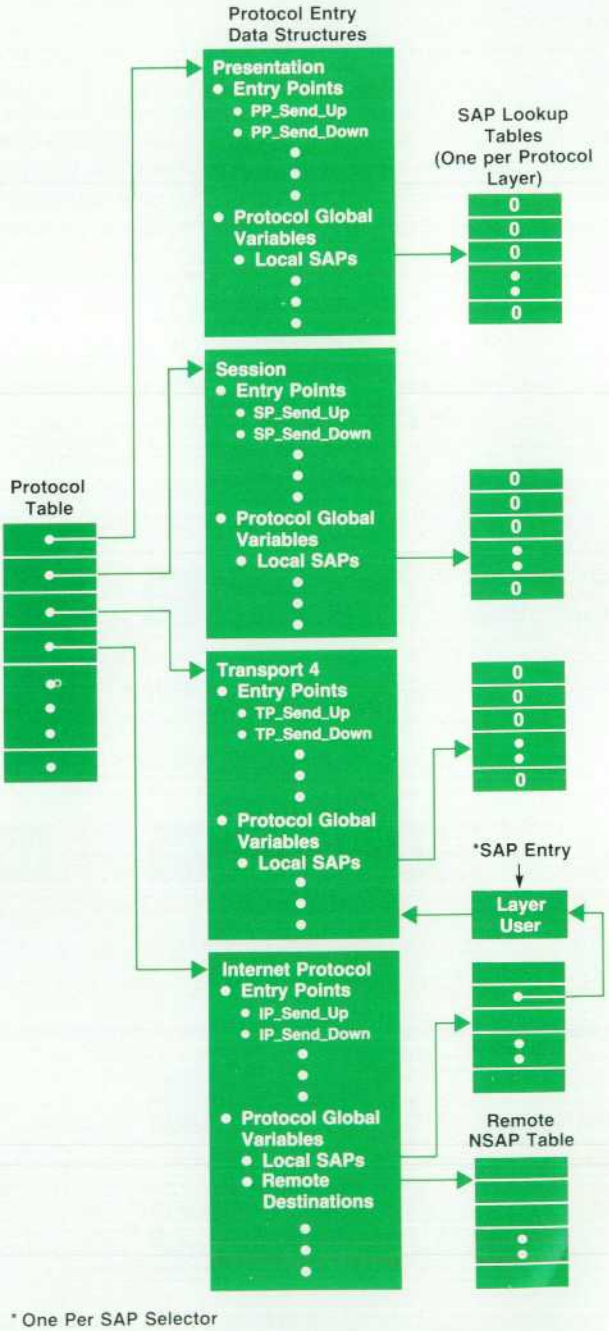


Fig. 6. *Protocol entry data structures right after power-up.*

Tying it Together

The protocol entry, path, and SAP entry data structures together provide the framework that enables protocol modules to create and maintain network connections between applications on different machines. When a user application makes an outbound connection, it directly or indirectly passes down all the related local and remote address information needed to identify the remote machine and all the modules on each end of the connection. CONE uses this information when setting up the path data structure and its relationship to local SAP entries. As the protocols send packets to the remote node to set up the connection, the address information is carried by the protocol headers. For connections coming alive in the inbound direction, the address information in the protocol headers is used by each protocol module to find a SAP entry that contains the information needed to initialize its entry in a fresh path data structure. Inbound paths are initialized upward, one protocol layer at a time. When the incoming connection reaches the user application, the path data structure is a mirror image of the one built for the outbound path on the initiating machine. At any time during the life of the path, CONE can be requested to extract all the address (and protocol parameter) information from a path. This information can be used by a user application to call a remote user back, or during an error log for precise identification of all the modules on each end of a connection having a problem.

Surrounding these common data structures is an extensive list of rules related to how these structures are used and what can and cannot happen as a result of a protocol interface event procedure call. These rules specify:

- What services a protocol at a given layer can rely on from the protocol layer below it without binding itself to a specific lower protocol. This is needed for supporting protocol replaceability (e.g., OSI internet protocol can work with IEEE 802.3, IEEE 802.4, X.25, LAPB, test modules, etc.).
- How protocol facilities are enabled and disabled, and how protocol-specific information is passed to a module in the middle of the protocol stack without the modules around it having to know what is happening. This is needed for protocol module independence and also for protocol replaceability.
- How paths are used when connections at various levels have different lifetimes, or when multiple connections multiplex onto each other.
- Which modules have the right to read or write each of the fields in the common data structures.
- At what times the data structure fields are known to be valid or assumed to be invalid.
- How data sent or received on the network wire (line data) flows from layer to layer.

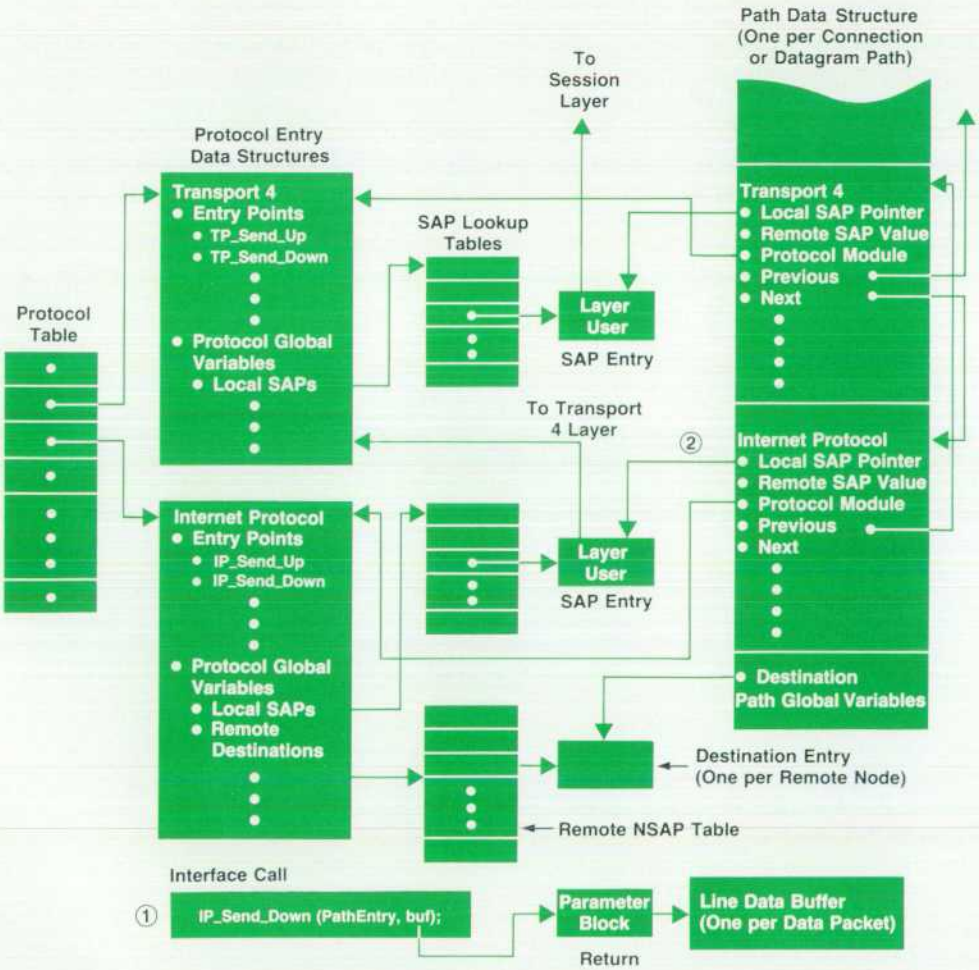


Fig. 7. *Relation between path and protocol entry data structures when applications on different machines are communicating over a network.*

- How buffer space is managed for multiple protocol layers and what layer has the right to touch a line data buffer (buffer containing data packets) at any given time.
- How a line data buffer is to be segmented and reassembled at a given layer when multiple layers have this ability.
- How flow is controlled on a system-wide basis. For example, when there are multiple connection-oriented protocols, buffers do not need to be reserved by each layer to handle its own flow control, retransmission, and queuing requirements. All layers know collectively what will happen to data buffer memory entering or exiting the system.
- How to handle arbitrarily complicated OSI protocol interface events with a minimum number of simple, standard buffer structures and interface calls.
- How to handle error situations, especially under race conditions where things are going wrong on both ends of the path at the same time.

Process Model

One of the major goals of CONE is to provide an architecture where protocol modules can be easily ported to different environments. To provide a portable architecture, it is essential that a well-defined process structure be ported as well. This allows the protocol modules to be designed with a specific process structure in mind.

The underlying process model for the CONE protocol code is procedure-oriented. The CONE process model differs from a typical time-sliced dispatching algorithm in that once a task is dispatched, it is run to completion. CONE performs a sort of "pseudo-multitasking" in that the system depends on the timely completion of a task rather than incurring the overhead of process preemption and context-switching. A task can be thought of as an event handler. When a CONE task is invoked, the dispatcher makes a procedure call to the related event handler procedure. The event handler is then free to do whatever it likes but must eventually return to the dispatcher. When there is no work to be done, the card is idle waiting for an external event to occur. When an external event occurs, the handler for the event is scheduled.

A scheduled event handler is represented by a small data structure called a token. The protocol module provides the space for the token as part of its path data structure. The token contains, among other things, the entry point of the event handler. When an event handler is scheduled, the token is added to the end of a global FIFO task queue. The dispatcher simply calls the event handling routine when the routine's token reaches the front of the queue. Because of the potential overhead, task priorities are avoided as much as possible.

All CONE-based event handlers are considered to be tested, trusted system-quality code. With this type of process model, the protocol modules must abide by two rules. First, the protocol module must complete execution as quickly as possible. Waiting in a loop for an external event is not allowed because it would delay other tasks from running and degrade performance. Second, a protocol module is not allowed to reenter the protocol module that called it. Disallowing a protocol module from being reentered avoids the possibility of infinite loops, and makes coding of the protocol modules much simpler because only one protocol module at a time can be changing the common data structures. Reentrance in a procedure-based system is a fertile bug source. For a small cost in performance, reentrance can be avoided by simply scheduling a task to call the other layers back only after they have exited back to the dispatcher.

An example of the CONE dispatcher behavior is illustrated in Fig. 8. In this example a packet is received that requires a TP4 AK (transport 4 acknowledgment) packet to be sent back out on the LAN. When a packet for the OSI stack is received from the LAN, a frontplane interrupt is generated. The frontplane interrupt service routine will service the hardware, queue the packet, schedule the inbound task of the data link protocol module (LLC = logical link control), and exit. At ① in Fig. 8, the CONE dispatcher calls the LLC inbound task scheduled by the frontplane interrupt service routine. LLC processes the packet and calls the network layer's protocol module (IP), which processes the packet and calls the transport protocol module (TP4). Since TP4 was entered via its inbound packet interface call, it is not allowed at this time to call an outbound interface routine to send an AK. Therefore, it must schedule an AK task ② to send the AK packet out after the inbound routines are done. After processing of the inbound packet, TP4 returns to IP ③, which returns to LLC, which returns to the CONE dispatcher. The CONE dispatcher then moves on to the next pending event, namely the AK task, and wakes up TP4 to handle the event ④. Since TP4 was entered directly from the dispatcher, it is now free to send outbound (or inbound) packets, since no other protocol modules are in danger of being reentered. At ⑤, TP4 calls IP to send the AK, which calls LLC to put the packet on the LAN.

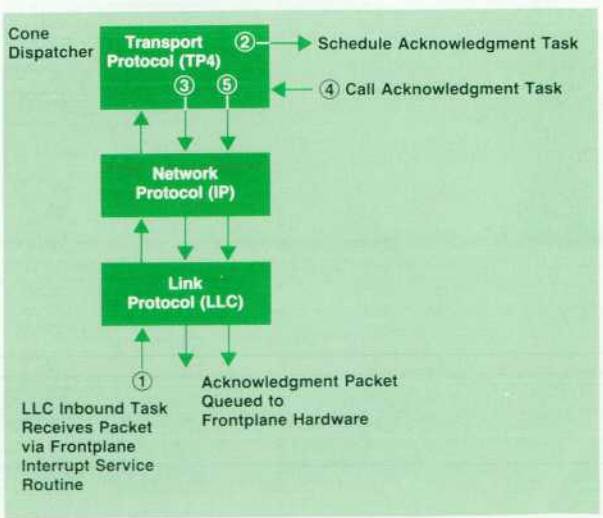


Fig. 8. *Example of the behavior of the CONE dispatcher.*

Memory Management

CONE provides two types of memory: memory objects and line data buffers. A memory object is a contiguous block of memory (heap space). The intended use of a memory object is to hold a data structure for direct use by a CONE-based module. Memory objects can be shared by multiple modules but there is always a single, well-defined owner which changes very little over the lifetime of the object.

Line data buffers hold data that is sent and received on the network wire or line. Unlike memory objects, line data buffers are passed, created, and destroyed outside the CONE environment. To ensure portability, all CONE-based modules allocate, deallocate, write, read, and manipulate line data buffers through macro calls to the CONE buffer manager. Since protocol modules aren't coupled to specific buffer structures, only the buffer manager needs to be changed to use a different underlying structure for efficient interaction within another operating system. Line data buffers are not guaranteed to be contiguous and may consist internally of several smaller memory objects chained together.

CONE's use of memory is optimized for speed in allocating and deallocating memory objects and line data buffers. At the same time, it is designed to make maximum use of available memory by taking advantage of the predetermined characteristics of protocol memory use. This can be contrasted with the memory managers in many conventional operating systems which are not optimized for speed of allocation and deallocation, since most regular processes allocate arbitrary-size memory objects and keep them until the process dies. The CONE buffer manager also plays a major role in card flow control, ensuring that all users can continue to run in worst-case memory situations. Refer to the article on page 36 for a detailed discussion on OSI Express card flow control.

Memory Object Allocation

A fundamental element of any memory management system is the ability to allocate and deallocate contiguous blocks of memory dynamically. Although a basic function, the method chosen can have a significant effect on performance. We studied the first-fit, best-fit, and buddy system memory allocation algorithms and these methods proved to be slower and more complicated than we needed. Networking applications typically make repeated requests for memory objects that fall into a small number of fixed sizes. Since the number of different memory object sizes is small, a two-level scheme is used in which memory is first divided into one of two block sizes, and then small blocks are subdivided to fill memory object pools. Having only two block sizes greatly reduces the time necessary to allocate and deallocate a memory block. A memory block is allocated by removing the block at the head of a free list. A memory block is deallocated by inserting the block at the head of the free list. Large block sizes are only used to grant large line data buffer requests, while the small block sizes are used for both small line data and memory objects.

Dividing the entire memory into fixed-size blocks eliminates external fragmentation because there are no wasted chunks of memory between blocks. However, internal fragmentation can still be a problem since the memory block may be larger than needed. To reduce internal fragmentation a pool manager was developed. The pool manager takes the smaller-size blocks described above and divides them into even smaller blocks of various fixed sizes so that they fit the groups of memory objects used by CONE-based modules. There are several pools, each managing a different object size. By studying the distribution of memory object sizes that are allocated, we determined that four different pool object size groups were needed. With the four pool object size groups and the two original block sizes, wasted space resulting from internal fragmentation was reduced to approximately 10 to 15 percent. CONE-based modules are unaware of whether a memory object comes from a pool or directly from the free list, since this detail is hidden behind the CONE interface.

The pool manager is designed to allocate and deallocate memory objects very quickly. The speed of the pool manager, combined with the simplicity of the memory free list, reduces the time required to allocate and deallocate memory to a very small portion of the overall processing time.

Line Data Buffer Structure

The structure of a line data buffer is a key part of the CONE design. For portability, the internal structure of line data buffers is hidden from CONE-based modules. Line data buffers are passed from module to module as protocol interface events propagate through the stack. To the layer users, a line data buffer is represented by a pointer to a standard data structure in a memory object called a parameter block which invisibly references the memory area that actually stores line data (see Fig. 9). The parameter block functions like a baton that is passed from module to module. The layer currently holding the parameter block is the one that has the right to work with the buffer. The parameter block has a fixed part that carries the standard parameters every protocol module must recognize, such as the current amount of line data contained in the buffer, whether the buffer contains a packet fragment or the end of a fragment train, and what protocol interface event the packet is related to. The rest of the parameter block can be used for storing protocol dependent parameters related to the interface event. This structure allows the protocol interface procedure calls to have a very small number of parameters, speeding up procedure calls from layer to layer. It also provides the space for queuing event-related information in the suboptimal case where the event can't immediately be acted upon and propagated through the stack.

Data copying is kept to a minimum in the buffer manager design, both to maximize performance and to minimize

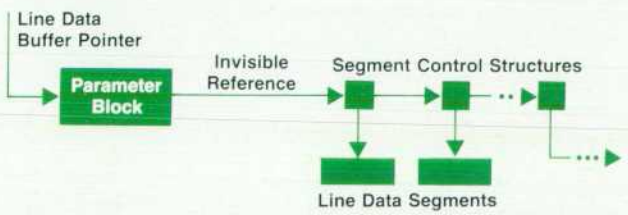


Fig. 9. *The components of a line data buffer.*

memory use. This led to a special feature in the parameter block design. Many connection-oriented protocols, such as OSI transport, need to keep a copy of each transmitted packet until an acknowledgment of delivery is received from the remote machine in case the packet needs to be retransmitted. Rather than allocate another buffer and copy the data, the parameter block is simply marked with a pointer to the protocol module's entry point so that the buffer can be given back when the lower protocol layers have finished with it. When this entry point is called, the protocol module queues the original packet, rather than a copy. The retransmission timer is also started during this buffer return call made by the lower layers after the previous transmission has left the machine. This avoids the embarassing problem of having multiple retransmitted copies of the same packet piling up in the lower layers.

Line Data Buffer Manager

The design of the primitives for the line data buffer manager was driven by what the module designers needed to implement the protocol layers' functionality. Primitives exist for allocating and deallocating buffers, reading or writing data in a line data buffer, adding or removing header data in a line data buffer, disabling or enabling line data flow inbound or outbound for a path, pacing of line data buffer use for each path, and a variety of other functions. Line data buffers are allocated asynchronously. If a module requests a line data buffer and one is not available, the buffer manager will schedule an event and inform the module when the buffer is available.

There are many line data buffer management functions. However, the two most important functions are responsible for fragmenting a packet for transmission and reassembling a packet when it is received.

Fragmenting a Data Packet. When a protocol module, such as the module for the transport layer, receives an outbound packet that is larger than it can legally send, the packet must be fragmented and sent as several smaller data packets. When the transport layer fragments a packet it must attach a header to each fragment. The buffer manager provides a primitive that allows the protocol module to attach its header, which is in a separate buffer, at any point in the data packet without having to copy data from the original buffer. By changing fields in the segment control structures (see Fig. 10) within the line data buffer, the header can be attached without copying data by making the new fragment buffer point into the relevant data portion of the unfragmented buffer. This method significantly improves performance because it avoids data copying.

Reassembling a Data Packet: Some protocol modules, such as the network layer, need to reassemble a fragmented inbound packet before delivering it to the layer above. The buffer manager provides a primitive for reassembling a data packet. This routine will handle out-of-order, duplicate, and overlapping fragments. Again, links in internal buffer segment control structures can be manipulated to avoid recopying the data in the buffers being coalesced.

Allocation versus Preallocation

Establishing a connection requires both types of memory, memory objects for connection-specific data structures and line data buffers to send and receive packets. The buffer manager design evolved from a method in which line data buffers were preallocated for each connection based on where they were most needed. When a connection was established, enough line data buffers were preallocated to ensure that the connection could always make progress. Any line data buffers that were not preallocated could be shared by all other connections to increase performance. The idea was to ensure that each connection had enough buffers to make progress in worst-case memory situations, but allow connection performance to increase when extra, uncommitted line data buffers were available.

Since the OSI Express card has a limited number of buffers, it became apparent that preallocating line data buffers restricted the total number of connections that the card could support. We wanted to support a greater number of connections. Good performance can be achieved as long as too many of the open connections do not try to send or receive data at the same time. The phone company is again a good analogy. Everyone has a phone and performance is generally good, even though there isn't enough switching equipment for everyone to make a call at the same time.

The algorithm used is to have all connections share a pool of line data memory, rather than preallocate buffers when a connection is established. When a moderate number of connections are active, performance is good. As more connections become active at the same time, connection performance degrades since the aggregate system performance is divided among the active connections. This proved to be a good compromise. Good performance was achieved while allowing a large number of connections.

Timer Management

Networking stacks use a large number of timer wakeups. Each connection needs one or more problem timers to detect when an expected event is overdue and recovery action is necessary. Other timers are used to generate protocol messages to check back with the remote machine before its problem timers wake up, and to avoid long delays when the remote machine can't send because of the flow control rules of OSI transport. Unlike timers for most other applications, network timers rarely expire in normal operation, since the expected event usually occurs. Instead, they are

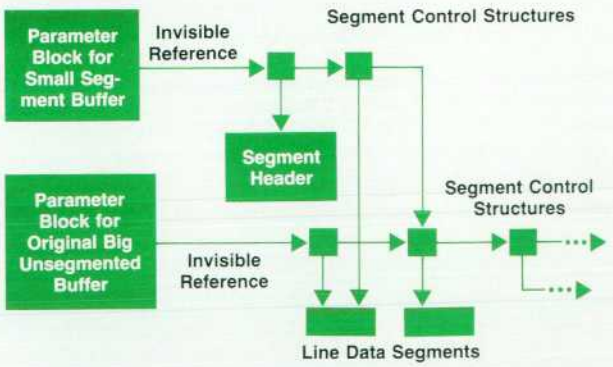


Fig. 10. *This shows what the buffer shown in Fig. 9 looks like after fragmentation. The original buffer is still intact and the new buffer points into it.*

canceled or restarted. A resolution of 100 to 200 ms is just fine since the timers are for exceptional events anyway.

Traditional timer manager implementations have kept the timers in a linked list. This makes it very easy to deal with the expiration of a timer because it is simply removed from the front of the list. However, restarting a timer is slow because the list has to be scanned to locate the proper place to insert it. In the case of the OSI Express card this proved to be too slow. During normal data transfer there are four timers for every connection, at least two of which have to be restarted every time a data packet is sent or received. A quick analysis showed that with just 50 connections, timer insertion could take as long as all other protocol stack processing combined, causing the timer manager to become a performance bottleneck.

What we needed was a way to restart timers quickly. The solution is to keep timers in two unordered lists called the short-term bin and the long-term bin. Timer wakeups are represented by the same tokens that were described earlier for event handlers, with the addition of a "time left until expiration" field. When a timer is restarted it is simply inserted at the head of the appropriate bin. No scanning has to be done every time a packet arrives. Periodically, a monitor task runs that scans the entire short-term bin looking for timers that have expired since the last time it ran. Those timers are removed from the bin and passed to the scheduler to be put on the task queue. Every ten times the timer monitor task runs, it also scans the long-term bin looking for timers that are getting close to expiring and need to be moved down into the short-term bin.

The central idea of this algorithm is to spread the timer list scanning overhead among many packets. To be successful the timer monitor task has to run at some large multiple of the packet arrival rate. If a packet arrives every 5 ms, the timer monitor task can't run every 10 ms or there would be little savings. We found that a period of 100 ms is a good compromise between precision and performance.

System Interface

The system interface is a collection of functions that provides the OSI Express card with an interface to the embedding operating system on the card and communication with the host system housing the OSI Express card. These functions include interrupt service routines and message channels for the card-to-host error reporting, tracing, and network management.

Interrupt System

There are eight available interrupt levels on the OSI Express card. Level zero has the lowest priority and level seven the highest priority. The first three levels are soft interrupts in that they are generated by a processor write to a special hardware register. The rest of the interrupt levels are devoted to interrupt handlers for the various card hardware components. They include a timer hardware interrupt, DMA hardware interrupt, LAN frontplane hardware interrupt, host backplane interrupt, powerfail interrupt, and memory parity and bus error interrupts.

The OSI Express card contains two types of code: the full OSI protocol stack and the card monitor/debugger. The full OSI stack runs at interrupt level zero, which is the card's background level, and the card monitor/debugger runs at interrupt level two. Interrupt level one is reserved for applications that may need to preempt the normal OSI protocol activities. The OSI stack is the largest and most active level since it contains all the protocol modules commonly used for general-purpose networking applications. The card debugger runs at a higher interrupt level than the OSI protocol stack and level one applications so that it can preempt all protocol activity, allowing card diagnosis when either the OSI protocol stack or other applications are stuck in loops.

There are two CONE dispatcher task queues, one for the full OSI stack and one for the card monitor/debugger. Each task queue represents a separate independent instance of the simple CONE process model. When a task queue becomes empty the CONE dispatcher will return to the module that called it. In the full-stack OSI case, the dispatcher will return to the card background process, which is simply an infinite loop that calls the CONE dispatcher. Since the card monitor/debugger runs at interrupt level two, the CONE dispatcher is called from the level two interrupt service routine.

Backplane Message Interface

The OSI Express card's backplane interface is message-based, in that an interface event (transfer of data inbound or outbound) is represented as a message with all the event parameters and line data serially encoded into a string of bytes. The string is sent via DMA between the host computer RAM and the OSI Express card.

Since the CONE protocol module interfaces are procedure-based, a module called the backplane message interface, or BMI, is used to translate CONE events (inbound packets) into messages that are sent to the host operating system and eventually to user applications. For outbound packets the backplane message interface converts messaged-based requests into CONE procedure calls. Because of the way the backplane message interface and the CONE protocol module interface are designed, any protocol module can be accessed across the backplane without the protocol module's knowing whether the entity above its interface is adjacent to it inside CONE.

The following factors affected the design of the OSI Express card's message-based backplane:

- High-performance LAN interface chips require rapid, high-bandwidth access to buffer memory when data is being sent or received on the line. Line signaling is synchronous, meaning that once started, data flows continuously, one bit after another with no wait signals. For these reasons, the buffers accessed by the LAN chips are located in RAM on the OSI Express card, rather than in the host computer.
- A specific word of host RAM cannot be rapidly read or written by the OSI Express card's processor, nor can card RAM be rapidly read or written by the host processor. Instead, so that many cards can share access to host RAM, the backplane is optimized for very high-speed DMA bursts. This minimizes the amount of bus bandwidth lost during bus access arbitration (see article on page 8 for more about DMA and the OSI Express

card).

- The OSI Express card's processor operates as an asynchronous, independent front end to the host processor. Very little card-related processing occurs inside the host operating system, but rather in user-space processes belonging to OSI applications. Since the user-space applications can be busy, timesharing the host with other applications, swapped out to disk, and so on, the coupling between the protocols on the card and the host system is very loose.
- The backplane hardware supports a large number of independent DMA channels. Each CONE path that is tied to a user application is allocated an inbound and an outbound DMA channel at path-creation time. There are also fixed DMA channels for trace messages (inbound), log messages (inbound), nodal management messages (inbound and outbound), debug/monitor messages (inbound and outbound), expedited data which bypasses normal flow control on each path (inbound and outbound), and backplane messages which set up and tear down paths and manage dynamic DMA channel assignment (inbound and outbound).

Conclusion

CONE provides a system design for supporting system-wide and module-internal optimization. Flexibility in the overall framework supports interchangeability of individual protocol modules and protocols from multiple protocol families, as well as portability of CONE-based code to almost any system. Having a coordinated overall framework also makes the system much more instrumentable and supportable. Finally, because of this system-wide orientation, the overall system performance and the number of connections supported for a given amount of RAM are much higher than they would otherwise be.

Acknowledgments

The members of the interdivisional task force that defined CONE include: Sanjay Chikarmane, Allwyn Sequeira, Collin Park, and Dean Thompson. Other contributors who provided additional details and site representation in other areas include Gerry Claflin, Steve Dean, Doug Gregory, and Lori Jacobson. CONE system designers include some of those already mentioned, as well as Bill Gilbert and Dave Woods. As mentioned earlier, CONE was built with the best ideas taken from previous products. CONE represents the work of many people, but the work of one person particularly stands out, Carl Dierschow, who originated the leaf node architecture which greatly influenced the CONE design.

References

1. J.J. Balza, H.M. Wenzel, and J.L. Willits, "A Local Area Network for the HP 9000 Series 500 Computers," *Hewlett-Packard Journal*, Vol. 35, no. 3, March 1984, pp. 22-27.
2. C. Dierschow, "Leaf Node Architecture," *Hewlett-Packard Journal*, Vol. 37, no. 10, October 1986, pp. 31-32.
3. K.J. Faulkner, C.W. Knouse, and B.K. Lynn, "Network Services and Transport for the HP 3000 Computer," *Hewlett-Packard Journal*, Vol. 37, no. 10, October 1986, pp. 11-18.
4. D.M. Tribby, "Network Services for HP Real-Time Computers," *Hewlett-Packard Journal*, Vol. 37, no. 10, October 1986, pp. 22-27.
5. S.J. Leffler, W.N. Joy, R.S. Fabry, and M.J. Karels, *Networking Implementation Notes, 4.3 Edition*, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Revised June 5, 1986.

The Upper Layers of the HP OSI Express Card Stack

The upper three layers of the HP OSI Express card share the same architecture and use tables to simplify their implementations of the OSI stack. The application and presentation layers are implemented in the same module.

by Kimball K. Banker and Michael A. Ellis

THE TOP THREE LAYERS of the OSI Reference Model consist of the session layer, the presentation layer, and the application layer. The purpose of the session layer is to provide organized and synchronized exchange of data between two cooperating session users—that is, two presentation layers in different applications. The session layer depends on the services of the transport layer to provide the end-to-end system communication channels for data transfer. The presentation layer's job is to negotiate a common transfer syntax (representation of data values) that is used by applications when transferring various data structures back and forth. The application layer is the high-

est layer of the OSI Reference Model and does not provide services to any other layer. This layer uses the common protocol called Association Control Service Element, or ACSE, to establish and terminate associations between applications and to negotiate things that are common to applications.

Session Layer

The OSI Express card's implementation of the session layer provides services to the presentation layer that enable it to:

- Establish a virtual connection with a peer session user to exchange data in a synchronized manner and release the connection in an organized manner
- Negotiate for the use of tokens to exchange data and arrange for data exchange to be half-duplex or full-duplex
- Establish synchronization points within the session connection dialogue so that in the event of errors, dialogue can be resumed from the agreed synchronization point
- Interrupt a dialogue and resume it later from a prearranged point.

Session Architecture

The OSI session protocol is now an international standard which is specified in ISO documents 8326 and 8327. However, defect reports and enhancements continue to be made to the base standard. These changes will continue to occur long after the first release of the first OSI Express product. Therefore, one of the key design considerations for our implementation of the session protocol was to provide for easy maintenance of the software. Another design goal was to isolate the protocol software from machine and system dependencies, thus allowing the protocol software to be portable from machine to machine with little or no changes. The common OSI networking environment (CONE) architecture enabled us to achieve our portability goal.

The session software is designed to separate those functions that pertain specifically to the OSI protocol and those that are called local matters. Local matters are primarily tasks that are not included in a protocol specification because they depend on specific system capabilities, such as user interfaces and memory management. As shown in Fig. 1, the OSI Express implementation divides session functions into two main modules, the session CONE manager and the SPM (session protocol machine).

The session CONE manger is primarily responsible for servicing local matters and providing a clean interface between CONE and the SPM. Some of the major functions of the session CONE manager include:

- Translating CONE interface macros into a form the SPM can act upon
- Providing session memory requirements using the CONE buffer manager
- Providing session timer requirements using the CONE timer manager
- Providing much of the session abort processing capabilities
- Managing the underlying transport connection.

The SPM is responsible for servicing the OSI session protocol requirements. The three primary functions the SPM performs are:

- Coordinating state table transitions
- Encoding SPDUs (session protocol data units)
- Decoding SPDUs.

Most of the future changes to the session standard will affect these three SPM operations. Therefore, maintainability was a critical concern in design decisions for the SPM.

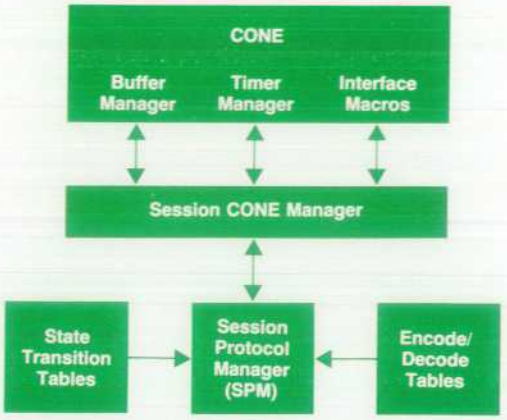


Fig. 1. *Architecture for the OSI Express implementation of the session layer.*

Session State Table

Aside from some clarifying text, the entire OSI session protocol can be defined in terms of tables. Ten separate tables dictate session protocol behavior. A portion of a typical session state table is represented in Fig. 2. The intersection of any given session event (outbound session primitive or inbound SPDU) with a valid session protocol state indicates a set of specific actions and the new protocol state to enter. For example, once the underlying transport connection is established, the SPM is in state STA01C. When a CN event arrives (indicating a successful connection with another session layer) the SPM will change state if the proper predicate conditions are met. In this example, if the predicate condition ˆp01 is satisfied, a transition to state STA08 occurs, which causes the SPM to generate a session connect indication (SCONind) to its session user.

A fully functional OSI session service implementation is responsible for coordinating the intersection of approximately 80 different session events with 32 different protocol states. This creates 2560 possible state table transitions. Close examination of the session state tables reveals that only 600 of the 2560 possible state table transitions are considered to be valid. Also, many of the valid intersections result in the same actions and next states.

A straightforward and common approach to implementing the behavior of these state tables is to create a massive series of if-then-else and/or switch statements that account for each of the valid session event-state intersections. With 600 valid intersections to account for, the code's complexity is high and its maintainability low.

For the OSI Express card implementation of the session protocol the objective was to exploit the tabular structure of the OSI session protocol as much as possible. By creating a structure of multidimensional arrays corresponding to

the OSI session state tables, a direct relationship can be maintained between the OSI standard and the implementation. As illustrated in Fig. 3, the basic scheme is as follows:

- Enumerated values of the current session SPM state and the incoming session event are used as indexes into a combination of arrays that generate a pair of event and state indexes.
- Ten two-dimensional static arrays are defined, one for each of the 10 protocol state tables defined in the OSI session protocol standard. These arrays are called sparse state table arrays. Each element in a sparse state table array is an unsigned byte that represents an index for a unique C function that is responsible for processing the specific actions of an event-state intersection. The event and state indexes generated above are used to select the correct sparse state table array and serve as indexes into the state table array to generate the corresponding function index.
- The function index is used to select a specific pointer to a function from an array of function pointers. The selected function is then invoked to service the requirements dictated by the session event and the SPM state. Invoking functions from pointer arrays (also known as jump tables) is one of the rarely used yet very powerful capabilities of the C programming language.

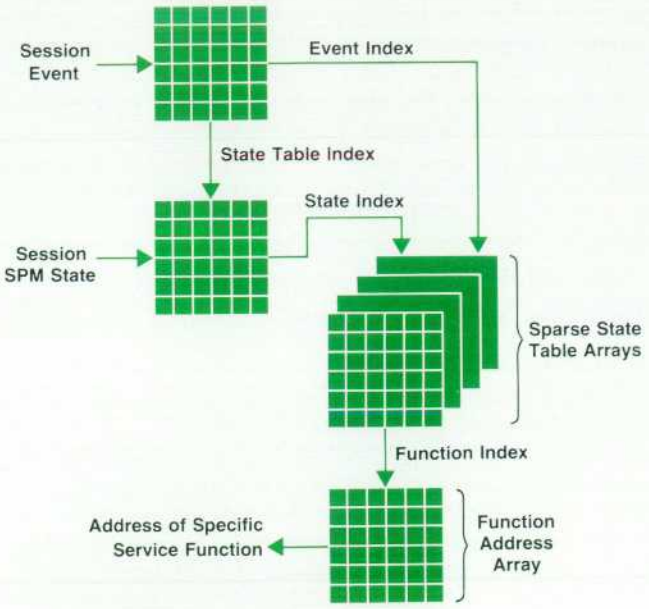


Fig. 3. *Multidimensional arrays used to implement the OSI Express version of the OSI session state tables. Event and SPM state identifiers are used to index into the arrays and acquire pointers to the functions that carry out the actions required.*

Encoding and Decoding SPDUs

Array manipulation also plays a key role in how the session implementation performs the tasks of encoding and decoding SPDUs. SPDUs are constructed in a fairly simple variable format that can be nested three levels deep.

As illustrated in Fig. 4, the mandatory SI (SPDU identifier) value identifies the type of SPDU. The LI (length indicator) following the SI value indicates how many bytes remain in the SPDU. The remainder of the SPDU consists of an optional combination of PGI (parameter group identifier) units and PI (parameter identifier) units to define the particular parameters of the SPDU. PI units are used to encapsulate parameter values such as token items and reason codes, while PGIs are primarily used to encapsulate groups of related PI units. Each PI and PGI unit consists of a PI or PGI value identifying the type of parameters, followed by a length value. The PI unit terminates with the parameter value while the PGI unit follows with either a parameter or one or more encapsulated PI units. The order in which PI and PGI units appear in an SPDU is also important and is uniquely specified for each SPDU.

Session State / Session Event	STA01 Idle no TC	STA01A Await AA	STA01B Await TCONcnf	STA01C Idle TC Con	STA02A Await AC	STA08 Await SCONrsp	STA16 Await TDISind
AC	//	STA01A	//	TDISreq STA01	SCONcnf + [5] [11] STA713 [6]	*	STA16
CN	//	TDISreq [3] STA01	//	∧p01 SCONind STA08 p01 TDISreq STA01		*	TDISreq [3] STA01
RF–nr	//	STA01A	//	TDISreq STA01	SCONcnf– TDISreq STA01	*	STA16
TCONind	TCONrsp [1] STA01C	//	//	//	//	//	//

// = States that Are not Logically Possible

* = Invalid States

Fig. 2. *A portion of a typical state table for the session protocol.*

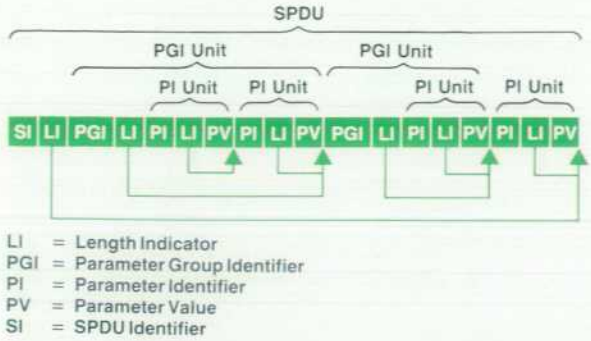


Fig. 4. *Session protocol data unit (SPDU) format.*

A fully functional session implementation is required to encode and decode approximately 20 different types of SPDUs. There are about 30 different types of PI or PGI units that make up these 20 SPDUs, with many SPDUs using the same type of PI and PGI units. PI and PGI units have certain parameter attributes associated with them, such as the maximum number of bytes the parameter may occupy in an SPDU. Because so many of the SPDUs contain the same types of parameters, and since the same parameter attribute information is needed for both encoding and decoding the SPDUs, the decision was made to define the parameter and ordering attributes only once and make this information available for both the encoding and the decoding processes.

Fig. 5 illustrates the manner in which the SPM encodes and decodes SPDUs. Once the SPDU identifier value for the SPDU is determined, it serves as an index into an SPDU script directory array which contains the script index (location) and size of an SPDU script located in the SPDU script array. The SPDU script array contains scripts that define the order in which parameters should appear in each SPDU and indicate whether the parameters are mandatory or optional in that particular SPDU. For each parameter of the SPDU, the SPDU script array also provides an index that selects parameter attribute information from the parameter attribute array.

Two independent programming modules are required to build and parse the SPDUs. They share the information provided by the SPDU and parameter structures defined above.

ACSE and Presentation Layer

The Association Control Service Element (ACSE) is the common protocol for the seventh layer of the OSI hierarchy.[1,2,3] ACSE is meant to be used to establish and terminate an association between applications and to negotiate things that are common to applications, which can be on separate systems. The most important function provided by this common protocol is the negotiation related to the application context parameter. This parameter is a registered name that is passed between applications in the ACSE connect PDU. The application context parameter defines the scope of an application's functionality and is used by local applications to ensure that the remote application is appropriate for a particular association.

The presentation layer is the sixth layer of the OSI model.[4,5] The presentation layer's job is to negotiate common transfer and abstract syntaxes that can be used by applications when transferring various data structures back and forth. Abstract syntax refers to the meaning of the data, and transfer syntax refers to the manner of encoding the data bits.

An application can use the presentation layer to specify several abstract syntaxes for use during an association. For example, an application might specify ACSE and virtual terminal as two abstract syntaxes to be used together in a specific association. The presentation layer will negotiate these two abstract syntaxes during the connection establishment and add the transfer syntaxes for each of the abstract syntaxes it is able to support. If these combinations are acceptable to both sides, subsequent data transfers are transferred with presentation tags denoting the particular abstract syntax (and thus which process should receive this data). The data is encoded in the negotiated transfer syntax and transformed to and from the local representation by the presentation service.

OSI Express Implementation

In the OSI Express card, the ACSE and presentation layers are located in the same code section because both layers share the same challenge in their implementation. The main complexity encountered in implementing ACSE and presentation service on a card involved the encoding and decoding of the ACSE and presentation protocol data units (PDUs) specified by Abstract Syntax Notation One (ASN.1).[6] The protocol data units contain the protocol con-

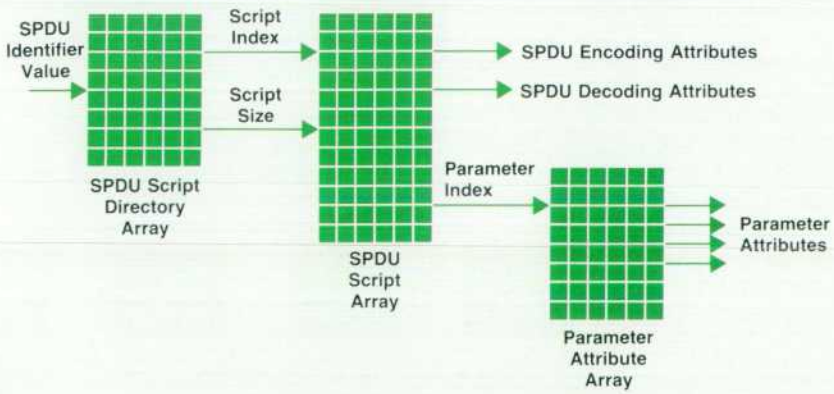


Fig. 5. *The method by which the session protocol manager (SPM) encodes and decodes SPDUs.*

trol information and data that are exchanged between two instances of any protocol. Most protocols (e.g., transport, network, LLC) specify the contents (transfer and abstract syntax) of their PDUs by means of text within their protocol specification document. Most of the upper-layer protocols, such as FTAM, directory services, ACSE, and presentation, use standard and more formal specifications contained in ASN.1.

The OSI Express implementation separates the presentation service into two parts: the protocol that provides transfer and abstract syntax negotiation for applications, and the transformation of user data from ASN.1 transfer syntax to the local representation specified by the abstract syntax and vice versa. Only the protocol is implemented on the card and described in this article, while the remaining transformation of user data occurs in the host system. User data is delivered to the host fully encoded in the agreed upon transfer syntax for a particular abstract syntax. Host software recognizes the abstract syntax from a tag (presentation context identifier) in the PDU and directly transforms the data from the transfer syntax into a local form recognizable to the particular application service element. Except for encoding and decoding PDUs, implementation of the presentation protocol was straightforward.

Two key considerations were identified during the design phase: memory use and the stability of the OSI ACSE and presentation standards. For memory use our goal was not to require a contiguous block of physical memory for either encoding or decoding since large memory buffers in our memory management scheme are not guaranteed. This consideration quickly eliminated many alternative designs. When we were doing our design the OSI standards were just gaining draft approval status with many changes promised in the future. Therefore, our design and architecture had to be easy to modify. The structure of the ACSE/presentation module is shown in Fig. 6. This architecture is similar to that used by other layers in the OSI Express card. The protocol machine is isolated from the CONE architecture by the ACSE/presentation CONE manager. The CONE manager provides a simple interface to the protocol machine and insulates the protocol machine from concerns of state transitions and memory availability. CONE is described in the article on page 18. The heart of the ACSE and presentation protocol implementation is the PDU encoder and decoder. Understanding some basic attributes of ASN.1 provides some insight into the technical solution of encoding and decoding PDUs for the presentation and ACSE protocols.

ASN.1

ASN.1 defines a means to specify the different types of data structures that can be transferred between protocol layers. The ASN.1 standard does not specify the encoding to be used for each type. A companion standard[7] defines the encoding rules which together with the ACSE and presentation specifications define the bit encodings used between the ACSE and presentation protocol layers. The following discussion does not differentiate between the term ASN.1 and the encoding rules since only one set of encoding rules exists for ASN.1.

The basic concept underlying ASN.1 encoding is quite simple. Primitive values are encoded as tag, length, and value. The tag identifies the type of value, length indicates the length of the value, and value represents the contents of the PDU being encoded. Simple primitive types predefined by ASN.1 include character string, Boolean, integer, and real. Primitive types can also be bit string or octet string. However, the encoding of these types is optional. Primitive values are values that cannot be broken down further into other ASN.1 values. ASN.1 also defines complex types, whose values can be broken into additional types.

To accommodate the need to encode complex types, values can be constructed within outer structure definitions. The encoding rules allow a value to consist of another tag, length, and value. Structure definitions for these complex types include:

- Sequence. A fixed ordered list of types.
- Sequence Of. An ordered list of a single type.
- Set. A fixed unordered list of types.
- Set Of. A fixed unordered list of a single type.
- Choice. A fixed unordered list of exclusive types.

These constructed types can be composed of additional constructed types. ASN.1 allows recursive PDU definitions that result in an unbounded collection of permissible sequences. The OSI Express presentation layer has several unbounded sequence types within its connect PDUs. Since values can represent constructed values of tags, lengths, and other values, nesting is prevalent in ASN.1 encodings. In fact, encodings of nested tags and lengths often make up a major portion of an encoded PDU.

An example of an ASN.1 representation of a single presentation PDU, connect confirm negative, is shown in Fig. 7. The PDU description has six parameters, which are defined as follows:

- Protocol-version. The presentation protocol version being used (currently only one exists).
- Responding-presentation-selector. This is presentation layer addressing information.
- Presentation-context-definition-result-list. This structure contains information about which abstract syntaxes are accepted at the initial connection and which transfer syntax is accepted for the transfer of PDUs encoded in the selected abstract syntaxes.
- Default-context-result. This structure specifies whether the

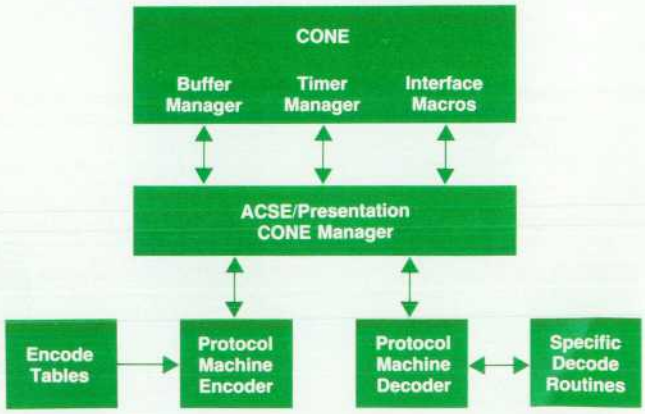


Fig. 6. *OSI Express card ACSE/presentation architecture.*

proposed default context is accepted.

- Provider-reason. This structure contains fields for declaring the reason for refusing the connection associated with a particular PDU.
- User-data. This is data that an application wishes to include on this presentation service primitive.

This PDU is a complex type of Choice. It is a Choice of either Set or Sequence, and in this case Sequence is always used. The first parameter, Protocol-version, has a context-specific tag of 0, as denoted by the [0]. Protocol-version is further defined as a BIT STRING, with the only acceptable value being version-1 the value of 0, as denoted by (0). The Presentation-context-definition-result-list is a complex type with three primitive types: Result, Transfer-syntax-name, and Provider-reason. The values in parentheses to the right of the six parameters denote the values for specific semantics. For example, a value of (1) for a Default-context-result means that the application rejected this default context.

Multiple levels of nesting also make decoding and verifying the length fields challenging. Length fields can be encoded in one of two ways: definite and indefinite. Definite lengths must be kept and verified during decoding and indefinite lengths require the decoder to keep track of where end-of-contents (EOC) flags appear in the PDU definition. Definite and indefinite lengths can appear together in the same PDU at the discretion of the encoder.

Fig. 8 shows the encoded PDU defined by the ASN.1 declaration in Fig. 7. The numbers are the hexadecimal values derived by using the basic encoding rules defined in reference 7, and they represent the values used to describe the semantics defined in Fig. 7. Each line represents a tag and a length. The values for complex types appear on the lines following the complex type declaration, and primitive types include the value on the same line. Tag values are derived from the encoding rules with each bit indicating the tag type (complex of primitive type) and the value of the tag. Fig. 8a begins with 30, which is the universal tag type for Sequence, followed by 80, which represents an indefinite length. For each indefinite length field, a corresponding EOC flag consisting of two octets of zeros must follow. Only complex types can be encoded using indefinite lengths. Fig. 8b shows the same encoded PDU using definite length encoding. Note that 80 is replaced with the definite length indicator 2B. Also note that the

```
CPR-type :: = CHOICE

{SET {x.410-1984 APDUs.RTORJapdu}

 SEQUENCE
 { [0]  IMPLICIT Protocol-version   DEFAULT {version-1},
                 Protocol-version ::= BIT STRING {version-1 (0)}

   [1]  IMPLICIT Responding-presentation-selector            OPTIONAL,
                 Responding-presentation-selector ::= OCTET STRING

   [5]  IMPLICIT Presentation-context-definition-result-list  OPTIONAL,
                 Presentation-context-definition-result-list ::=
                 SEQUENCE OF SEQUENCE
                 {[0] IMPLICIT Result
                               Result ::= INTEGER{acceptance (0),
                                                  user-rejection (1),
                                                  provider-rejection (2)
                                                 }

                  [1] IMPLICIT Transfer-syntax-name OPTIONAL,
                               Transfer-syntax-name ::= OBJECT IDENTIFIER

       provider-reason[2] IMPLICIT INTEGER
                          {reason-not-specified                      (0),
                           abstract-syntax-not-supported             (1),
                           proposed-transfer-syntaxes-not-supported  (2),
                           local-limit-on-DCS-exceeded               (3)
                          }  OPTIONAL
                        }

   [7]  IMPLICIT Default-context-result     OPTIONAL,
                 Default-context-result ::= INTEGER
                                        {acceptance (0),
                                         user-rejection (1),
                                         provider-rejection (2)
                                        }
   [10] IMPLICIT Provider-reason  OPTIONAL
                 Provider-reason ::= INTEGER
                                {reason-not-specified  (0),
                                 temporary-congestion  (1),
                                 local-limit-exceeded  (2),
                                 called-presentation-address-unknown (3),
                                 protocol-version-not-supported (5),
                                 default-context-not-supported (6),
                                 user-data-not-readable (6),
                                 no-PSAP-availlable     (7) }

   User-data  OPTIONAL

 }
}
```

Fig. 7. *An ASN.1 specification for the presentation connect confirm negative PDU.*

Responding-presentation-selector is encoded in two ways, both of which are valid since octet strings can be encoded as constructed types at the discretion of the sender. In Fig. 8a a constructed type is used to break the Responding-presentation-selector value into three primitive encodings, each with a tag of 04 (universal tag type for octet string) and a length of 02. Fig. 8b merely encodes the entire value as a primitive with a length of 06. ASN.1 encoding rules are not deterministic because the encodings given Figs. 8a and 8b are valid for the same PDU.

Another important aspect of ASN.1 is the concept of a context-specific tag. Some tag values are universal in scope and apply to all ASN.1 encodings. Other tag types assume values whose meaning is specific to a particular PDU. For example, context-specific tag value [0] identifies the presentation Protocol-version in Fig. 7. This tag value only means Protocol-version when encountered in a presentation connect confirm negative PDU. In another PDU, the value [0] means something else entirely.

Context tags allow a protocol designer to assign a tag value such that the value of the tag determines the type of value. To decode and validate the PDU, the decoder must have knowledge of a protocol's context-specific values, their meanings, and the order and range of the PDU primitive values. This means that some parts of an ASN.1 decoder may be generic to any ASN.1 encoded PDU (such as an ASN.1 integer decode routine), while other parts of the decoder are quite specific to a single PDU (such as the checking needed to verify that presentation transfer syntaxes are in the appropriate sequence).

A final key to understanding ASN.1 encoding rules is that in almost all cases, the sender chooses which options to use. These options include the way in which lengths are encoded and when constructed elements may be segmented. Octet strings, for example, may optionally be sent as a contiguous string or parsed into a constructed version with many pieces, which may themselves be segmented. A decoder must handle any combination of the above. Thus, the decoder must be able to handle an almost infinite number of byte combinations for PDUs of any complexity. This makes the decoder more complicated to construct than an encoder. For example, Fig. 8 shows that the Responding-presentation-selector can be encoded in two ways—both valid.

Encoder

The encoder is responsible for encoding outbound data packets based on the ASN.1 syntax. Because the encoder can select a limited set of options within the rather large ASN.1 set of choices, encoding is much easier than decoding. The main requirement of the encoder is to know the syntax of the PDU to be constructed. In particular, it needs to know the order and values of the tags and be equipped with the mechanisms to encode the actual lengths and values.

The OSI Express card implementation encodes PDUs front to back using indefinite length encoding. An alternative, encoding ASN.1 back to front, has the advantage of being able to calculate the lengths and allow definite length encoding. Once all of the primitive values are encoded, the encoder can work backwards, filling in all of the constructed tag lengths. However, encoding back to front does not allow data streaming, since all of the PDU must be present and encoded (including user data) to calculate the lengths. Without data streaming, large pieces of shared memory must be used, thus making memory unavailable to the rest of the card's processes until all of the PDU and its user data has been encoded.

The encoder is table-driven in that a set of tables is used for each type of PDU. Each table contains constants for the tag and length and an index to a routine for a particular value. A generic algorithm uses the tables to build each PDU. The tables allow modifications to be made easily when there are changes to the OSI standards. OSI standards for tag values and primitives changed constantly during our implementation. However, these changes merely meant changing a constant used by the table (often a simple macro

```
30 80          SEQUENCE

   80 02 07 80      Protocol-version

   a3 80            Responding-presentation-selector
      04 02 01 02
      04 02 03 04
      04 02 05 06
   00 00            EOC

   a5 80            Presentation-context-definition-
                    result-list
                    IMPLICIT SEQUENCE

      30 80   SEQUENCE
         80 01 00      Result
         81 02 51 01   Transfer-syntax-name
      00 00   EOC

      30 80  SEQUENCE
         80 01 00      Result
         81 02 51 01   Transfer-syntax-name
      00 00  EOC

      30 80  SEQUENCE
         80 01 01      Result
      00 00  EOC

   00 00   EOC

   87 01 03    Default-context-result

   8a 01 00    Provider-reason

00 00   EOC
```

(a)

```
30 2B                        SEQUENCE

   80 02 07 80                 Protocol-version

   83 06 01 02 03 04 05 06     Responding-presentation-selector

   a5 17                       Presentation-context-definition-
                               result-list
                               IMPLICIT SEQUENCE

      30 07                      SEQUENCE
         80 01 00                  Result
         81 02 51 01               Transfer-syntax-name

      30 07                      SEQUENCE
         80 01 00                  Result
         81 02 51 01               Transfer-syntax-name

      30 03                      SEQUENCE
         80 01 01                  Result

   87 01 03                    Default-context-result

   8a 01 00                    Provider-reason
```

(b)

Fig. 8. *Encoding for the PDU shown in Fig. 7. (a) Indefinite length encoding. (b) Definite length encoding.*

update). Changing the order or adding or deleting a value was also easy because only the table entries had to be altered.

Decoder

The decoder presented a more significant challenge in the ACSE/presentation protocol machine. In an effort to reduce memory requirements, the decoder does not depend upon having the entire PDU in memory to decode. Pieces can be received separately, and these need to be decoded and the memory released. The decoder also does not require contiguous memory. PDU segments can be received from the session layer according to the transport segment size. In addition, the memory manager on the card presents PDUs in separate physical buffers called line data buffers (see the article on page 18, which describes the CONE memory manager).

The main job of the decoder is to find the primitive values encoded within the complex nesting of tags and values, and extract those primitives. Along the way, the decoder must also verify that the outer constructed tags are correct, and that the lengths associated with all the constructed tags are correct.

The decoder uses a mathematical calculation to predict and check directly the appropriate tag values. The idea is to generate a unique token that directly identifies particular primitive values. This unique tag is calculated by successively using the outer nested tag values to create a unique number that can be predicted a priori. For example, a simple method to calculate a unique value for any primitive is to take every constructed tag value and add it to the total calculated from previous constructed tags, and then multiply the new total by some base. This calculation derives a unique value for every primitive in a PDU. The unique value can be calculated statically from the standard. Our implementation uses the same constants as were used in the encoding tables above to construct a compiled constant. The unique value can then be calculated dynamically as the decoder goes through a received PDU. Thus, as the decoder is parsing a PDU and successively reading constructed tags, it is calculating the current_unique_tag = (old_unique_tag × base) + tag_value.

The advantage of this method is that a generic decode routine can be used to validate ASN.1 syntax, and as soon as a primitive is reached within a nested PDU, the generic routine can jump directly to a specific routine to deal with the primitive. The value can be checked for specifics and then used or stored. The generic routine is relatively simple. It merely loops looking for a tag, length, and value. If the value is not a primitive it calculates the unique tag. Otherwise it uses the calculated unique tag to know which routine to call. Much of the syntax is automatically verified during the calculation.

The disadvantage of using such a calculation is that while it guarantees a unique number, the number may grow quite large as the depth of nesting within a PDU grows. The problem is that the base used must be at least as large as the total number of tag values. Thus, the unique tag must be able to represent a number as large as the base to the nth power, where n is the depth of nesting required. PDUs that allow very large nesting may not be suitable for unique tag calculation if the largest reasonable number cannot hold the maximum calculated unique tag. Calculating a unique tag has proven to be fairly quick in comparison to using a structure definition to verify each incoming PDU.

Once a primitive tag value is reached, the derived unique tag is used to vector to a procedure specific to that primitive. The procedure contains the code to deal with the primitive. The decoder has a switch table of valid tags, as well as a bit table used to determine correct orders of values and mandatory or optional field checks. This mechanism allows the decoder to identify quickly the primitives nested within a complex PDU, verify correctness, and take the necessary action.

The decoder must perform two types of length checking: definite lengths in which lengths must be kept and verified, and indefinite lengths in which the decoder must keep track of end-of-contents flags. Definite and indefinite lengths can appear together in the same PDU at the discretion of the encoder. The decoder uses two stacks in parallel to check the lengths, one for definite values, and one for EOCs. The definite length stack pushes a value for each constructor type encountered and subtracts a primitive length from each of the appropriate constructor values in the stack. When the last, innermost primitive is subtracted, the appropriate constructor values are popped from the stack. Using and saving stacks allows the decoder to receive PDU segments and decode part way, stop, save the stack values, and resume decoding when the next PDU segment is received. Thus, a complete PDU does not have to be received before memory can be released back to the card memory pool. With this design we have not noticed any difference in the amount of time it takes to decode definite and indefinite length types.

Using a Compiler

During the design phase, the option of using an ASN.1 compiler was considered for the ACSE and presentation protocol machines. The main advantage of a compiler is that once the compiler is written, any protocol specification that uses ASN.1 can be compiled into useful object code. The object code then interacts with the protocol machine via a set of interface structures. The disadvantages of compilers are that they are complicated to write and existing compilers expect PDUs to be decoded from contiguous buffers. The generic code produced is also larger than the specific code necessary for relatively small protocols. Given the requirement to stream PDUs in memory segments, to use as little memory as possible, and to decode only ACSE and presentation PDUs, the compiler alternative was not as attractive as it might be in other applications.

References

1. *Information Processing Systems - Open Systems Interconnection - Application Layer Structure*, ISO/DP 9545, ISO/TC97/SC21/N1743. July 24, 1987. Revised November 1987.
2. *Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*, ISO 8824: 1987 (E).
3. *Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, ISO 8825: 1987 (E).
4. *Information Processing Systems - Open Systems Interconnec-*

tion - Service Definition for the Association Control Service Element, ISO 8649: 1987 (E) (ISO/IEC JTC1/SC21 N2326).
5. Information Processing Systems - Open Systems Interconnection - Protocol Definition for the Association Control Service Element, ISO 8650: 1987 (E) (ISO/IEC JTC1/SC21 N2327).
6. Information Processing Systems - Open Systems Interconnection - Connection Oriented Presentation Service Specification, ISO 8822: 1988 (ISO/IEC JTC1/SC21 N2335).
7. Information Processing Systems - Open Systems Interconnection - Connection Oriented Presentation Protocol Specification, ISO 8822: 1988 (ISO/IEC JTC1/SC21 N2336).

Implementation of the OSI Class 4 Transport Layer Protocol in the HP OSI Express Card

The HP OSI Express card's implementation of the transport layer protocol provides flow control, congestion control, and congestion avoidance.

by Rex A. Pugh

THE TRANSPORT LAYER is responsible for providing reliable end-to-end transport services, such as error detection and recovery, multiplexing, addressing, flow control, and other features. These services relieve the upper-layer user (typically the session layer) of any concern about the details of achieving reliable cost-effective data transfers. These services are provided on top of both connection-oriented and connectionless network protocols. Basically, the transport layer is responsible for converting the quality of service provided by the network layer into the quality of services demanded by the upper layer protocol.

This article describes the OSI Express card's implementation of OSI Class 4 Transport Protocol (TP4). The OSI Express TP4 implementation extends the definition of the OSI transport layer's basic flow control mechanisms to provide congestion avoidance and congestion control for the network and the OSI Express card itself. Because we have requirements to support a large number of connections on a fairly inexpensive platform, the memory management and flow control schemes are designed to work closely together and to use the card's limited memory as efficiently as possible. This efficiency also includes ensuring fair buffer utilization among connections.

Flow Control Basics

An introduction to the basic concepts of flow control, congestion control, and congestion avoidance is useful in setting the stage for a discussion of the OSI Express card TP4 implementation. These concepts are related because they all solve the problem of resource management in the network. They are also distinct because they solve resource problems either in different parts of the network or in a different manner.

Flow Control

Flow control is the process of controlling the flow of data between two network entities. Flow control at the transport layer is needed because of the interactions between the transport service users, the transport protocol machines, and the network service. A transport entity can be modeled as a pair of queues (inbound and outbound) between the transport service user and the transport protocol machine, and a set of buffers dedicated to receiving inbound data and/or storing outbound data for retransmission (see Fig. 1). The transport entity would want to restrain the rate of transport protocol data unit (TPDU*) transmission over a connection from another transport entity for the following reasons:

- The user of the receiving transport entity cannot keep up with the flow of inbound data. In other words, the inbound queue between the transport service user and the transport protocol machine has grown too deep.
- The receiving transport entity does not have enough buffers to keep up with the flow of inbound data from the network.

Note that analogous situations exist in the outbound direction, but they are usually handled internally between the transport user and the transport entity. If the sending transport entity does not have enough buffers to keep up with the flow of data from the transport user, or the sending transport entity is flow controlled by the receiving transport

* A TPDU contains transport layer control commands and data packets.

entity, then the transport user must be flow controlled by some backpressure mechanism caused by the outbound queue's growing too deep.

Thus flow control is a two-party agreement between the transport entities of a connection to limit the flow of packets without taking into account the load on the network. Its purpose is to ensure that a packet arriving at its destination is given the resources it needs to be processed up to the transport user.

Congestion Control

While flow control is used to prevent end system resources from being overrun, congestion control is used to keep resources along a network path from becoming congested. Congestion is said to occur in the network when the resource demands exceed the capacity and packets are lost because of too much queuing in the network.

Congestion control is usually categorized as a network layer function. In an X.25 type network where the network layer is connection-oriented, the congestion problem is handled by reserving resources at each of the routers along a path during connection setup. The X.25 flow control mechanism can be used between the X.25 routers to ensure that these resources do not become congested. With a connectionless network layer like ISO 8473, the routers can detect that they are becoming congested, but there are no explicit flow control mechanisms (like choke packets[1]) that can be used by the OSI network layer alone for controlling congestion.

The most promising approach to congestion control in connectionless networks is the use of implicit techniques whereby the transport entities are notified that the network is becoming congested. The binary feedback scheme[2] is an example of such a notification technique. The transport entities can relieve the congestion by exercising varying degrees of flow control.

Thus congestion control is a social agreement among network entities. Different connections may choose different flow control practices, but all entities on a network must follow the same congestion control strategy. The purpose of congestion control is to control network traffic to reduce resource overload.

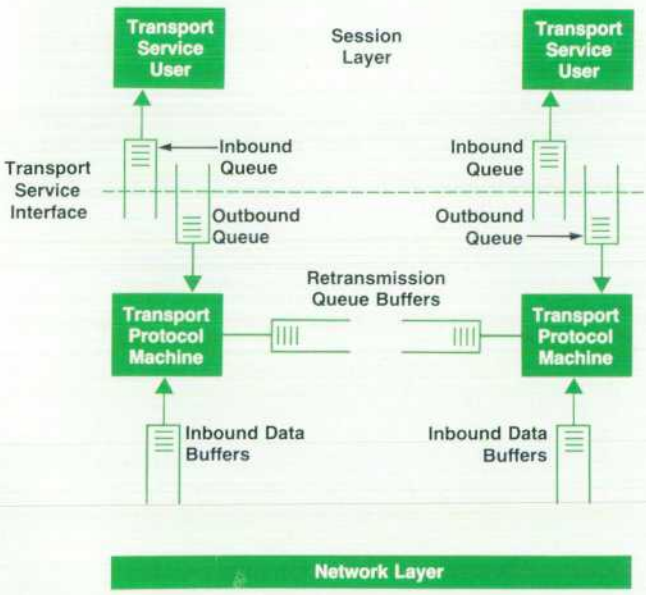


Fig. 1. *Model of a transport entity.*

Congestion Avoidance

Congestion control helps to improve performance after congestion has occurred. Congestion avoidance tries to keep congestion from occurring. Thus congestion control procedures are curative while congestion avoidance procedures are preventive. Given that a graph of throughput versus network load typically looks like Fig. 2, a congestion avoidance scheme should cause the network to oscillate slightly around the knee, while a congestion control scheme tries to minimize the chances of going over the cliff. The knee is the optimal operating point because increases in load do not offer a proportional increase in throughput, and it provides a certain amount of reserve for the natural burstiness associated with network traffic.

Flow Control Mechanisms in TP4

The OSI Class 4 Transport, or TP4, protocol is described in ISO document number 8073. It provides a reliable end-to-end data transfer service by using error detection and recovery mechanisms. Flow control is an inherent part of this reliable service. This section will describe the protocol mechanisms that are used to provide flow control in OSI TP4. These mechanisms make use of the TP4 data stream structure, TPDU numbering, and TPDU acknowledgments.

TP4 Data Stream Structure

The main service provided by the transport layer is, of course, data transfer. Two types of transfer service are available from TP4: a normal data service and an expedited data service. Expedited data at the transport layer bypasses normal data end-to-end flow control, so we need not concern ourselves with expedited data when discussing TP4 flow control.

The OSI transport service (TS) interface is modeled as a set of primitives through which information is passed between the TS provider and the TS user. Normal TS user data is given to the transport layer by the sending TS user in a transport data request primitive. TS user data is delivered to the receiving TS user in a transport data indication

Fig. 2. *A typical graph of throughput versus network load. A congestion avoidance scheme should cause the network to oscillate around the knee, while a congestion control scheme tries to minimize the chances of going over the cliff.*

primitive.

The data carried in each transport data request and transport data indication primitive is called a transport service data unit (TSDU). There is no limit on the length of a TSDU. To deliver a TSDU, the transport protocol may segment the TSDU into multiple data transport protocol data units (DT TPDUs). The maximum data TPDU size is negotiated for each connection at connection establishment. Negotiation of a particular size depends on the internal buffer management scheme and the maximum packet size supported by the underlying network service. The maximum TPDU sizes allowed in TP4 are 128, 256, 512, 1024, 2048, 4096, and 8192 octets.*

TPDU Numbering

The error detection, recovery, and flow control functions all rely on TPDU numbering. Unlike ARPA TCP, where sequencing is based on numbering each byte in the data stream since connection establishment, TP4 sequencing is based on numbering each TPDU in the data stream since connection establishment. A transport entity allocates the sequence number zero to the first DT TPDU that it transmits for a transport connection. For subsequent DT TPDUs sent on the same transport connection, the transport entity allocates a sequence number one greater than the previous one, modulo the sequence space size (see Fig. 3).

The sequence number is carried in the header of each DT TPDU and its corresponding AK (acknowledgment) TPDU. The sequence number field can be either 7 or 31 bits long. The size of the sequence space is negotiated at connection establishment. Since a transport entity must wait until the network's maximum packet lifetime has expired before reusing a sequence number, the 31-bit sequence space is preferred for performance reasons.

TP4 Acknowledgments

An AK (acknowledgment) TPDU is used in OSI TP4 for the following reasons:

- It is the third part of the three-way handshake that is used for connection establishment (see Fig. 4). It acknowledges the receipt of the CC (connect confirm) TPDU.
- It is used to provide the connection assurance or keep-alive function. To detect an unsignaled loss of the network connection or failure of the remote transport entity, an inactivity timer is used. A connection's inactivity timer is reset each time a valid TPDU is received on that connection. If a connection's inactivity timer expires, the connection is presumed lost and the local transport entity invokes its release procedures for the connection. The keep-alive function maintains an idle connection by periodically transmitting an AK TPDU upon expiration of the window timer. Thus the interval of one transport entity's window timer must be less than that of its peer's inactivity timer. Since there is no mechanism for sharing information about timer values, a transport entity must respond to the receipt of a duplicate AK TPDU not containing the FCC (flow control confirmation) parameter by transmitting an AK TPDU containing the FCC parameter. Thus, a transport entity can provoke another transport entity into sending an AK TPDU to keep the connection alive by transmitting a duplicate AK TPDU.
- It is used to acknowledge the in-sequence receipt of one or more DT TPDUs. Since OSI TP4 retains DT TPDUs until acknowledgment (for possible retransmission), receipt of an AK TPDU allows the sender to release the acknowledged TPDUs and free transmit buffers. To acknowledge the receipt of multiple DT TPDUs, an implementation of OSI TP4 may withhold sending an AK TPDU for some time (maximum acknowledgment holdback time) after receipt of a DT TPDU. This holdback time must be conveyed to the remote transport entity at connection establishment time.

*An octet is eight bits.

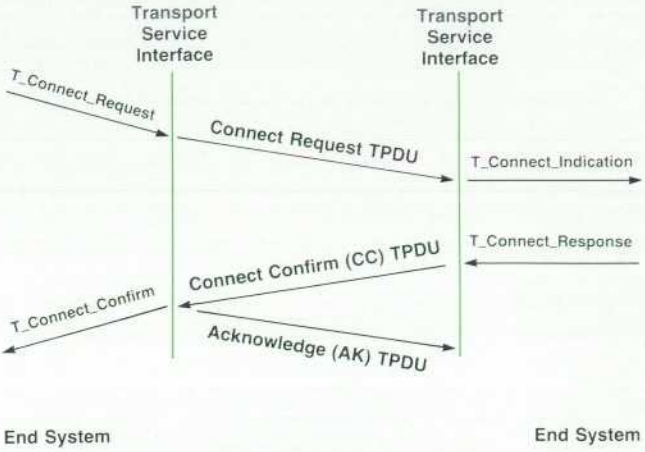


Fig. 4. *Three-way handshake used for connection establishment.*

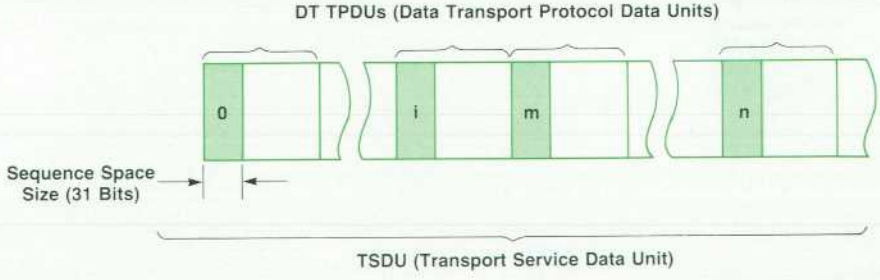


Fig. 3. *Transport data service unit (TSDU) format and the DT TPDU numbering scheme.*

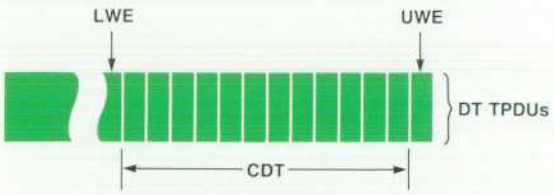
- It is used to convey TP4 flow control information, as described in the next section.

TP4 Flow Control

OSI TP4 flow control, like many other schemes, is managed by the receiver. TP4 uses a credit scheme. The receiver sends an indication through the AK TPDU of how many DT TPDUs it is prepared to receive. More specifically, an AK TPDU carries the sequence number of the next expected DT TPDU (this is called the LWE or lower window edge) and the credit window (CDT), which is the number of DT TPDUs that the peer transport entity may send on this connection. The sequence number of the first DT TPDU that cannot be sent, called the upper window edge (UWE), is then the lower window edge plus the credit window modulo the sequence space size (see Fig. 5). As an example, say that the receiving transport entity has received DT TPDUs up through sequence number 5. Then the LWE or next expected DT TPDU number is 6. If the receiver transmits an AK TPDU with a CDT of 10 and an LWE of 6, then the transmitter (receiver of the AK TPDU) has permission to transmit 10 DT TPDUs numbered 6 through 15. The transmitter is free to retransmit any DT TPDU that has not been acknowledged and for which it has credit. A DT TPDU is acknowledged when an AK TPDU is received whose LWE is greater than the sequence number of the DT TPDU.

Credit Reduction

OSI TP4 allows the receiver to reduce the credit window as well as take back credit for DT TPDUs that it has not yet acknowledged. The LWE cannot be reduced, however, since it represents the next expected DT TPDU sequence number and acknowledges receipt of all DT TPDUs of lower number. Another way of saying this is that the UWE need not move forward with each successive AK TPDU, and in fact it may move backwards as long as it isn't less than the LWE. As will be discussed later, the OSI Express card's TP4 takes advantage of this feature to provide memory congestion control by closing the credit window (AK TPDU with CDT of zero) under certain circumstances.



UWE = Upper Window Edge
(Sequence Number of the First DT TPDU that Cannot Be Sent)
LWE = Lower Window Edge
(Sequence Number—Carried by an AK TPDU—of the Next Expected DT TPDU)
CDT = Credit Window or Window Size
(Number of DT TPDUs a Receiver Can Handle)

Fig. 5. *Parameters associated with a buffer of DT TPDUs.*

OSI Express Card TP4

The OSI Express card's implementation of TP4 (hereafter called the Express TP4) flow control and network congestion control and avoidance policies use many of the basic protocol mechanisms described above.

Flow Control

In Express TP4 the maximum receive credit window size (W) is a user-settable parameter. A similar parameter (Q) is used to provide an upper limit on the number of DT TPDUs a given connection is allowed to retain awaiting acknowledgment. The Express TP4 dynamically changes the window size and queuing limit based on the state of congestion, so W and Q are treated as upper limits. An application can set values for W and Q for a particular connection during connection establishment. A set of values may also be associated with a particular TSAP (transport service access point) selector, so that applications can select from different transport service profiles. In lieu of a connection using one of the two methods just described, configured default values are used.

There is no real notion of flow control in the outbound direction, although TPDU transmissions are paced during times of congestion. The Express TP4 continues to send TPDUs until it has used all the credit that it was allocated by the peer entity, or it has Q TPDUs in its retransmission queue awaiting acknowledgment, whichever comes first.

Ignoring any congestion control mechanisms for the moment, inbound flow control is also fairly simple. When the Express TP4 sends an AK TPDU, its goal is to grant a full window's worth of credit. The CDT field of the AK TPDU is set to W, and the LWE field is set to the sequence number of the last in-sequence DT TPDU received plus one (i.e., the next expected DT TPDU). The key to the efficient oper-

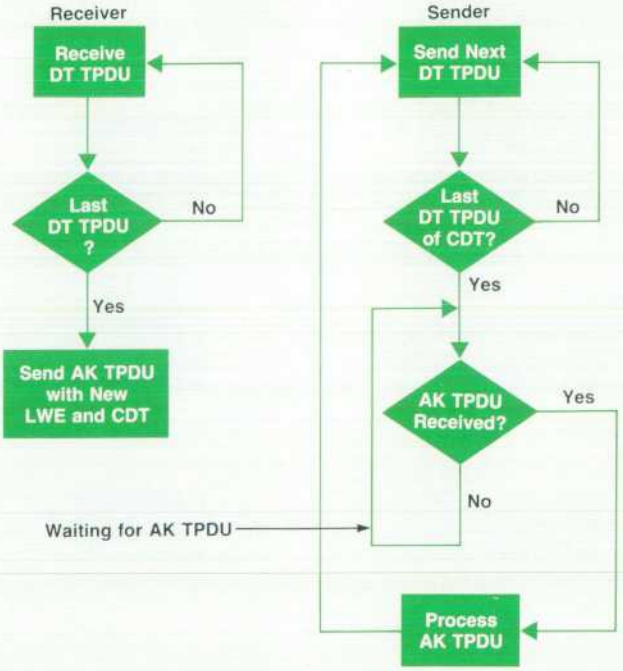


Fig. 6. *Simple flow control policy.*

ation of the flow control policy is the timing of the AK TPDU transmissions.

A simple flow control policy (see Fig. 6) could be to send an AK TPDU granting a full credit window when the last in-sequence DT TPDU of the current credit window has been received. This policy would degrade the potential throughput of the connection, however, because it neglects the propagation delays and processing times of the DT TPDUs and AK TPDUs. After transmitting the last DT TPDU of the current credit window, the sender is idle until the AK TPDU is received and processed. After sending the AK TPDU, the receiver is idle until the first DT TPDU of the new credit window has propagated across the network. These delays could be lengthy depending on the speed of the underlying transmission equipment and on the relative speeds of the sending and receiving end systems.

A more efficient flow control policy, like that implemented in the Express TP4, sends credit updates such that the slowest part of the transmission pipeline (sending entity, receiving entity, or network subsystem) is not idle as long as there is data to be transmitted. This is done by sending an AK TPDU granting a full window's worth of credit before all of the DT TPDUs of the current credit window have been received. The point in the current credit window at which the credit-giving AK TPDU is sent is called the credit acknowledgment point (CAP). Thus the CAP is the sequence number of a DT TPDU in the current credit window whose in-sequence receipt will generate the transmission of an AK TPDU. The AK TPDU's LWE will be the sequence number of the DT TPDU causing the generation of the AK TPDU and the CDT field of the AK TPDU will contain the value of W. The CAP is calculated each time an AK TPDU is sent, and is just the sum of the credit acknowledgment interval (CAI) and the current LWE. CAI represents the number of data packets received before an AK TPDU is sent.

Example

Consider a hypothetical connection where two end systems are connected through an intermediate system via two 9600-baud full-duplex serial links. Fig. 7 shows the progression of DT TPDUs and the flow control pacing AK TPDUs across the links of this connection. At time T0, end system 1 has received the DT TPDU whose sequence number is the CAP. End system 1 then places an AK TPDU in the transmission queue of link A′, thereby granting a new credit window. Meanwhile links A and B are busy processing DT TPDUs numbered CAP + 1 and CAP + 2 respectively. At time T1, the AK TPDU has made it to the link B′ transmission queue and the DT TPDUs have advanced one hop, allowing DT TPDU number CAP + 3 to be inserted in the link B transmission queue. Finally, at time T2, the AK TPDU has made it to end system 2, and again the DT TPDUs have advanced one hop, allowing DT TPDU number CAP + 4 to be inserted in the link B transmission queue. Note that for simplicity, it is assumed that the propagation delay of a DT TPDU across a link is equal to that of an AK TPDU. In reality, DT TPDUs are larger than AK TPDUs and would take longer to propagate.

For this example, the minimal CAI needed to keep the links busy is four, and the minimal window size W is eight. Thus the AK TPDU would carry a CDT of eight, so that end system 2 has credit to send DT TPDUs numbered CAP + 5 through CAP + 8 at the time it receives the AK TPDU (time T2). DT TPDU number CAP + 4 would trigger end system 1 to send another credit-granting AK. The CAI should not be greater than W − 4 for this example, or end system 1 will notice an abnormal delay in the packet train because end system 2 does not have enough credit to keep the links busy while the AK TPDU is in transit. Any CAI less than W − 4 would avoid the delay problem, but the increase in AK TPDU traffic tends to decrease the amount of CPU and link bandwidth that can be used for data trans-

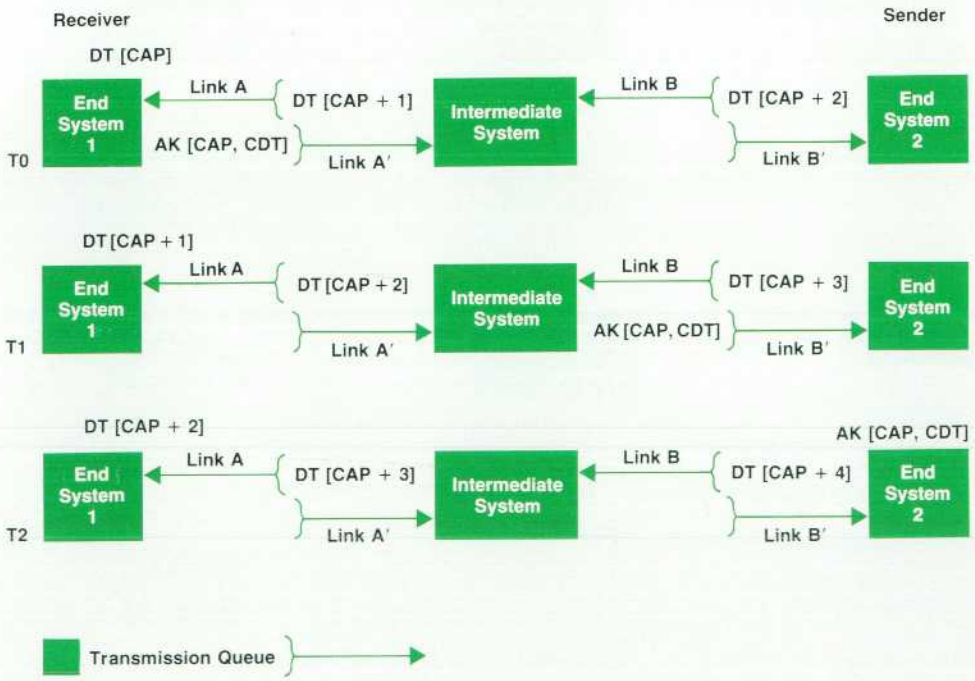


Fig. 7. *Hypothetical connection of two end systems through an intermediate system via two 9600-baud full-duplex serial links.*

mission. The optimal CAI for this example would be W − 4 since that avoids the credit delay and minimizes the number of AK TPDUs. The graph in Fig. 9 on page 56 shows the effect on throughput of different values for W and different CAIs (packets per AK TPDU) for each of the window sizes. This graph was created from a simulation of the Express TP4 implementation running a connection between two end systems connected to a single LAN segment. This simulation data and analysis of real Express TP4 data have shown that a maximum CAI of W/2 yields the best performance with the least amount of algorithmic complexity.

Optimal Credit Window

In the Express TP4, the CAI initially starts at half the credit window size (W/2), but can be reduced and subsequently increased dynamically to reach and maintain the optimal interval during the life of the connection. The optimal value, as shown in the above example, is large enough to ensure that the sender receives the AK TPDU granting a new credit window before it finishes transmitting at least the last DT TPDU of the current window, but not larger than the number of DT TPDUs the sender is willing to queue on its retransmit queue awaiting acknowledgment (note that this scheme relies on the setting of sufficiently large values for Q and W such that the optimal CAI can be reached). If the sending transport entity does not allow W/2 DT TPDUs to be queued awaiting acknowledgment, then as a receiver, the Express TP4 will decrease the CAI to avoid waiting for the CAP DT TPDU that would never come. This situation is detected with the maximum acknowledgment holdback timer. Since any AK TPDU that is sent cancels the acknowledgment holdback timer, expiration of the holdback timer indicates that the sender may not have sent the CAP DT TPDU. When the timer expires, the CAI is decreased to half the number of DT TPDUs received since the last credit update. This is done to preserve the pipelining scheme, since it has been shown that it is better to send AK TPDUs slightly more often than to allow the pipeline to dry up. The amount of credit offered to the receiver is not shrunk (unless congestion is detected), so if the sender devotes more resources to the connection, it can take advantage of the larger window size. The CAP will increase linearly as long as the sender is able to send up to the CAP DT TPDU before the acknowledgment holdback timer expires. The linear increase allows the Express TP4 to probe the sender's transmit capability, and has proved fairly effective.

A more effective mechanism for matching the receiver's AK TPDU rate to the sender's needs has reached draft proposal status as an enhancement to OSI TP4. That mechanism allows the sending transport entity to request an acknowledgment from the receiving transport entity.

Congestion Control and Avoidance

Several network congestion control and avoidance algorithms are used in the Express TP4. All of these algorithms have been described and rationalized in reference 3. This section provides a basic description of each algorithm and how they were effectively incorporated in the Express TP4 implementation. There is also a description of how these algorithms are used together with the dynamic credit window and retransmit queue sizing algorithms to provide congestion control of card resources and network resources.

Slow Start/CUTE Congestion Avoidance

Two very similar congestion avoidance schemes have been described by Jacobsen[3] and Jain.[4] The fundamental observation of these two algorithms is that the flow on a transport connection should obey a "conservation of packets" principle. If a network is running in equilibrium, then a new packet isn't put onto the network until an old one leaves. Congestion and ultimately packet loss occur as soon as this principle is violated. In practice, whenever a new connection is started or an existing connection is restarted after an idle period, new packets are injected into the network before an old packet has exited. To minimize the destabilizing effects of these new packet injections, the CUTE* and slow start schemes require the sender to start from one packet and linearly increase the number of packets sent per round-trip time. The basic algorithm is as follows:

- When starting or restarting after a packet loss or an idle period, set a variable congestion window to one.
- When sending DT TPDUs, send the minimum of the congestion window or the receiver's advertised credit window size.
- On receipt of an AK TPDU acknowledging outstanding DT TPDUs, increase the congestion window by one up to some maximum (the minimum of Q or the receiver's advertised credit window size).

Note that this algorithm also is employed when the retransmit or retransmission timer expires. The CUTE scheme proposes that a retransmission time-out be used as an indication of packet loss because of congestion. Jacobsen[3] also argues, with some confidence, that if a good round-trip-time estimator is used in setting the retransmit timer, a time-out indicates a lost packet and not a broken timer (assuming that a delayed packet is equated with a lost packet). For a LAN environment, packets are dropped because of congestion.

The Express TP4 uses the slow start algorithm (if configured to do so) when a connection is first established, upon expiration of the retransmission timer, and after an idle period on an existing connection. An idle period is detected when certain number of keep-alive AK TPDU's have been sent or received. The slow start and CUTE schemes limit their description to sender functions. The Express TP4 provides the slow start function on the receive side as well, to protect both the network and the OSI Express card from a sender that does not use the slow start scheme. The receiver slow start algorithm is nearly identical to the sender's and works as follows:

- When starting or restarting after an idle period, set a variable congestion window to one.
- When sending an AK TPDU, offer a credit window size equal to the congestion window to the sender.
- On receipt of the CAP DT TPDU, increase the congestion window by one up to some maximum W as described

*CUTE stands for Congestion control *U*sing *T*ime-outs at the *E*nd-to-end layer.

above.

Round-Trip-Time Variance Estimation

Since the retransmit timer is used to provide congestion notification, it must be sensitive to abnormal packet delay as well as packet loss. To do this, it must maintain an accurate measurement of the round-trip time (RTT). The round-trip time is defined as the time it takes for a packet to propagate across the network and its acknowledgment to propagate back. Most transport implementations use an averaging algorithm to keep an ongoing estimation of the round-trip time using measurements taken for each received acknowledgment.

The Express TP4 uses RTT mean and variance estimation algorithms[3] to derive its retransmission timer value. The basic estimator equations in a C-language-like notation are:

$$\begin{aligned} \text{Err} &= \text{M} - \text{A} \\ \text{A} &= \text{A} + (\text{Err} >> \text{Gain}) \\ \text{D} &= \text{D} + ((|\text{Err}| - \text{D}) >> \text{Gain}) \end{aligned}$$

where:

M = current RTT measurement
A = average estimation for RTT, or a prediction of the next measurement
Err = error in the previous prediction of M which may be treated as a variance
Gain = a weighting factor
D = estimated mean deviation
>> = C notation for the logical shift right operation (a division of the left operand by 2 to the power of the right operand).

The retransmission timer is then calculated as:

$$\text{retrans\_time} = \text{A} + 2\text{D}.$$

The addition of the deviation estimator has provided a more reactive retransmission timer while still damping the somewhat spurious fluctuations in the round-trip time.

Exponential Retransmit Timer

If it can be believed that a retransmit timer expiration is a signal of network congestion, then it should be obvious that the retransmission time should be increased when the timer expires to avoid further unnecessary retransmissions. If the network is congested, then the timer most likely expired because the round-trip time has increased appreciably (a packet loss could be viewed as an infinite increase). The question is how the retransmissions should be spaced. An exponential timer back-off seems to be good enough to provide stability in the face of congestion, although in theory even an exponential back-off won't guarantee stability.[5]

The Express TP4 uses an exponential back-off with clamping. Clamping means that the backed-off retransmit time is used as the new round-trip time estimate, and thus directly effects the retransmit time for subsequent DT TPDUs. The exponential back-off equation is as follows:

$$\text{retrans\_time} = \text{retrans\_time} \times 2^{n}$$

where n is the number of times the packet has been transmitted.

For a given DT TPDU, the first time the retransmission timer expires the retransmission time is doubled. The second time it expires, the doubled retransmission time is quadrupled, and so on.

Dynamic Window and Retransmit Queue Sizing

The slow start described earlier provides congestion avoidance when used at connection start-up and restart after idle. It provides congestion control when triggered by a retransmission. The problem with it is that a slow start only reduces a connection's resource demands for a short while. It takes time $\text{RTT}\log_2\text{W}$, where RTT is the round-trip time and W is the credit window size, for the window increase to reach W. When a window size reaches W again, congestion will most likely recur if it doesn't still exist. Something needs to be done to control a connection's contribution to the load on the network for the long run.

The transport credit window size is the most appropriate control point, since the size of the offered credit window directly effects the load on the network. Increasing the window size increases the load on the network, and decreasing the window size decreases the load. A simple rule is that to avoid congestion, the sum of all the window sizes (W_i) of the connections in the network must be less than the network capacity. If the network becomes congested, then having each connection reduce its W (while also employing the slow start algorithm to alleviate the congestion) should bring the network back into equilibrium. Since there is no notification by the network when a connection is using less than its fair share of the network resources, a connection should increase its W in the absence of congestion notification to find its limit. For example, a connection could have been sharing a path with someone else and converged to a window that gave each connection half the available bandwidth. If the other connection shuts down, the released bandwidth will be wasted unless the remaining connection increases its window size.

It is argued that a multiplicative decrease of the window size is best when the feedback selector signals congestion, while an additive increase of the window size is best in the absence of congestion.[3,6] Network load grows nonlinearly at the onset of congestion, so a multiplicative decrease is about the least that can be done to help the network reach equilibrium again. A multiplicative decrease also affects connections with large window sizes more than those with small window sizes, so it penalizes connections fairly. An additive increase slowly probes the capacity of the network and lessens the chance of overestimating the available bandwidth. Overestimation could result in frequent congestion oscillations.

Like the slow start algorithm, the Express TP4 uses multiplicative decrease and additive increase by adjusting W on a connection's receive side and by adjusting Q on a connection's send side. This allows us to control the injection of packets into the network and control the memory utilization of each connection on the OSI Express card. The amount of credit given controls the amount of buffer space needed in the network and on the card. The size of Q also controls the amount of buffer space needed on the

card, because TSDUs are not sent to the card from the host computer unless the connection has credit to send them or there are less than Q TPDUs already queued awaiting acknowledgment. The Express TP4 uses the following equations to implement multiplicative decrease and additive increase.

Upon notification of congestion (multiplicative decrease):

$$W' = W'/2 \quad (1)$$
$$Q' = Q'/2. \quad (2)$$

Upon absence of congestion (additive increase):

$$W' = W' + W'/4 \quad (3)$$
$$Q' = Q' + Q'/4. \quad (4)$$

W′ and Q′ are the actual values used by the connection and W and Q are upper limits for W′ and Q′ respectively.

The window and queue size adjustments are used with the retransmit timer congestion notification in the following manner:

- Expiration of the retransmit timer signals network congestion and Q′ is decreased.
- The slow start algorithm is used to clock data packets out until the congestion window equals Q′.
- As long as no other notifications of congestion occur, Q′ is increased each time an AK TPDU is received, up to a maximum of Q.

OSI Express Congestion Control

One of the main design goals of the OSI Express card was to support a large number of connections. To achieve this goal, the memory management scheme had to be as efficient as possible since memory (for data structures and data buffers) is the limiting factor in supporting many connections. OSI Express memory management is provided by the CONE memory buffer manager (see page 27 for more about CONE memory buffer manager).

Initially, the memory buffer manager was designed such that each connection's packet buffers were preallocated. A connection was guaranteed that the buffers it needed would be available on demand. This scheme provided good performance for each connection when there were many active connections, but it would not support enough active connections. The connections goal had to be met, so the memory buffer manager was redesigned such that all connections share the buffer pool. Theoretically,there can be more connections active than there are data buffers, so this scheme maximizes the number of supportable connections at the cost of individual connection performance as the ratio of data buffers to the number of connections approaches one.

The Problem and The Solution

With a shared buffer scheme comes the possibility of congestion. (Actually, even without a shared buffer scheme, other resources such as CPU and queuing capacity are typically shared, so congestion is not a problem specific to statistical buffering.) Since no resources are reserved for each connection, congestion on the card arises from the same situations as congestion in the network. A new connection coming alive or an existing connection restarting after an idle period injects new packets into the system without waiting for old packets to leave the system. Also, since there can be many connections, it is likely that the sum of the connections' window sizes and other resource demands could become greater than what the card can actually supply.

A shared resource scheme also brings the problem of ensuring that each connection can get its fair share of the resources. Connections will operate with different window sizes, packet sizes, and consumption and production rates. This leads to many different patterns and quantities of resource use. As many connections start competing for scarce resources, the congestion control scheme must be able to determine which connections are and which connections are not contributing to the shortage.

The problem of congestion and fairness was addressed by modeling the card as a simple feedback control system. The system model used consists of processes (connections) that take input in the form of user data, buffers, CPU resources, and control signals, and produce output in the form of protocol data units. To guarantee the success of the system as a whole, each process must be successful. Each process reports its success by providing feedback signals to a central control decision process. The control decision process is responsible for processing these feedback signals, determining how well the system is performing and providing control information to the connection processes so that they will adjust their use of buffers and CPU resources such that system performance can be maximized.

Control System

Certain measures are needed to determine the load on the card so that congestion can be detected, controlled, and hopefully avoided. When the card is lightly loaded, fairness is not an issue. As resources become scarce, however, some way is needed to measure each connection's resource use so that fairness can be determined and control applied to reduce congestion.

Two types of accounting structures are used on the OSI Express card to facilitate measurement: accounts and credit cards. Since outbound packets are already associated with a connection as they are sent from the host to the card, each connection uses its own account structure to maintain its outbound resource use information. All protocol layers involved in a particular connection charge their outbound operations directly to the connection's outbound account. For inbound traffic, when a packet is received from the LAN, the first three protocol layers do not know which upper-layer connection the packet is for. Therefore, a single inbound account is used for all inbound resource use information for the first three protocol layers, and some combined resource use information for upper-layer connections. This provides some level of accountability for inbound resource use at the lower layers such that comparisons can be made to overall outbound resource use. Since a single inbound account exists for all connections, credit cards are used by the upper four layers (transport and up) to charge their inbound operations to specific connections. Thus each connection has an outbound account and a credit card for the inbound account.

The protocol modules and CONE utilities are responsible

for updating the statistics (i.e., charging the operations) that are used to measure resource use. These statistics include various system and connection queue depths, CPU use, throughput, and time-averaged memory utilization. When summed over all of the connections, these statistics are used along with other signals to determine the degree of resource shortage or congestion on the card. The individual connection values indicate which connections are contributing the most to the congestion (and should be punished) and which connections are not using their fair share of resources (and should be allowed to do so).

Flow Control Daemon

The control decision and feedback filtering function is implemented in a CONE daemon process aptly named the flow control daemon. Using a daemon allows the overhead for flow control to be averaged over a number of packets. The daemon periodically looks at the global resource statistics and then sets a target for each of the resources for each connection. The target level is not just the total number of, say, buffers divided by the number of connections. Targets are based on the average use adjusted up or down based on the scarcity of various resources. This allows more flexibility of system configurations since one installation or mix of connections may perform better with different maximum queue depths than another. It is also the simplest way to set targets for things like throughput since total throughput is not a constant or a linear function of the number of connections.

Control signals are generated by the flow control daemon as simple indications of whether a connection should increase, decrease, or leave as is its level of resource use. The daemon determines the direction by comparing the connection's level of use with the current target levels. There is a separate direction indication for inbound and outbound resource use.

The fairness function falls out very simply from this decision and control scheme. Any connection that is using more than its fair share of a resource will have a level of use greater than the average and thus greater than the target when that resource is scarce. In other words, the "fair share" is the target.

The control signals are generated when a connection queries the daemon. The most likely point for querying the daemon is when a connection is about to make a flow control decision. That decision point is, of course, in the TP4 layer of the OSI Express card.

Effects of the Daemon

The effects of flow control notifications to a connection regarding decreasing or increasing resource use vary according to whether the direction of traffic is inbound or outbound.

Outbound. The Express TP4 queries the flow control daemon for outbound congestion/fairness notification when it receives an AK TPDU. It is at this point that DT TPDUs are released from the retransmission queue, and it can be decided if more or fewer DT TPDUs can be queued until the next AK TPDU is received.

If the connection is using more than its fair share of outbound resources (because of congestion or just overzealousness), the daemon will return a decrease notification. A decrease notification causes the Express TP4 to reduce the connection's retransmit queue size (Q′) using equation 2. The slow start algorithm is then used to clock DT TPDUs out until the congestion window equals Q′.

If Q′ is equal to one when a decrease is signaled, the Express TP4 goes into DT TPDU send delay mode. In this mode, transmission of successive DT TPDUs is spaced by a minimum delay (D) to produce an interpacket gap that will slow down the connection's demand for resources. If further decrease signals are received in delay mode, the minimum delay is increased using $D = D \times 2$.

If the connection is using less than its fair share of outbound resources, the daemon will return an increase notification. An increase notification causes the Express TP4 to increase the connection's retransmit queue size (Q′) up to a maximum of Q, using the additive increase equation. If an increase signal is received in delay mode, the minimum delay is decreased using $D = D - D/4$.

Inbound. The Express TP4 queries the flow control daemon for inbound congestion/fairness notification when it sends an AK TPDU. At this point the decision needs to be made whether more or fewer DT TPDUs should be allowed in the pipeline until the next AK TPDU is sent. If the connection is using more than its fair share of inbound resources, the daemon will return a decrease notification. A decrease notification causes the Express TP4 to reduce the connection's receive window size (W′) using equation 1. The slow start algorithm is then used to clock AK TPDUs out with credit window (CDT) values increasing from one to W′.

If W′ is equal to one when a decrease is signaled, the Express TP4 goes into credit delay mode. In this mode, transmission of AK TPDUs containing a CDT of one are spaced by a minimum delay to produce an interpacket gap between incoming DT TPDUs that will slow down the connection's demand for resources. If further decrease signals are received in delay mode, the minimum delay is increased using $D = D \times 2$.

If the connection is using less than its fair share of inbound resources, the daemon will return an increase notification. An increase notification causes the Express TP4 to increase the connection's credit window size (W′) up to a maximum of W, using equation 3. If an increase signal is received in delay mode, the minimum delay is decreased using $D = D - D/4$.

Severe Congestion Notification

The flow control daemon also provides an emergency notification to Express TP4 in cases where transient shortages of memory are severe enough to jeopardize the existence of connections. Because the OSI Express card uses statistical buffering, there is a possibility that a large burst of outbound data could queue up in the Express TP4 retransmission queues, while inbound data is flowing in and getting queued because the host computer is not reading data from the card. If the situation is such that buffers may not be available to receive or send AK TPDUs, the daemon will give an emergency notification to the Express TP4.

Upon receipt of this notification, the Express TP4 sends an AK TPDU with a CDT of zero, closing the credit window. Thus DT TPDUs received that are outside the new credit

window are thrown away so as to avoid memory deadlock. The Express TP4 also decreases the credit window W' and the retransmit queue size Q' using equations 1 and 2. The slow start algorithm is used to get the inbound and outbound data traffic flowing again.

Acknowledgments

A special thanks to Ballard Bare who participated in the design and development efforts for the OSI Express TP4 implementation, and to Mike Wenzel who contributed to the design efforts.

References

1. J.C. Majithia, et al., "Experiments in Congestion Control," *Proceedings of the International Symposium on Flow Control in Computer Networks*, Versailles, France, February 1979.
2. K. K. Ramakrishnan and Raj Jain, *Congestion Avoidance in Computer Networks with a Connectionless Network Layer. Part II: An Explicit Binary Feedback Scheme*, Digital Equipment Corporation, Technical Report #TR-508, August 1987.
3. V. Jacobsen, "Congestion Avoidance and Control," *Computer Review: Communications Architectures and Protocols (SIGCOMM '88)*, Vol. 18, no. 4, August 1988.
4. Raj Jain, "A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, no. 7, October 1986.
5. D. J. Aldous, "Ultimate Instability of Exponential Back-off for Acknowledgment Based Transmission Control of Random Access Communication Channels," *IEEE Transactions on Information Theory*, Vol. IT-33, no. 3, March 1987.
6. K. K. Ramakrishnan and Raj Jain, *Congestion Avoidance in Computer Networks with a Connectionless Network Layer*, Digital Equipment Corporation, Technical Report #TR-506, August 1987.

Data Link Layer Design and Testing for the OSI Express Card

The modules in the data link layer occupy the bottom of the OSI Reference Model. Therefore, it was imperative that they be finished first and that their reliability be assured before use by the upper layers of the OSI stack.

by Judith A. Smith and Bill Thomas

THE DATA LINK LAYER is the second layer in the OSI Reference Model. Its function is to provide access to the LAN interface for the OSI network layer (layer 3), and transmitting and receiving of data packets to or from the physical layer (layer 1). This article describes the data link layer, particularly the OSI Express card's implementation of this protocol layer. The box on page 49 provides a brief description of the OSI network layer.

The data link layer consists of two sublayers: the LLC (logical link control) sublayer and the MAC (media access control) sublayer (see Fig. 1). The LLC sublayer provides a hardware independent interface to the upper-layer protocol. The LLC used for the OSI Express card implementation is specified in ANSI/IEEE standard 802.2. The OSI Express card uses the Type 1 LLC protocol described within this specification. Type 1 LLCs exchange PDUs (protocol data units) between themselves without the establishment of a data link connection. This is also called connectionless network protocol. The MAC sublayer controls access to the shared physical signaling and medium technologies (e.g., coaxial cable, twisted pair, fiber optic cables, and even radio signals). The MAC protocol used by the OSI Express card implementation is specified in IEEE standard 802.4. Besides requiring that the OSI Express card implementation conform closely to the IEEE standards, the goals that guided our design included:

- Hiding the upper LLC interface details from the data link layer user (network layer).
- Making the LLC support multiple MAC sublayers.
- Making the lower LLC interface simple and flexible enough to promote testability and ease of integration.
- Providing a loopback mechanism in the LLC.
- Creating and porting the MAC code to the OSI Express card before all other protocol layers.
- Designing the MAC code and MAC test environment so that some portions are leverageable to other MAC implementations.

Since the data link layer module had to be the first protocol module completed, another goal was to ensure that the design and development process produced simple and

reliable code.

The Data Link Layer and CONE

The data link layer uses the facilities provided by CONE (common OSI networking environment) to provide services to the protocol layer above it and to communicate with the protocol layer below it. These facilities include data structures for service access points (SAPs), interfaces to the protocol layer routines, and the path data structure which represents an individual connection between applications on different machines. CONE facilities and SAPs are described in detail in the article on page 18.

The protocol layer above the data link layer is called the data link layer user. This is the network layer. Since the LLC is the top layer of the data link layer, the network layer is also the LLC user. Similarly the MAC user is the LLC. A SAP is an addressable point at which protocol services are provided for a layer user. SAPs are identified by address information found in the headers (protocol headers) of data packets arriving at each layer. For the LLC layer a SAP address is called an LSAP. Packets arriving at the LLC layer usually have two addresses. One indicates where the packet came from (source) and the other indicates the packet's destination. The from address is called the source service access point, or SSAP, and the destination address is called the destination service access point, or DSAP.

CONE provides three data structures for all the protocol layers that enable them to communicate with each other. The first is the protocol entry data structure, which contains pointers to all the procedures required by a particular protocol layer. For example the following procedures are part of the data link layer protocol and are used by the network layer to command the data link layer to perform certain actions.

- DL_Add_SAP. Set up an LSAP.
- DL_Send_Down. Send a data packet.
- DL_Control_Down. Send an XID or TEST command packet.
- DL_Start_Down. Set up a path between the data layer and its user.
- DL_Delete_SAP. Remove an LSAP.
- DL_Stop_Down. Remove a path.

Pointers to these procedures are set in the CONE protocol data structure when the LLC initialization procedure is called. Also at initialization, an LLC SAP data structure is set up so that the data link layer can find the network layer.

When a connection is established with a remote application, CONE creates a data structure called a path. A path represents the intramachine route taken through the protocol layers by packets on a given connection from the application to the LAN interface. It consists of an ordered list of data structures that contain, among other things, pointers to the SAP entries of the protocol layers involved in the conversation between the two applications. Fig. 7 on page 23 shows the CONE data structures.

Logical Link Control Sublayer

The LLC sublayer on the OSI Express card performs two kinds of functions. It sends and receives packets for the users and sends and responds to XID (exchange identification) and TEST commands. The XID command is used to describe the capabilities of the LLC sublayer on one machine to the LLC sublayer on another machine. The XID command is sent as a single packet containing the DSAP and SSAP addresses, a control field set to the XID command, and the XID information which describes the functions the LLC supports. The LLC on the receiving machine sends a response packet to the sender describing itself. The receipt of the XID command is not reported to the LLC user because it is handled internally by the LLC sublayer. The TEST command is used to test the integrity of the communication link between the LLC sublayers on two communicating machines. Therefore, the TEST command also causes the receiving LLC to send a response. The response data from the receiving machine is expected to be the same data that is sent in the command packet. Like the XID command, the TEST command is not sent to the LLC user. The kinds of DSAP addresses in the XID and TEST commands include individual, group, and global addresses. The individual

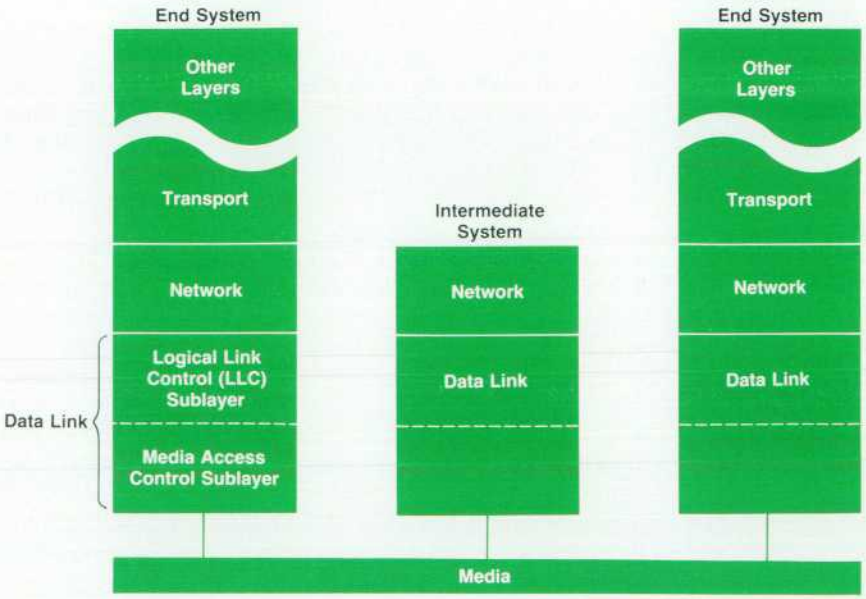


Fig. 1. *Overview of the data link layer and its sublayers.*

address is used when the response packet is to be sent for one particular LSAP address. The group address is used when the response is to be sent for a group of LLC users. The global address is used when the response is to be sent for all of the active LLC SAP addresses. A TEST packet sent to the global address should result in responses from address zero and from each of the other active SAPs. One of the individual addresses, address 0, designates the sending LLC itself and is always active. Therefore, an XID or TEST command sent to this address will always generate a response.

Media Access Control Sublayer

The MAC sublayer is responsible for sending and receiving data from the media. To fulfill this responsibility the MAC sublayer performs:

- Conversion of outbound data into a form acceptable to the hardware that sends the packet onto the media. It performs the reverse transformation for inbound packets
- Checking to ensure that received packets have a MAC address that is acceptable to the OSI Express card and that there are no detectable transmission errors
- Managing how many times retransmission of a packet should be attempted if there are transmission errors.

The MAC sublayer maintains a SAP table with one entry for each active MAC address. Two addresses are always active: the local individual MAC address and the broadcast MAC address. The individual MAC address is stored in nonvolatile memory on the card and is unique for every individual card made. The assignment of this address is managed on a worldwide basis. The broadcast address is one that all MAC sublayers are required to accept. Additional addresses, such as multicast addresses, may also be activated. These multicast addresses are used by the network layer.

LLC and MAC Interface

The procedures contained in the LLC and MAC sublayers are designed to conform closely to IEEE standards 802.3 and 802.4 and to maximize the independence between the two sublayers. The procedures provided by the MAC sublayer include:

- Send_Packet. This procedure is used by the LLC sublayer to request the MAC sublayer to send a data packet out onto the media.
- Activate_MAC_Addr and Deactivate_MAC_Addr. These procedures are used as their name implies, to activate and deactivate MAC addresses. When a MAC address is activated, an entry is made in the MAC SAP lookup table. A MAC address may be activated more than once if several LLC users (with different LSAPs) use the same MAC address. The data structure containing the MAC SAP has a reference counter that contains a count of the number of times the address is activated by one of the LLC users. When the MAC address is deactivated, the count is reduced, but the MAC address itself is not deactivated until the count is reduced to zero.
- Check_MAC_Addr and Store_Indiv_MAC_Addr. These procedures are used to provide independence between the LLC and MAC sublayers.

The procedures provided by the LLC for the MAC sublayer include:

- Check_Packet and Receive_Packet. These procedures are used to send packets received from the media by the MAC sublayer to the LLC sublayer, which in turn sends them to the data link layer user. The Check_Packet procedure was developed to improve performance. When the MAC layer receives a packet from the media it is in a format used by the hardware to interface to the media. Therefore, the data must be converted to the format used by the OSI protocol stack. This effort is wasted if there is no data link layer user to accept the packet. Therefore, before the MAC does the conversion, it calls the Check_Packet procedure to check that the packet's LLC header is valid and that its destination address has an active LSAP set up for it. The LLC then returns a pointer to the LSAP to the MAC sublayer if and only if the packet is acceptable. If a pointer is returned, the MAC sublayer

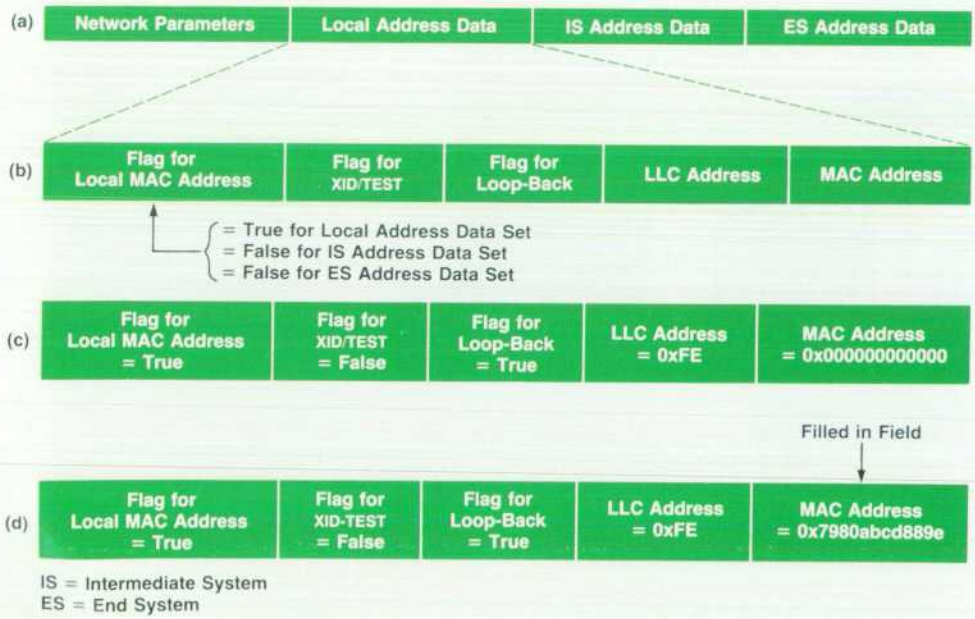


Fig. 2. *(a) Configuration data for the network layer initialization. (b) Data fields associated with each address data set. (c) Address fields of local address set sent down with DL_Add_SAP call. (d) Address fields of local address set after the MAC address is inserted.*

does the conversion and then passes the packet and pointer to the LLC sublayer using the Receive_Packet procedure.

- Return_Sent_Packet. This procedure is used by the MAC sublayer to return the data structure of the packet that the MAC sublayer has sent onto the media. The LLC sublayer will return the data structure to whatever protocol wants it back.

Design Decisions

The network layer and the LLC and MAC sublayers work together successfully because of the decisions we made to simplify the design and to minimize the amount of information each layer needed to have about the other layer. One of these decisions was that the network layer and the LLC and MAC sublayers are to return no error messages about whether or not a packet is successfully sent. This decision stemmed from trying to decide how a layer user should respond to an error from lower layers. Since these errors are characteristic of the particular lower layer in use, handling these errors could result in a great deal of dependency in an upper layer on what was going on in a lower layer, and would change if the lower layer changed (e.g., if the IEEE 802.4 MAC was replaced with IEEE 802.3).

After reviewing the functions each layer was required to provide, we realized that the transport layer had the responsibility for end-to-end communication and also that the transport layer contains algorithms for ensuring the integrity of the connection no matter how the packets are lost. Some packets transmitted with no errors will fail to arrive at their destination because of network errors on the media. It was decided to allow the transport layer to detect the loss of any packets and handle all error recovery. This relieves the transport layer from having to check status information from the lower levels on every packet.

One area we went to great length to simplify is address handling. The individual MAC address is a good example. The network layer needs, as part of its protocol, to know which of three MAC addresses (two multicast addresses and the individual address) a received packet has as its destination address. One method is to pass the individual MAC address to the network layer. This has the drawback that the network layer would have to know the format of the address and the value of the individual address. To eliminate the need for the network layer to know this information, LSAPs are set up for each set of LLC and MAC addresses the network layer might use. Fig. 2a shows the configuration data the network layer receives at initialization. The network parameters are used internally by the network layer and each of the sets of address data is used to add an LSAP for the network layer. Fig. 2b shows the data items associated with each set of address data. The network layer sets up an LSAP with the DL_Add_SAP procedure, which is in the LLC sublayer. To get the MAC address field initialized for the local address data, a call is made to the DL_Add_SAP procedure with one of the parameters pointing to the local address data shown in Fig. 2c. The DL_Add_SAP procedure examines the address data fields and if the field containing the flag for the local MAC address is true, the LLC calls the MAC sublayer routine Store_Indiv_MAC_Addr and passes to the routine a pointer to the place in the address data where the MAC address is supposed

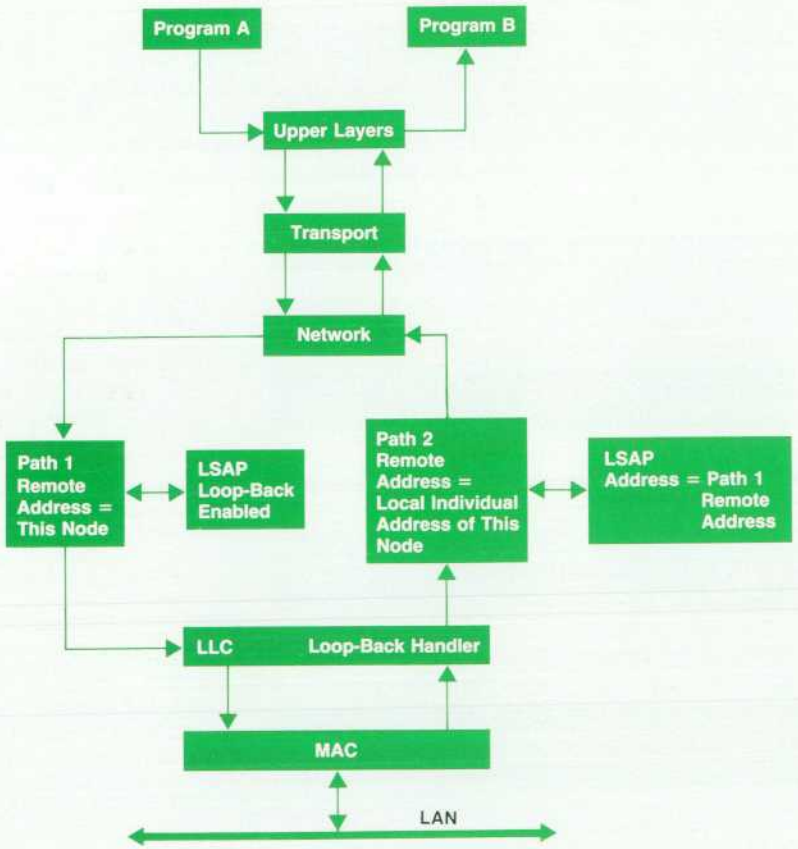


Fig. 3. *Loop-back flowchart.*

The OSI Connectionless Network Protocol

The network layer is the third layer of the OSI Reference Model. It provides network service to the transport layer and uses the data link service as provided by the data link layer. Two different types of service are defined for the OSI network layer: the connection-oriented network service using the protocol defined in ISO 8208 (CCITT Recommendation X.25) and the connectionless network service provided by the connectionless network protocol (CLNP) defined in ISO 8473. The OSI Express card relies on a LAN subnetwork technology, so it uses the connectionless network layer protocol. The OSI Express card also uses the end-system-to-intermediate-system routing exchange protocol defined in ISO 9542 to discover the existence of other end systems or the existence of one or more intermediate systems on the same subnetwork (LAN segment). An end system (ES) is defined as a system in which there is a transport entity in an instance of communication. An intermediate system (IS) is a system that provides the routing and relaying functions of the OSI network layer. End systems rely on intermediate systems to deliver network protocol data units (NPDUs) from the source ES to the destination ES across multiple subnetworks.

Service Provided by CLNP

The connectionless network service (CLNS) provides a datagram service to the transport layer. Each NPDU contains the source and destination end system addresses, and is routed to the destination as an autonomous unit (i.e., not associated with any connection between the end systems). The CLNS may misorder, duplicate, or lose packets. Therefore, it is up to an upper-layer protocol, such as the transport layer, to perform error checking.

The connectionless network service provides only two service primitives to the transport layer: an N-UNITDATA request and an N-UNITDATA indication. The transport layer initiates the transmission of a TPDU or TPDUs by issuing an N-UNITDATA request. The transport layer receives TPDUs via the N-UNITDATA indication. The parameters of both CLNS primitives are the network source address, the network destination address, the network quality of service, and the network service user data.

The source address and the destination address parameters are OSI network service access point (NSAP) addresses. An NSAP has two parts: the network entity title part which uniquely identifies the ES or IS within the global OSI environment, and the selector part which identifies the network service user within the ES.

ES-to-IS Exchange Protocol

The ES-to-IS routing exchange protocol, which is specified in ISO 9542, provides solutions to the following practical problems.

- How do end systems discover the existence and reachability of intermediate systems that can route NPDUs to destinations on subnetworks other than the ones to which the ES is directly connected?
- How do end systems discover the existence and reachability of other end systems on the same subnetwork?
- How do intermediate systems discover the existence and reachability of end systems on each of the subnetworks to which they are directly connected?
- How do end systems decide which intermediate system to use to forward NPDUs to a particular destination when more than one IS is accessible?
- The ES-to-IS protocol is connectionless and operates as a protocol within the network layer, specifically in conjunction with the CLNP, ISO 8473. The ES-to-IS PDUs are carried as user data in data link PDUs just like ISO 8473 NPDUs. Certain ES-to-IS protocol functions require that the subnetwork (i.e., data link service) supports broadcast, multicast, or other forms of multidestination addressing for n-way transmission.

to be. When this routine is finished the local address data looks like Fig. 2d. The availability of the Store_Indiv_MAC_Addr ensures that the LLC does not have to know what the MAC address is or where it is stored. When control is returned to the LLC sublayer, it uses the modified address data buffer to add an LSAP just as if the MAC address had been supplied when DL_Add_SAP was initially called.

The network layer does not have within its protocol the concept of XID and TEST commands or responses. Either the network layer must detect and reject these packets or the LLC must not send them to the LLC user. Some LLC users do want to receive these packets. To prevent the network layer from having to check the LLC control fields of every packet, special flags were added to the LSAPs for XID and TEST packets. When the LSAP is activated, the network layer designates that the XID and TEST flags be set to prohibit the reception of these responses at this particular SAP. LLC users that do want to receive XID and TEST packets would not set these flags.

Loop-Back

Loop-back is the process by which the card is able to receive or appear to receive something it has sent. Loop-back is often used for testing, but it is also required for the normal operation of the card. If two programs that are written to communicate with each other over the network are run on the same machine, loop-back is necessary for them to communicate with each other. A data packet from either of these programs must travel the entire protocol stack because some of the layers of the network provide services such as data transformations as well as transporting the packet from one program to the other. Another reason for traversing the entire stack is that the card cannot know whether a packet being sent is also one that the card should receive unless the entire address of the packet is evaluated. The task of loop-back, that is, the process of generating a receive packet from a packet being sent, is the responsibility of the LLC sublayer in this implementation.

The network layer does not want all packets looped back to itself. For instance, if all packets sent out with one of the multicast addresses as the destination address were looped back, the network layer would be burdened with spurious packets and would have to check each packet's network address to be sure it was not one it had sent. Since one possible error in a network is for two network layers

to have the same network layer address, even the detection of unwanted looped-back packets could be impossible, since the MAC individual address, which would decide the issue, is not available to the network layer. The solution is to have a loop-back flag in the LSAP data structure. When the loop-back flag is set, the LLC knows that packets sent on a path using the LSAP should be looped back if the remote address of the path is the one on which the card receives packets.

The data flow of a loop-back packet is shown in Fig. 3. The packet is sent from program A to program B. Program A sends the packet down to the upper layers just as it would send a packet to a program on another node. From there it is sent to the transport layer and then the network layer. The network layer sends the packet to the LLC using path 1, which has its remote address set to the node of program B. In this case since program B is on the same node, the remote address is the one on which the local node itself receives packets. The LLC sublayer sends the packet to the MAC sublayer where it is sent out onto the network. (Loop-back packets are also sent out onto the network because the remote address can be one that other nodes also receive.) The MAC returns the packet to the LLC after it is sent. The LLC checks to see if the packet is a loop-back packet. Since it is, the LLC starts the packet up the stack via path 2, which has as its remote address the local address of the original packet. The LSAP associated with path 2 has as its address the destination address of the original packet. The network layer receives this packet the same way it would if it came from another node. The packet is then passed up the stack to program B.

Rather than do a full LLC and MAC address comparison each time a packet is returned from the MAC sublayer, a flag in the path is tested. This flag is set when the path is set up, based on whether the LSAP associated with the path allows loop-back and whether the remote address of the path is one on which the node receives packets. This flag must be updated each time an LSAP is added or deleted. Since LSAPs are usually added at initialization and never deleted, the updating does not add any overhead to the card's operation.

The checking of a path's remote address against addresses that are active in the LLC and MAC sublayers is done by a method that maintains as much independence between the two sublayers as possible. The LLC sublayer uses the MAC procedure Check_MAC_Addr to check a remote address. The MAC sublayer returns a flag that indicates whether or not the address is an active MAC address. Thus, the LLC does not have to know the format of the MAC address or how it is stored in the MAC sublayer. If the MAC address is active, the LLC checks its own LSAPs to determine if one of them will accept the remote address of the path as a legitimate destination address.

LLC and MAC Testing

Once the LLC and MAC interface design was completed, testing became the next critical issue. The OSI Express project required that the MAC interface software be one of the first functional modules on the OSI Express prototype card. A high percentage of its functionality had to be very reliable so that code for the LLC and other layers of the OSI stack could begin to run on the card. Since the prototype card was not immediately available, another method of testing had to be developed to make immediate progress. The scenario interpreter and test harness environment had already been developed for the HP 9000 Series 300 HP-UX environment, so we decided to leverage the tools from this existing testing environment. The scenario interpreter is a software test tool that handles the sending and receiving of data packets to and from the software under test, and the test harness enables testing in different environments. Both of these test tools are described in the article on page 72. Testing the MAC interface in the scenario interpreter and test harness environment also allowed the LLC and other modules that have interfaces to the MAC software to exercise this interface without writing special test code. It was also necessary to be able to do a majority of the debugging in the friendly HP-UX environment. Since the Motorola 68824 token bus controller chip (TBC) had been previously tested and had proven to be reliable, it was decided that the TBC could be emulated, thereby avoiding the need to wait for the hardware prototype to be ready.

As shown in Fig. 4, the MAC interface testing environment used the existing scenario interpreter and its scenario syntax and the existing test harness. In place of the generic bounce-back module, a special MAC interface bounce-back module was written. The generic bounce-back module is used by any module that needs to make it look as though it is receiving data packets from the layer below it. It takes the data transmitted to it and calls the receive routine of the layer configured above it. The MAC interface could not use this module because there is no layer below it and so special code had to be written in the emulator. In a typical testing instance, the scenario interpreter reads a scenario that tells it to send a specific amount of data to the configured layer. The test harness reads the data, which eventually gets sent to the LLC sublayer. The LLC puts its header on the data packet and calls the MAC module. The MAC module prepares all the data structures needed by the TBC

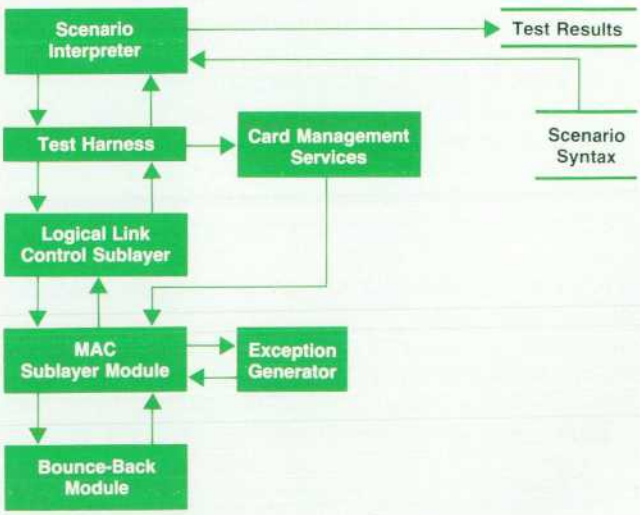


Fig. 4. *MAC interface software test environment.*

and transmits the packet. The special MAC interface bounce-back module is then called. This module performs the tasks that the hardware and the TBC normally perform; it sets status in the packet to make it appear that it has been transmitted onto the network and copies the information in the transmitted packet into buffers in the inbound buffer pool to make it appear that a packet has been received from the network. It then causes a receive packet interrupt, which causes the MAC code responsible for receiving the packet to be invoked. The transmitted and received packets are processed and forwarded to the LLC software as though the code was running on the OSI Express card. When the received data reaches the scenario interpreter, the interpreter compares it to the data that was sent and saves the results of the comparison in the test results file.

Conclusion

The network layer and the data link layer with its LLC and MAC sublayers provide the network layer user, the transport layer, with the ability to send a packet efficiently to any accessible node given just the network layer address. The network layer locates the destination node even if it is not on the local area network. The LLC separates packets it receives that are for the network layer from those that are for other data link layer users on the OSI Express card. The MAC sublayer provides an interface to the media that is independent of the media. This achievement was accomplished by adherence to international standards and a design that minimizes the dependencies of the protocols upon each other's internal operations.

Acknowledgments

We would like to acknowledge and thank the card and chip hardware design team which consisted of Mike Perkins, Mark Fidler, Paul Zimmer, Alan Albrecht, Dan Dove, and Nancy Mundelius. Mike Perkins and Mark Fidler were also vital in the early debugging and testing of the TBC chip. Mike Wenzel provided vital insights on how to incorporate the data link layer into the CONE environment. Curtis Derr provided a ROM version of the LLC/MAC and TBC interface software which is used with the hardware diagnostic program. He also coordinated the COS (Corporation for Open Systems) testing of the data link layer. Special thanks to Motorola's technical support staff, especially Rhonda Alexis Dirvin, Paul Polansky, and Robert Odell, who provided excellent technical support of the TBC.

HP OSI Express Design for Performance

Network standards are sometimes associated with slow networking. This is not the case with the HP OSI Express card. Because of early analysis of critical code paths, throughput exceeds 600,000 bytes per second.

by Elizabeth P. Bortolotto

PERFORMANCE ANALYSIS of the HP OSI Express card began during its early design stages and continued until the product was released. During the course of the project several different analysis techniques were applied. These included simple analytic modeling, path length estimation, simulation, and prototype measurement. Several tools were developed to make the prototype performance measurements. Many estimations of throughput and delay were made during the development phases of the OSI Express project. These intermediate results led to redesign or code reduction efforts on the bottlenecks in the software.

In the end, we far exceeded our initial performance expectations. Early performance investigation was invaluable in pinpointing potential bottlenecks when there was still time to make design changes. We learned that the most fertile areas for performance enhancement and code path reduction are usually in module redesign, not code tuning.

Static Analysis

The earliest OSI Express performance activity was to estimate the amount of code in "typical" inbound and outbound data paths. A typical inbound data path was defined as the code executed when a data packet is received from the LAN going to the host service. For this estimate, it was assumed that the packet arrives without errors. Some assumptions were also made about what processing was typical or most common. These assumptions were periodically revised as we learned more about the system.

Once the path estimates were derived, throughput and delay measurements could be obtained. This process was referred to as static analysis because the statistics obtained were best-case and worst-case estimates without any reference to how a dynamic system behaved. The static analysis process derived these statistics by comparing the number of CPU (and DMA) cycles required by a single packet to the total number of cycles available in the hardware.

The first path measurements were made in units of 68020

assembler instructions. An early analysis revealed that using ten CPU cycles per assembler instruction was a fairly safe (and usually conservative) estimate. This was true unless the design engineer used a number of multiply or divide instructions. In fact, early analysis showed the high cost of these two instructions and steps were taken to avoid using multiplies and divides unless necessary.

Because the earliest performance estimates were attempted before much code was written, it was necessary to study each software module carefully to understand all of the tasks that the software would be required to perform. The typical paths (inbound and outbound) could then be roughly pseudocoded. A second analysis during this time revealed that the C compiler on the development systems typically generated three to four 68020 assembler instructions per line of simple C code. A simple C code line was defined as a line in which only one operation is performed. Therefore, if a line of C (or pseudo C) was

```
a = (b & a) | (c << d);
```

it was estimated as four simple C instructions and therefore twelve to sixteen 68020 assembler instructions.

The inbound and outbound paths were estimated separately because independent estimates for each path were needed to understand the complete set of tasks necessary to transfer a packet from one node to another. The two parts are not the same length. We expected to find the inbound path longer (in terms of instructions) than the outbound path.

Once the estimation had been completed, the number of assembler instructions in both paths was multiplied by 10 (ten cycles per 68020 instruction). The result is the number of processor cycles used in transmitting and receiving one typical data packet by the OSI Express card. Since the basic hardware architecture of the OSI Express card was in place, it was relatively easy to estimate the maximum possible throughput and minimum possible delay. The following is an example of a static analysis throughput equation for the OSI Express card.

Throughput in bytes per second =

$$[TC/(PW(RC + WC) + IC)](P - H)$$

where TC = total available cycles per second
PW = size of the packet in words (16 bits)
RC = number of cycles per read access
WC = number of cycles per write access
IC = number of instruction cycles in receive data path
P = packet size in bytes
H = header size.

Some of these values were slightly variable. Average or typical values were often used, and care was taken to estimate conservatively.

First Path Estimation

During the course of the OSI Express project, two complete data path estimations were made. The first estimate was made during the design phase, before much coding had begun. The second estimate was made after most of the code had been written.

The first code path length estimate was done while the project was in the early design phase. Only a portion of the code was written. To get the path length for the code that was written, a mixed listing of the code was obtained. A mixed listing in this case was an assembled listing of the 68020 instructions intermixed with the original C instructions. The data path was then identified and the assembly instructions counted. In addition to giving us the instruction count, this exercise also educated us on how the C compiler was behaving and what sort of assembly code was generated.

As discussed before, most of the code was written at the time of the first path length estimation when most of the development engineers were working on their external designs. The estimation method used was to read the ISO specifications for each layer and the ERS for CONE (common OSI networking environment), and write pseudocode for the data path. The pseudocode was then translated into

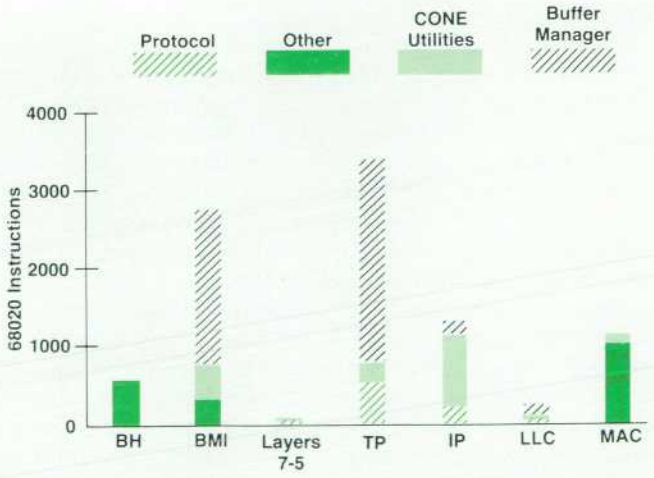


Fig. 1. *First-estimate (early design phase) OSI Express card instruction count summary for 1K-byte packets outbound to the network.*

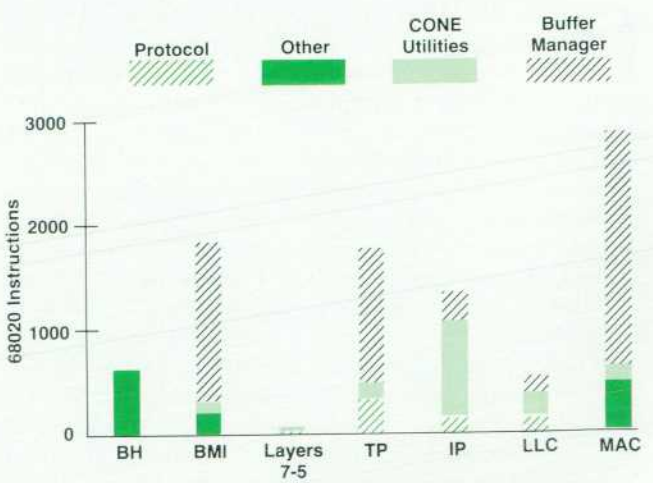


Fig. 2. *First-estimate OSI Express card instruction count summary for 1K-byte packets inbound from the network.*

68020 instructions using the the multiplier factors discussed above. This entire process took about six months.

Figs. 1 and 2 show the results of this first estimation process. Fig. 1 displays the number of instructions in the outbound data path and Fig. 2 displays the number of instructions in the inbound data path. The graphs show that the largest code segment in the data path at that time was the memory management code. We therefore decided to redesign the memory manager code to reduce the number of instructions in the most common data path.

A number of smaller code changes were also made as a result of this first performance investigation. Redundant instructions, excessive multiplies, unnecessary initialization, and more streamlined code processes were identified. In addition, the team learned more about code modules that were influenced by decisions in distant code modules.

Second Path Estimation

The second estimate was made after the code was basically written but before much unit testing had been done. This estimate was quite a bit quicker because there was no pseudocoding to do. In addition, the data path was pretty well understood by this point. Therefore, mixed listings of all the code modules (and protocol layers) were obtained and a walkthrough of the data path was performed. Again, care was taken to be as accurate as possible, since the performance statistics resulting from the code count were only as good as the data.

Code was counted for both the inbound and the outbound data paths. By the time the second count was made there had been a number of design changes and developments. Figs. 3 through 6 show the results of these changes. The backplane handler code had exploded into a much larger module than was initially expected. This module then became the primary target of a performance redesign effort.

As before, a number of performance opportunities were identified as a result of the second walkthrough. In addition, we learned more about how the OSI Express card would behave when parameters were varied in the FTAM, IPC, and CIA host code.* Several changes were suggested to the designers of these modules. In one case, we found that performance was severely impacted during file transfers when the data was presented to the OSI Express backplane in 256-byte buffers instead of kernel clusters (2K-byte buffers).

*FTAM = File Transfer Access and Management. IPC = Interprocess Communication. CIA = CONE (Common OSI Networking Environment) Interface Adapter.

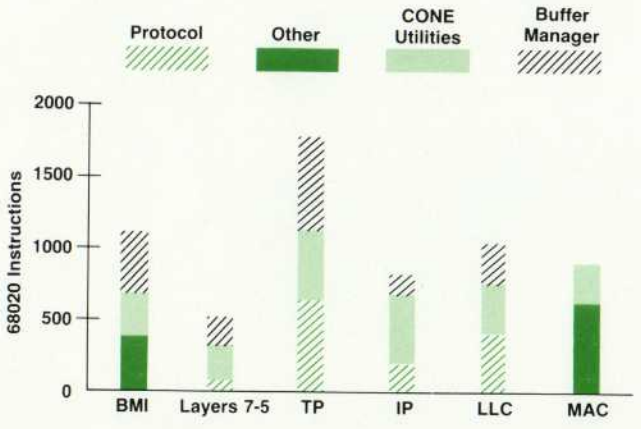


Fig. 3. *Second-estimate (after coding and several design changes) instruction count summary for 1K-byte packets outbound to the network.*

Connection Establishment Path

In addition to the common data path, the connection establishment path was also analyzed during the OSI Express performance investigation. This analysis was made a little later in the project after the second path estimate had been completed. For the sake of speed, this path was counted in lines of simple C. By this time we had gained quite a bit of confidence in our estimation method and in our knowledge of the code processes. This estimation took much less time than the other two.

It was discovered that the amount of code required to secure a connection was quite a bit larger than that required to send or receive a data packet. Of course, we knew that this was true before even beginning the connect path analysis. We just did not know how large it was. Our investigation showed us that the connect code path was 91,424 lines of C code (simple) in a typical case. In other words, it would take approximately 366 milliseconds for a connect to complete successfully. (We assumed four 68020 instructions per C instruction).

It was also discovered that the connect path provided many opportunities for path reduction. Once a particular code path is fully understood, performance opportunities are usually obvious. This was definitely the case in this analysis and both of the previous path estimation exercises.

Benefits of Early Performance Walkthroughs

There are a number of benefits to performance analysis during all of the phases of new product design. The benefits far outweigh the cost of the additional engineer (or two) if one of the project goals is good performance. The benefits are obvious when path analysis reveals code redundancy or other time-saving opportunities. Other benefits that provide big paybacks may not be so obvious. The following is a list of the less obvious benefits we found during OSI Express performance analysis.

- Design inconsistencies were exposed.

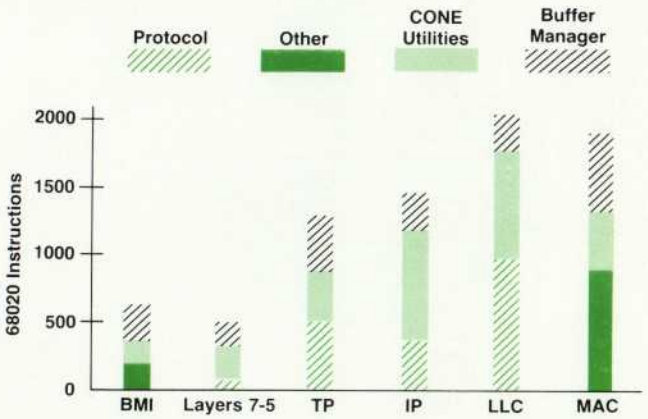


Fig. 4. *Second-estimate instruction count summary for 1K-byte packets inbound from the network.*

- The design engineers became performance conscious and wrote cleaner code.
- There was time for redesign of bottleneck areas.
- We became much more proficient in performance analysis. Future products benefit from this kind of education.

Simulating Flow Control

The second major step in the OSI Express performance study was to create a simulation model to aid us in discovering how configurable parameters in the OSI Express stack affected performance. The static or path flow analysis that was discussed above had yielded best possible throughput and delay statistics. In other words, the static analysis had given us an idea of what the upper performance bounds were, given our code paths. What quickly became apparent was that it was quite improbable that we could achieve these upper bounds unless the card was configured with optimal parameters and all other conditions were perfect. Fig. 7 shows the difference in throughput when only one parameter (packet size) is varied.

The reason that packet size plays such a substantial role in throughput is that it takes approximately the same amount of work to process an 8K packet (the maximum packet size allowable by the IEEE 802.4 standard) as it does a 1K packet. At least this is true if the memory management design is optimal for fast throughput.* Larger packets generally require more CPU cycles to process (for memory copies, DMA transfers, checksum operation, etc.). However, the difference in the cycles required to process two packets of different sizes is proportionally smaller than the difference in the number of bytes transferred. Additionally, processors with cache memories can minimize the difference in the CPU overhead between large and small packets because copies and checksum operations are repetitive looping functions.

*Sometimes memory management designs are optimized for efficient memory use at the expense of fast throughput. In the OSI Express project, we attempted to optimize for both.

Transport Layer

The OSI transport layer (layer 4) is the layer where the packet size is determined. Other transport parameters also have values that can dramatically influence system throughput and delay. The parameters that govern the flow of data from one node to another were the major topics of our simulation study.

The transport layer parameters have significant impact on the communication performance of a network node. The flow control algorithm in the transport layer is responsible for the dynamic end-to-end pacing of conversation between two nodes. Its main purpose is to ensure that one node does not send data faster than another node can receive it. Given two connected nodes, one node will usually be able to execute faster than the other. The best throughput between these two nodes is achieved when the slowest node is kept completely busy. If the flow control algorithm allows the slower node to become idle, throughput will be lower than its potential maximum. If the flow control algorithm allows too much data to be sent to the slower side (usually the receiving side), the slow side will eventually be filled to capacity and be unable to accept more data. This results in lost data, which must be resent. Resending data also causes performance degradation.

The flow control algorithm usually has a number of parameters that can be set by the system manager. These parameters are available so that the algorithm can be tuned to provide the best performance in a specific user environment. Some of the parameters at the transport layer include the transport segment size (the maximum amount of data in each packet), the transport window size (the maximum number of packets that can be sent at one time), the amount of credit to extend to a peer, the frequency of acknowledgment packets, and the length of the retransmission timer.

Simulation Model

The simulation model of the OSI Express card was written in a language called PC Simscript II.5. It was primarily designed to expose and isolate the dynamic elements of

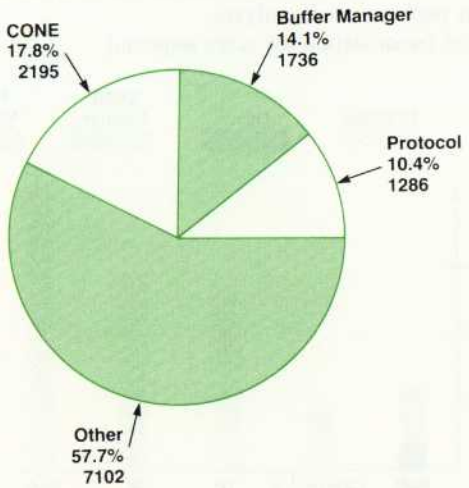


Fig. 5. *Second-estimate code breakdown by module for 1K-byte outbound packets in number of instructions and percentage of total.*

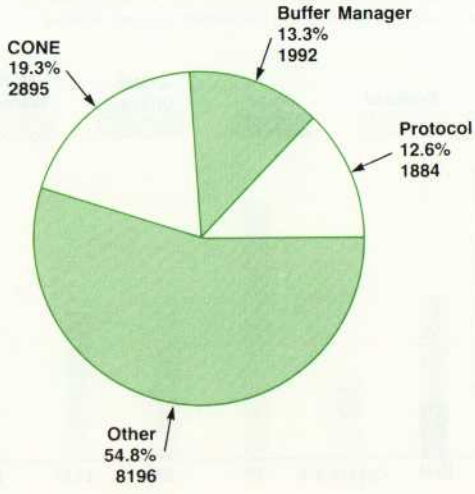


Fig. 6. *Second-estimate code breakdown by module for 1K-byte inbound packets in number of instructions and percentage of total.*

the OSI Express system. Therefore, the transport layer, the backplane message interface layer (because of the segmentation capability at the backplane), and the CONE scheduler were simulated in great detail. The upper layers (ACSE, presentation, and session) were not really simulated at all because they do very little processing for a data packet. Instead the simulation merely "worked" for the amount of time that the upper layer headers would typically require for processing.

The simulation model was specifically designed to allow a user to vary parameters, getting a performance report at the end of each simulation run. The idea was that the simulation would help the OSI Express team define which parameter values gave the best throughput and delay values and why.

A number of assumptions were made in the simulation model that are not necessarily true in the actual OSI Express system. The reason for these simplifying assumptions is that they streamlined the simulation implementation and facilitated the experimentation process. Since the simulation was written to isolate dynamic behavior, details that might obscure or complicate the simulation were ignored. Although the system representation had been simplified extensively, an attempt was made to be meticulous in simulating those parts of the real system that have an impact on dynamic behavior on the OSI Express card. To a large extent, the art of simulation is knowing what *not* to simulate.

The following is a list of the major assumptions made during the design of the simulation program:

- All packets arrive in order and without error.
- All data transmissions from the host contain the same amount of data for all connections (the amount of that data is a parameter).
- Since packets are never lost, no retransmission timers or AK* delay timers are included in the transport simulation.
- The two target nodes transmit all data at the highest priority level (IEEE 802.4 specifies four priority levels: 0, 2, 4, and 6).
- There is no simulated connect setup or tear-down time. The assumption is that connections are fully established before the data is sent to the card.
- All packets sent onto the simulated network are either data packets or AK/credit packets. None of the routine features in the internet protocol are simulated. Consequently, there are no end-system or intermediate-system hello packets to contend with.
- The packet headers are 80 bytes long.
- Card memory is a user-configurable parameter. However, the inbound packet data memory is assumed to be half of the total data memory. The outbound data packet memory is also assumed to be half of the total data memory.
- The host data can be sent to the card faster than the card can consume it. Also, on the receiving side, the host can consume the data faster than the card can send it. In short, the host is assumed to be an infinitely fast source and sink.
- The maximum speed of the token bus is 10 Mbits/s. An assumption is made that the speed with which packet data can travel is 1 Mbyte/s. This is because there is overhead for the IEEE 802.4 protocol that prevents the data packets from traveling much faster.

*AK = Acknowledgment packet

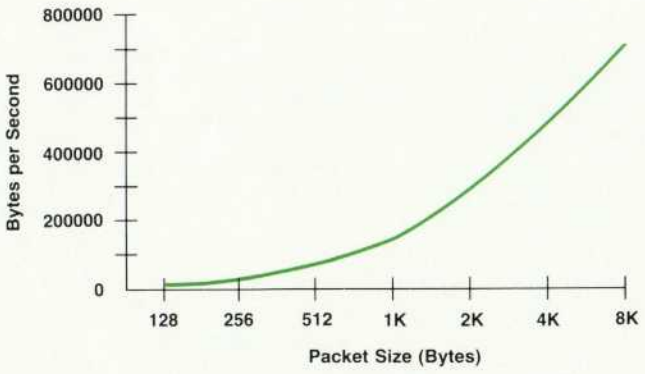


Fig. 7. *Throughput versus packet size for a window size of 10.*

Simulation Model Features

The simulation model has a number of features that increase its usability. The model can be run in either half-duplex or full-duplex mode. In half-duplex mode, one of the two communicating nodes is a sender and one is the receiver. In full-duplex mode, both nodes send and receive simultaneously.

The model has the capability of varying four parameters automatically and running a complete simulation for each value of the parameters. Each of the four parameters can be given a range of values and a step size to vary. Statistics are collected for each of the simulation runs and saved in a file.

The model allows the communicating nodes to have a number of connections alive at the same time. In this mode, the model can calculate statistics for each connection, as well as global statistics.

The model has various debugging levels that can be turned on to enable the user to understand better what is happening during a simulation run.

There is a separate default parameter generator program that enables the user to specifiy default parameters easily. The generator program then creates a default file that is used by the simulation program.

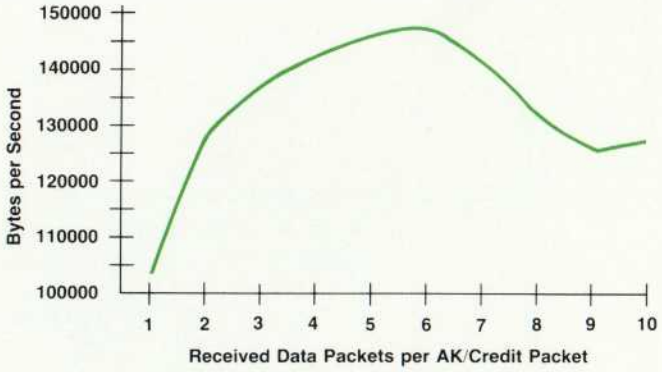


Fig. 8. *Throughput versus credit frequency for a packet size of 1K bytes.*

The simulation model generates and saves a number of useful statistics during execution. These are formatted and saved in a file for later examination. Some of these statistics are:

- Throughput in bytes per second
- Total simulaton delay in bytes per second
- Mean packet delay in milliseconds
- Maximum packet delay in milliseconds
- Mean transport-to-transport delay in milliseconds
- Mean acknowledge delay in milliseconds
- Maximum acknowledge delay in milliseconds
- Average interval time between packets, in milliseconds
- Total interval time between packets, in milliseconds
- Percentage of CPU idle time
- Maximum and minimum queue depths for five system queues.

The simulation model has a very friendly user interface to simplify the selection of the system parameters. In addition, the user interface displays the parameters obtained from the default file and allows the user to change them if necessary.

Simulation Study Results

Once the simulation was written and verified (by hours of painstaking cross-checking) a number of simulation experiments were run. Time and space prevent describing all of the results except the most interesting: what happens when the transport window size and the frequency of sending AK/credit packets are varied.

Figs. 8 and 9 show the impact of varying these two parameters in the simulated system. Each of the data points in these graphs represents one complete simulation run with a particular set of parameter values. To get Fig. 9, the window size was set to 10 and the packet size fixed at 1K bytes. For the sake of simplicity, it was assumed that incoming packets were only acknowledged when it was time to send more credit (permission for the transmitting node to send more packets) to the peer node. The frequency of sending credit packets was varied from one to ten. In other words, during the first simulation run, the window size was ten and the receiving node sent out an AK/credit packet to the sending node after each packet was received and processed. The AK/credit packet acknowledged the packet that was received and gave the sending node permission to send another. During the next simulation run, the credit frequency parameter was set to two. An AK/credit packet was therefore sent after two packets had been received by the receiving node and processed. In this case the receiving node acknowledged reception of two packets and gave permission to send two more.

As shown above, the best throughput value is achieved when the receiving node sends an AK/credit packet every sixth packet. This point represents a balance between sending AKs too frequently and not sending them frequently enough to keep the system fully pipelined. If too many AKs are sent, they effectively increase the CPU overhead required to process packets (Fig. 10). That is because the number of instructions required to construct and send (and receive) AK packets is significant.

On the other hand, if AK packets are not sent frequently enough, the sending node will run out of packets to send and will have to wait for an AK before starting to send more (the window size limits how many packets can be sent without an AK). When the sending side stops sending packets (even for a short while), interarrival time between incoming packets at the receiving node will, in general, increase.

Dozens of simulation experiments were run during the course of the OSI Express project. The flow control parameter defaults were set based on the information from the simulations. In addition, we learned a great deal about the behavior and resulting statistics of the transport stack. Some design decisions were changed based on the results of the experiments. For example, we decided not to give priority to inbound packets by allowing a logical link control (LLC) process to execute until all the receive packets were processed. We found out to our surprise that the simulated throughput dropped sharply when we experimented with this design. The reason was that the AK/credit packets were being excessively delayed and the queues between the transmitting and receiving node were therefore emptying.

Performance Measurement

The final challenge of the OSI Express performance project was to measure the product, compare the measured performance with the estimates, and identify any bottle-

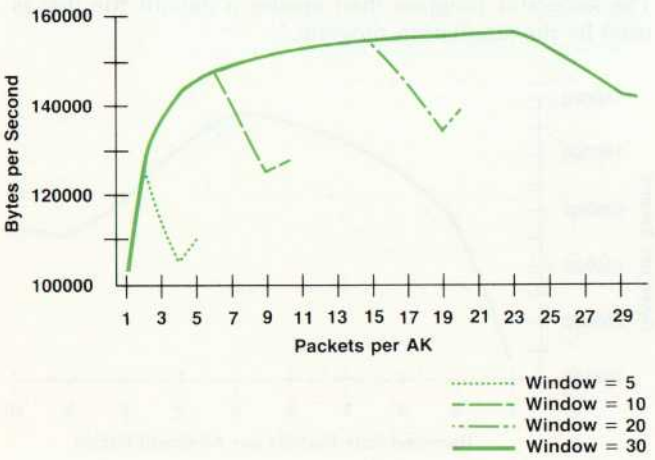


Fig. 9. *Throughput as a function of window size and packets per AK for a packet size of 1K bytes.*

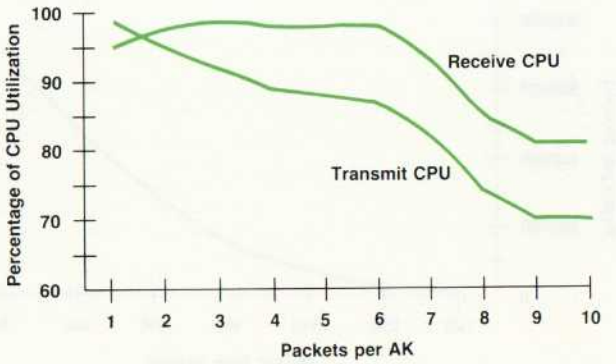


Fig. 10. *CPU utilization versus credit frequency for a packet size of 1K bytes.*

neck code modules. Several tools were designed and written to help us get real-time performance measurements. These tools were basically designed solely for prototype measurement, not for field or customer use. In the following paragraphs, three tools are briefly described. These tools are the real-time procedure tracer, the statistics monitor, and the statistics formatter.

Real-Time Procedure Tracer. This tool consists of a special entry and exit macro call that was put after the entry and before the exit of every procedure in the OSI Express code. Each module in the OSI Express code was assigned a hexadecimal number range. The designers of each module then assigned an even number within that range to each routine in the module. A second value (1 + even number) was reserved for the exit macro. These numbers were passed as parameters in the macro calls. Both the entry and the exit macros caused the passed hexadecimal value to be written into a reserved memory location called CISTERN. The idea is that using a logic analyzer (such as the HP 64000, HP 1630, or HP 1650), a user can trace writes to the CISTERN location and see the procedures being executed in real time. The hexadecimal value ranges assigned to each module allow the user to limit the values read to a specific number range. This way, the user can choose to see only the transport layer executing, if desired.

To make the traces more readable, a formatter program was written for HP 64000 trace files. The formatter required a file that defined the hexadecimal values for specific procedures. It then produced very readable formatted traces. Fig. 11 is an example of one of these formatted traces.

The procedure traces were used extensively once integrated OSI Express code measurements could be made. These traces allowed us to see how long each module was executing in as much detail as we cared to see. Code could be quickly tuned and remeasured. In addition, the trace macros were optionally compiled, ensuring that they did not provide needless overhead in the final product code.

Statistics Monitor. A second tool that was designed into the OSI Express stack was the statistics monitor. A number of primitive statistics are kept in the OSI Express code (see Fig. 12). In addition, statistics are kept (these optionally compiled) about each of five major queues in the OSI Express system (see Fig. 13). These statistics can be retrieved and displayed upon command. The statistics can be cleared, read, or read and cleared. The clear command clears all of the statistics except for current-value statistics such as the current queue depths.

These statistics made it possible to get real-time throughput values at at the card level. In addition, the queue statistics provided some troubleshooting capability because certain queue depths signaled flow control problems.

Statistics Formatter. The OSI Express statistics formatter is a tool designed to allow a user to run a user-level test program a number of times automatically, varying OSI Express parameters each time. The transit statistics are cleared at the beginning of each test run and sampled at the end.* The purpose of this tool was to find the optimal parameter set automatically on the working prototype. The simulation model had this basic capability, so in effect, we were simulating the simulation model.

Once all of the test program runs have executed, the formatter can retrieve the file with the statistical samples and display the results in several ways. Fig. 14 is an example of one of the types of displays that can be obtained. The user now has the opportunity to identify, for example, the highest throughput obtained when packet size is varied because the test program was repeated for several possible packet sizes.

Performance Results

After the OSI Express prototype testing had been completed, final performance measurements were made. Of course, numerous performance values are possible, depending on how the card is configured. However, our best-case throughput for 8K packets was approximately 600,000 bytes per second.

This result reflects numerous redesign, code rewrite, and code tuning efforts made by the whole team during the entire lab prototype phase of the project. Many mil-

*Actually, there are transient start-up and cool-down pipelining effects that tend to distort the sample. To prevent distortion, the statistics are cleared after the start-up transient has died out and sampled before the cool-down transient begins.

```
# call BH_get_data for the outbound data
#
#
# BH_get_data entry
#
ENTER BH_GET_DATA_p (0x1204)
  ENTER BH_QUEUE_IRS_F_p (0x125c)
  EXIT BH_QUEUE_IRS_F_p (0x125c) gross 12.40 uS, net 12.40 uS
  ENTER BH_MAIN_ISR_F_p (0x1290)
    ENTER BH_MPX_F_p (0x1284)
    EXIT BH_MPX_F_p (0x1284) gross 61.60 uS, net 61.60 uS
    ENTER BH_PROCESS_IRS_F_p (0x128c)
      ENTER START_REQ_p (0x1274)
      EXIT START_REQ_p (0x1274) gross 38.40 uS, net 38.40 uS
      ENTER START_DMA_READ_p (0x124c)
        ENTER END_DMA_READ_p (0x1254)
        EXIT END_DMA_READ_p (0x1254) gross 56.80 uS, net 56.80 uS
      EXIT START_DMA_READ_p (0x124c) gross 146.32 uS, net 89.52 uS
      ENTER CONTINUE_REQ_p (0x127c)
        ENTER DO_QUAD_FETCH_p (0x1268)
        EXIT DO_QUAD_FETCH_p (0x1268) gross 91.20 uS, net 91.20 uS
        ENTER DO_DMA_CMD_p (0x1278)
          ENTER DO_CCMD_LINK_FN_p (0x126c)
          EXIT DO_CCMD_LINK_FN_p (0x126c) gross 63.20 uS, net 63.20 uS
        EXIT DO_DMA_CMD_p (0x1278) gross 147.00 uS, net 83.80 uS
      EXIT CONTINUE_REQ_p (0x127c) gross 313.50 uS, net 75.30 uS
      ENTER REQ_COMP_OUT_p (0x1270)
      EXIT REQ_COMP_OUT_p (0x1270) gross 33.70 uS, net 33.70 uS
      ENTER DO_QUAD_FETCH_p (0x1268)
      EXIT DO_QUAD_FETCH_p (0x1268) gross 16.30 uS, net 16.30 uS
    EXIT BH_PROCESS_IRS_F_p (0x128c) gross 709.72 uS, net 161.50 uS
    ENTER BH_MPX_F_p (0x1284)
    EXIT BH_MPX_F_p (0x1284) gross 18.80 uS, net 18.80 uS
  EXIT BH_MAIN_ISR_F_p (0x1290) gross 853.14 uS, net 63.02 uS
EXIT BH_GET_DATA_p (0x1204) gross 970.84 uS, net 105.30 uS
#
# BH_get_data exit
```

Fig. 11. *An example of a formatted procedure trace.*

Current Counters and Queue Depths

Number of Open Connections:
Global Retransmissions:

	Bytes	Packets	Throughput
Frontplane Transmissions to Host:			
Frontplane Transmissions to Network:			
Backplane Transmissions to Host:			
Backplane Transmissions to Network:			

Current Queue Depths

Number of Messages in Backplane Queue (to Host):
Number of Packets in Frontplane Queue (to Host):
Number of Packets in Frontplane Queue (to Network):
Number of Packets in Retransmission Queue:
Number of Packets in Transport Segment Queue (to Network):
Number of Tasks in Scheduler Queue:

Fig. 12. *Primitive statistics kept in the OSI Express code.*

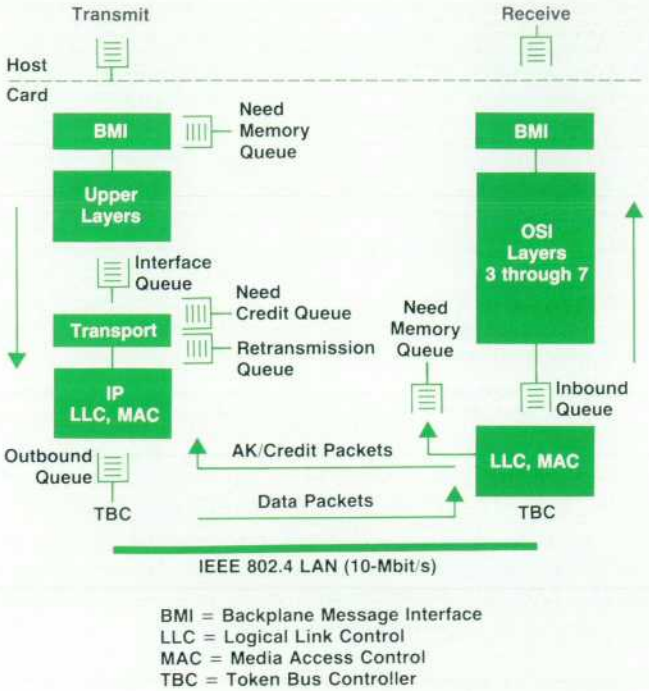


Fig. 13. *Locations of the five queues in the simulated system on which statistics are kept.*

liseconds were cut out of the code path based on information uncovered by these investigations. The majority of these improvements were made well before most code tuning efforts began. There is no way that the same code reductions could have been made after the code had been integrated.

Conclusion

Early performance investigation and prediction is vital to performance sensitive projects, especially if they are large and involve a number of design engineers. A large amount of very useful data can be retrieved with very little investment if it begins early enough in the project and continues through code integration. Full performance investigations should be a part of every product life cycle.

TMF Table

Statistic		Functional Unit	
Current Counters & Queue Depths			
0) Thruput-BP Bytes in [B/s]	308593	114988	
1) FP Packets Out	44	18	
2) LFP Bytes Out	1684	1077	
3) FP Packets In	40	27	
4) FP Bytes In	5443	6673	
5) BP Packets Out	9	5	
6) BP Bytes Out	354	298	
7) BP Packets In	27	24	
8) BP Bytes In	4140	5164	
9) Number of Timer Ticks	0	0	
10) Timer Tick Value [ms]	0	0	
11) Number of Global Retrans	0	0	
12) Number of Open Connections	1	1	
13) Cur. IFACE Queue Depth	0	0	
14) Cur. Retrans Queue Depth	1	1	

CTRL Graph# (0, 0)
Parameters Dump Exit Up Down Left Right X# Y# Max: (0, 4)

Fig. 14. *One of the types of displays produced by the OSI Express statistics formatter.*

Acknowledgments

Many thanks to Mike Wenzel at Roseville Network Division and Martin Ackroyd at HP Laboratories in Bristol for their expert consultation and assistance. Thanks also to the rest of the OSI Express team for their patience and many excellent suggestions. Special thanks to Glenn Talbott for writing the CISTERN formatter, and to David Ching for writing the statistics formatter tool.

The HP OSI Express Card Software Diagnostic Program

The software diagnostic program is a high-level mnemonic debugger. The structure definition utility isolates the diagnostic program from compiler differences and data definition changes.

by Joseph R. Longo, Jr.

APPROPRIATE DIAGNOSTIC AND DEBUGGING TOOLS are essential to any successful software or hardware development effort. A project as large as the HP OSI Express card development effort posed some challenging opportunities. Not only was most of the technology for the card, both software and hardware, still being defined, but the target computer line was still under development as well. Tools such as the HP 64000-UX microprocessor development environment and the HP 1650 logic analyzer were evaluated to understand what was already available. These tools provided features such as single-stepping and data tracing and were indispensable for doing low-level debugging. However, a much higher-level debugger was also necessary to observe protocol operations and system dynamics. Obtaining this information by deciphering screens of hexadecimal data would be very tedious and time-consuming. Also, until the card management tools were in place much later in the development cycle, there would be no means of monitoring the utilization of resources on the card.

For these reasons, it was decided to pursue the development of in-house debugging and diagnostic tools. The following design goals were established:

- No existing functions duplicated
- Modular design
- Evolving feature set
- Minimal impact on product performance
- Minimal impact on card software size
- No additional hardware on card required
- No additional coding in product modules required
- Can be used when all other debugging hooks are removed.

The design goals can be summarized as: (1) use the limited available time and engineers to develop new functions rather than trying to duplicate features provided elsewhere, (2) provide flexibility to accommodate changes in the development environment and new requests from the customer base, and (3) ensure that nothing special needs to be done to use these tools and that their use does not impact the product being developed. While these goals may appear to be unattainable, their intent was to focus the project so that something usable could be provided in a reasonable time and the effort would not collapse under its own weight by trying to be the last word in diagnostics. The result of all this was the development of two modules: the structure definition utility, which provides a dictionary of data definitions that can be accessed programatically, and the software diagnostic program, which is a high-level mnemonic debugger that can monitor the resources on the card and allow the user to view data from the card in various formats (see Fig. 1).

Structure Definition Utility

During the early stages of the development of the card software, the definitions of the internal data structures were constantly in a state of flux. Any module or program referencing these data types was constantly being recompiled in an effort to keep it up to date. It was quickly recognized that it would not be practical or productive if the diagnostic tools, test programs, and formatters had to be recreated every time a data type changed. Also, at any time in the development process there could be different versions of protocol or environment modules under test. It would be impractical to require that a different version of the diagnostic and test programs be used depending on which version of a module was being tested.

A second obstacle in the creation of the diagnostic tools had to do with the two compilers that were to be used.

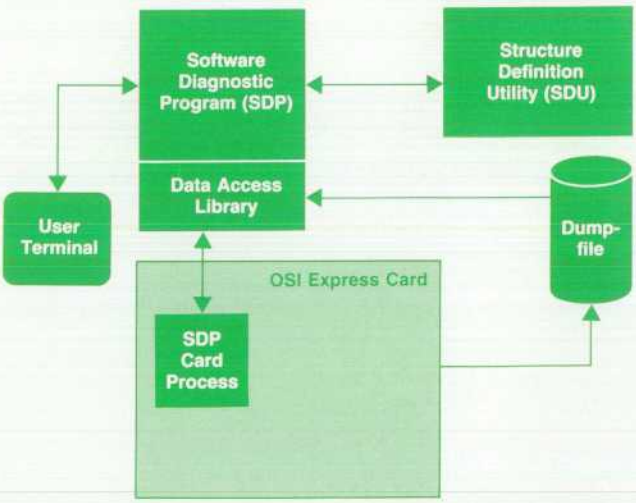


Fig. 1. *Special diagnostic and debugging tools created for the OSI Express card development project consisted of the software diagnostic program, which includes the data access library, and the structure definition utility.*

The card code was to be compiled with the 68000 C compiler. The diagnostic programs, which resided on the host, used the standard UNIX C compiler. The primary difference between these two compilers has to do with the way data types are aligned and padded. The 68000 compiler aligns types such as ints on 2-byte boundaries while the UNIX compiler aligns ints on 4-byte boundaries. Therefore, a data buffer retrieved from the card could not be interpreted by the host program if the same data types were used. These differences prevented the host diagnostics from compiling with the same C header files as the card code.

It was obvious that some mechanism was needed to isolate the test and debugging programs from both the fluctuations in the data structure declarations and the differences in the compilers. The structure definition utility (SDU) was developed for this purpose. The SDU is used to create a data dictionary containing the C data type definitions. The definitions stored in the dictionary can then be accessed via standard SDU library routines. When a data type changes, the new definition is loaded into the dictionary and the engineer can continue testing and debugging without recompiling.

The SDU consists of three parts: a stand-alone parser/compiler program, sdu.build, which processes the C type definitions and creates the data dictionary, the dictionary file, which is generated by the sdu.build program, and the dictionary interface library, which allows applications to access the information stored in the dictionary (Fig. 2).

When designing the SDU it was necessary to keep in mind that regardless of how creative the end product was, no one would ever use it if it was too complicated, took too long to operate, or required that data be maintained in more than one location. Given the number of type definitions, it was especially important that the sdu.build program be able to accept standard C include files as input. This also meant that the sdu.build parser had to recognize as many of the C data type constructs as possible. After these two criteria were satisfied the whole process of creating and accessing the dictionary still had to remain relatively simple and fast.

Input Format

The input to the SDU parser is a C include file containing the C data types, type definitions, and #defines from the program header (.h) files. To provide for portability between compilers and to simplify the parser design, some minimal structure had to be imposed on the input data. The basic format for the input data is:

```
(*
type specifiers
!!
type definitions, #defines, and
data default values
*)
```

The input is divided into two parts: the type specifiers and the type declarations. The punctuation denotes the beginning and end of input and separates the two sections. The type specifiers are optional, but the punctuation is required even if the specifiers are not entered. While syntax is important, the input format is relatively free-form. For example, there are no restrictions on the number of statements per line. At least one blank must separate identifiers on an input line, but for the most part, separators (blanks, tabs, newlines) are ignored.

All data declarations are defined from the atomic C data types (int, char, short, etc.). The alignment and sizes of the C basic types are preloaded into the data dictionary. These values can be redefined and/or new values added using the type specifiers input. The primary reason for redefining the basic type values is the use of a different C compiler. At least two and possibly three different C compilers were expected to be used during the development of the card code. The main differences between the compilers were the alignment of the data types and the padding of struct/union data types. The SDU compiler defaults to the alignment requirements of the HP 9000 Series 300 and 68000 C compilers. The syntax for a type specifier entry is:

```
type, type_len, alignment, format;
```

Type is an ASCII string representing the name of the type specifier to be loaded. Type_len is a decimal value indicating the storage requirements of the type specifier in bytes (e.g., storage for the C type char is one byte). Alignment is a decimal value indicating the byte alignment of the type when it appears within a struct/union type declaration. The value is in bytes and must be greater than zero. The value is used to determine to what boundary (byte, even byte, double word, etc.) the type should be aligned. The value is also used to determine the padding within the struct type. The format field is a single character indicating the default display form for this data type (x = hexadecimal, d = decimal, a = ASCII).

Variable Definitions and Constants

The C variable definitions and constants are specified in the second part of the SDU parser input. The variable definitions must be in standard C format as defined in the C reference manual.[1] Data declarations (e.g., int abc;) and type

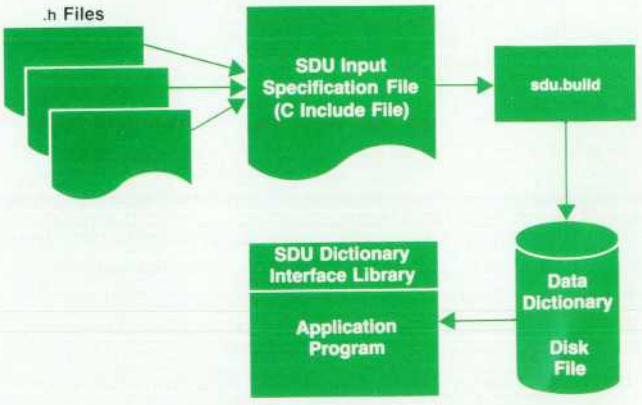


Fig. 2. *The structure definition utility consists of a stand-alone parser/compiler program (sdu.build), a dictionary file built by the program, and a dictionary interface library. Input to the SDU is a C include file containing C data types, type definitions, and #defines from the program header (.h) files.*

UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

definitions (typedefs) are accepted as input. Both simple and complex (struct/union) definitions can be loaded. Constants are loaded using the C preprocessor #define statement. The constant values can be used in subsequent #define statements or to specify the size of an array in a type definition. Application programs can access the #define values once the dictionary is created. The SDU compiler will also recognize C comments (/* */) and some forms of compiler directives (#ifdef, #else).

It is not necessary to define all the variables and constants explicitly in the same file as the basic type specifiers. It is not even necessary to have them all in a single text file. The SDU parser allows the user to specify the name of the file or files containing the definitions instead of the definitions themselves. Given the name of the file bracketed by percent signs (%name%), the SDU parser will open the specified file and load the definitions. This feature allows the variable and constant definitions to be used directly by the C programs since any special SDU symbols can be restricted to the input specification file and do not have to be put in with the types.

Default Information

The SDU provides routines that allow applications to create data buffers based on definitions loaded in the dictionary. These buffers can then be used by the applications for various purposes such as testing, debugging, and validation. The SDU provides mechanisms for storing default values for the data definitions in the dictionary. The default values can then be loaded into the data buffers created for the applications. The default information is loaded at the same time as the data definitions using the format:

```
definition name = default value;
```

The definition must have already been loaded into the dictionary. If the definition name is an item within a struct or union type then it must be fully qualified.

Creating the Dictionary

The data dictionary is created by the sdu.build program from the C include files. Depending on the amount of information to be processed, the creation of the dictionary can be a time-intensive task. So that every application does not have to incur this overhead cost each time it wishes to access the dictionary, the sdu.build program is run as a standalone program. The sdu.build program must be run whenever new data definitions are to be added to a dictionary. Once the dictionary is created, the dictionary can be accessed by multiple applications.

Building the dictionary is a two-step process. The first step is to create the dictionary in the internal memory of the sdu.build program. As the data declarations are read they are loaded into the internal tables and data structures of the dictionary. The SDU compiler is responsible for reading and verifying the input definitions and loading the information into the tables. Each #define constant and data declaration will have at least one entry in a table (struct/union data types have one entry for each element defined as part of the struct/union declaration). Any errors encountered during the processing will cause the program to terminate and display an appropriate message. The second step is to save the table information from the internal memory into something more accessible by the user applications. Once the dictionary has been successfully loaded the memory image is written to an HP-UX disk file. The name of this file is specified in the run string when the sdu.build program is executed.

Accessing the Dictionary

Applications planning to use the data dictionary must link with the dictionary interface library. This library contains all the routines for accessing information stored in the dictionary. The first library call made by the application must be the one to load the dictionary information from the disk file into the application's internal memory. The application passes the name of the dictionary file to the load call. The load routine allocates memory for the dictionary and reads the data into memory. The amount of space required was written to a header record in the disk file by the build program. The dictionary loaded is now an exact copy of the dictionary created by the sdu.build program.

The load routine performs one more task before the data can be accessed by the calling application. The internal design of the dictionary requires numerous pointers to link various pieces of information together. These pointers, which are really just memory addresses, are valid only in the original memory space where the dictionary was created. Although the system call malloc is used in both the build and the load processes, it cannot be guaranteed that the memory obtained from the call will be in exactly the same address location each time. Therefore, the internal pointers must be modified to reflect the location of the data in the new address space.

The pointers are adjusted by comparing the load address and the build address (which was stored in the image file). The required pointer adjustment is the difference between the starting address for the build and the starting address of the internal memory for the load. This adjustment value (positive or negative) is added to all pointers in the internal dictionary structures. When the pointers are adjusted the load process is complete and the dictionary is ready for use by the application.

Developing sdu.build

Developing a program that can recognize C-language data declarations in all forms is akin to writing a mini version of the C compiler. Development of the SDU parser/compiler program sdu.build would have been a formidable task had it not been for the tools yacc and lex available under the HP-UX operating system.[2] Yacc is a generalized tool for describing input to programs; it imposes a structure on the input and then provides a framework in which to develop routines to handle the input as it is recognized. The parser generated from yacc organizes the input according to the specified structure rules to determine if the data is valid. Lex is used to generate the lexical analyzer, which assembles the input stream into identifiable items known as tokens, which are then passed to the parser. Lex has its own set of rules called regular expressions,[3] which define the input tokens. Regular expressions are patterns against which the input is compared; a match represents a recognized token.

The parser and lexical analyzer are combined to create the SDU compiler known as sdu.build.

The first step in using yacc is to define the set of rules, or grammar, for the input. A grammar specifies the syntactic structure of a language, with the language in this case being the C data declarations. The syntax is used to determine whether a sequence of words (or tokens) is in the language. Describing the syntax of a language is not as hard as it sounds. A notation known as Backus-Naur form (BNF)[4] already exists for specifying the syntax of a language. Converting the C data declarations to BNF was simplified by the fact that a partial grammar already existed.[3] Elements not supported by the SDU were eliminated from the grammar.

The grammar consists of a sequence of rules. A rule is written with a left-hand side and a right-hand side separated by a colon. The left-hand side consists of a single unique symbol called a nonterminal. The right-hand side consists of a sequence of zero or more terminals and nonterminals sometimes called a formulation. One or more formulations may appear on the right-hand side of a rule. A rule must exist for every nonterminal symbol. Terminal symbols, which are synonymous with tokens, are not defined further in the grammar but are returned from the lexical analyzer. Examples of grammar rules used for describing some simplified mathematical expressions are:

```
expression  : primary
            | '(' expression ')'
            | '-' expression
            | expression '+' expression
            | expression '-' expression
            | expression '*' expression
            | expression '/' expression

primary     : identifier
            | constant
```

The symbols expression and primary are nonterminals while identifier and constant are terminals. Values enclosed in single quotes are literals and must be recognized from the input stream along with the terminals. The vertical bar (|) means "or" and is used to combine formulations for the same nonterminal symbol. The nonterminal symbol on the left-hand side of the first rule is called the start symbol. This symbol represents the most general structure defined by the grammar rules and is used to denote the language that the grammar describes.

Once the grammar is defined in BNF, it is a very simple process to convert it to a form that is acceptable to yacc. Because terminals and nonterminals look alike, yacc requires terminals to be defined using the %token statement in a declarations section ahead of the grammar. Any grammar that involves arithmetic expressions must define the precedence and associativity of the operators in the declarations section to avoid parsing conflicts. Some additional punctuation, such as semicolons (;) at the end of each grammar rule, and double percent signs (%%) to separate the declarations section from the grammar, must also be added before the file can be processed by yacc. With these modifications the specifications can now be turned into a C program by yacc that will parse an input stream based on the grammar rules.

The function of the lexical analyzer is to read the input stream a character at a time and assemble tokens from the unstructured data. Tokens can be anything from operators to reserved words to user-defined constants and identifiers. Separating the tokens can be any number of white-space characters (blanks, tabs, and line separators), which are typically ignored. The most time-consuming part of creating the lexical analyzer is defining the regular expressions, or patterns, which are used to recognize the input tokens. The patterns must be general enough to recognize all forms of the tokens and yet be specific enough to exclude tokens that are not of the desired class. The syntax for defining regular expressions is similar to the pattern matching features found in most editors. A pattern to match C identifiers might look like:

```
[A-Za-z_][A-Za-z0-9_]*
```

C identifiers start with a letter or underscore followed by an arbitrary number of letters, digits, or underscores. In the case where a token matches more than one pattern, lex attempts to resolve the conflict by first choosing the pattern that represents the longest possible input string, and then, if the conflict still exists, by choosing the pattern that is listed first. Once a pattern is matched, lex executes any action associated with the pattern. Actions can be specified along with the patterns; they consist of one or more lines of C code that perform additional processing on the tokens. For example, when an identifier is recognized it can be a user-defined value or a C reserved word such as typedef or struct. The action associated with the identifier pattern can be used to search a table of reserved words to determine the type of identifier found. This information can then be returned to the parser along with the token.

Using the lexical analyzer and the parser as just described, we now have a program that will read and validate the input data. There is still one more step before this program can be used to create the data dictionary. Now that we know the information is acceptable we have to do something with it. This requires going back to the specifications for yacc and adding actions for each grammar rule. The actions consist of one or more C statements that are performed each time a rule is recognized. Unlike the lex actions, these actions may return values that can be accessed by other actions. They can also access values returned by the lexical analyzer for tokens. In the sdu.build program, the purpose of the yacc actions is to load the C data declarations into the internal structures of the data dictionary. With the addition of the yacc actions the sdu.build program is now complete.

Software Diagnostic Program

The software diagnostic program (SDP) is an interactive application program that runs under the HP-UX operating system on HP 9000 Series 800 computers. It provides diagnostic and debugging features for the software downloaded to the OSI Express card. The primary function of the diagnostic program is to provide a means for dynamically accessing data structures on the card and then displaying

the data in an easily readable format. The SDP also allows the user to monitor certain aspects of the card's operation and to gather and report performance related statistics. Some of the features provided include:

- Dynamic access to card-resident data structures
- Data formatting capabilities
- Single-character commands
- Statistical displays
- Mnemonic access to global symbols
- Per-path state information displays
- Print and log functions
- Breakpoints, traps, and suspend function
- Card death display
- Dumpfile access.

The diagnostic program consists of two primary modules: the data access routines and the user interface module. The access routines provide the mechanisms to read and write information between the application and the card or the dumpfile. The user interface module handles all the interactions with the user, makes the necessary access routine calls to read or write data, and does the formatting and displaying of information to the terminal screen. The user interface and the data access routines were developed in a modular fashion with a documented interface between the two. While the library routines were originally intended for use only by the user interface module, the interface is designed to allow other applications access to the functions.

Data Access Routines

The data access routines provide the mechanisms for reading and writing information between the host application and the card or the dumpfile. The data access routines consist of three major components: the host-resident library routines, the dumpfile access module, and the card-resident process. The library is a well-defined set of calls that provide the application interface to the various data access operations. The library routines do all the error checking on the call parameters and then route the request to either the card process or the dumpfile access module. The library routines decode any received responses and return the appropriate data and status information back to the host application program. The most important service provided by the library routines is providing a transparent interface to the data. The same library calls are used to access both the dumpfile and the card.

The card process is downloaded to the card along with the networking software. It receives messages from the host library via an established communication channel and then performs the requested operation on the card. Status information and any data retrieved are returned to the host via the same communication channel. For the card process to be able to carry out its duties, it must operate independently from the networking software and it must not rely on any services provided through CONE (common OSI networking environment). The process must also be able to interrupt the networking operations when necessary, and be able to operate when the networking software has died. Most of this independence is achieved by communicating directly with the backplane handler (on the card) and the driver (from the host). This interface bypasses most of the standard communication paths used by the networking software. The card process manages all its own data buffers and has no dependencies on external data structures. Also, the card process is designed to operate at a higher interrupt level than the network protocols. This allows the diagnostic module to gain control of the card processor when necessary.

In some debugging situations it is not always possible or practical to access the OSI Express card directly. During development, for example, if the card died abnormally the developer might not be able to get to the problem for some time. Rather than tie up the hardware for an extended period of time or attempt to try to reproduce the problem at a later time it is often better to save the card image and attempt to diagnose the problem off-line. The facility exists for dumping the card image to a disk file. However, most engineers prefer something other than digging through stacks of hexadecimal listings. In fact, the preferred method is to use the same debugging tool on both the card and the dumpfile. For this reason, the library routines provide access to both the card and the dumpfile, the only change being the parameters that are passed to the call that initiates the connection. Once the connection is established, card and dumpfile operations are identical, with the exception that write operations are not allowed to the dumpfile. What is going on is completely transparent to the user sitting at the terminal.

User Interface

When developing the user interface it was important to keep in mind some basic concepts. First, the users of the diagnostic program would be in the process of learning many new debugging tools such as the symbolic debuggers on the HP 9000 Series 300 (cdb) and 800 (xdb) and the HP 64000-UX development environment at the same time. It was important to keep the interface simple and the number of special keys to a minimum so as not to make the learning curve too long or steep. Also, where possible, functions or data input operations should be handled in the same way as the corresponding operations in the other debuggers. Something as simple as entering numeric information should not require users to learn two different formats. Second, the development time for providing a useful debugging tool required that the complexity of the interface be kept to a minimum so the functionality would be available on time.

When the diagnostic is initially invoked the user is presented with a menu listing the major functional areas available, such as resource utilization or data retrieval. Submenus may be displayed detailing the operations available within a particular functional area depending on the selection on the main menu. Once a specific operation has been selected, the appropriate screen is displayed containing any data retrieved from the card and a list of commands available for that display.

The user interface has a two-tiered command structure consisting of global and local commands. Both global and local commands are typically single keyboard characters which are acted on as soon as they are typed (Return is not required). Global commands are active for every display within the program and can be entered whenever a com-

```
memory: 00800000h - 009fffffh                          cstate: RUNNING
Address     Data:       (long)                            Ascii:
00834338    00000000    00010000    0000e000    00000000    ........  ........
00834348    00000000    00000000    60092800    00846a14    ........  `.(...j.
00834358    00846a14    00000000    00000096    532c0000    ..j.....  ....S,..
00834368    00846a14    00841cac    00846a14    00846a14    ..j.....  ..j...j.
00834378    00845cd8    00846a14    00846a1c    00846a1c    ..\...j.  ..j...j.
00834388    008424fc    00846a1c    00846a14    00000000    ..$...j.  ..j.....
00834398    00845e38    00846a14    00846a1c    00846a1c    ..^8..j.  ..j...j.
008343a8    00846a14    00845f18    00846a14    00846a14    ..j..._.  ..j...j.
008343b8    00845ff8    008457fc    00846a14    00846a1c    ..._...W.  ..j...j.
008343c8    0088d20c    0083441c    bcbcbcbc    009659c8    ......D.  ......Y.
008343d8    00839820    00000000    00000000    00000000    ... ....  ........
008343e8    00000000    00000000    00000000    00000000    ........  ........
008343f8    00000000    00000000    0000000b    00000000    ........  ........
00834408    00006400    00000002    00966f26    00966d40    ..d.....  ..o&..m@
00834418    00966b5a    0083443a    aaaaaaaa    00000000    ..kZ..D:  ........
00834428    00000084    24880001    6b640000    00000096    ....$...  kd......
00834438    251c0083    4458aaaa    aaaa0000    00000000    %...DX..  ........
00834448    00842488    00016b64    00000000    00000000    ..$...kd  ........

DISPLAY:  1=memory  2=cast  3=struct  4=path states  5=card info  (+)=more
```

Fig. 3. *Raw-form display of data retrieved from the card.*

mand is expected as input. Some examples of global commands include: help (?), quit (Q), shell escape (!), and main menu (M). Local commands are specific to the display with which they are associated and are only available when that display is current (appearing on the terminal). The local commands for a particular display are shown at the bottom of the terminal screen. Local commands perform operations such as reread statistics, reformat data, and retrieve a global data structure from the card. While global commands are unique for the entire program the local commands are unique only within the associated display. The same keyboard character may invoke entirely different functions in different displays.

The software diagnostic uses the HP-UX curses[5] screen control package to create displays and handle all interactions with the terminal. Curses is designed to use the terminal screen control and display capabilities. Briefly, curses uses data structures called windows to collect the data to be displayed. The application program writes the data to be displayed to the current window and then makes the appropriate curses calls to transfer the window to the terminal screen. The primary benefit of using curses is that it relieves the application of the overhead of dealing with different terminal types and cursor movements. It also minimizes the amount of information that must be redisplayed on the screen by only transmitting the text information that has changed from the previous display.

Data Access Operations

The data access operations are all functions and commands for accessing, formatting, and manipulating information from the card. As with most debuggers, the ability to view data is one of the most frequently used. Data retrieved from the card can be displayed in two forms: raw and cast. In raw form (Fig. 3) the data is displayed in columns of four-byte integers. The first column is the RAM address of the first byte of data in each row. The address and data values are hexadecimal. The right two screen columns contain the ASCII representation of each byte of data in the row if it is printable. If the byte is not a printable character then a period is shown as a placeholder. The user also has the option to change the data format from hexadecimal to decimal and from four-byte integers to columns of two-byte words. The NEXT and PREVIOUS functions can be used to page through memory from the initial display address.

```
memory: 00800000h - 009fffffh                          cstate: RUNNING

0x834338 bmi_globals_t = struct {
  mod_globs = struct {
    valid_drain_list = 0;
    module_id = 0x1;
    trace_mask = 0;
    log_mask = 0xe0000000;
    diag_mask = 0;
    mod_glob_stats = struct {
      item_ptr = 0;
      item_size = 0;
    };
    canonical_addr = 24585;
    path_report_pid = "("    (040);
    nm_req_rtn = 0x846a14;
    nm_event_rtn = 0x846a14;
  };
  proto_globs = struct {
    sap_t_addr = 0;

Press Return to continue, SPACE to stop
```

Fig. 4. *Cast-form display of data retrieved from the card.*

The second form of data formatting is the cast function. The data retrieved from the card can be displayed based on a specified C data type (Fig. 4). When the cast function is selected the user is prompted for the RAM address from which to retrieve the information and the name of a data type, which will define the formatting of the data. To use the cast function the specified data type must be in the SDU data dictionary and the dictionary must have been loaded into the user interface module. The data type is displayed and the information is formatted based on the data type. Data can be reformatted simply by specifying a different data type. If the data type exceeds a single screen the user is allowed to page through the displays. The user can switch between the raw and cast displays without having to reread the data from the card.

Address values can be entered in either numeric or mnemonic forms. Numeric addresses can be either hexadecimal, decimal, or octal values. Mnemonic addresses are entered by typing the name of a global variable or procedure. C variables and procedure names must be preceded by an underbar (_) while assembly variables and labels may or may not require an underbar depending on how they are declared in the code. The address value is obtained by searching the linker symbol file (.L), which corresponds to the download file on the OSI Express card. In addition to other information, the symbol file contains global symbol records,[6] which provide the names of global symbols (variables and procedures) and their relocated addresses. The address stored in the file for the symbol entered is then used to retrieve the information from the card. Use of the mnemonic address is recommended whenever possible. Not only does it eliminate the need to look up the address of the variable in the first place, it ensures that the address will be correct regardless of the version of the card software being accessed.

One level of addressing indirection can be accessed by preceding the address values, either numeric or mnemonic, by an asterisk (*). The address location on the card is then interpreted as containing the address of the data to be retrieved. In other words, the address specified is really a pointer to the data rather than the data itself. All address values, either direct or indirect, are checked to ensure that they are in the range of accessible addresses on the card. Both read access and write access are allowed to RAM memory, while only read access is permitted to EEPROM addresses.

Card Death Display

Whenever the OSI Express card dies abnormally, either from a software exception (address error, divide by zero, etc.) or an internal error (disaster log), or is halted from the host, a fatal error routine is invoked on the card to save the state of the card processors and record the error information at the time the card halted. The routine also sends an error indication to the host which reports that the card has died. During development and testing these situations were common. At such times, the process of gathering the data to determine why the error occurred can be time-consuming and involved. The type of error and even the size of the RAM memory can influence the location of the information to be read. Once the error is known a text file must still be searched to determine the meaning of the error.

The card death information display attempts to provide on one screen all the error information necessary to determine where and possibly why the card died. The diagnostic program gathers the information concerning the card death from the various memory locations and, after analyzing the data, displays on the screen the values that relate to the type of death that occurred (Fig. 5). The processor registers, including the stack pointer, the program counter, the status register, and the data and address registers, are retrieved and displayed in the center of the screen. When a card module dies gracefully it stores information in a disaster record. This information is retrieved, if available, and displayed at the bottom of the screen. The program also evaluates the error and supplies an apparent reason, or best guess, as to why the card died. On this screen the user should have enough data to understand why the card died and be able to locate any additional information.

Resource Utilization

The displays available under the resource utilization selection are intended to provide information on the operational state of the various modules and resources on the

```
memory: 00800000h - 009fffffh                          cstate: RUNNING

Type of Death: cmd.stop issued                 Subsys Id: 0
Type of Error: 0                               Location : 0

Apparent Reason: Card stopped from host

PROCESSOR REGISTERS  (SF_cmdstop_CPU_regs)

 Stack Ptr: 009ff5bc       Program Ctr: 00849844       Status Reg: 2004

 D0-D7  00000000 00002004 000000df 00001a88 008210e8 009f0000 00820000 00000600
 A0-A7  00400018 0083a066 00208c68 00833188 00008090 00200000 00203c42 009ff5bc

DISASTER RECORD
    Current Module: 0                          Myentry Pointer: 0
    Current Region: 0   ISO                    Event Pointer  : 0
    Flags         : 0                          Event Length   : 0

DISPLAY:  1=memory  2=cast  3=struct  4=path states  5=card info  (+)=more
```

Fig. 5. *Card death display.*

```
Flow Control                                          cstate: RUNNING

CONNECTIONS                                  ERAS
  Active Inbound : 1                           Era Boundaries    : 457
  Active Outbound: 1                           Eras Memory Tight: 0
  Active Retrans : 1                           Eras CPU Tight    : 0
                                               Era Period        : 1000

THROUGHPUT                    Actual         Target         Scaled
  Card CPU Packets In :       0              1000           1000
  Card CPU Packets Out:       0              1000           1000
  Throughput Bytes In :       0              1000000        1000000
  Card Memory Out     :       0              500000000      500000000

  Card Throughput Bytes: 0

MEMORY MANAGER
  Available BLOCKS on FREE LIST  : 52
  Available LARGE buffer segments: 313
  Available SMALL buffer segments: 1581
  Available TINY  buffer segments: 2


' '=Read stats     1=Read and Clear     2=Clear and Read
```

Fig. 6. *Resource utilization display showing flow control statistics.*

OSI Express card. For the most part, the displays contain various combinations of statistics gathered from the card that can be monitored to determine such things as throughput, flow control (Fig. 6), and memory utilization (Fig. 7). There are basically two types of statistics that are maintained; cumulative and actual. The cumulative statistics represent values that have accumulated over a time period. Examples of cumulative statistics include frontplane packets transmitted, number of global retransmissions, and backplane bytes transferred. These statistics can be cleared to zero by the user. Actual statistics reflect the conditions as they currently exist on the card. Number of open connections, available buffer manager memory, and scheduler queue depth are examples of actual statistics. Actual statistics cannot be cleared.

Trap/Breakpoint/Suspend

When attempting to debug problems on the card it is often necessary to stop the processing on the card to examine the current state of the processor or a global variable before continuing. The diagnostic program provides three mechanisms for stopping the card: breakpoints, traps, and suspend.

The breakpoint feature is similar in implementation to breakpoints in other debuggers. The user specifies the address of the instruction on the card where the breakpoint should be set. When that location is reached in the processing stream the card is stopped and a message is sent to the host application, which notifies the user. The card remains stopped until the user tells it to continue. The card then resumes processing from the instruction at the breakpoint location.

Traps are basically predefined breakpoints hardcoded in the networking software that can be turned on and off as needed. The locations of the trap calls are determined by the code developers and can be anywhere in the executable code. When a trap is encountered a diagnostic procedure on the card is called. The diagnostic procedure checks the trap type with a global mask to determine whether this trap is on or off. The trap type is one of the parameters

```
Buffer Manager Utilization                            cstate: RUNNING


Total Memory (bytes) : 1646544
Available Memory     : 1430312
Available Percent    : 86

Available BLOCKS on FREE LIST  : 52
Available TINY  buffer segments: 2              segment size: 220
Available SMALL buffer segments: 1581           segment size: 480
Available LARGE buffer segments: 313            segment size: 2064

POOL MANAGER              Pool1    Pool2    Pool3    TinyBS
  Number of segments :    1        0        2        3
  Objects per segment:    10       6        2        2
  Objects in use     :    4        0        3        4
  Object size (bytes):    44       76       228      228

Subtasker Queue Depth   : 0
LLC Inbound Queue Depth : 0


' '=Read stats
```

Fig. 7. *Resource utilization display showing memory utilization statistics.*

passed on the trap call and is defined by the code developer. The global mask is configurable from the user interface module. If the trap is off then the call returns and the code continues without any break. If the trap is on then the card is stopped and a message is sent to the host application. Again, the card remains stopped until the user tells it to continue. Processing resumes from the instruction after the trap call.

The suspend operation gives the user the ability to stop the card at any moment in time. This is a global command issued from the user interface. When the request is received by the card process a routine is invoked that interrupts the networking protocols and places the card in an idle loop. The card timer manager interrupts are also suppressed by this routine. The suspend will remain in effect until a resume command is issued by the user. The purpose of the suspend function is to give the user the opportunity to take a quick look around without having data change or move before it can be examined.

Summary

The success of these modules is evidenced by their acceptance as the tools of choice for much of the debugging, diagnostic, and testing efforts. The use of these tools significantly reduced the time needed to isolate many of the defects encountered in the card software. The statistical displays provided valuable information on throughput and flow control early enough in the development cycle to allow time to make any necessary adjustments.

Acknowledgments

The following individuals have contributed to the success of these tools through their work on either the design or the coding of certain functions: Gerry Claflin, Steve Dean, John Nivinski, and Chuck Black. Also, I would especially like to mention David Ching, who provided the routines for processing the linker symbol file, and Chwee Kong Quek for his work on the dumpfile access module.

References

1. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
2. *HP-UX Concepts and Tutorials, Volume 3: Programming Environment*, Hewlett-Packard Company, 1986.
3. A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1979.
4. A.T. Schreiner and H.G. Friedman, Jr., *Introduction to Compiler Construction with UNIX*, Prentice-Hall, 1985.
5. *HP-UX Concepts and Tutorials, Volume 4: Device I/O and User Interfacing*, Hewlett-Packard Company, 1985.
6. *File Format Reference for the HP 64000-UX Microprocessor Development Environment*, Hewlett-Packard Company, 1987.

Support Features of the HP OSI Express Card

The HP OSI Express card offers event logging and tracing to facilitate troubleshooting in multivendor networks.

by Jayesh K. Shah and Charles L. Hamer

TODAY'S STATE-OF-THE-ART automated factories require the seamless interaction of systems and devices supplied by a diverse set of vendors. To manage this complex environment effectively and keep it operating smoothly, users must be able to resolve problems quickly. The HP OSI Express card incorporates several powerful new features to aid the troubleshooter. This article highlights the support features of the HP OSI Express card and illustrates their use in two troubleshooting scenarios.

Architecture Overview

The support architecture of the HP OSI Express card was an important consideration since the development of OSI protocols was a new area of endeavor for HP as well as for other computer companies. Numerous communication problems with other OSI implementations were expected. Therefore, a superior set of diagnostic capabilities was needed to resolve problems quickly in an I/O card environment. To achieve this functionality it was decided to extend the host's own nodal management facilities to include the HP OSI Express card. This design provides a single nodal management mechanism for event logging and protocol tracing for both host and card modules and provides the user with several benefits. The user does not have to be concerned whether a layer, module, or service resides in the host or on the card. The same set of tools with the same capabilities can be used to manage all aspects of the product. In addition, the trace and log output from both host- and card-based modules are identical in format because they share a common header and terminology for describing the severity of an error or the type of message being traced.

The OSI Express support architecture is shown in Fig.

1. The numbered arrows show the initial flow of control and information to enable a log class (logging severity level) and then to send log information from a card-based layer to the file system. Log classes are controlled by the user via the nodal management applications osiconfig and osicontrol. When the user enters a command to enable a particular log class in a particular layer, a request is passed by the nodal management application to the trace/log facility, which validates the request and ensures that various trace/log resources have been allocated. The request is then passed to subsystem management services (SMS), which provides facilities that allow the user to access management services (parameter manipulation, statistics collection, status, and control) and sends the request to card management services (CMS). CMS, which is the card-based counterpart of SMS, provides nodal and network management services to both the host-based management applications and the card-based protocol and system modules. After receiving the request from SMS, CMS forwards the request to the appropriate protocol layer or system module.

When an event that must be logged occurs in a card-based protocol layer, the event is passed from the protocol stack to CMS which communicates through the kernel with the log daemon. The log daemon receives the event (log) messages from the OSI Express card, obtains the system time (timestamps the message) and formats a log call to the host trace/log facility. Unformatted log messages are then written to the file system. When the user reads the log file, the trace/log formatter osidump is used. Osidump writes formatted log entries to the log file or terminal.

Event Logging

Logging is used to record abnormal or unusual networking events such as the receipt of an inbound packet with invalid protocol information (remote protocol error) or a remote system's refusal to accept a connect request. This is different from tracing. Tracing is used to record all information of a particular type or types from one or more layers or modules.

Log Headers

Log (and trace) messages have two parts: the header part and the data part. The header consists of the first eight lines (see Fig. 2). It includes the timestamp and other identifiers. The contents of the header are very important because the data in the header usually determines the formatting capabilities of the trace/log formatter. The data portion of the message that follows the header contains the description of the event (error message text).

One of the more important fields in the header is the log class. This is the severity of the event being logged. When logging is enabled the severity can be selected by the user. The user can choose to ignore event messages that are by nature informational, but when problems occur the user can modify the log class to obtain informational messages. Log messages have four classes of severity: disaster, error,

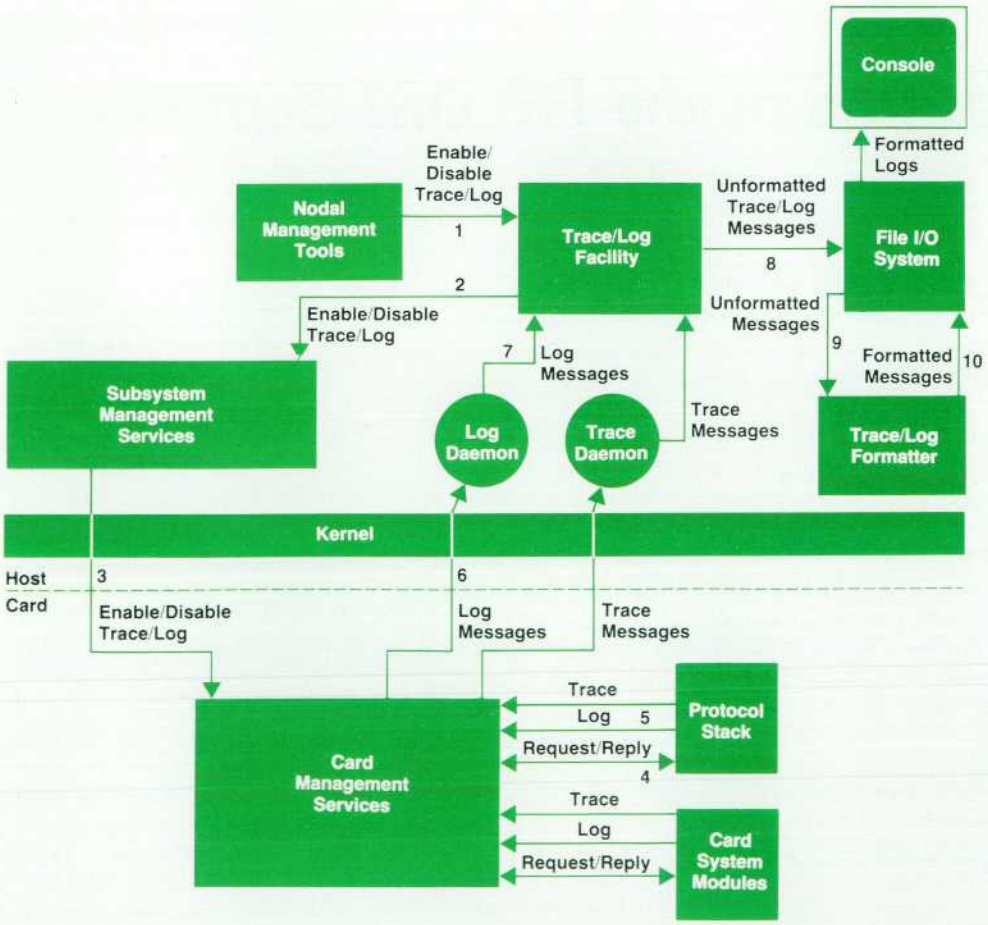


Fig. 1. *HP OSI Express card support architecture. The numbers show the sequence of operations for getting log information from a card-based layer to the file system.*

warning, and informational. Disaster class messages are logged when a condition occurs that could jeopardize the integrity of the system or network. Discovering that another system on the network is using the identical NSAP address is one example of a disaster class event. Once a disaster event occurs, the event is logged and the OSI Express card is taken off-line. The next lower event classification is error class. Errors are events that cause a user application to fail or take extra steps to recover. This is the default log class for most of the product's layers or modules. The definition of an error class event put the additional burden on the OSI Express card software developers of understanding the end result of an event. It could only be classified as an error once it was understood that it would adversely impact a user application. The expiration of one connection's transport inactivity timer is an example of an error class event. The error class designation is not as significant for what is included as an error as it is for what is excluded as an error. The error class definition prevents events that are interesting but not a problem (from the application point of view) from being logged. This not only saves disk space but frees the user from having to consider events that do not affect applications. Warning and informational class events are the next-lower-severity log classes. Warning events such as "Destination NSAP Unreachable," which impacts the network layer, or informational events such as receipt of a duplicate connection request, which impacts the transport layer, have no impact on user applications except for time loss. These events are probably most useful for performance analysis.

Two fields in the header provide connection information: the connection identifier (CID) and the path identifier (path ID). The CID is used by host software to reference a connection and is returned to the user. The path ID identifies a specific communication path on the OSI Express card and thus serves the same purpose in the card environment as the CID does in the host environment. From a troubleshooting perspective the path ID is very useful when problems occur on inbound connection requests that fail before reaching the host. In this situation, a CID will not exist since the request does not reach connection management and hence a CID is not issued.

```
Timestamp          : Fri Jun 9 1989 16:04:23.056389
Process ID         : 5679                    Subsystem          : SESSION
UID                : 4223                    Log Class          : ERROR
Device ID          : 0                       Path ID            : 19
Connection ID      : 1                       Log Instance       : 32771
Location           : 00144
Card Time Stamp    : 0x 7c50                 Card Location      : 0x 5000090
Previous Error     : 0x f00500               Current Error      : 0x 10120d31

  Remote protocol error. Received an SPDU that contains an SI which is not valid
  for the current state. An S-P-ABORT will be issued. (SS00144) Refer to troubletree
  Card_04 in the HP MAP 3.0 troubleshooting manual.


Session state vector  :
Protocol state        : 713 Data transfer state
Remote selector       :
52 65 6d 6f 74 65 00 00 00 00 00 00 00 00 00 00          Remote . . .
Requirements          : FD
Tokens available      :                         Tokens owned       :
Restart sync number   : 0                       Lowest confirm sync : 0
Next sync number      : 0                       Lowest restart sync : 0
IB expected data In   : 0                       IB accumulated In  : 0
OB expected data In   : 0                       OB accumulated In  : 0
XP connection         : Initiator               XP connection reuse : FALSE
FN SPDU collision     : FALSE                   DN after FN collisn : FALSE
SSYNmrsp allowed      : FALSES                  Activity in progress : FALSE
MAA/AEA/rx'd/tx'd     : AEA                     XP expedited avail : TRUE
Inbound CONE state    : IDLE                    Inbound SI         : MIP
Outbound CONE state   : IDLE                    Outbound SI        : CN
Horseshoe reason      : No_PENDING_REASON       Timer reason       : No_PENDING_REASON
Memory get reason     : No_PENDING_REASON       Abort reason       : ISSUE_U_ABORT
Session version       : TWO                     Resync kind        : No resync
Top of path           : FALSE                   Start timer        : FALSE
IB interface queue    : Not Empty               IB pending queue   : Empty
OB interface queue    : Empty                   OB pending queue   : Empty
IB SPM queue          : Empty                   Concatenation queue : Empty
SPDU concatenated     : FALSE                   Category 0 SI      : DT_OB or ER
T-disc req delayed    : FALSE                   Dequeue spdu       : FALSE
T-disc req issued     : FALSE
Stack 0 CONE event    : T_Connect req           Stack 1 CONE event : T_CONNECT cnf
Stack 2 outbound SI   : CN                      Stack 3 inbound SI : AC

SPDU offset:
  0: 00 00 00 09 -- -- -- -- -- -- -- -- -- -- -- -- ....................
SPDU:
  0: 31 07 0f 01 01 2a 02 00 02  -- -- -- -- -- -- 1....*.............

MIP                LI=2
  Sync type item   LI=2              : Explicit confirmation not required
  Serial number    LI=2              : 00 02
```

Fig. 2. *A formatted remote protocol error log message. There are two parts: the header and the data.*

Another field in the header and one of the most significant contributions to the supportability of this product is the log instance. The log instance is an identifier that threads log messages together. When a module first detects an error, it obtains a new unique log instance identifier and logs the event. The log instance is then passed with the error to the calling entity. If the calling entity also logs an error as a result of processing the error it receives, it logs the error as well as the log instance passed to it. The calling entity then returns the log instance to its caller. In this manner, the log instance is propagated all the way up to the user application. Log events with the same log instance are related. The earliest event with the same log instance is the root of the problem. Without a log instance mechanism, a user might think that several errors had occurred when in fact only one had occurred. Once the error is returned to the user application the log instance is available via a special function call to the service interface. Thus, the log instance provides an audit trail from the module that first detects an error all the way back to the user application.

Other fields in the header of interest to users include the user identifier (UID), and the process identifier (PID). The UID is the HP-UX user identifier of the user that created the connection. The PID is the identifier of the process that created the connection.

Error Messages

Special attention was focused on the content of error messages. All error messages include the problem category, the cause of the problem, and the corrective action recommended to resolve the problem. At all points in the code where an error might be logged, the protocol developer had to resolve the problem and not merely report it. It was also generally agreed to return any helpful information that was available to the user that would aid problem resolution. For this reason the session state vector is appended to the error text in Fig. 2.

The product troubleshooting guide is tightly coupled to the error messages. In Fig. 2, for example, the user is referred to troubletree Card_04 in the troubleshooting guide. Card_04 is a troubleshooting procedure designed to lead the user through the process of resolving a remote protocol error. The technique of referring to a specific troubleshooting procedure in the troubleshooting manual is used when the resolution procedure is longer than what could easily be described in a log message.

In addition, since usability was of great concern, we wanted to avoid terse log messages that required interpreting to understand what transpired. Therefore, error messages were reviewed and reworked to ensure that the text was clear. As a result of the efforts to make error messages more usable, an error messages manual was not required as part of the product's documentation.

CMS Informational Log

Another feature designed to aid troubleshooting is the CMS informational log message. Recall that CMS is used by the protocol stack and system modules to log event messages and trace protocol and system module activity. When CMS receives a request to log a message it checks to see if it has logged a message on that path before. If it has, it just performs the log or trace task requested by the calling software module. If it has not logged a message on that path before, it logs a CMS informational message and then logs the message requested by the calling software module. The informational message logged by CMS includes as much of both the local and remote applications' presentation addresses as is known. An application's presentation address is also often referred to as its PST-N selectors. This information is logged in the data portion of the log message and is especially useful for remotely initiated connections as is typical on server nodes. Now, when an error occurs, information is available that provides the presentation addresses of the affected applications.

A Troubleshooting Scenario

Two sample scenarios will illustrate the use of the troubleshooting features described above. Troubleshooting scenario 1 is shown in Fig. 3. Assume that user application 1 (UA1), an HP MMS (Manufacturing Message Service) client, on node A wants to communicate with user application 2 (UA2) on node B. Furthermore, assume that UA1's connect request to UA2 fails because UA2 has a different presentation address from the one UA1 is trying to communicate with. This can occur when the same presentation address is maintained in two separate locations. For example, a shop-floor-device OSI implementation may not provide a directory service user agent for directory access. Instead, it may locally manage presentation addresses, thereby providing an opportunity for address inconsistency.

In Fig. 3 the dotted line represents the SAPs (service access points) that have been activated by UA2 to receive

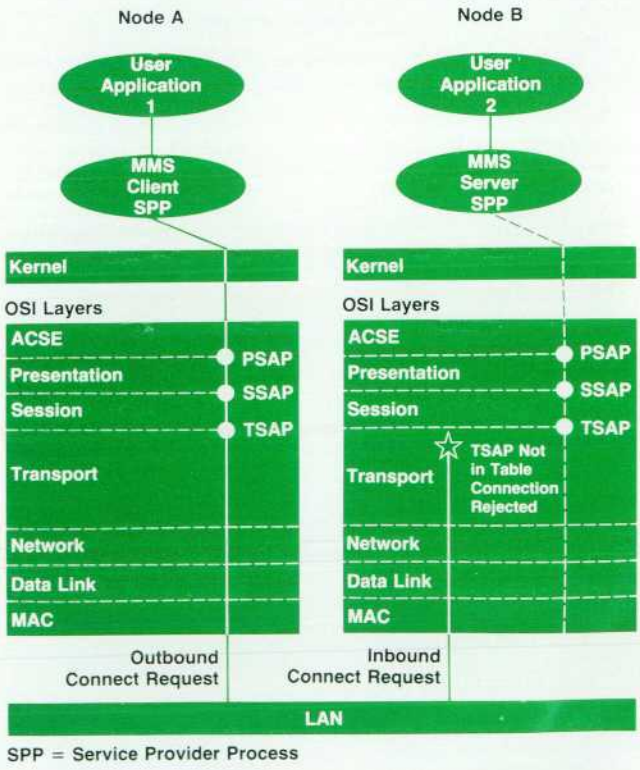


Fig. 3. *Troubleshooting scenario 1.*

requests from MMS client applications. UA1 obtains the PST-N selectors for UA2 and sends an mm_connect request to UA2. When the connect request is received by the transport layer on node B, the transport layer finds that the destination TSAP (transport service access point) is not in its table and rejects the request. The transport layer on node B rejects the connect request by sending a disconnect request to the transport layer on node A. The transport layer on node A logs this event. When it logs the event, it gets a unique log instance value. The transport layer on node A returns an error along with the log instance to the session layer. If any other module in the propagation path logs additional information, the log instance will be identical to the one originally logged by the transport layer. As explained above, the log instance is a mechanism that threads together all errors related to a specific error.

Error information returned to UA1 from the service interface includes the log instance. The user can then use the log instance as a key to query the log file for the underlying cause of the problem. All necessary data required to resolve the problem is logged along with the error message. In this example, the transport layer on node A will log the disconnect request TPDU (transport protocol data unit) in the data portion of the message. In this way, fault isolation and correction are facilitated by the use of the log instance, a detailed error message, and a comprehensive troubleshooting procedure.

Another Scenario

Fig. 4 illustrates troubleshooting scenario 2. In this scenario, assume that an FTAM (File Transfer Access and Management) initiator application on a remote system receives an abort indication while transferring a file to the local HP system. Also, assume that the remote system has limited troubleshooting capability. Thus, we need to isolate and resolve the problem from the responder side. Assume that the cause of the problem is that the remote system has sent an invalid session PDU (protocol data unit) and the local session entity aborted the connection.

When the connection was aborted on the responder side, a connection information message was logged by CMS with the complete presentation address of both the initiator and the responder along with the path identifier of the aborted connection. To resolve the problem, the user searches the log file for the CMS message with the appropriate initiator and responder presentation addresses. Locating this log message provides the user with the path ID, which can be used as a key to query the log file for errors that occurred on the aborted connection. The abort event message that the user obtains informs the user that the type of problem encountered was a remote protocol error (see Fig. 2). The event message also specifies the exact nature of the problem: the received PDU had an incorrect value for the session indicator. This type of problem is typically caused by a defect in the remote vendor's code and can be resolved only by a code change in the remote vendor's implementation. Therefore, the corrective action in the error message tells the user to follow a procedure that recreates the problem with tracing turned on. The additional trace information will help the remote system's vendor understand the context in which the problem occured so that an appropriate fix can be made.

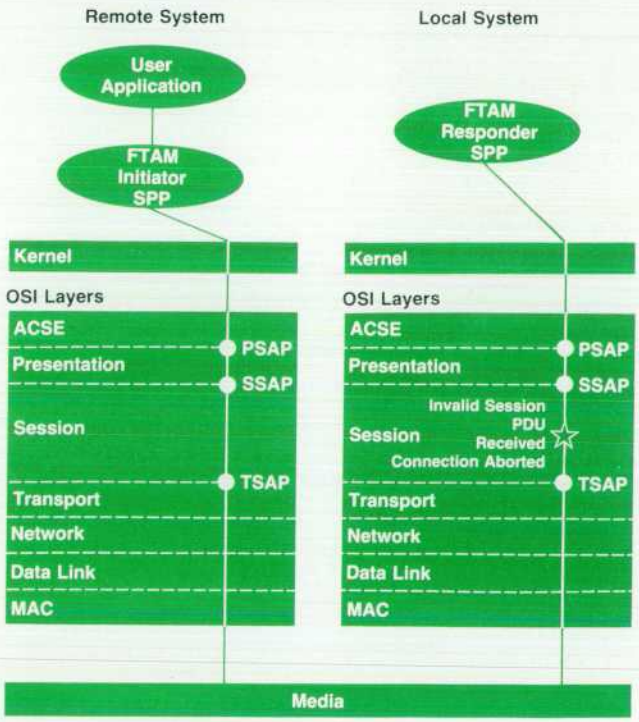
Tracing

Tracing is used to record all activity of a specific kind. It provides the contextual information that may be necessary to determine the cause or the activities that led up to a networking event. Both normal and abnormal events are recorded and, in fact, the trace utility cannot distinguish between the two. Tracing is a very useful tool for isolating remote protocol errors (interoperability problems) or internal defects.

Typically, a troubleshooter uses network tracing as a last resort to identify a problem. This is because configuration problems and user application problems are much more common, and because the use of trace tools and the analysis of the output require significant expertise. A major problem, therefore, is knowing when to use tracing. The log message in troubleshooting scenario 2 is typical of remote protocol error log messages generated by protocol modules. The message is intended to define the problem clearly and guide the troubleshooter to a procedure to isolate it.

The user can enable several types of tracing for each subsystem. The most commonly used trace kinds are listed below.

- Header Inbound. Traces protocol headers received from the next-lower protocol layer before decoding is done.
- Header Outbound. Traces protocol headers after encoding is complete before they are sent to the next-lower protocol layer.
- PDU Inbound. Traces the whole protocol data unit as it is received.
- PDU Outbound. Traces the whole protocol data unit as



SPP = Service Provider Process

Fig. 4. *Troubleshooting scenario 2.*

it is being sent.

- State Trace. Traces protocol state information.

A fundamental problem with tracing in general is that the person analyzing the trace file must recognize an abnormal event and so must have a fairly intimate knowledge of the protocol. The logging trace is a special trace type that writes a copy of the log message to the trace file. For instance, when tracing is enabled at the transport layer and this layer logs a message, that message is written to the log file (this is normal) and also to the trace file. The logging trace message acts as a marker within the trace file to help the person analyzing it locate the area of interest.

Acknowledgments

We are grateful to Mike Wenzel for his patience and for helping us evaluate various technical alternatives. Rich Rolph was instrumental in the development of usable error messages and troubleshooting procedures. We would like to thank the entire OSI Express team for suggesting and implementing supportability and usability features.

Integration and Test for the OSI Express Card's Protocol Stack

Special test tools and a multidimensional integration process enabled engineers to develop, test, and debug the firmware for the OSI Express card in two different environments. In one environment an emulation of the OSI Express card was used and in another the real hardware was used.

by Neil M. Alexander and Randy J. Westra

THE OSI EXPRESS PROJECT consisted of many independent project teams (made up of one or more engineers) working on specific portions of the protocol modules or support code. Each team needed the ability to test and develop code independent of others. However, periodically they needed to have a set of stable and tested code from other teams to enable them to test their own code. Since each engineer was involved in testing, test environments were designed to maximize their efforts. One environment consisted of an emulation of the OSI Express card on the development machines and another test environment consisted of a real OSI Express card connected to a target machine. Both the target and the development machines were HP 9000 Series 800 computers running the HP-UX operating system. Because of the number of engineers working on the project, multiple development and test machines were configured as a network. These test and development environments are shown in Fig. 1.

Test Architecture

Each protocol module was first tested in isolation before the module was integrated with the rest of the modules of the OSI Express stack. The CONE (common OSI networking environment) protocol module interface facilitates this module isolation since a stack can be built that does not contain all seven protocol modules. Protocol modules do not call each other directly to pass packets but instead make calls to CONE. A data structure called a path report is used to specify the modules configured into a stack. Protocol modules not specified in a path report will not be called by CONE and do not need to be in the stack. However, even with this modular design, several test modules are needed to test the stack fully.

The architecture and the modules involved in testing the OSI Express card firmware are shown in Fig. 2. This architecture was used on the host (running in user space) to test and debug protocol modules before the hardware was ready. When the hardware was ready, this same architecture was used on the target machines to test the protocol modules in the real environment.

Exception Generator

The exception generator is a test module that is configured in the stack below the module being tested. Packets moving inbound to the protocol module under test and moving outbound from the module are operated on by the exception generator. Packets not operated on by the exception generator are simply passed through to the next layer.

The exception generator can intercept, modify, generate, or discard packets as they are moving up or down the stack. Packets intercepted are placed in the exception generator packet queue. Up to ten packets can be saved in the queue at one time. Packets stored in this queue can be modified

and then sent up or down the protocol stack. In this way, PDUs that occur rarely can be constructed. Also, errors in transmission can be simulated by corrupting a packet in the queue and then sending it.

Scenario Interpreter Agent

The scenario interpreter agent performs functions similar to the exception generator. Whereas the exception generator is configured below the module under test, the scenario interpreter agent is positioned above the module under test. The scenario interpreter agent operates on inbound packets coming from the module under test and outbound packets going to the module under test. A packet can be intercepted as it moves down the stack and placed in the packet queue of the scenario interpreter agent. A saved packet is sent to the module under test by releasing it from the save queue of the scenario interpreter agent.

Bounce-Back Module

The bounce-back module sits at the bottom of the stack, and as its name implies, it enables packets heading down the stack to be sent (bounced) back up the stack. Normally, a protocol stack runs in a two-node configuration consisting of a sender and a receiver with the two nodes connected by a communication medium such as coax cable. When testing of the protocol stack first started, all testing was done in a single-node configuration. Packets were sent down the stack, turned around by the bounce-back test module, and then sent back up the stack. To make one stack act as both the incoming and the outgoing protocol stacks, the bounce-back module maintains a set of tables. The tables contain the proper inbound CONE call for each outgoing CONE call.

The bounce-back module makes different calls to CONE depending on which layer is configured above it in the stack. Thus, a separate table is maintained in the bounce-back module for each protocol layer that may be above it. For example, a stack can be configured for testing that consists only of the session layer above the bounce-back module. The session layer is a connection-oriented protocol layer and receives different incoming CONE calls than a connectionless layer such as the network layer. In this example, a packet would flow outbound from the session layer and be received by the bounce-back module. The bounce-back module would look in the session table to find the corresponding incoming call for the session layer. The packet would be copied and sent back up to the session layer, which would accept the incoming call as if it were part of the receiving node in a two-node test.

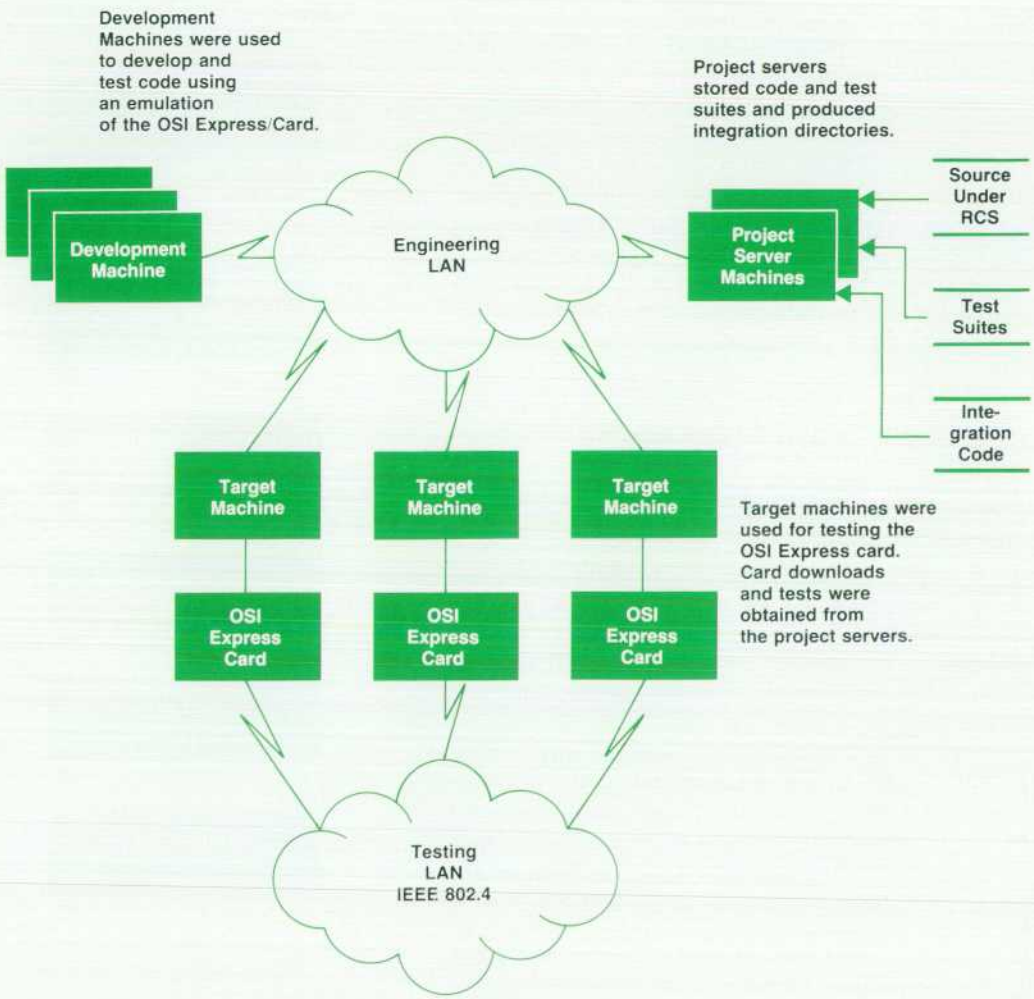


Fig. 1. *OSI Express card testing environments.*

Error Handling

Every CONE call returns an error value to the protocol module making the call. Normally, the exception generator and the scenario interpreter agent would simply propagate the error value returned to them to the next layer. However, the error value returned can be changed by the test modules. In this way error paths can be executed in the protocol modules for unusual error return values from CONE calls. Since the bounce-back module is at the bottom of the stack and cannot propagate error return values, tables were used as explained above for the return value of each CONE call.

Scenario Interpreter

To generate packets to send down the stack, the scenario interpreter is used. Scenarios are test specifications that tell the scenario interpreter what packets to send and what packets to expect to receive. Each scenario has two sides, which can be thought of as a sender and a receiver. Packets are defined using packet definition commands. These constructed packets are sent down the stack using packet send/receive commands. A parameter tells the scenario interpreter whether to send or expect to receive a packet. When a packet is received it is compared to the packet specified in the scenario. If the packets do not match, an error is reported. Repeating sequences of data are generated by macros in the scenario interpreter. For example, a repeating sequence of 5000 bytes is generated with the simple macro !5000. The value of each byte is one greater than the previous byte, modulo 256.

The scenario interpreter also controls the exception generator, bounce-back module, and scenario interpreter agent test modules. Commands to these test modules are sent down the protocol stack in special command packets. Command packets are created in the same fashion as data packets. A parameter indicates whether the packet is a data packet or a command packet. The command packets are absorbed by the test module they are intended for. A test module can also send a command packet to the scenario interpreter. For example, the scenario interpreter can send a command packet to the exception generator telling it to signal the scenario interpreter when a certain number of outbound packets have passed through the exception generator. After sending the packet, the scenario interpreter waits for a response. When the exception generator determines that the specified number of packets have passed through, it sends a command packet to the scenario interpreter telling it that the specified number of packets were sent. After the scenario interpreter receives the expected response it can then proceed. The scenario interpreter can also wait for inbound packets to pass through a test module.

This interaction between the scenario interpreter and the test modules is used to test the many states of a protocol layer. One example is the session layer. Several special packets that the session layer sends to its peer on another machine are preceded by a prepare packet. The two packets are sent one after the other (prepare packet followed by a special packet). However, some states in the session protocol state machine are only entered when a data packet is sent after the prepare packet is received but before the special packet is received (see Fig. 3). To test this case, a prepare and special packet combination is sent down the stack. The special packet is caught and saved by the exception generator. On the receiving side the scenario interpreter waits to receive the prepare packet. After receiving the prepare packet, the scenario interpreter sends a data packet and the receiving side enters the desired state. Finally, the special packet previously captured by the exception generator is released. Without this kind of control, hitting the desired state on the receiving side would only result as a matter of chance.

Another example of packet timing involves the transport layer. The transport layer receives acknowledgments from its peer on another node for the packets it sends. The timing of these acknowledgments is not deterministic. Testing all the transport protocol states requires sending certain packets after an acknowledgment is received. To send a packet after the transport layer receives an acknowledgment requires the scenario interpreter to wait for the exception generator to signal that the acknowledgment packet has arrived.

The scenario interpreter interfaces to the stack via the test harness. The test harness operates in the two different

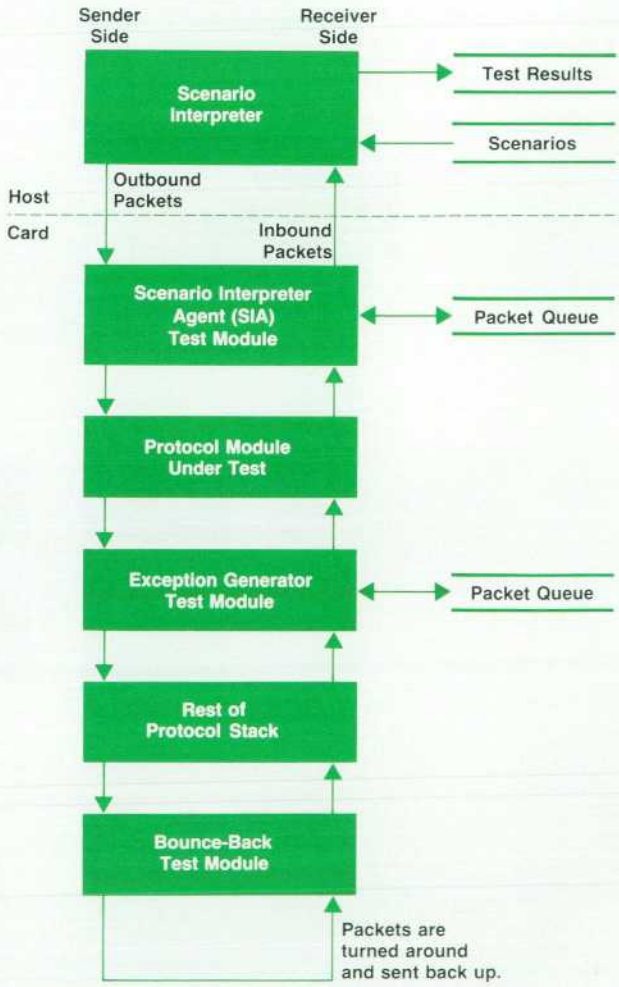


Fig. 2. *Architecture for testing protocol modules.*

environments. In the environment where the stack is actually running on the OSI Express card, the test harness uses a tool called UL-IPC (upper-layer interprocess communication) to communicate with the card. In the user space environment on the development machine, the test harness uses shared buffers (HP-UX IPC) to communicate with the protocol stack which is also running in user space. Fig. 4 shows these environments.

Integration Process

System integration in its simplest form is the process of creating a set of deliverables (e.g., executable product code, test code, etc.) from some source code. For the OSI Express card the integration process was driven by project goals, project size, and environment.

Goals and Results

The integration process for the OSI Express card was designed with follow-on products in mind. CONE exemplifies how this works. CONE allows the protocol modules to be combined in different ways to create new protocol stacks. The integration process also needed the ability to produce additional products without modification to the build process. Like CONE, this involved combining existing code in new ways to produce additional products. The whole problem can be thought of as multidimensional, in that the integration process for the OSI Express card needed to run in a multiple-machine environment, where there were multiple products, each product having multiple versions, each version's code subject to compilation in multiple ways.

The challenge was to create a process that would run effectively in a network environment, supply timely and accurate integration services, and be flexible enough to produce all the targeted outputs required. Other goals for the integration process included quick response to changes by developers, sufficient tracking to create a history of the events that occurred during any given integration, and production of metrics for managing the project.

Although the integration process was modified over the course of the project, what eventually developed was a set of structures and concepts that make integration in this multidimensional environment possible. A successful integration for a given version of a product produced a download file that was able to run on the card, an emulation testing environment to run on development machines, and host-based tools to run on the host machine housing the card. The test environments were similar in that they used the same set of source code to build from. They were different in the deliverables that came out of the environment and the compilers required to produce them. The deliverables for each of these environments was built separately in its own integration directory. These integration directories were built in a standard way so that they had the same look to the build processes regardless of the type of deliverables being built. Standardization of integration directories made it easy to support multiple products, versions, and types of compiles. Having an integration directory with a standard structure residing in a known directory, it was easy to build tools that performed their functions simply by being passed only the name of the integration directory. The flexibility to perform different types of integrations within the integration directory came from the control files (inputs to HP-UX scripts) contained within each integration directory. This information included what source to use, what to build, and compiler options. The integration scripts could then use these files to determine exactly what needed to be done for a particular integration space.

The Process

To get a better understanding of how the integration process functions, let's see what happens when a a new version of a source code module is added to the system. Fig. 5 shows the data flows between some of the components

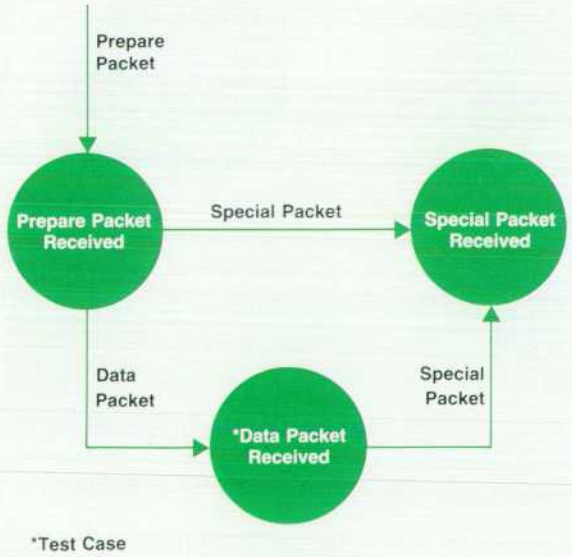


Fig. 3. *Session state machine for the prepare packet and special packet scenario.*

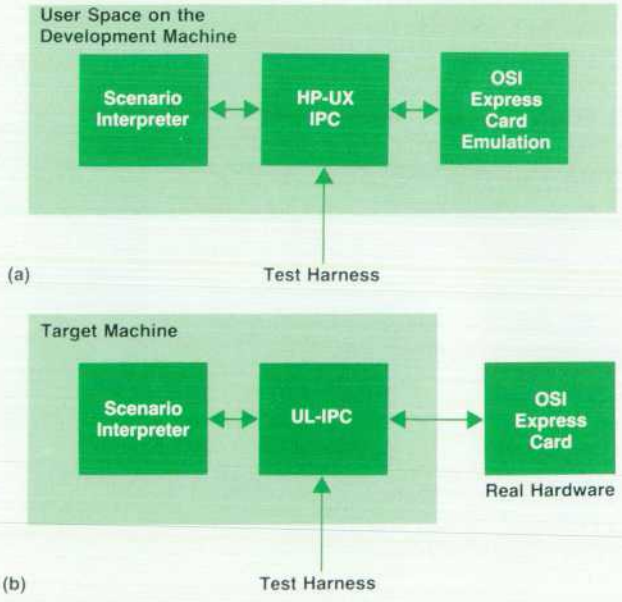


Fig. 4. *(a) Test harness in user space. (b) Test harness used with the real OSI Express card.*

involved in the integration process. Assume that a developer would like to make a bug fix to an existing integration. The first step would be to check out the source code using the HP-UX revision control system (RCS) and put it into a directory on the development system. The specific version is identified by an RCS tag. The RCS tag associates a name with a revision number, so in this case the developer would check out the source code using a tag that is associated with the integration version in which the bug is being fixed. The developer would then make the changes necessary to the source, compile it, and test it using a standard test suite that uses code from the integration directory associated with the change. After the change has passed testing in the emulation space it can be checked back into the common source directories. At this time the new versions are tagged to indicate that they are the latest tested versions and are ready to be integrated. This tag serves as a communication vehicle to tell the integration process that a new version of some module needs to be brought into a specific set of integration directories.

Within each integration directory, a source map contains the name, version, and location of each piece of source code that is needed for a specific integration. The location serves a dual role in that it is the subdirectory path within the source directory of where to get the source code and it is the subdirectory path of where the code belongs within the integration directory. An updated version of this map can be generated by finding out which version of a file needs to be used. To do this a process is run that selects a version of the code to use based on one or more tags. In this case the tag that the developer put on the code would be used. When the module that was updated is looked at, the process would discover that a new version of the file is now needed and a new source map would be created to reflect these changes. The next step would be to place the correct version of the source code into the appropriate directory within the integration directory. This process is accomplished by using the source map previously generated to direct RCS as to what version to check out and where to put it. Other checking is done at this point to make sure the source code residing in a directory is actually the version specified by the source map.

Once valid source code has been placed in an integration directory it is compiled or assembled as required to create relocatable object files, which are then linked into a library. The compiler to use is determined by parsing the integration directory name. Based on a subfield within the name, one of three compiles is chosen: Express card downloads, host-based debugging tools, or card emulations. Each of these types of integration requires that a different compiler be used. The scripts that perform this process verify that the compiler or assembler needed is available on the machine that they are being run from. The compiler or assembler options needed are collected from three locations.

- Options that are specific to the compiler being used are contained in the script that calls the compiler.
- Options that are specific to a library being built come from a control file that describes the name of the library to build, where to build it, where the source can be found, and what compiler options are needed.
- Options that apply to all compiles within an integration

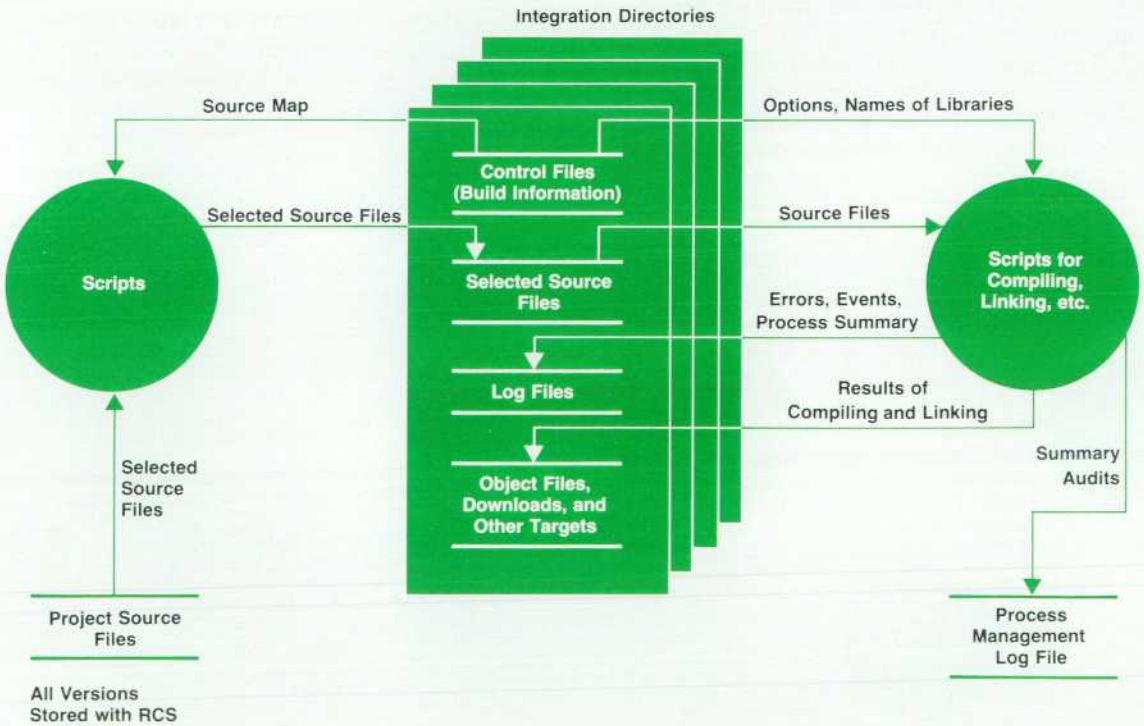


Fig. 5. *Data flow for the portion of the OSI Express card integration involved in extracting the desired source files from the project source directory and building the new object files, download files, and logging files.*

directory come from a global flags file.

These three sets of options are combined and passed to the compiler. Note that since the global flags file and the file that describes how to build libraries are both contained in the specified integration directory, they are unique to a particular integration.

After all the libraries are built other types of targets can be built. These could be programs or downloads depending on the type of integration directory. Again there are control files that specify what target outputs to build, where the inputs can be found, the tool that needs to be called, and where to store the result. If the above steps have all run successfully then the integration directory is again up to date and ready for use by the rest of the team.

The mechanism used to deliver an integration to developers is network mounting. Mounting allows a given machine to have access to another machine's files as if they were stored locally. This method avoids the problem of developers working with out-of-date copies of an integration, and provides immediate availability of an updated integration to all developers.

Data Logging

Since there is a standard set of scripts that provide integration build services, inclusion of consistent logging and error handling was straightforward. The scripts log information to four different log files.

Error Log. The most detailed log contains the warning messages and compile errors generated from calling compilers and other tools needed to produce the target outputs. This file also contains separators indicating what was built and whether or not the process was successful.

Process Log. The process log is a process summary indicating whether a program or library was built successfully. The log file and the error log file are useful for identifying details about process failures or simple compile errors.

Event Log. This file is an event history of the actions upon every module involved in a compile. Every time a module is checked out, compiled, or archived into a library, a record is written to the event log with the time and date, the module name, the version number, what was done (compiled, archived), and if it was successful. This file is established when an integration process is started and is never purged until the the integration directory is removed. The event log provides a useful audit trail to track down things like when a given module changed, what else might have changed at the same time, or whether a particular fix was made.

Process Management Log. This logging file is used to manage the overall integration process. Since there are generally over ten integration directories active at any given time, looking at logging files within each directory to determine what needs to be done is time-consuming and provides no overview of how the integration process is working. Any integration build run on any machine in the network logs to this file to indicate if a major integration process was successful. Information in each record includes the start and stop time, the process run, and the name of the integration directory that was processed.

Acknowledgments

Tim McGowen produced the early version of the OSI Express card integration tools from which the current process evolved. Jeff Ferreira-Pro was instrumental in setting the long range direction of the integration process and the use of RCS, and Kevin Porter and Mike McKinnon produced, tuned, and managed the integration process. Also contributing to the success of the test tools through their work on designing, coding, and enhancing the tools were: Lynn Vaughan, Meryem Primmer, and Jon Saunders.

Authors

February 1990

6 — HP OSI Overview

William R. Johnson

Bill Johnson helped design, develop, and test the session protocol software for the OSI Express card and contributed to the development of standards produced by the U.S.A. National Institute of Standards and Technology (NIST) for the upper layer of OSI. After joining HP in 1985, he helped implement MAP 2.1 networking for HP 1000 A-Series computers. Before coming to HP, he worked for a year in software development at IBM. Bill is now manufacturing networks program manager, responsible for market planning, trade show support, and field education. He is a 1985 graduate of California State University at Chico, and has a BS degree in computer science. Born in Galt, California, he has two children and resides in Auburn. He enjoys golfing, skiing, family outings, and home improvement projects.

8 — OSI Backplane Handler

Glenn F. Talbott

Glenn Talbott has been a development engineer for HP since he graduated from the University of California at Irvine in 1977 with a BS degree in electrical engineering. He helped develop the backplane architecture and design for the OSI Express card. He also worked on the remote console and boot code for the MAP 2.1 project and developed the HP 98645A measurement library. A member of the IEEE, Glenn's professional interests focus on firmware and low-level software and hardware interfaces. A U.S. Marine Corps veteran, he was born in Washington, D.C., and now lives in Auburn, California. His hobbies include softball, skiing, and camping.

18 CONE Software Environment

David A. Kumpf

A graduate of the University of California at Davis with a BS degree in computer science and mathematics, Dave Kumpf joined HP in 1985 as a software engineer right after he graduated. He helped develop and increase the functionality of the test harness, improve performance of the timer manager, and implement the backplane message interface for the OSI Express card. Before that, he worked on the synchronous data link control product for HP's business operating system, MPE XL. Born in Milwaukee, Wisconsin, Dave has two children and resides in Roseville, California. He enjoys backpacking, mountaineering, and reading.

Steven M. Dean

As a member of the Roseville Networks Division technical staff, Steve Dean designed and implemented the buffer manager and scheduler, which are parts of the common OSI networking environment (CONE) for the OSI Express card. After joining HP in 1984, he worked as a process engineer with the manufacturing team that automated the autoinsertion process for building I/O and networking boards. Before that, he served as a development engineer at Amdahl Corporation, working on a small operating system that was used to run diagnostic programs for final reliability tests before product shipment. Steve earned his BS (1982) and MS (1986) degrees in computer science at the California State University at Chico. Born in Chico, he now lives in Rocklin, California, where he enjoys golf, basketball, and fishing.

H. Michael Wenzel

The space shuttle was Mike Wenzel's major concern for five years when he served as a contract officer on the shuttle program and as a captain in the U.S. Air Force. His first project after coming to HP in 1974 was the development of firmware for a raster printer. As a systems designer, he was the primary designer for the OSI Express card and is currently working on I/O systems design and performance. He has developed software for several data communications and network projects, including design of the message manager and architecture for the HP LAN/9000 Series 500 network subsystem. Mike has previously authored an article for the HP Journal (March 1984). He received his BSEE (1969) and MSEE (1971) degrees from the University of Denver. He was born in Alton, Illinois and lives in Granite Bay, California. The father of two children, he enjoys music, hiking, family activities, and investing.

28 OSI Upper-Layer Protocols

Kimball K. Banker

As an R&D engineer in HP's Data Systems Division, Kim Banker helped develop the HP 2250A measurement and control processor. He has worked on other HP factory automation products and on the HP 12065A video output card for the HP 1000 A-Series computer, and helped design and develop the session protocol layer for the OSI Express card. For the past three years, he has participated in ANSI and OSI protocol standards committees and has designed and implemented these standards into HP networking products. Kim serves as ANSI's U.S. session editor and as session rapporteur in ISO SC 21, and is a member of the IEEE 802.6 committee on metropolitan area networks. He came to HP after earning his BS degree in electrical engineering from the University of California at Davis in 1977 and an MS degree in computer engineering at Carnegie-Mellon University in 1979. Kim lives in Rocklin, California and enjoys bicycling, basketball, skiing, and swimming.

Michael A. Ellis

Along with a passion for tandem mountain biking in Northern California, Mike Ellis also is a software engineer with the HP Roseville Networks Division. He designed and implemented the ACSE and presentation protocol modules for the OSI Express card. He also participated in the development of standards and functional specifications for OSI, ACSE, and presentation protocols. After joining HP in 1983, Mike worked on the installation and support of marketing and manufacturing systems. Earlier, he designed and developed an interactive real-time data base management system for use in jail facilities in the United States. His professional society memberships have included chairing the U.S.A. National Institute of Standards and Technology Special Interest Group on Upper Layer Architecture (1987) and membership in the ANSI X3T5.5 Upper Layer Architecture Committee (1986-87). He received a BS degree (1978) in genetics from the University of California at Davis and an MS degree (1983) in computer science from Sacramento State University. Born in Albuquerque, New Mexico, Mike resides in Sacramento, California, and enjoys mountain biking, skiing, windsurfing, and music.

36 OSI Class 4 Transport Protocol

Rex A. Pugh

As a member of the R&D design team at the Roseville Networks Division, Rex Pugh co-designed and developed the OSI class 4 transport protocol for the HP OSI Express card. He currently is investigating routing protocols, focusing on development and standardization of an OSI connectionless routing protocol. He worked on the development of SNA layers 3 and 4 to provide connectivity for the HP 3000 MPE XL operating system with IBM systems. Rex came to HP in 1984 as a software design engineer after graduating from the University of California at Davis with a major in computer science and mathematics. He is a member of the American National Standards Institute X3S3.3 subcommittee, which is developing OSI network and transport layer standards. Born in Arvada, Colorado, he lives in Sacramento, California, and enjoys water and snow skiing.

45 OSI Data Link Layer Design

Judith A. Smith

Shortly after graduating from California State University at Sacramento with a BS degree in computer science in 1985, Judy Smith came to HP as a software development engineer. She designed, implemented and tested software used in the OSI Express card and in the past has tested LAN cards. Currently, she is working on diagnostics for LAN interface cards for the HP-UX and MPE XL operating systems. Born in Sacramento, California, Judy is an active member of the Sacramento Valley Chapter of the Society of Women Engineers. She resides in Roseville, California and enjoys singing with the Sweet Adelines, cycling, and downhill skiing.

Bill Thomas

Bill Thomas's major professional interest is designing the hardware-software interfaces between computers and their peripherals. Before helping design and implement the HP OSI Express card, he designed hardware and software for IC test systems and HP-IB interface cards. He also designed software that created an automated test and measurement environment for HP 1000 computer systems. Bill came to HP in 1969. He received his BS degree in 1969 at the University of California at Berkeley and an MS degree in 1972 at Colorado State University, both in electrical engineering. At Berkeley, he was a member of the engineering fraternity, Eta Kappa Nu, and is now a member of the IEEE. Active in amateur radio emergency communications, he lives in Carmichael, California.

51 OSI Design for Performance

Elizabeth P. Bortolotto

Liz Bortolotto served as a performance engineer working with the Roseville Networks Division R&D team that developed and implemented the OSI Express card. She came to HP in 1983 as a laboratory firmware engineer, and was responsible for firmware design for the HP 98642 four-channel multiplexer and the HP DMI/3000 system. She is cur-

rently a technical marketing engineer. Liz earned a BS degree (1982) and an MS degree (1988) in computer science from California State University at Chico. Born in Kansas City, Missouri, she and her husband live in Loomis, California, and are expecting their first child. Liz is very interested in environmental concerns and also enjoys skiing, biking, dancing, volleyball, and music.

59 OSI Software Diagnostic Program

Joseph R. Longo, Jr.

Rick Longo is a software development engineer who was responsible for the design and development of diagnostic and debugging tools for the HP OSI Express card. He came to HP as a summer student in 1980 and joined full-time in 1981, working on manufacturing applications and technical support projects before moving to the Roseville Networks Division laboratory in 1985. Rick is currently researching network management software for LAN devices. While studying for his BS degree (1980) in computer science from the California State University at Chico, he worked for a year at Burroughs Corporation. Born in Los Molinos, California, he resides in Roseville, California. Rick enjoys volleyball, golf, softball, hiking, and family outings.

67 OSI Support Features

Jayesh K. Shah

Jay Shah has traveled a long way from his birthplace of Aden in Southern Yemen to his present hometown of Citrus Heights, California. He came to HP in 1988 as a development engineer in the Roseville Networks Division, working on troubleshooting methods and format definition for the HP OSI Express card. Currently, he's testing LAN cards for HP Vectra computer systems. Jay earned his BSE degree (1986) in computer systems engineering from Arizona State University and his MS degree (1988) in computer science from the University of California at Los Angeles. He is a member of the IEEE and the ACM. His hobbies include reading, skiing, and traveling.

Charles L. Hamer

Chuck Hamer is a technical marketing support engineer in HP's Roseville Networks Division. He developed the support strategy for the MAP 3.0 product, the troubleshooting methodology for the OSI Express card, and the MAP 3.0 troubleshooting training module. He also worked on the MAP 2.1 product and provided on-line support for DS/1000 and X.25/1000 systems at the HP Network Support Center. Before coming to HP, Chuck was a systems engineer for Data Switch Corporation, responsible for sales support of peripheral and communications matrix switches, and a network designer with Bechtel Power Corporation, where he designed an internal communications network. Born in Spokane, Washington, Chuck has a BA degree in economics from the University of California at Berkeley. He lives in Ophir, California, and enjoys canoeing and hobby farming. He is the treasurer of the Ophir Elementary School parent-teachers association.

72 OSI Protocol Stack Integration

Neil M. Alexander

As a software development engineer, Neil Alexander provided software configuration management processes and tools for the HP OSI Express project. Before he came to HP in 1988, Neil supervised the information center at the Sacramento Municipal Utility District. He also served as a lead programmer at Wismer and Becker Construction Engineers, as an application analyst at Control Data Corporation, and an associate engineer at Lockheed Missiles and Space Company. Born in Sacramento, California, he received a BS degree (1972) in computer science and mathematics from the California State University at Chico. He currently resides in Roseville, California.

Randy J. Westra

Randy Westra came to HP in 1983 as a development engineer shortly after he earned BS and MS degrees in computer science from the University of Iowa. He worked in the R&D laboratory at the Logic Systems Division developing an editor, compilers, and debuggers for the HP 64000 microprocessor development system. After transferring to the Roseville Networks Division, Randy developed test tools and the session protocol layer for the OSI Express card. Born in Sioux Center, Iowa, Randy lives in Roseville, California with his wife and new daughter. He enjoys swimming, reading, and traveling.

80 Lightwave Signal Analysis

Christopher M. Miller

As a project manager for the past five years at HP's Signal Analysis Division, Chris Miller was responsible for development of the HP 71400A lightwave signal analyzer. Before that, he was the project manager for the HP 71300A millimeter spectrum analyzer. Earlier, he was with HP Laboratories, where he designed high-speed bipolar and GaAs integrated circuits. He came to HP in 1979 from Hughes Aircraft Company, where he designed electronic systems for laser target designators and cryogenic coolers for infrared sensors. Chris coauthored an International Microwave Symposium paper on a high-speed photoreceiver and has written several other symposium papers on RF and lightwave subjects. Born in Merced, California, he earned a BSEE degree (1975) from the University of California at Berkeley, and an MSEE degree (1978) from the University of California at Los Angeles. Married and the father of two sons, he lives in Santa Rosa, California, where he enjoys wine tasting, running, skiing, body surfing, and camping.

92 Fiber Optic Interferometer

Douglas M. Baney

Now a doctoral candidate in applied physics at the Ecole Nationale Supérieure des Télécommunications in Paris, Doug Baney has been with HP's Signal Analysis Division since 1981, specializing in the design of microwave amplifiers and frequency multipliers. Most recently, he contributed to the development of the HP 11980A lightwave interferometer. He earned his BS degree (1981) in electronic engineering from the California Polytechnic State University at San Luis Obispo and an MSEE degree (1986) from the University of California at Santa Barbara with an HP fellowship. Doug is an author and coauthor of several scientific and conference articles published in English and French on the laser power spectrum, and is named a co-inventor in a patent for an optical measurement technique. Born in Wayne, New Jersey, Doug now lives in Paris. When he returns to California, he plans to continue his favorite activity, sailing his Hobie Cat.

Wayne V. Sorin

Wayne Sorin developed new fiber-optics-based measurement techniques and instrumentation after his arrival at HP Laboratories in 1985. He also contributed to the idea for the gated delayed self-homodyne technique during development of the HP 11980A fiber optic interferometer. While attending Stanford University, he studied evanescent interactions in single-mode optical fibers. Born in New Westminster, British Columbia, Wayne earned a BS degree (1978) in physics and a BS degree (1980) in electrical engineering from the University of British Columbia, and MSEE (1982) and PhD degrees (1986) from Stanford University. Wayne holds five patents on fiber optics components and is a member of the IEEE and the OSA. He is the author of 14 technical papers in the field of fiber optics components and lasers, and teaches a fiber optics course at California State University at San Jose. His major professional interests are developing new fiber optics-based measurement techniques and instrumentation. Married and the father of a son, he enjoys playing tennis and soccer in his hometown of Mountain View, California.

High-Speed Lightwave Signal Analysis

This analyzer measures the important characteristics of high-capacity lightwave systems and their components, including single-frequency or distributed feedback semiconductor lasers and broadband pin photodetectors.

by Christopher M. Miller

THE LOW PROPAGATION LOSS and extremely broad bandwidth of single-mode optical fiber have contributed to the emergence of high-capacity digital transmission systems and analog-modulated microwave-frequency systems. New lightwave components have been developed to support these high-speed systems. Most notable among these components are single-frequency or distributed feedback semiconductor lasers and broadband pin photodetectors.

The HP 71400A Lightwave Signal Analyzer has been designed to measure the important characteristics of these lightwave components and systems, such as signal strength and distortion, modulation depth and bandwidth, intensity noise, and susceptibility to reflected light. When the lightwave signal analyzer is used in conjunction with the HP 11980A Fiber Optic Interferometer (see article, page 92), the linewidth, chirp, and frequency modulation characteristics of single-frequency lasers can be measured.

System Description

The HP 71400A Lightwave Signal Analyzer, Fig. 1, is part of the HP 70000 Modular Measurement System, which provides RF, microwave, and now lightwave measurement capability. The HP 70000 is an expandable system and can be upgraded as requirements grow and new modules become available. For example, the HP 71400A can measure lightwave modulation up to 22 GHz. However, substitution of a 2.9-GHz RF front-end module makes the system an HP 71401A, which for certain applications may be a more cost-effective solution. In addition to being lightwave signal analyzers, the HP 71400A and HP 71401A also function as microwave and RF spectrum analyzers. The current offering of HP 70000 modules is shown in Fig. 2.

A simplified block diagram of the HP 71400A is shown in Fig. 3. The key module in the system is the HP 70810A Lightwave Receiver. Light from the input fiber is collimated by a lens and focused onto a high-speed pin photodetector. The optical attenuator in the collimated beam prevents overload of the front end. The photodetector converts

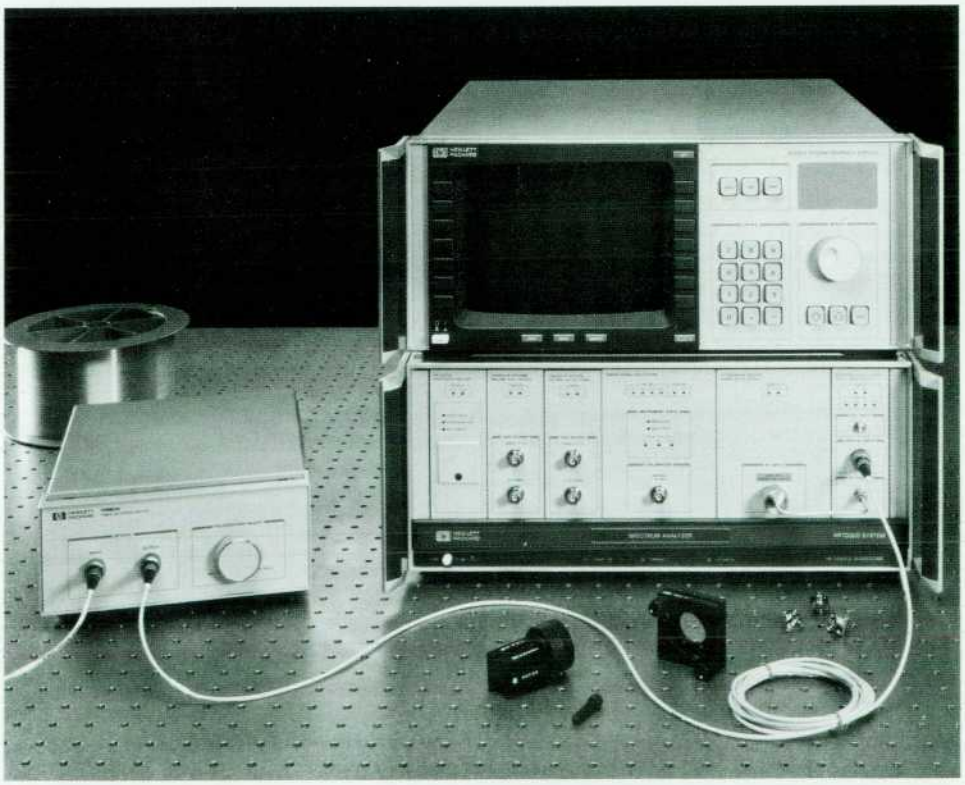


Fig. 1. *An HP 70000 Modular Measurement System configured as an HP 71400A Lightwave Signal Analyzer, shown with the HP 11980A Fiber Optic Interferometer.*

Fig. 2. *The HP 70000 family of modular instruments includes lightwave signal analyzers, RF, microwave, and millimeter-wave spectrum analyzers, tracking generators, microwave power meters, digitizers, and vector voltmeters.*

photons (optical power) to electrons (photocurrent). The time varying component of this photocurrent, which represents the demodulated signal, is fed through the preamplifier to the input of the microwave spectrum analyzer. The dc portion of the photocurrent is fed to a power meter circuit. Thus the same detector is used to measure both the average power and the modulated power.

The lightwave signal analyzer is often confused with an optical spectrum analyzer (also called a spectrometer). Although both instruments have frequency-domain displays, the information they provide is quite different. The optical spectrum analyzer shows the spectral distribution of average optical power and is useful for observing the modes of multimode lasers or the sidelobe rejection of single-frequency lasers. Its measurement resolution is typically about 0.1 nm or approximately 18 GHz at a wavelength of 1300 nm. The lightwave signal analyzer displays the total average power and the modulation spectrum, but provides no information about the wavelength of the optical signal. This distinction is shown in Fig. 4.

Lightwave Receiver Design

Four major subassemblies make up the lightwave receiver module. They are the optoblock, the optical microcircuit, the average power circuitry, and the optical attenuator control circuitry. The optical and high-frequency RF circuits are located close to the front-panel connectors, as shown in Fig. 5.

The optoblock is essentially an optical-mechanical assembly that serves two functions. It collimates the light at the input and refocuses it onto the detector, and along the way it allows for attenuation of the light. The input to the optoblock uses the fiber optic connector adapter system developed by HP's Böblingen Instruments Division. The adapter system is based on the precision Diamond® HMS-10/HP fiber optic connector.[1] This adapter design allows easy access to the ferrule for cleaning, provides a physical, low-return-loss contact to the input fiber, and allows mating to any of five different connector systems: HMS-10/HP,

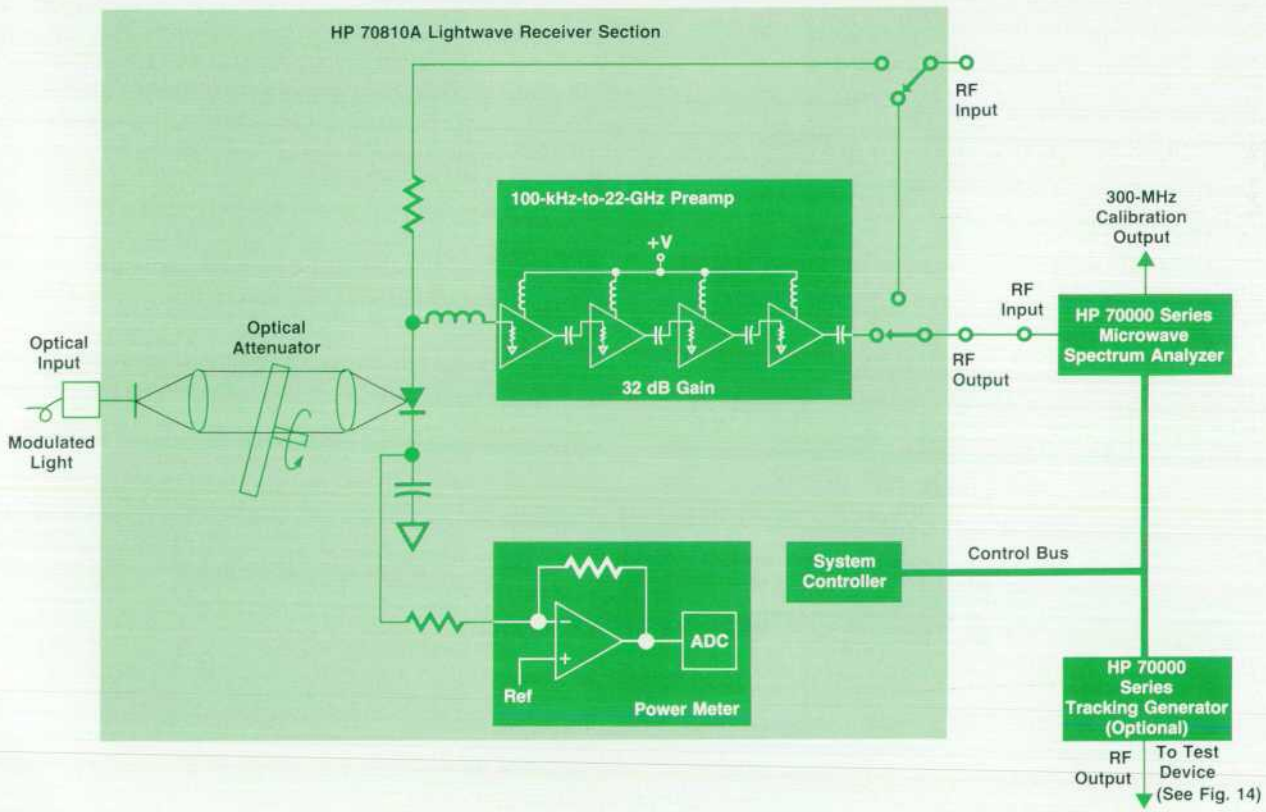


Fig. 3. *The lightwave signal analyzer system, highlighting the HP 70810A Lightwave Receiver section.*

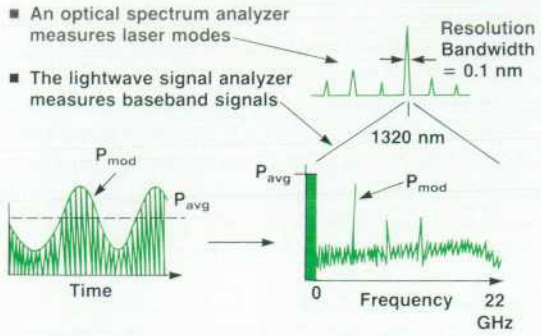


Fig. 4. *Measurement differences between a lightwave signal analyzer and an optical spectrum analyzer.*

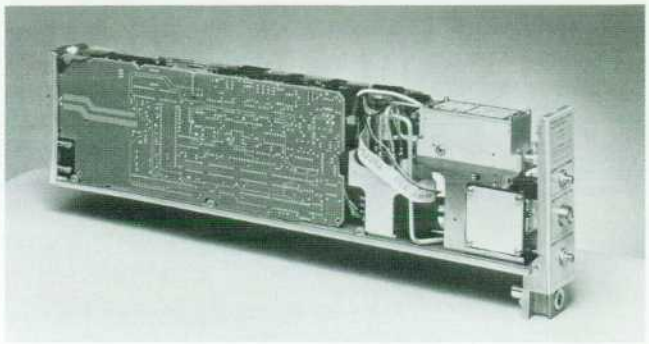


Fig. 5. *HP 70810A Lightwave Receiver, showing the proximity of the optoblock and optical microcircuit to the front panel.*

FC/PC, ST, biconic, and DIN. Internally, the fiber butts up against a piece of glass on the backside of the connector. Index matching fluid at this interface and an antireflection coating on the glass-to-air surface help maintain the connector's good input return loss.

Exiting the input connector, the light passes into air. The diverging beam is collimated into an expanded parallel beam, which then passes through a continuously variable 0-to-30-dB circular filter. The filter is coated with a metallic neutral density layer which reduces the wavelength dependence of the optical attenuation. The filter is angled to the optical axis to prevent reflection back to the optical connector. The positioning of the filter with the drive motor, optical encoder, and drive electronics will be described later.

A mirror positioned at a 45-degree angle to the optical path directs the light to the output lens, which focuses it onto the detector. The mirror is partially transmissive, which allows the light to be aligned to the detector by viewing the reflected light from the illuminated detector with a microscope objective, as shown in Fig. 6.

Optical Microcircuit

The optical microcircuit containing the pin photodiode and microwave preamplifier is mated to the optoblock. The pin detector works by converting received optical power into an electrical current. Light at wavelengths between 1200 and 1600 nm enters through the antireflection-coated top surface, and passes through the transparent InP p layer. Electron/hole pairs are created when the photons are absorbed in the InGaAs i region. Reverse bias is applied across the device, sweeping the electrons out through the bottom n-type InP substrate, while the holes are collected by the p-type top contact. The active area is only 25 μm in diameter, which keeps the device capacitance low. This, along with the short transit time across the i layer, contributes to a 20-GHz device bandwidth.

Electrical photocurrent from the photodiode's anode is terminated in a preamplifier that has an input impedance of 50Ω and a bandwidth of 100 kHz to 22 GHz. The cathode side of the photodiode is bypassed by an 800-pF capacitor to provide a good RF termination. The preamplifier helps overcome the relatively high noise figure of the microwave spectrum analyzer shown in Fig. 3. It also improves the overall system sensitivity. The preamplifier has about 32 dB of gain, provided by a cascade of four microwave monolithic integrated circuit (MMIC) amplifier chips, each with a nominal gain of 8 dB (see box, page 84).

The optical microcircuit package includes the bias board assembly. This was done to shield the bias lines from any radiated electromagnetic interference (EMI). In addition, a

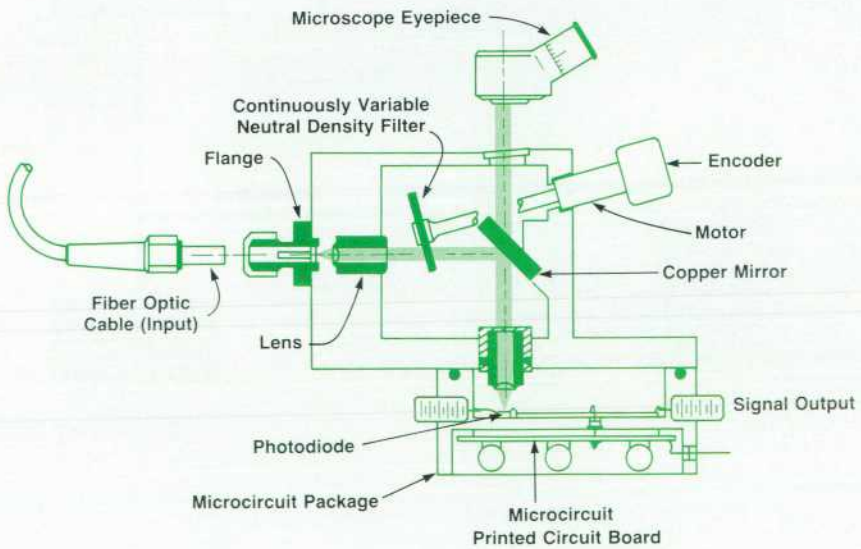


Fig. 6. *Optoblock and microcircuit assembly showing optical alignment.*

spiral wound gasket is placed at the microcircuit-optoblock interface to reduce the likelihood of any EMI pickup. A rubber O-ring gasket is also placed at this interface to help seal the microcircuit assembly.

Average Power Circuitry

Connected to the cathode of the photodetector is a transimpedance amplifier, which is the input circuit for the average power circuitry. The design of the average power meter was highly leveraged from the HP 8152A Optical Average Power Meter.[2] Fig. 7 shows the block diagram of the average power circuitry, which incorporates four key elements: a transimpedance amplifier, offset correction, wavelength gain correction, and digitization.

In this design, the transimpedance amplifier serves a dual role. It converts photocurrent into an equivalent voltage depending on which feedback resistor is selected. In addition, it provides the reverse bias for the photodiode. The input amplifier is an OPA111BM, which was chosen for its low input offset characteristics. The transimpedance amplifier is followed by a difference amplifier which removes the bias voltage component from the signal component being measured. This amplifier is followed by an internal gain-adjust amplifier, which is set to produce a 4-volt output when −20 dBm of optical power is present at the input.

The two values of feedback resistors, along with the three values of step gain, provide six different range settings. The proper range is automatically selected as a function of input power level. The design allows a measurement range of +3 dBm to less than −60 dBm when there is no optical attenuation present. With the attenuator set to 30 dB, power levels up to +33 dBm can be measured. In the lowest range the feedback resistor is 3.33 MΩ and at −60 dBm the photocurrent is less than 1 nA, so guarding is used to prevent offset errors resulting from leakage currents.

λ DAC, ADC, and Offset DAC

To compensate for the photodiode's responsivity variations with wavelength, a multiplying digital-to-analog converter called the λ DAC is used as a variable-gain amplifier. The average power reading of the HP 71400A is calibrated at two wavelengths: 1300 nm and 1550 nm. The responsivity at 1300 nm is defined as 0 dB and the relative responsivity value at 1550 nm is within ±0.5 dB of this value. To calibrate the HP 71400A to an external reference or if the customer chooses to operate at another wavelength, the value of the λ DAC can be varied by ±3 dB using the USER CAL function.

The operation of the analog-to-digital converter (ADC) circuitry is identical to that of the HP 8152A.[2] An AD7550 13-bit ADC is used, with the following relationship for a 10-dB range step:

$$n = A_{in}(4096/V_{fs}) + 4096,$$

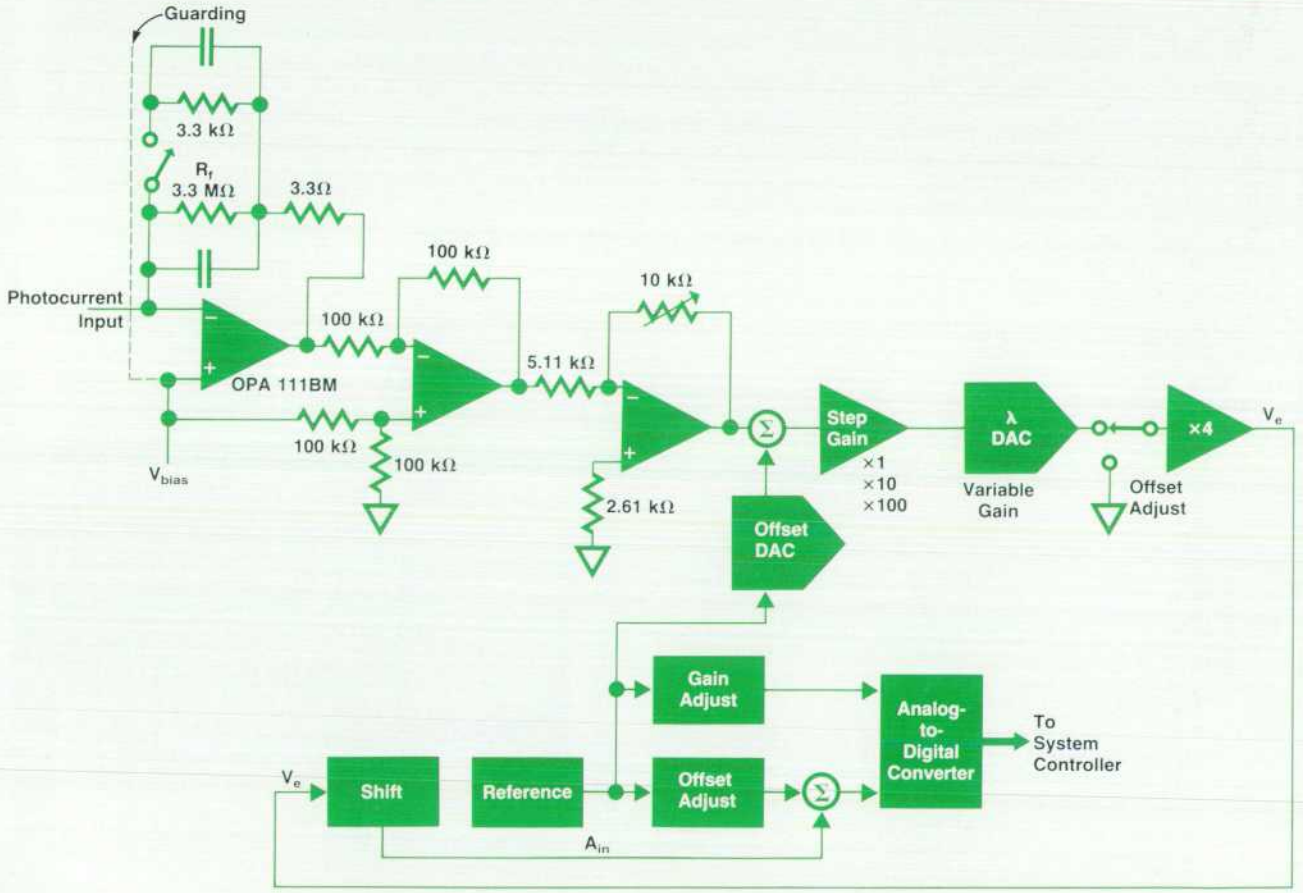


Fig. 7. *Block diagram of the average power meter circuit.*

A Broadband Instrumentation Photoreceiver

A broadband microwave amplifier is the key to achieving good photoreceiver sensitivity without compromising the system's bandwidth. The amplifier in the HP 70810A Lightwave Receiver consists of four microwave monolithic distributed amplifiers[1] that have their low-frequency corner extended down to 100 kHz. Each amplifier chip is an independent distributed amplifier consisting of seven GaAs FETs spaced along synthetic 50Ω input and output transmission lines. In this distributed design, the signal from each FET adds to that of its neighbors to produce gain at frequencies beyond the cutoff of the individual FETs.

between the 10-pF on-chip bypass capacitor and the inductance of the bond wire connected to the external bypass capacitor. The amplifier bias is fed into the reverse termination end of the drain line through a bias choke. This feed point has the advantage of less sensitivity to bias choke shunting impedances. The bias choke is constructed by close-winding insulated, gold-plated copper wire around a high-magnetic-loss cylindrical core. Interstage coupling is through a 1000-pF TaO_5 thin-film integrated circuit capacitor in parallel with a 0.047-μF ceramic capacitor. The integrated capacitor has good microwave performance and

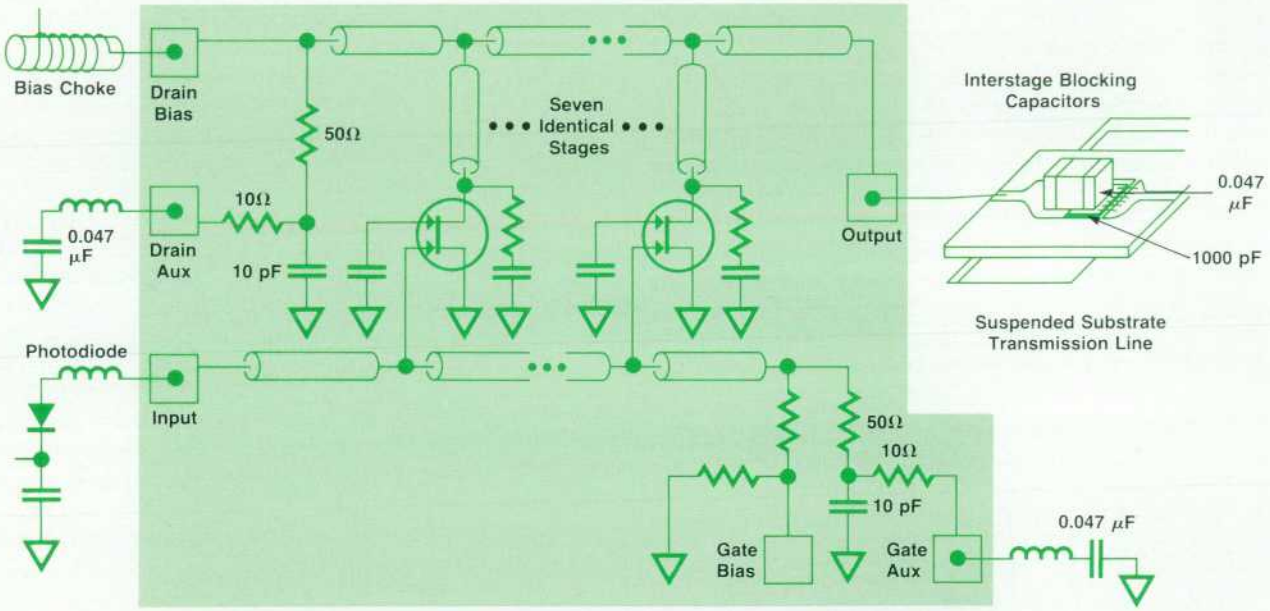


Fig. 1. *Detail of single amplifier stage, showing bias choke, external bypass capacitors, and the interstage coupling capacitors for the MMIC chip.*

Details of the input stage of the photoreceiver (Fig. 1) show how the good low-frequency and high-frequency performance of the amplifier cascade is achieved. The gate and drain artificial transmission lines are externally bypassed with 0.047-μF ceramic capacitors. A 10Ω resistor is used to prevent parallel resonance

the large ceramic capacitor is mounted on a short suspended-substrate transmission line segment to reduce parasitic capacitance to ground. Typical gain and noise figure for the cascade are shown in Fig. 2.

To achieve maximum photoreceiver sensitivity, the photo-

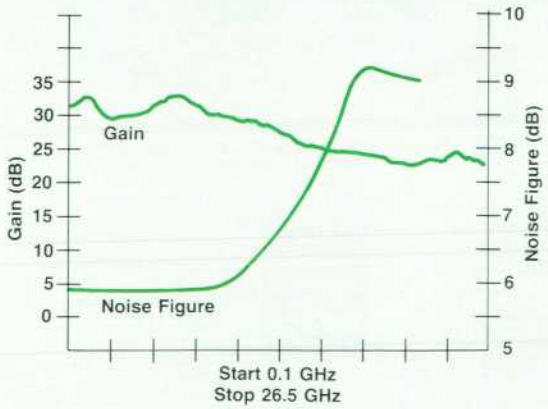


Fig 2. *Gain and noise figure of the four-stage amplifier cascade over the 100-MHz-to-22-GHz frequency range.*

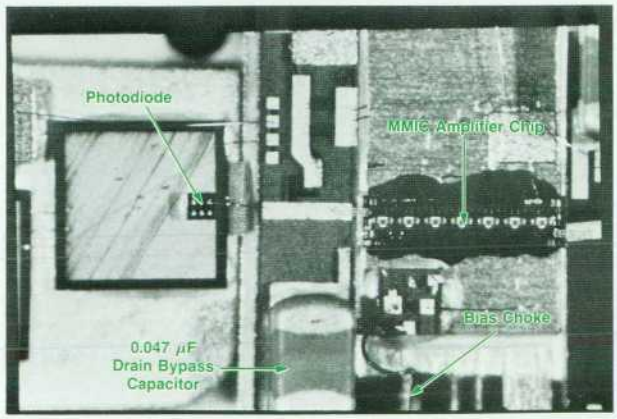


Fig. 3. *Photograph of the photodiode and first stage of the amplifier.*

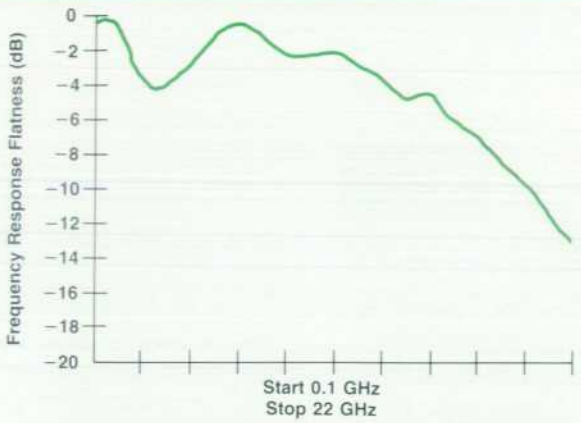


Fig. 4. *Combined frequency response flatness of the photodetector and the amplifier.*

detector is not back terminated. Consequently, the photodiode is placed as close as possible to the amplifier input to minimize mismatch loss. This is shown in Fig. 3. The combined frequency reponse of the photodetector and amplifier is shown in Fig. 4. Overall freqency response roll-off for the optical receiver microcircuit is 13 dB, of which 8 dB is from the amplifier and 5 dB is from the photodiode.

Reference

1. J. Orr, "A Stable 2-26.5 GHz Two-Stage Dual-Gate Distributed MMIC Amplifier," *IEEE-MTT-S International Microwave Symposium Digest*, HH-4, 1986, pg. 817.

Dennis Derickson
Development Engineer
Signal Analysis Division

where n is the number of ADC counts, A_{in} is the analog input voltage, and V_{fs} is the full-scale input voltage. A one-millivolt change in the voltage at V_e in Fig. 7 produces a one-count change in the ADC reading. To center the input voltage range on the ADC range, V_e is shifted down by 3.901V to produce the following relationship:

V_e(V)	A_{in}(V)	ADC Counts	Relative Power
>7.996		overflow	
7.996	4.095	8191	2.0
4.000	0.099	4195	1.0
0.400	−3.501	595	0.1
0.000	−3.901	195	0.0
−0.195	−4.096	0	
<−0.195		underflow	

The relative power in a given range is computed by subtracting 195 from the ADC counts and then dividing by 4000 counts.

Because the reverse-biased, uncooled InGaAs photodiode has a substantial dark current of several nanoamperes that is present under no illumination, offset compensation had to be designed to correct for offsets that could be larger than the signal in the most sensitive range. There are two convenient places to put the offset correction DAC: before or after the step-gain amplifier. Placing the offset correction after the step-gain amplifier has the advantage that the resolution of the offset correction is constant and independent of range, and there can be a one-to-one correspondence between an ADC count and an offset DAC count. However, the disadvantage is that the effective offset correction range, referenced to the input, decreases as the step gain is in-

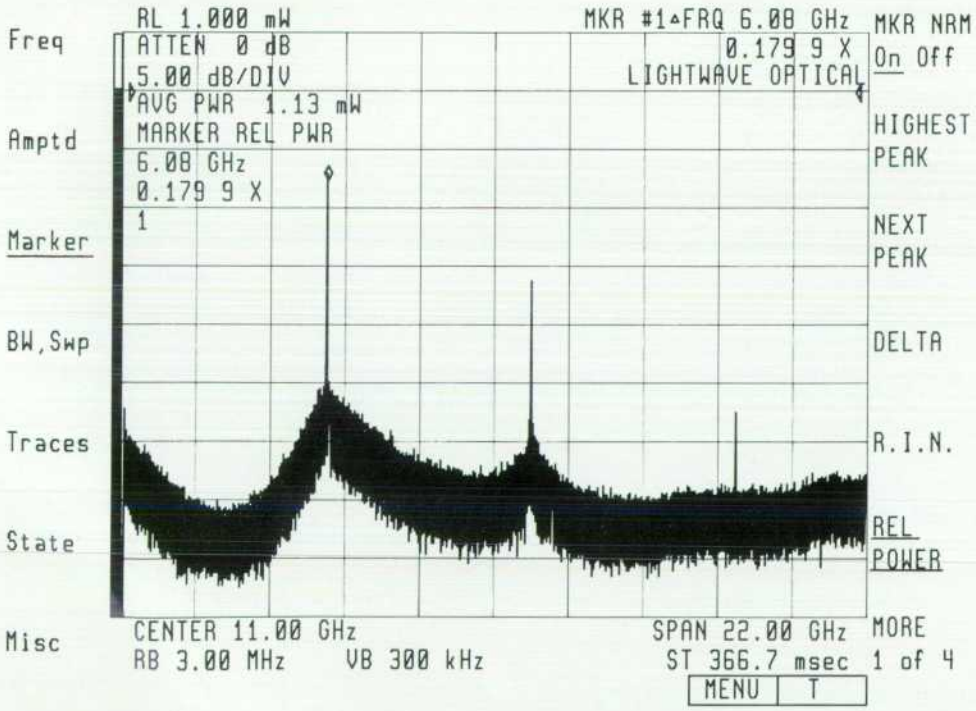


Fig. 8. *Typical display of the lightwave signal analyzer showing the menu key labels (firmkeys and softkeys), the average power bar, and the modulated lightwave spectrum.*

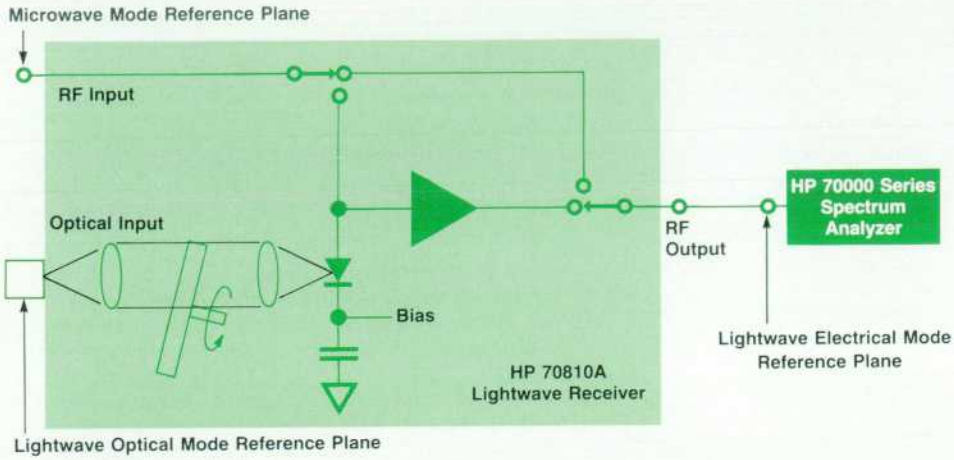


Fig. 9. *Diagram indicating the measurent reference plane for each mode of the lightwave signal analyzer.*

creased. In this design, a significant offset can exist because of a large dark current component, particularly at the maximum instrument operating temperature of 55°C. Therefore, a 12-bit offset DAC is used to supply an offset correction at the input of the step-gain amplifier that can compensate for as much as 250 nA of dark current at the photodiode. This causes a corresponding loss in offset resolution in the most sensitive range, and these leftover residual offset counts are recorded and subtracted in the firmware.

Input Optical Attenuator

The control circuitry for the input optical attenuator was highly leveraged from another instrument, the HP 8158A Optical Attenuator.[3] The digital motor controller uses an 8-bit microprocessor with 128 bytes of internal RAM and a 16-bit internal timer. This processor sets the pulse width of the motor drive, whose period is 31.25 kHz. The motor driver itself is a simple transistor full bridge circuit. An optical encoder, driven in quadrature mode, provides an effective resolution of 2048 positions per revolution. The positions, corresponding to 1-dB steps of the linear filter wheel, are measured at both 1300 nm and 1550 nm, and these positions are stored in EEPROM. The HP 71400A uses the same motor control firmware as the HP 8158A, which is based on a PD (proportional differential) algorithm.[3]

Display and User Interface

The goal of the display and user interface design was that both optical and microwave scientists and engineers would be comfortable with it. Basically, the design follows the HP 70000 electrical spectrum analyzer formats, and integrates the optical functionality into this context.

Primarily menu-driven, the user interface consists of a set of firmkeys on the left side of the display. As shown in Fig. 8, these firmkeys are the basic analyzer control function headings, which when selected, pull up submenus on the softkeys on the right side of the display. These softkeys for the most part represent immediately executable functions. Control of optical parameters such as wavelength calibration, optical attenuation, power meter offset zeroing, and optical marker functions is offered on submenus with related analyzer functions.

Displayed intensity modulation of a lightwave carrier has essentially the same appearance as the electrical modulation spectrum, so the basic display format mimics that of the electrical spectrum analyzer with one important difference. This difference is the display of the average power bar on the left side of the screen (see Fig. 8). In addition to providing an accurate average power indication, the graphical power bar representation makes optical alignment much easier. The average power and modulated power displays are coupled in that they are have the same scale and are referenced to the same absolute amplitude level.

The lightwave signal analyzer has three measurement modes. Two modes are for making lightwave measurements—the input is the optical input of the lightwave section. The difference in these two modes is in the display units. In lightwave-optical mode, the display is referenced to the optical input connector and the display is calibrated in optical power units. In lightwave-electrical mode, the display is referenced to the input of the electrical spectrum analyzer and the display is calibrated in electrical power

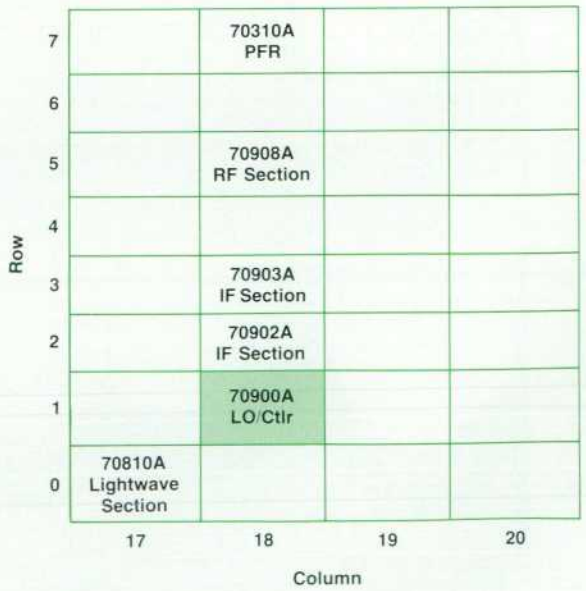


Fig. 10. *Module address map for the lightwave signal analyzer.*

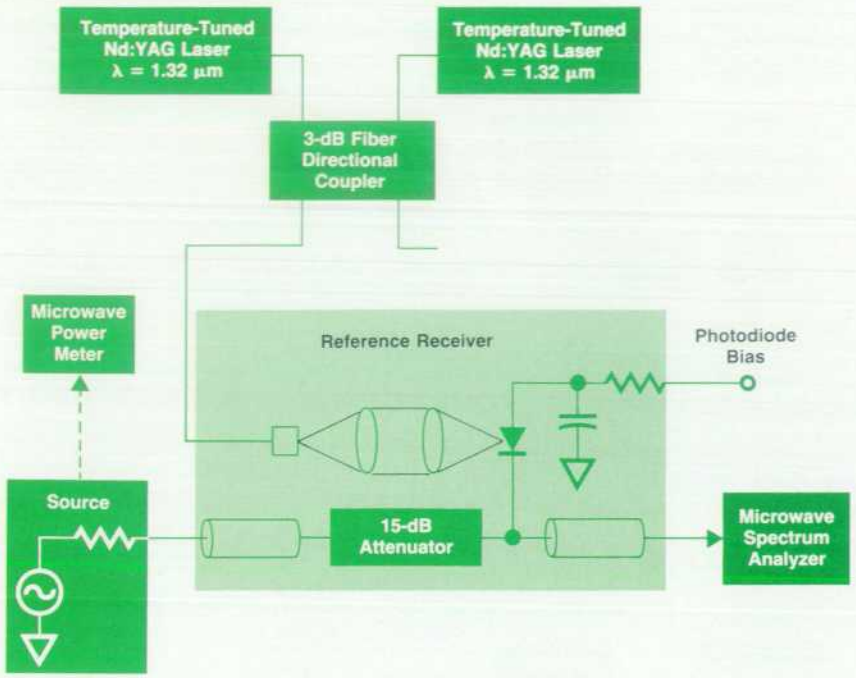


Fig. 11. *Diagram of the reference receiver used to calibrate the lightwave signal analyzer.*

units. This mode was implemented because, before the lightwave signal analyzer was developed, customers became accustomed to using electrical spectrum analyzers to make these lightwave measurements, and have specified some of these measurements in electrical power units. The display units of these two modes are related by the following equation:

$$P_{elec}(dBm) = 2P_{opt}(dBm) + 10\log[(1\ mW) \times r^2 \times 50\Omega \times G_{v(lin)}{}^2]$$

where r is the responsivity of the photodiode and $G_{v(lin)}$ is the linear voltage gain of the microwave preamplifier.

The third measurement mode, the microwave mode, is for making strictly electrical measurements. In the microwave mode the RF input of the lightwave section is used and the optical path is bypassed. The three modes are shown in Fig. 9.

Firmware Design Overview

As previously mentioned, the HP 71400A Lightwave Signal Analyzer is part of the HP 70000 Modular Measurement System (MMS). In this system, certain instrument modules, designated as masters, can control the operation of other modules, designated as slaves. Communication between modules occurs over the internal high-speed modular system interface bus (HP-MSIB). Whether a module operates as a master or a slave is determined by the module's internal firmware design and its relative position in the module address map.[4] The address map for the lightwave signal analyzer, indicating the row and column positions of the modules in the system, is shown in Fig. 10. The HP 70810A lightwave section, in the row 0, column 17 location, is the master module, controlling all the modules at higher row and column addresses up to the column where another master is present on row 0. Thus, a number of independent instruments can be configured in the system, simultaneously making measurements.

Firmware for the lightwave module is written in the C programming language, and the compiled code runs on a Motorola 68000 microprocessor. The firmware consists of three major components:

- The pSOS operating system, written by Software Component Group, Inc.[5] This is a full multitasking operating system.
- The MMS instrument shell. This is a large, integrated collection of support routines and drivers intended to supply functionality to most HP 70000 Series modules.
- Lightwave-section-specific code written on top of the instrument shell and the pSOS operating system.

The lightwave-specific code encompasses a number of elements. Communication, measurement coordination, and control of the HP 70900A local oscillator module must be established and maintained. The HP 70900A local oscillator module is the controller of the electrical spectrum analyzer and is allocated a display subwindow for presenting the lightwave modulation spectrum. An array containing the flatness corrections for the frequency response of the optical microcircuit is stored in the HP 70810A's EEPROM and is passed over the HP-MSIB to the HP 70900A to apply as a correction to the displayed trace. A small vertical stripe on the left edge of the window is reserved for the average power bar, which the HP 70810A generates. The HP 70900A is relied upon to display all annotation normally associated with the spectrum analyzer except for the active parameter area, the message area, the mode annotation, and the average power and optical attenuation annotation, for which the HP 70810A is responsible. The manual interface is handled entirely by the lightwave section. All remote commands and parameters are parsed by the HP 70810A. Commands that are intended to modify the spectrum analyzer are passed along to the HP 70900A.

When the HP 70810A is operated without an HP 70900A as its slave, it operates in a stand-alone mode. In this mode the module can be used as a lightwave converter, can make

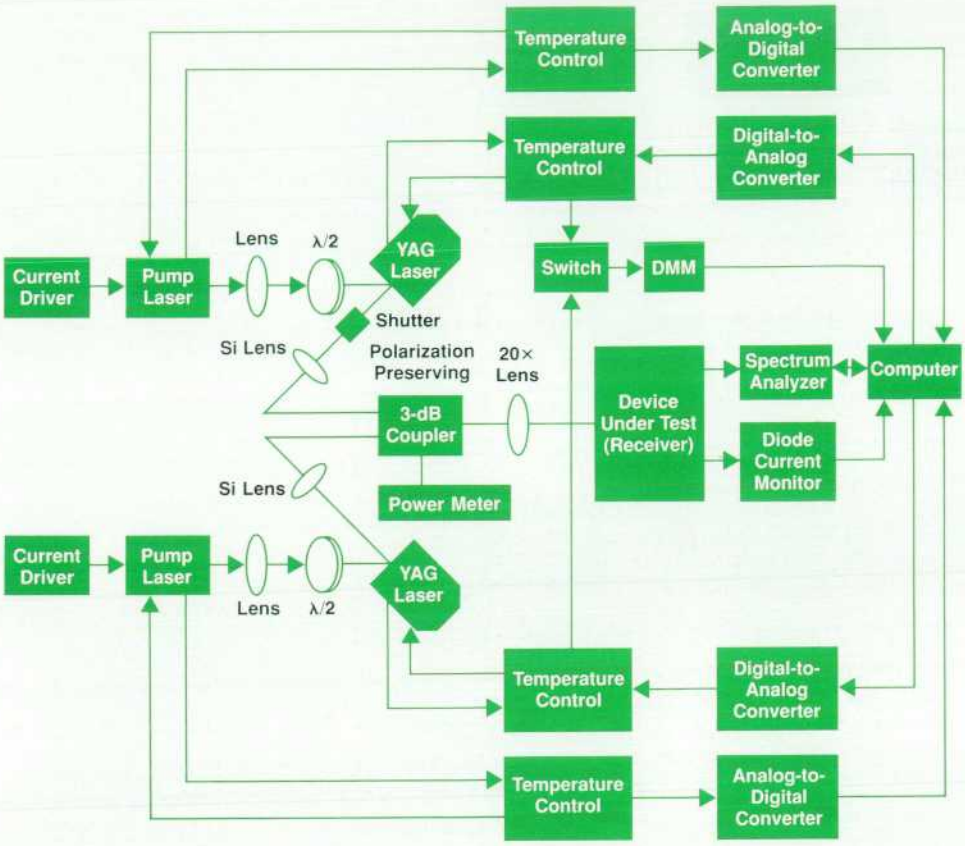


Fig. 12. *Heterodyne laser system for calibrating reference receivers.*

average optical power measurements, and can control the optical attenuation.

Calibration

A major contribution of the HP 71400A is its optical calibration. To our knowledge it is the only lightwave product that is calibrated in both relative and absolute power levels out to a modulation bandwidth of 22 GHz. The lightwave signal analyzer is calibrated by comparing its response at 250 frequency points to that of a reference receiver. This specially packaged reference receiver is calibrated as shown in Fig. 11. All sources of electrical frequency response error, including detector capacitance, mismatch loss, cable loss, and spectrum analyzer amplitude errors, are measured by feeding a power-meter-calibrated microwave signal through the fixture and into the spectrum analyzer. The frequency response of the reference detector's photocurrent is then calibrated by turning off the microwave signal and injecting a constant amplitude-modulated optical signal whose modulation frequency is determined by the heterodyne interaction of two quasiplanarring, diode-pumped Nd:YAG lasers,[6] one of which is temperature tuned over a 22-GHz range.

These two highly stable single-line lasers produce a beat frequency with a linewidth less than 10 kHz, which is essential for accurate repeatable measurements. As shown in Fig. 12, the system is constructed with polarization-preserving fiber to avoid amplitude variations of the beat frequency caused by a change in the relative polarizations of the two laser signals. The output powers of the lasers are monitored during the calibration process, eliminating another potential error source.[7]

After the reference receiver is calibrated, it is used to calibrate lightwave signal analyzer systems. To calibrate a system, a gain-switched diode laser's output is measured with the reference receiver. The calibrated laser response is then used to calibrate the system under test.

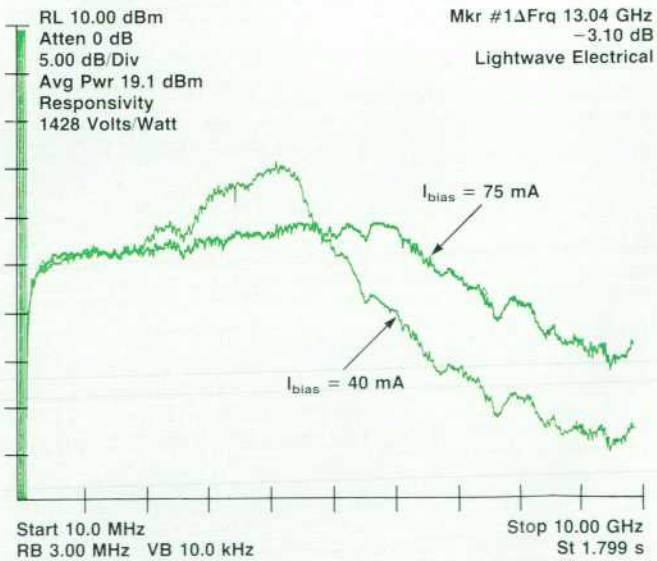


Fig. 13. *Modulation frequency response measurement of a high-speed laser.*

System Performance

The HP 71400A Lightwave Signal Analyzer offers advanced lightwave measurement performance. The combination of the broad-bandwidth pin photodetector, the high-gain, low-noise microwave preamplifier, and Hewlett-Packard's highest-performance spectrum analyzer offers excellent measurement sensitivity out to 22 GHz. The displayed average optical noise floor in a 10-Hz resolution bandwidth is typically better than −68 dBm from 10 MHz to 16 GHz, allowing optical signals below −60 dBm (1 nW) to be detected easily. With the built-in 30-dB optical attenuator, intensity modulation up to +15 dBm (31.6 mW) can be displayed.

Modulated power frequency response is flat within an excellent ±1.0 dB from 100 kHz to 22 GHz. This is a result of the optical heterodyne calibration technique and the method of calibrating the HP 71400A as a system. The system calibration corrects for the roll-off of the HP 70810A lightwave section and the frequency response of the spectrum analyzer. The mismatch loss and cable loss between the lightwave section and the spectrum analyzer are also corrected.

Measurements

The HP 71400A can make a number of measurements on lasers, optical modulators, and receivers.[8] Only a few can be described here.

A key parameter in any lightwave system is the modulation bandwidth of the optical source. Current-modulated semiconductor lasers today have bandwidths that are approaching 20 GHz. This bandwidth is achieved by optimization of the laser construction and selection of the appropriate current bias point. Fig. 13 shows a measurement of intensity modulation frequency response on a semiconductor laser designed particularly for high-frequency operation. As can be shown analytically,[9] the modulation bandwidth increases as a function of bias. In addition, the peaking in the response decreases, which is generally advantageous. If the current is increased beyond the critically damped response point, the bandwidth decreases.

This intensity modulation response measurement was made with the HP 71400A in conjunction with the HP 70300A tracking generator (20 Hz to 2.9 GHz) and the HP 70301A tracking generator (2.7 GHz to 18 GHz). Fig. 14 shows the setup. These tracking generators are also modules in the HP 70000 MMS family, and produce a modulation signal that is locked to the frequency to which the analyzer is tuned, thus making stimulus-response measurements easy and straightforward.

In most applications the laser noise spectrum is very important for a number of reasons. It obviously impacts

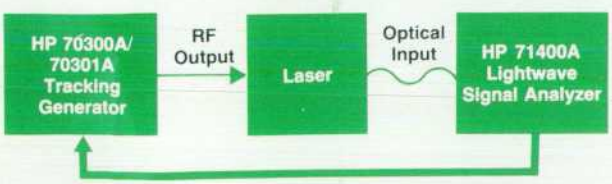


Fig. 14. *Block diagram of the instrumentation for high-speed laser modulation frequency response measurements.*

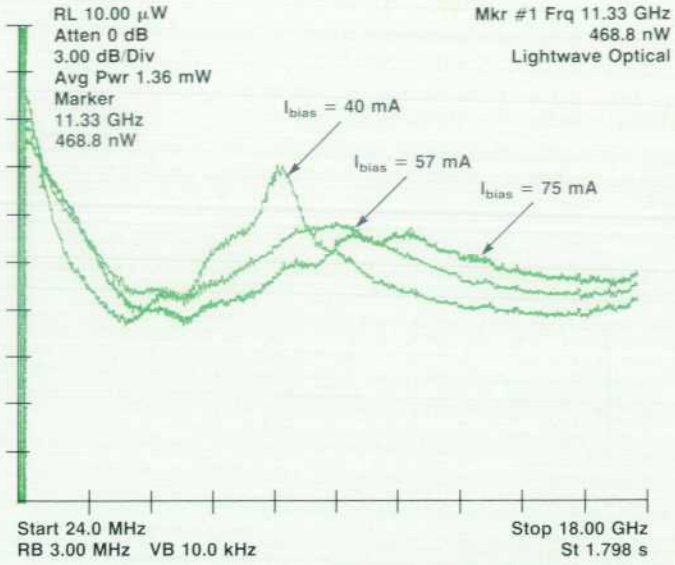


Fig. 15. *Intensity noise measurement of a high-speed laser showing the intensity noise peaking.*

the signal-to-noise ratio in a transmission system. Furthermore, it can be shown that the intensity noise spectrum has the same general shape as the intensity modulation response, and can be used as a indicator of potential modulation bandwidth.[9] The characteristic noise peak of the intensity noise spectrum also increases in frequency and decreases in amplitude as the bias current is increased. This is shown in Fig. 15.

The laser intensity noise spectrum can be greatly affected by both the magnitude and the polarization of the optical power that is fed back to the laser. This is called reflection-induced noise and is typically caused by reflections from optical connectors. This reflected power upsets the dynamic equilibrium of the lasing process and typically increases the amplitude of the intensity noise as shown in Fig. 16.

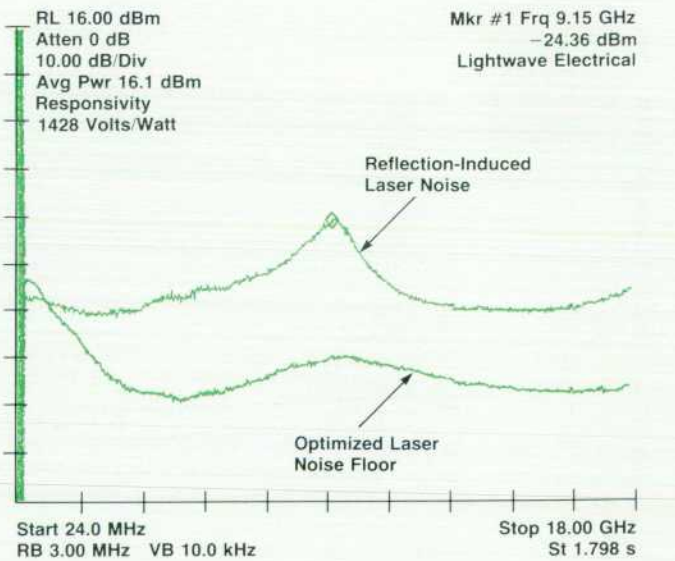


Fig. 16. *Effects of optical reflections on the laser intensity noise.*

It also can induce a ripple on the spectrum with a frequency that is inversely porportional to the round-trip time from the laser to the reflection. It should be noted that other instruments, such as an optical time-domain reflectometer, can measure the magnitude of a reflection, but the lightwave signal analyzer is the only instrument that can measure the effect of these reflections on the noise characteristic of the laser under test.

An important quantity related to signal-to-noise ratio is the relative intensity noise (RIN). It is a ratio of the optical noise power to the average optical power, and is an indication of the maximum possible signal-to-noise ratio in a lightwave system, where the dominant noise source is the laser intensity noise. In the lightwave-optical measurement mode, the HP 71400A makes the following measurement when the RIN marker is activated:

$$\text{RIN} = P_{noise}/P_{avg}$$

where P_{noise} is the optical noise power expressed in a 1-Hz bandwidth, and P_{avg} is the average optical power. This measurement can be made directly because of the built-in power meter function.

Before the development of the lightwave signal analyzer, customers used a photodiode and a microwave spectrum analyzer to make this noise measurement, and an ammeter to monitor the photocurrent. This has led to an alternate expression of RIN in electrical power units, since these were the units of the measurement equipment being used. The HP 71400A has the ability to express this RIN measurement in electrical power units in the lightwave-electrical measurement mode. Fig. 17 shows an RIN measurement in electrical power units of −143 dB at 4.65 GHz for this semiconductor laser. Notice that the noise floor of the HP 71400A is 10 dB lower than the laser noise floor in this measurement.

The HP 71400A can make a number of useful measurements involving large-signal digital modulation of lasers.

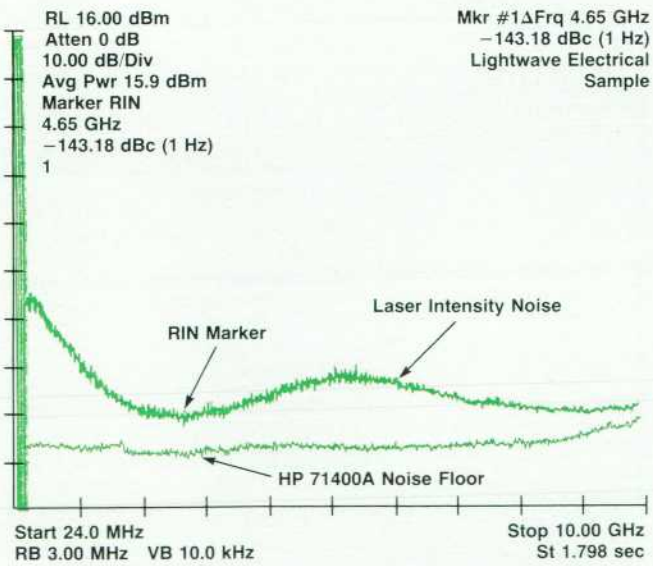


Fig. 17. *Relative intensity noise (RIN) measurement of a high-speed laser.*

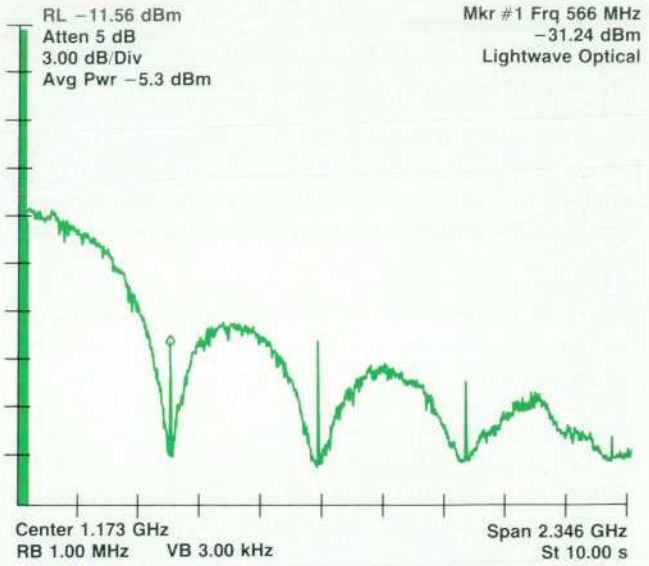


Fig. 18. *Broadband sweep of a laser modulated with a 565-megabit-per-second PRBS data pattern.*

Fig. 18 shows a laser transmitting pseudorandom binary sequence (PRBS) intensity modulation at 565 megabits per second. This sequence is a widely used test signal usually observed as an eye diagram in the time domain. In the frequency domain, an envelope that is the Fourier transform of the pulse shape is displayed. Nonideal characteristics, such as clock feedthough, are evident. As shown in Fig. 19, a narrower frequency sweep reveals that the signal is divided into discrete frequencies whose spacing is equal to the clock rate divided by the sequence length. Noise is also visible. In fact, different signal-to-noise ratios are observable as the feedback to the laser is adjusted. It is likely that this is the only way to measure transmitter-related

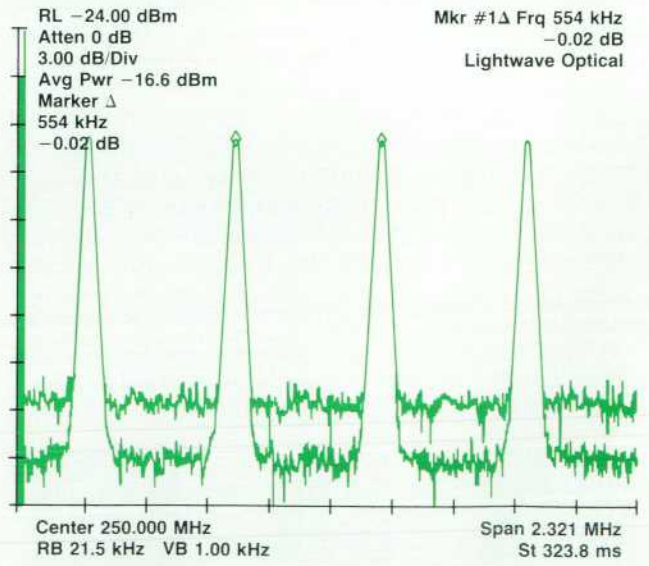


Fig. 19. *Narrowband sweep of a laser modulated with a PRBS data pattern, showing the individual frequency components and the effect of the polarization of the reflected light on the signal-to-noise.*

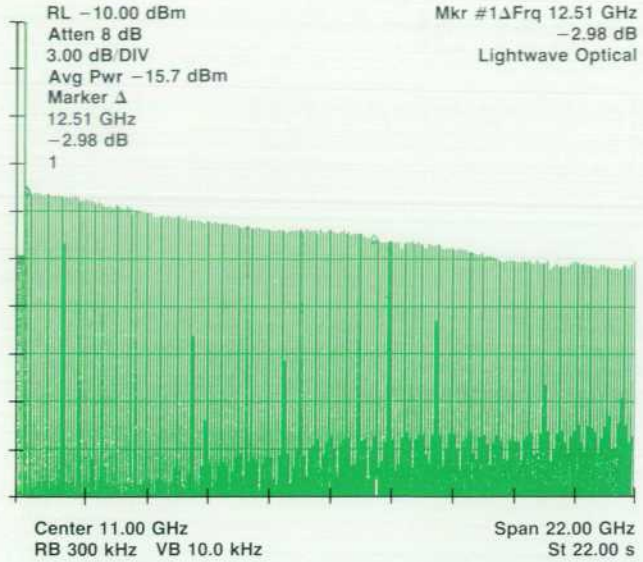


Fig. 20. *Modulation spectrum of a pulsed laser.*

noise problems under large-signal modulation. In principle, it is possible to estimate bit error rate from this signal-to-noise ratio.

High-speed pulse modulation can also be displayed on the HP 71400A. Fig. 20 shows the frequency-domain spectrum of a laser being driven at 100 MHz and generating 35-picosecond-wide pulses. The spacing between the individual discrete frequencies is equal to the pulse repetion rate. Once again, the envelope is the Fourier transform of the pulse shape. The pulse width can be determined from the 3-dB bandwidth, here 12.5 GHz, by assuming the pulse shape is Gaussian and using the following relationship:

$$\text{Pulse Width} = 0.44/\text{Optical 3-dB Bandwidth.}$$

This technique may be just as accurate as measuring the pulse width on a sampling oscilloscope, where the rise time of the scope must be deconvolved to get the correct answer.

Acknowledgments

The development of the HP 71400A Lightwave Signal Analyzer depended on contributions of a number of individuals, spread over a number of Hewlett-Packard divisions. This project had the potential to be a project manager's nightmare with all the personnel interdependency and new technology development, along with an aggressive introduction schedule. It was the dedication and perseverance of the following people that made this product happen. The project team and their responsibilities were: Dennis Derickson, microcircuit design, Roberto Collins, digital design, Jimmie Yarnell, mechanical design, Dave Bailey and Zoltan Azary, firmware design. The instrument shell design team completed their portion of the firmware under significant time constraints. This product was based on a new technology for Signal Analysis Division, and the effort of the NPI team to get this product into production was commendable. The major contributions of this instrument were dependent on the high-speed pin photodiode, distributed MMIC amplifier, and TaO_5 thin-film integrated capacitor technology, all developed by the engineers at Microwave Technology Division. Portions of the design were leveraged from existing products with the help of the engineers at Böblingen Instruments Division. I would like to thank Jack Dupre and the rest of the management team at Signal Analysis Division who supported the development of this product. Finally, I would especially like to acknowledge the efforts of Rory Van Tuyl, who started our lightwave program at Signal Analysis Division and whose vision this product reflects.

References

1. W. Radermacher, "A High-Precision Optical Connector for Optical Test and Instrumentation," *Hewlett-Packard Journal*, Vol. 38, no. 2, February 1987, pp. 28-30.
2. H. Schweikardt, "An Accurate Two-Channel Optical Average Power Meter," *Hewlett-Packard Journal*, Vol. 38, no. 2, February 1987, pp. 8-11.
3. B. Maisenbacher, S. Schmidt, and M. Schlicker, "Design Approach for a Programmable Optical Attenuator," *Hewlett-Packard Journal*, Vol. 38, no. 2, February 1987, pp. 31-35.
4. *Product Note 70000-1, HP 70000 System Design Overview*, Hewlett-Packard Publication No. 5954-9135.
5. *pSOS-68K User's Manual*, The Software Components Group Inc., Doc. No. PK68K-MAN.
6. W. R. Trutna Jr., D. K. Donald, and M. Nazarathy, "Unidirectional Diode Laser-Pumped Nd:YAG Ring Laser," *Optical Letters*, Vol. 12, 1987, pg. 248.
7. T. S. Tan, R. L. Jungerman, and S. S. Elliot, "Calibration of Optical Receivers and Modulators Using an Optical Heterodyne Technique," *IEEE-MTT-S International Microwave Symposium Digest*, OO-2, 1988, pp. 1067-1070.
8. *Application Note 371, Lightwave Measurements with the HP 71400 Lightwave Signal Analyzer*, Hewlett-Packard Publication No. 5954-9137.
9. C. Miller, D. Baney, and J. Dupre, "Measurements on Lasers for High-Capacity Communication Systems," *Hewlett-Packard RF, Microwave, and Lightwave Measurement Symposium*, 1989.

Linewidth and Power Spectral Measurements of Single-Frequency Lasers

A special fiber optic interferometer preprocesses optical signals for a lightwave signal analyzer to measure laser characteristics using delayed and gated delayed self-homodyne techniques.

by Douglas M. Baney and Wayne V. Sorin

WITH THE ADVENT OF SEMICONDUCTOR lasers and low-loss optical fibers, the possibility of achieving over 1000-Gbit·km/s bandwidth-distance products has propelled research towards improving the performance of the laser and the optical fiber transmission medium.[1] To minimize transmission penalties resulting from dispersion in long optical fiber communication links, high-performance lightwave communication systems require lasers that operate in a single longitudinal mode (i.e., single-frequency oscillation) and have minimal dynamic linewidth broadening (i.e., frequency chirp) under modulation. In coherent communications, the lasing linewidth becomes an important determinant of system performance. In the development of FSK modulated systems, which often rely on modulating the injection current to a semiconductor laser, the FM deviation as a function of both injection current and modulation frequency must be characterized.

Advances in laser technology necessary to meet the stringent requirements of communications system design have required similiar advances in measurement techniques and technology. The HP 11980A Fiber Optic Interferometer was developed to work as an accessory to the HP 71400A Lightwave Signal Analyzer (see article, page 80) to enable users to characterize many important spectral modulation properties of single-frequency telecommunication lasers.

Interferometer Design

The function of the HP 11980A is to act as a frequency discriminator, converting optical phase or frequency deviations into intensity variations, which can then be detected using a square-law photodetector (e.g., the high-speed photodiode of the HP 71400A). Inside the HP 11980A is an unbalanced fiber optic Mach-Zehnder interferometer (see Fig. 1). This type of interferometer has an input directional coupler, which splits the incoming optical signal into two equal parts. The two signals then travel along separate fiber paths where they experience a differential delay, τ_o. The two signals are then recombined using another directional coupler. Since the optical fiber does not preserve the polarization state, a polarization state controller is added to one arm of the interferometer. The controller is purely mechanical and consists simply of a loop of fiber that can be rotated. This adjustment allows the user to maximize the interference signal by ensuring similar polarization states at the combining directional coupler. The optical output can then be sent to the HP 71400A where intensity variations are converted to a time-varying photocurrent, which is displayed on a spectrum analyzer.

The HP 11980A interferometer is completely passive and has the same adaptable fiber optic connectors as the HP 71400A. The connectors are compatible with the HMS-10/HP, FC/PC, ST, biconic, and DIN connector formats. Fused single-mode fiber directional couplers from Gould, Inc. were chosen for their broad wavelength range from 1250 to 1600 nanometers, enabling coverage of the important 1300-nm and 1550-nm telecommunication windows. One arm of the interferometer is spliced to a 730-meter reel of Corning single-mode optical fiber to provide a differential delay of 3.5 microseconds. This delay permits laser

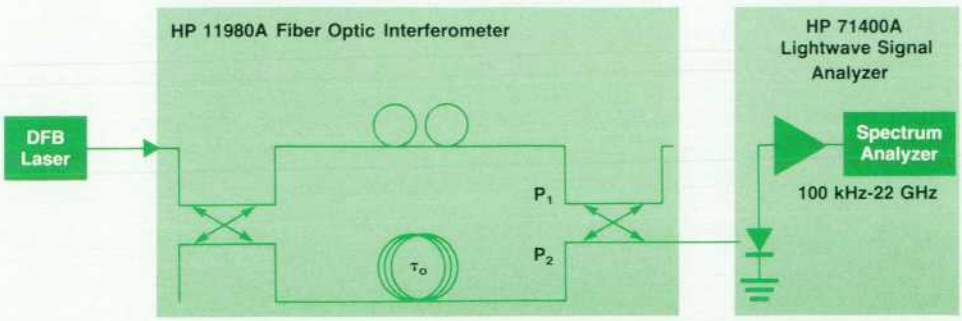


Fig. 1. *Distributed feedback (DFB) laser linewidth measurement using the delayed self-homodyne technique.*

linewidth measurements as low as 225 kHz (Lorentzian line shapes).

Laser Diode Linewidth

The most basic type of semiconductor laser uses reflections from cleaved end facets to provide the feedback needed for laser operation. One disadvantage of this Fabry-Perot type laser is that it generally operates in several frequency modes, each separated by about 100 GHz. This can produce effective laser linewidths greater than 500 GHz, which can limit data rates (because of dispersion) in long-haul fiber optic communication links. One possible solution for reducing the effects of dispersion is the development of DFB (distributed feedback) and DBR (distributed Bragg reflector) semiconductor lasers. In these lasers, a wavelength filter (a diffraction grating) suppresses all but one of the frequency modes of the laser. The resulting linewidths for these lasers are typically less than 50 MHz. Considering that the laser itself oscillates at a frequency of about 200,000 GHz, this is a relatively small fractional linewidth.

DFB and DBR lasers have a tendency to change their operating frequency for different levels of injection current. This causes the laser to frequency chirp while being amplitude modulated, which can also result in limited data rates because of dispersion. The magnitude of these frequency chirps can be in the tens of gigahertz. Measurements of linewidth and frequency chirp yield important information not only about the laser's performance in a lightwave link, but also about the physical characteristics of the laser itself.

Measuring Linewidth

The HP 11980A enables measurement of laser linewidth, $\Delta\nu$, by preprocessing the optical signal for the HP 71400A Lightwave Signal Analyzer. The block diagram of the measurement system is shown in Fig. 1. The single-frequency laser, typically a DFB or DBR laser, is coupled to an optical fiber. Isolators are often used to reduce perturbations of the laser by optical feedback arising from optical scattering in the fiber or at optical interfaces. The signal to be analyzed is then fed into the unbalanced Mach-Zehnder fiber optic interferometer inside the HP 11980A. Inside the interferometer the laser signal is split into two signals, which experience different delays before being recombined and sent to the photodiode of the HP 71400A. If the differential delay τ_o is larger than the coherence time τ_c of the laser,

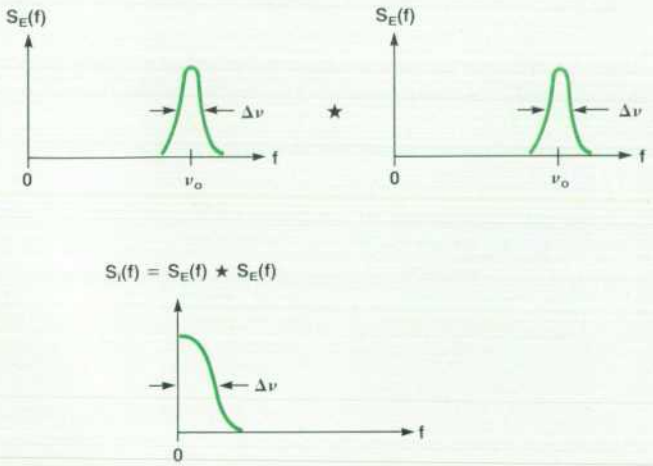


Fig. 2. *After detection in the HP 71400A Lightwave Signal Analyzer, the spectrum of the signal from the HP 11980A Fiber Optic Interferometer is the autocorrelation function of the laser's electric field spectrum $S_E(f)$. The ★ indicates correlation.*

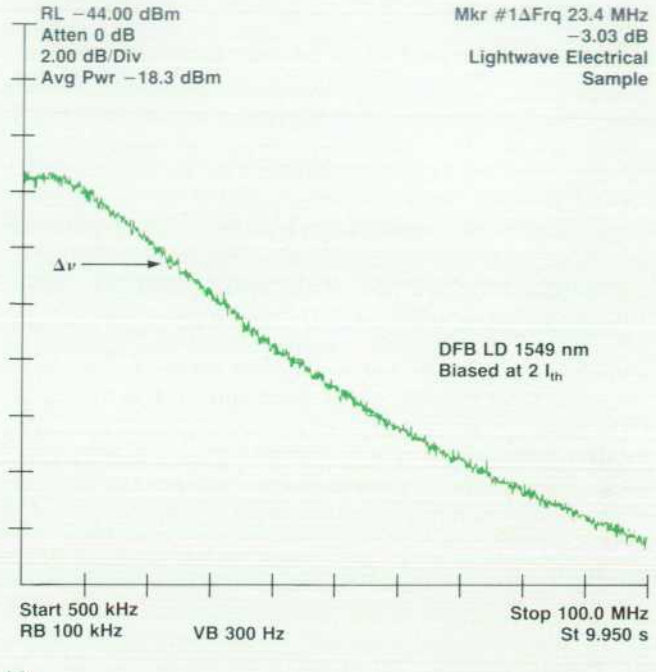


Fig. 3. *(a) A linewidth measurement for a DFB laser operating at 1549 nm. The linewidth is approximately 25 MHz. (b) Two-sided measurement of the same laser linewidth.*

the two combined signals become uncorrelated. This process is equivalent to mixing two separate laser signals, both having the same linewidth and center frequency. The mixing (i.e., multiplying) of these two signals is accomplished as a result of the square-law nature of the photodiode. The resulting photocurrent spectrum is the autocorrelation function of the laser's electric field spectrum $S_E(f)$ and is commonly referred to as the delayed self-homodyne linewidth measurement. This process is shown graphically in Fig. 2. Since the displayed spectrum is the autocorrelation function of the laser's line shape, its spectral width is approximately twice that of the laser linewidth.[2] For the special case of Lorentzian line shapes, the autocorrelation function is also Lorentzian and has a linewidth exactly twice that of the original line shape. For Gaussian line shapes, the autocorrelation function is also Gaussian but has a linewidth equal to $\sqrt{2}$ times that of the original line shape. Currently, most single-frequency semiconductor lasers are accurately described by Lorentzian line shapes.

For the delayed self-homodyne measurement to be valid, the combining signals from the two arms of the interferometer must be uncorrelated. For the HP 11980A, this means that the coherence time of the laser should be less than the interferometer delay of 3.5 microseconds. Since the coherence time is approximately equal to the inverse of the linewidth (i.e., $\tau_c \approx 1/\Delta\nu$), the HP 11980A can measure linewidths less than 300 kHz.

The signal-to-noise ratio of the displayed photocurrent spectrum can often be improved by manual adjustment of the front-panel knob on the HP 11980A. This polarization state adjustment can increase the interference between the two mixing signals by ensuring that their polarization states are closely matched. The shape of the displayed spectrum is not altered by this adjustment, only its size relative to the noise floor. It was decided not to automate this adjustment because of the additional complexity that would be required.

Fig. 3a shows a linewidth measurement for a DFB laser operating at 1549 nanometers. The linewidth, $\Delta\nu$, is found by placing the display delta marker at the −3-dB point from the peak. The half width is measured, since the autocorrelation process doubles the width of the laser's spectrum. For the conditions of Fig. 3a, the laser linewidth is measured to be approximately 25 MHz.

It is also possible to display a two-sided line shape by by applying a small amount of amplitude modulation to the laser and observing the linewidth convolved about one of the modulation sidebands.[3] This result is shown in Fig. 3b where the full double-width Lorentzian line shape is displayed. The linewidth is again measured to be about 25 MHz, which agrees with that obtained in Fig. 3a.

Modulated Laser Power Spectrum Measurement

Using a newly developed measurement technique,[4] the HP 11980A Fiber Optic Interferometer can used to measure laser chirp as well as intentional frequency modulation. Chirp can be thought of as the unwanted frequency deviation in the optical carrier of a modulated laser. There exist a variety of techniques to measure the modulated power spectrum of a single-frequency laser. These include grating and Fabry-Perot spectrometers and heterodyne down-conversion using two lasers. The technique presented here was developed in response to the shortcomings of previously known techniques. For example, it offers superior frequency resolution than grating spectrometers, which in practice are limited to a resolution of about 1 angstrom (approximately 15 GHz). Higher resolution (i.e., finesse) can be achieved with Fabry-Perot spectrometers, but the wavelength range is limited for a fixed pair of mirrors. In heterodyne techniques, two lasers are required and their wavelengths must be precisely controlled, which often requires a high degree of complexity. The technique presented in this section overcomes these problems, allowing homodyne frequency measurements to be made over a range of 300 kHz to 22 GHz.

Laser chirp in semiconductor lasers is caused by the dependence of the real and imaginary parts of the index of refraction on the injection current. Because of this effect, modulation of the injection current can result in large fluctuations of the lasing wavelength. This phenomenon is responsible for a substantial widening of the electric field modulation power spectrum, $S_m(f)$, beyond the Fourier transform limit of the information bandwidth. A wide power spectrum can impose severe transmission penalties in lightwave links with nonnegligible wavelength dispersion. Using the new gated delayed self-homodyne technique,[4] a homodyne measurement of $S_m(f)$ can be performed using the HP 11980A in conjunction with the HP 71400A.

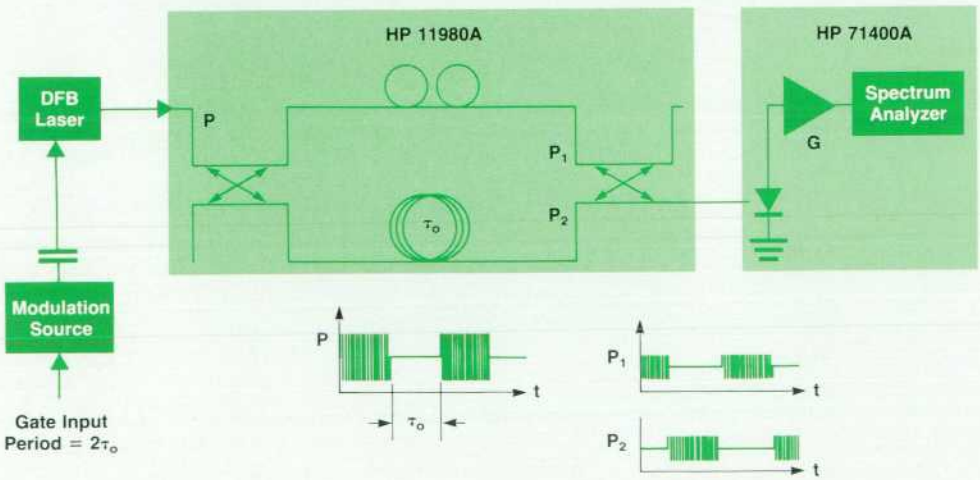


Fig. 4. *Gated delayed self-homodyne technique for measuring laser frequency chirp and FM deviation.*

Fig. 4 shows the measurement setup. With the laser biased above threshold, the injection current is gated between two states, one state modulated and the other state unmodulated. Thus, the laser behaves as a modulated laser for a period τ_o and an unmodulated laser, or local oscillator signal, for a sequential period τ_o. The period τ_o is chosen to equal the differential delay in the arms of the fiber optic interferometer, which is assumed to be longer than the coherence time of the laser. In the HP 11980A, there is a continuous combination of a modulated state with an unmodulated state. These states are then mixed in the photodetector of the HP 71400A. The power spectrum of the detector photocurrent, $S_i(f)$, is displayed by the HP 71400A. The homodyne down-conversion of the optical spectrum is illustrated in Fig. 5. In this figure, the modulated spectrum is shown to be asymmetrically located around the average frequency ν_o. This demonstrates the folding about zero frequency which is characteristic of homodyne mixing.

This spectrum, $S_i(f)$, for the case where $\tau_o > \tau_c$, can be approximated as:[5]

$$S_i(f) = S_D(f) + \left\{ \frac{\Delta\nu/\pi}{(\Delta\nu)^2 + f^2} \right\} \bigstar \{S_m(f) + S_m(-f)\}$$

where $S_D(f)$ is the direct intensity modulation that would be measured if the interferometer were not present, and the other terms describe the Lorentzian line shape of the laser crosscorrelated with the homodyne power spectrum of the lasers's electric field modulation. The ability to make this measurement while the laser is modulated allows the determination of the alpha factor,[6] which characterizes the coupling between gain and frequency chirp in semiconductor lasers.

Figs. 6a and 6b demonstrate some of the experimental results that can be obtained using this gated delayed self-homodyne technique. In Fig. 6a, the injection current to a DFB laser is sinusoidally modulated at a rate of 300 MHz. Besides introducing a small amount of intensity modulation, the optical frequency is also modulated. The modulation of the optical carrier results in an electric field spectrum whose peaks are spaced by 300 MHz and whose amplitudes are described in terms of Bessel functions as predicted by classical FM theory. By adjusting the injection current to null a specific Bessel sideband, the frequency modulation index β can be determined very accurately. This technique is useful for accurately determining the optical FM response at various modulation frequencies.

In Fig. 6b, the modulation frequency was reduced to 45 MHz, which results in a larger FM modulation index for

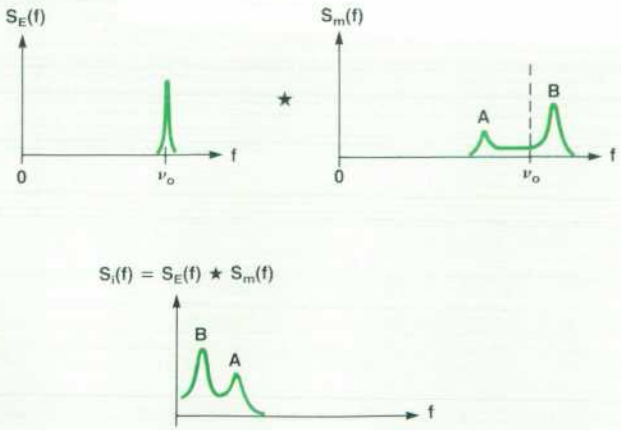


Fig. 5. *For the gated delayed self-homodyne technique, the power spectrum of the detector photocurrent is the crosscorrelation function of the laser electric field spectrum $S_E(f)$ and the modulated electric field spectrum $S_m(f)$. Homodyne detection results in the folding of the upper and lower sidebands, as illustrated in the displayed spectrum.*

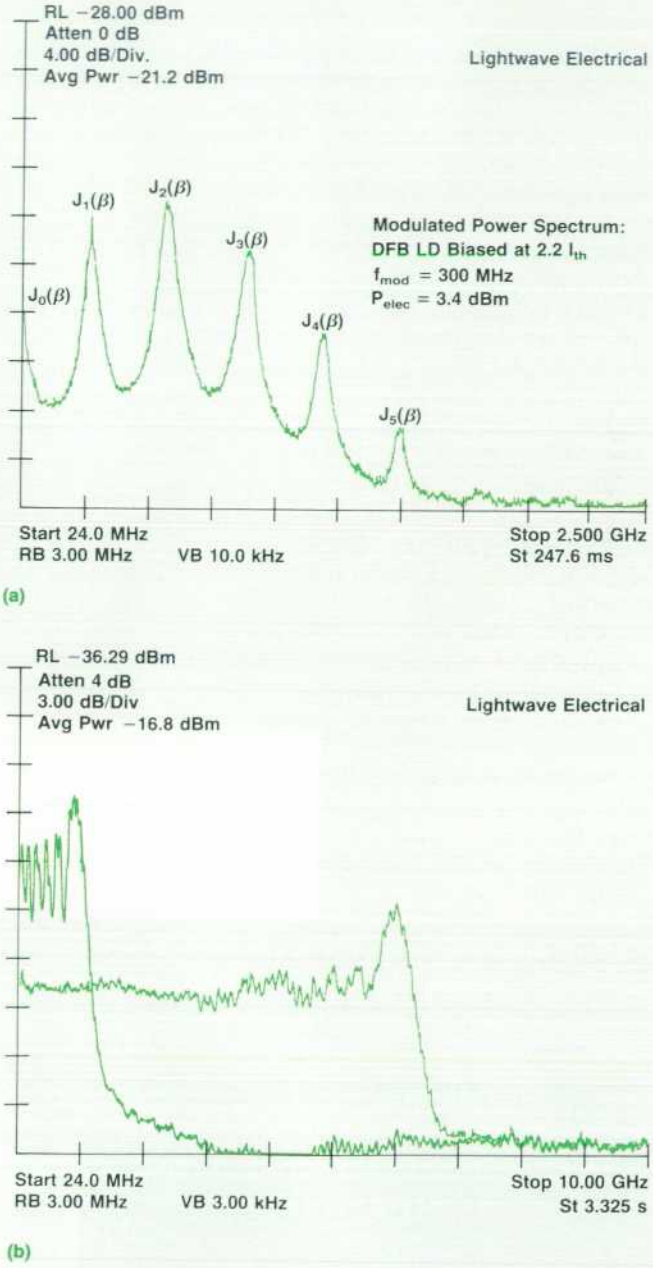


Fig. 6. *(a) Gated delayed self-homodyne power spectrum for a DFB laser with injection current sinusoidally modulated at 300 MHz. (b) Same measurement with the modulation frequency reduced to 45 MHz, resulting in a larger modulation index. The two curves represent different sinusoidal drive currents to the laser, resulting in different degrees of frequency chirping.*

the laser. The individual sidebands are no longer resolved because of the finite linewidth of the laser, and the spectrum takes on the shape of the probability density function for wideband sinusoidal FM modulation. The two curves in Fig. 6b indicate the progression of laser chirp with increasing modulation power. The difference between these two curves corresponds to a ratio of optical frequency chirp to injection current of 410 MHz/mA at a modulation frequency of 45 MHz.

The resolution of the technique is approximately equal to the laser linewidth and therefore can be significantly superior to that of the Fabry-Perot spectrometer while being able to operate over a wavelength range of approximately 1250 to 1600 nm. Compared to heterodyne techniques employing two lasers, this technique has the advantage of wavelength autotracking between the local oscillator and the modulated laser, since the same laser is used to generate both signals.

Summary

The HP 11980A Fiber Optic Interferometer was developed to enhance the measurement capabilities of the HP 71400A Lightwave Signal Analyzer. The fiber interferometer provides the ability to compare an optical signal with a 3.5-microsecond delayed version of itself. Using this type of comparision, information can be obtained about deviations in the optical carrier frequency. This enhancement allows the HP 71400A to measure laser linewidths as low as 225 kHz and frequency chirp (up to ±22 GHz) over a wavelength range of 1250 to 1600 nm.

Acknowledgments

Since the conception of the HP 11980A was based on a new measurement technique, it required allocation of previously unscheduled human and material resources. Many people put in the extra effort needed to make it a reality. Some key people and their responsibilities are as follows. Rory Van Tuyl provided indispensable management and technical support. Scott Conrad and Ron Koo provided manufacturing and production engineering support. Dean Carter performed mechanical design. At Hewlett-Packard Laboratories, Moshe Nazarathy and Steve Newton provided important technical support.

References

1. N.A. Olsson, G.P. Agrawal, and K.W. Wecht, "16 Gbit/s, 70 km pulse transmission by simultaneous dispersion and loss compensation with 1.5 μm optical amplifiers," *Electronics Letters*, Vol. 25, April 1989, pp. 603-605.
2. M. Nazarathy, W.V. Sorin, D.M. Baney, and S.A. Newton, "Spectral analysis of optical mixing measurements," *Journal of Lightwave Technology*, Vol. LT-7, 1989, pp. 1083-1096.
3. R.D. Esman and L. Goldberg, "Simple measurement of laser diode spectral linewidth using modulation sidebands," *Electronics Letters*, Vol. 24, October 1988, pp. 1393-1395.
4. D.M. Baney and W.V. Sorin, "Measurement of a modulated DFB laser spectrum using gated delayed self-homodyne technique," *Electronics Letters*, Vol. 24, May 1988, pp. 669-670.
5. D.M. Baney and P.B. Gallion, "Power spectrum measurement of a modulated semiconductor laser using an interferometric self-homodyne technique: influence of quantum phase noise and field correlation," *IEEE Journal of Quantum Electronics*, Vol. 25, October 1989, pp. 2106-2112.
6. C.H. Henry, "Phase noise in semiconductor lasers," *Journal of Lightwave Technology*, Vol. LT-4, 1986, pp. 298-311, 1986.

Hewlett-Packard Company, 3200 Hillview Avenue, Palo Alto, California 94304

ADDRESS CORRECTION REQUESTED

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD JOURNAL
February 1990 Volume 41 • Number 1

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Marcom Operations Europe
P.O. Box 529
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8579