SLAC – 297
UC – 32
(M)

THE VM VERSION OF INTERLAN'S NS4240 XEROX ITP NETWORK SOFTWARE*

Hans Frese
R. Leslie Cottrell
Teresa Downey

Stanford Linear Accelerator Center
Stanford University
Stanford, California 94305

April 1986

Prepared for the Department of Energy
under contract number DE-AC03-76SF00515

* Manual

Table of Contents

ACKNOWLEDGMENTS

This document is adapted from INTERLAN's NS4240 ITP/UNIX Network Software ,[4] Documentation Part Number 950-1015-AA. Permission to use large portions of the INTERLAN document was kindly given by INTERLAN.

1. PREFACE

This manual describes SLAC's VM adaptation of INTERLAN's NS4240 ITP Network Software. The ITP Network Software, hereafter called ITP/VM, is an implementation of the Xerox Network Systems (Xerox NS) Internet Transport Protocols (ITP). ITP/VM runs under the VM/SP operating system.

This manual explains how to write application programs that interface to ITP/VM. This manual also contains information about computer networks that use Xerox ITP-based protocols.

1.1 INTENDED AUDIENCE

This manual assumes you are familiar with the use of the VM operating system. It assumes you have experience assembling, linking, and running application programs on your system. This manual also assumes you have a basic familiarity with the concepts of computer networking, and that you understand the more specific concepts of Ethernet-based networks.

1.2 SUGGESTED READING

Before using ITP/VM, you should read the Xerox ITP Manual (Xerox publication "Internet Transport Protocols" XSIS 028112, December 1981). It describes in detail the protocols that ITP/VM implements and will be referred to as.[6]

There are two additional Xerox documents that describe other parts of the Xerox Network Systems (NS) architecture. These are not required reading to use ITP/VM, but they are useful if you are connecting to Xerox equipment. They are: "Level 0 Point-to-Point Protocol/Product Specification T33-2.0" XSIS 018201, January 1982[7] and "Courier: The Remote Procedure Call Protocol", XSIS 038112, December 1981.[8] These manuals are available from Xerox.

These three IBM manuals provide most of the operating system details: "VM/SP System Programmer's Guide",[1] "VM/SP CMS Command and Macro Reference",[2] and "VM/SP CMS User's Guide".[3]

2. INTRODUCTION TO ITP/VM

ITP/VM allows a virtual machine to communicate with other ITP-compatible hosts in a Xerox NS internet. You use ITP/VM with a DACU containing one INTERLAN NI1010A UNIBUS Ethernet controller board.

ITP/VM implements layers 0 through 2 of the Xerox Network Systems Internet Transport Protocols. These levels correspond to layers 1 through 4 of the International Standards Organization (ISO) Open Systems Interconnection (OSI) Model for Network Communications.

ITP/VM provides a program interface so you can write application programs that access ITP protocols such as Packet Exchange, Echo, and Sequenced Packet Protocols.

ITP/VM provides sample programs that demonstrate the use of these protocols.

2.1 FEATURES OF ITP/VM

The following sections give a general overview of the operation and interfaces of ITP/VM.

2.1.1 STRUCTURE OF ITP/VM

ITP/VM consists of two main pieces. These are:

1. a program called ITPACP running in the Disconnected Service Virtual Machine (DSVM) ITPACP1 and
2. a runtime library (SNITPUSR TXTLIB on the SLACNET disk), which contains subroutines that interface to ITPACP. These subroutines are called by your application programs running in a virtual machine other than ITPACP1.

Your application programs can be written in higher level languages that support Waterloo-C subroutines.

The ITP interface runtime library contains subroutines which your program will call to invoke the various services provided by ITP.

The ITPACP program contains all the Xerox NS ITP Level 1 and Level 2 protocols as well as part of Level 0.

2.2 XEROX ITP FUNCTIONS

This section briefly describes ITP/VM's support of the Xerox Internet Transport Protocols (ITP). Chapter 3 provides more information on ITP. For complete information on ITP, please consult the Xerox documentation mentioned in the Preface.

2.2.1 Protocols

Specifically, ITP/VM provides interfaces to the following ITP protocols:

1. Internet Datagram Protocol (IDP) and, indirectly, the Error Protocol (ERR) and Routing Information Protocol (RIP),
2. Packet Exchange Protocol (PEP),
3. Echo Protocol (ECH), and
4. Sequenced Packet Protocol (SPP), including Reliable Packet Mode, Packet and Byte Stream.

ITP/VM does not implement the following protocols:

1. Courier
2. Clearinghouse

These protocols are in the level above PEP, RIP, ERR, ECH, and SPP. ITP/VM does not implement any other application-specific protocols.

ITP/VM does provide easy-to-use interfaces on which you can implement higher-level protocols.

2.2.2 Routing

ITP/VM contains all router functionality. ITP/VM can route packets from

1. sockets to networks (for transmission),
2. networks to sockets (for reception),
3. sockets to sockets, and
4. (optionally) from networks to networks as an internet router.

2.3 NETWORK MANAGEMENT

ITP/VM maintains various statistics about the state of the internet and about sockets and connections. These statistics are available (both locally and remotely) to your programs.

ITP/VM includes a network monitoring utility, NETMON. NETMON is a menu-driven, screen-oriented program for 327x type terminals that allows you to inspect local and remote network statistics and configuraton information. For more information see Appendix D.

ITP/VM implements an integral Echo Request Server which listens at a well-known socket for Echo Protocol requests.

ITP/VM implements a Statistics Request Server which listens at a well-known socket for Remote Report Statistics requests. (Note that the Statistics Request Server is an INTERLAN extension to the Xerox ITP specification).

2.4 USER INTERFACE

You can write application programs that use the ITP/VM software. Chapter 4 describes how your programs use a set of runtime subroutines that in turn execute itpiucv() calls to communicate via the Inter-User Communications Vehicle (IUCV)[1:110–176] and[9] with the ITPACP1 virtual machine to access the various ITP/VM protocols.

The ITP/VM user interface was adapted from the ITP/UNIX user interface[4] for SLACNET needs. At the same time, an ITP/VMS interface was created with identical calling conventions to ITP/VM.

2.5 PRODUCT REQUIREMENTS

ITP/VM requires one INTERLAN Ethernet controller NI1010A in a UNIBUS slot of the DACU connected via a channel interface to a 30xx/43xx.

3. XEROX NS NETWORKS

This chapter describes some of the basic concepts of the Xerox ITP Specification. We provide this information for your convenience in configuring your own networks and in developing application programs that use the ITP/VM user interface.

For more detailed information on ITP, please consult the original Xerox ITP documentation mentioned in the Preface.

3.1 PHYSICAL CONFIGURATIONS

Although you may be using only one network transmission medium (an Ethernet), the Xerox protocols are general enough to support the connection of (incompatible) transmission media into networks of networks or "internets". This means, for example, that you could connect your Ethernet network to another Ethernet network via a phone line. At the junction of two (or more) networks, sits a store-and-forward device called an "internet router".

The following diagram illustrates your VM system participating in local and internet configurations.

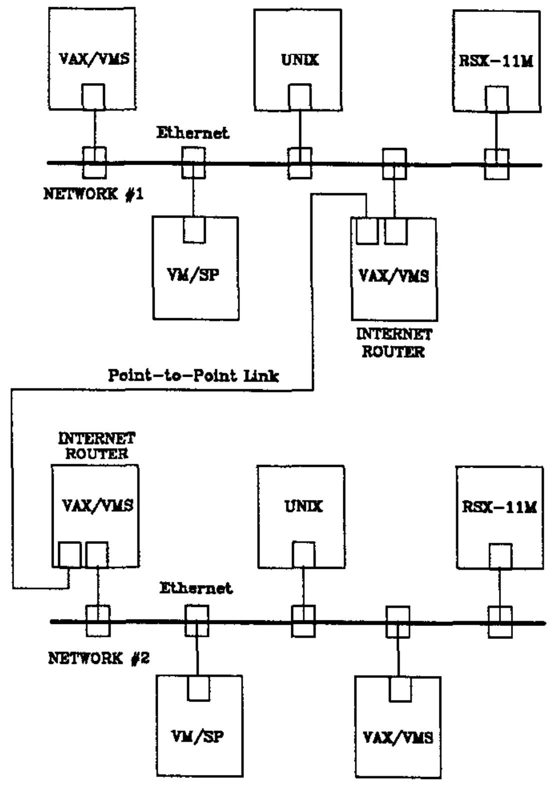

Figure 1

3.2 ITP PROTOCOL LEVELS

The Xerox ITP protocols break down the communication of data in an internet environment into clearly-defined sub-functions. These sub-functions form layers, one on top of another, with each sub-function using the services of the layer beneath it and providing services to the layer above it. These layers are called Level 0, Level 1, and Level 2. All hosts on an ITP-compatible network that engage in end-to-end communications implement all three Levels. The INTERLAN ITP/VM product provides a user interface to Level 1 and Level 2.

The next figure shows the relationship of the three levels.

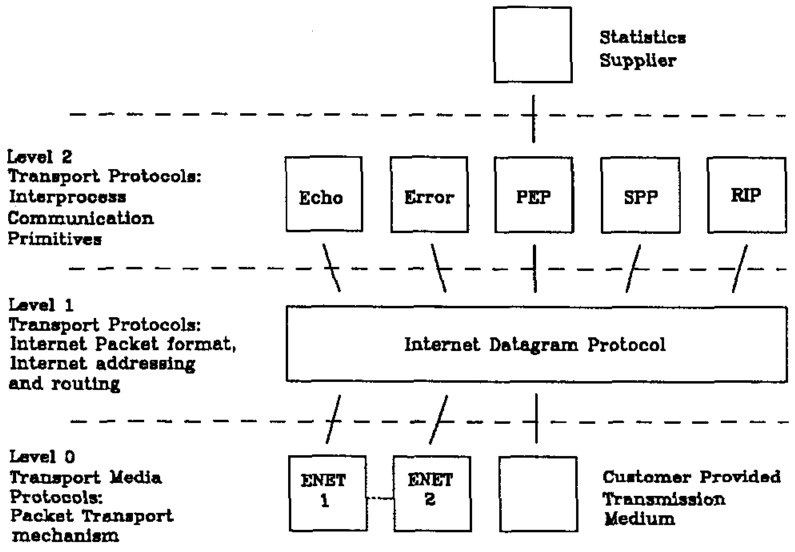
Statistics Supplier
Level 2 Transport Protocols: Interprocess Communication Primitives
Echo
Error
PEP
SPP
RIP
Level 1 Transport Protocols: Internet Packet format, Internet addressing and routing
Internet Datagram Protocol
Level 0 Transport Media Protocols: Packet Transport mechanism
ENET 1
ENET 2
Customer Provided Transmission Medium

Figure 2

3.2.1 ITP Level 0

Level 0 is also called the Transmission Media Protocol. In ITP/VM there is at least one network driver for the network controller connected to your VM host. The functionality of the Level 0 Ethernet network drivers in ITP/VM is shared among the Ethernet controller hardware, the driver software for this controller, NI, and the ITPACP program that uses this driver software.

The network driver has two main functions:

1. Accept data from Level 1 and send it to a specified destination on the transmission medium.
2. Accept data from the transmission medium, decapsulate any medium-specific information and pass the data up to Level 1.

Note that a network driver does not enhance the reliability of the network transmission medium. If data is lost due to errors, such as CRC failures or buffer limitations at the receiver, the corresponding network driver takes no special action. In fact, the driver may not even know of the demise of the data. Note also that a network driver makes no assumptions about how sequences of packets may be ordered or related. Level 0 is said to provide a "mostly reliable" datagram service between hosts on a common transmission medium.

Every network is a broadcast, multicast, or point-to-point network.

1. A broadcast network can deliver a packet to all hosts on the network.
2. A multicast network can deliver a packet to a subset of all hosts on the network.
3. A point-to-point network can deliver packets only from one host to another.

Ethernet is both a broadcast and multicast network.

3.2.2 ITP Level 1

Level 1 is also called the Internet Datagram Protocol (IDP).

Level 1 implements a packet addressing scheme that allows many unique local addresses inside any one host. These local addresses, called sockets, are the sources and destinations of all internet datagrams. In this way, Level 1 can multiplex and de-multiplex packet streams.

Level 1 implements a packet switch called an internet router. The router enhances the functionality of Level 0 by providing an internet delivery system between networks. Level 1 routers try to share a common description of the internet configuration (which may change over time) by exchanging topological information with each other. The Level 1 functionality in ITP/VM is in the ITPACP program running in the Disconnected Service Virtual Machine (DSVM) ITPACP1.

Level 1:

1. Routes outbound packets from local sockets to local sockets.
2. Routes outbound packets from local sockets to Level 0 network drivers.
3. Routes inbound packets from Level 0 network drivers to local sockets.
4. On internet routers only, routes inbound packets from one Level 0 network driver to another Level 0 network driver.

Note that, like Level 0, Level 1 does not enhance the capabilities of the underlying transmission media nor does it relate sequences of packets. Level 1 provides a "mostly reliable" internet datagram service between sockets. Figure 3 shows the delivery of a packet in an Ethernet environment and in an internet environment.

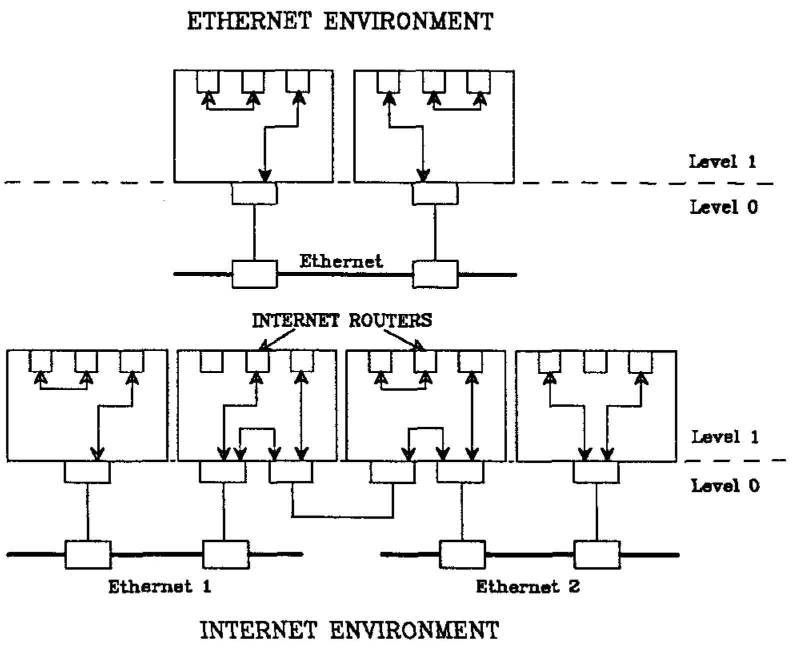

Figure 3

3.2.3 ITP Level 2

The highest Level of ITP, Level 2, is responsible for giving structure to a stream of related packets.

Different Level 2 protocols allow packets to be related in different ways. For example:

1. The sequenced packet protocol (SPP) ensures that an arbitrary number of transmitted packets are reliably delivered exactly once to the destination application in the same order as sent.
2. The packet exchange protocol (PEP) causes each transmitted packet to be answered by a packet from the receiver; this is useful for transaction-oriented communications.
3. The echo protocol (ECH) lets a transmitter "bounce" a packet off a host anywhere in the internet. ECHO is useful for finding out if a host exists and if there is a path to it.

Other Level 2 protocols exist mainly for communication between hosts and do not require explicit user initiation.

1. The error protocol (ERR) is for one host to inform another that a specific error has occurred.
2. The routing information protocol (RIP) is used by the Level 1 routers to communicate routing information among themselves.

ITP/VM allows you to use these protocols should your application require them.

The Level 2 functionality is in the ITPACP program running in the ITPACP1 DSVM.

3.3 IDP PACKET ADDRESSING

As we said, the Level 1 router routes internet datagrams between end-points called sockets. More than one socket exists in a host; more than one host (usually) exists on a network; and a network may be connected to other networks, which together are called an internet.

Every IDP packet contains a source network address and a destination network address. Each of these two network addresses has three parts: the network number, the host number, and the socket number.

The host number and socket number uniquely identify a socket, while the network number is used for internetwork routing. The host number of any host is the same as its 48-bit Ethernet address. The Xerox ITP specification gives a detailed description of the network address fields.

WARNING: The ITP specification restricts the values that these fields may have, and ITP/VM enforces these restrictions.

In the following discussion, UNKNOWN means zero (0) and ALL means all ones (the ones complement of 0).

1. A socket number cannot have the value UNKNOWN or ALL. The Level 1 router can't deliver to an UNKNOWN socket. Broadcasting to ALL sockets is not defined by the Xerox ITP specification.

   EXCEPTION: When your application program asks ITP/VM to allocate a local socket, specifying UNKNOWN means you don't care what socket number ITP/VM assigns. ITP/VM assigns a socket number in the range 3001 to 65534 (decimal).

2. The host number cannot have the value UNKNOWN. The Level 0 network drivers can't deliver to an UNKNOWN host. A destination host number

can be ALL or a multi-cast host number only if the destination network supports broadcast. (Ethernet supports broadcast).

3. The network number cannot be ALL. A broadcast to ALL networks ("global broadcast") is not supported. A destination network address that includes a destination network number of UNKNOWN causes transmitted packets to be sent on all networks to which your VM system is directly-connected. This is not as efficient as sending the packet on the correct directly-connected network and you should specify it only while the actual network number is unknown.

4. USER INTERFACE TO ITP/VM

4.1 OVERVIEW

User programs can use the ITP/VM runtime subroutines to interface to ITP/VM. These subroutines are contained in the library file SNITPUSR TXTLIB on the SLACNET disk.

To link these subroutines, you must load your program with a GLOBAL TXTLIB SNITPUSR in effect (plus any other TXTLIBs you might need). You may also use the GENPROG exec which is on the SLACNET disk.

The sample programs SOURCE and SINK in Appendix C demonstrate how to establish a virtual circuit between two programs.

Note that the calling conventions of the ITP/VM subroutines are identical to the calling conventions of the ITP/VMS interface produced for SLACNET.

The following figure is a block diagram of ITP/VM in operation.

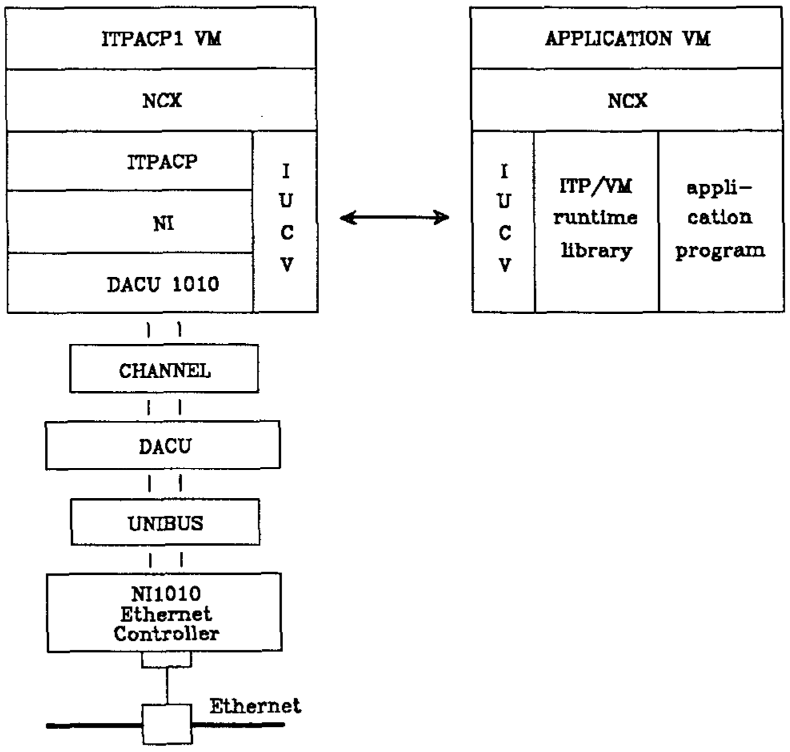

Figure 4

1. Your application program (running in the APPLICATION VM) issues an I/O request to ITP/VM. Such requests can open or close connections, send internet datagrams, get statistics, etc. This I/O request is passed to the ITPACP program (running in the ITPACP1 VM) via IUCV[1:110–176] and.[9]

   Your program will be in a wait for a Network Communications Executive (NCX) semaphore[5] until your request is satisfied or times out.
2. ITPACP performs your request. This may or may not involve sending packets on the network; the request may simply go from one of ITPACP's local sockets to another.
3. ITPACP returns status information to your program.

4.2 USER INTERFACE

Your VM program can access ITP/VM functionality in several ways. For example, your program may:

1. Open a virtual circuit connection to a remote program, or
2. Exchange related datagrams with a remote program, or
3. Send unrelated datagrams to a remote program, or
4. May request statistics about local Ethernet level and connection level performance.

Requests from an application program to ITP/VM typically follow this sequence:

1. The program requests the type of access it wants. For the four functions just mentioned, the type of access would be as a client of either:
   (a) Sequenced Packet Protocol
   (b) Packet Exchange Protocol
   (c) Internet Datagram Protocol
   (d) (No access required.)
2. The program then issues requests that are appropriate for the type of access. Following the four functions again:
   (a) Send and/or receive sequenced, reliable, unduplicated, flow-controlled arbitrary data.
   (b) Send and/or receive Packet Exchange requests and responses.
   (c) Send and/or receive unrelated Internet Datagram Packets.
   (d) Request link or connection statistics.
3. The program then terminates the transaction stream (deaccess).

The following figure shows a typical interaction (access) between a user-written application program and ITP/VM.

```
         APPLICATION VM                          ITPACP1 VM

      Access for Sequenced
      Packet Protocol
                               ------->
                                          Return Access-identifier
                                          and status
                               <-------
      Open connection on
      Access-identifier
                               ------->
                                          Open connection and
                                          return status
                               <-------
      Send and receive
      sequenced data on
      Access-identifier
                               ------->
                                          Perform operation and
                                          return status
                               <-------
      Deaccess Access-
      identifier
                               ------->
                                          Return status
                               <-------
```

4.3 GROUPS OF ITP/VM USER LIBRARY CALLS

The calls fall into eight groups:

1. IDP related
2. ECH related
3. SPP related
4. PEP related
5. Statistics related
6. General Deaccess Call
7. The Load-Multicast Call
8. Calls to initialize and terminate the user interface

The following list describes the calls in each group.

1. IDP related:
   (a) idpacc - access as client of the Internet Datagram Protocol.
   (b) idpxmt - transmit an Internet Datagram packet.

   (c) idptrcv - receive an Internet Datagram packet and truncate the remainder.

   (d) idphrcv - receive an Internet Datagram packet hold the remainder.

2. ECH related:

   (a) echacc - access as client of the Echo Protocol.

   (b) echreq - transmit an Echo packet.

3. SPP related:

   (a) sppacc - access as client of the Sequence Packet Protocol.

   (b) sppopen - open a Sequence Packet Protocol connection.

   (c) sppxmt - send a Sequence Packet Protocol packet over an SPP connection.

   (d) spptrcv - receive a Sequence Packet Protocol packet and truncate the remainder.

   (e) spphrcv - receive a Sequence Packet Protocol packet and hold the remainder.

   (f) sppforce - send a packet to a local SPP socket.

   (g) sppclose - close a Sequence Packet Protocol connection.

4. PEP related:

   (a) pepareq - access as a Packet Exchange Protocol requester.

   (b) pepxreq - transmit a PEP request and wait for response.

   (c) peptreq - receive PEP response and truncate the remainder.

   (d) pephreq - receive PEP response and hold the remainder.

   (e) pepares - access as a PEP responder.

   (f) pepxres - transmit a PEP response.

   (g) peptres - receive a PEP request and truncate the remainder.

   (h) pephres - receive a PEP request and hold the remainder.

5. STAT related:

   (a) statsgen - report general statistics.

   (b) statssock - report statistics on a specific socket.

   (c) statsconn - report statistics on a specific connection.

   (d) statnet - report statistics on a specific level 0 network interface.

   (e) statasock - report statistics on all sockets.

   (f) stataconn - report statistics on all connections.
   (g) statanet - report statistics on all level 0 network interfaces.

6. General Deaccess call:
   (a) deaccess - terminate an ITP access.
7. Load Multicast call:
   (a) loadmcast - load multicast address(es).
8. Init and Close calls:
   (a) itpini - initialize the ITP user interface.
   (b) itpclose - close the ITP user interface.

4.4 ACCESS IDENTIFIERS: The aid argument

Your program, which may have many concurrent program-to-ITP request streams, (e.g. many open connections), begins each stream by requesting an access for a particular purpose (e.g. "I want to be a client of the Sequenced Packet Protocol").

If ITP/VM successfully grants the access your program requested, it returns to your program a value called an access identifier or access-ID. The returned access-ID is your "handle" and you must specify it in future related I/O requests.

NOTE: It is possible to have more than one active access to the same local socket. Furthermore, these multiple accesses do not have to be of the same type.

Be careful if you use this capability since packets received at the socket are matched up to supplied receive buffers on a first-come-first-served basis. Unless you make other provisions, there is no way of determining which access will receive the packet.

4.5 REQUESTS TO ITP/VM

This section describes some general features of the ITP/VM calls.

4.5.1 ITP/VM calling format

All requests to ITP/VM use the format:

msgcode = call-name(arg1, arg2, ... argn);

where "msgcode" is the value returned by the call.

All ITP related subroutines in the ITP User Interface library (SNITPUSR TXTLIB) return an UNSIGN32 value. The three rightmost bits of the msg-code indicate SUCCESS or ERROR, and in the latter case, the severity of the error:

```
   0    Warning
   1    Success
   2    Error
   3    Informational
   4    Severe error, FATAL
  5-7   Reserved
```

XN_SUCCESS indicates a successful return. The files IUCVMSG H and XNMSG H (appendix A) on the SLACNET disk contain the complete catalog of possible error codes.

4.5.2 msgCnvrt - the MESSAGE facility

The <msgCnvrt> function on the SLACNET disk can be used to translate the 32-bit error code into a pointer to an ECBDIC string which then can be printed out. For details, see[10] and the examples in section 4.6

4.5.3 Timeouts

Many ITP/VM requests let you specify a timeout value. This allows you to abort a requested operation that has not completed after the specified time. The timeout value that you supply is in 10 millisecond units; for example a value of 600 decimal means "six seconds". A timeout value of zero indicates an indefinite wait, i.e., return only when the request is satisfied or aborted.

In VM, the clock granularity is one second. Timeout values of any fraction of a second are always truncated. For example, a timeout value of 620 is equivalent to 6 seconds.

4.5.4 Two Ways to Receive Packets

This section describes two ways for a user program to receive packets from ITP/VM: RECEIVE AND HOLD and RECEIVE AND TRUNCATE.

When your program supplies a buffer for packet reception, it usually has no way of knowing how large the next arriving packet will be and hence doesn't know how large a buffer it should supply. There are several ways to solve this problem; the solution usually depends on the application. Here are some solutions:

(a) Your program can always supply a buffer big enough to hold the largest packet it expects to receive. This method requires a lot of buffer space, and can be inefficient when the packets vary in length.

(b) Your program can have the protocol pass it the size of the received packet. This lets your program allocate a buffer of the correct size. This method is not as wasteful of memory space, but doubles the number of transactions that must occur between the program and the protocol.

(c) Your program can ask for the packet one (or a few) bytes at a time. This method simplifies your program's logic, it is usually inefficient because it requires a transaction for each byte or group of bytes in the packet.

There are other issues related to packet reception. For example, how can your program obtain the first part (the header) of a packet to see if it is interested in the remainder (the data)? Further, if your program doesn't want the remainder, how can it tell the protocol to discard it?

ITP/VM supports all of these strategies by providing a very general interface for receiving packets. This interface works as follows:

1. Your program issues one of two types of receive requests, either RECEIVE AND HOLD or RECEIVE AND TRUNCATE. With this request it specifies the address and size of a receive buffer. The size is allowed to be zero bytes.
2. As soon as there is received data available, ITP/VM transfers as much as will fit into your receive buffer. If the specified buffer size is zero, no transfer will take place. ITP/VM returns the number of bytes transferred and the number of bytes remaining (potentially zero) as described in each call below.
3. If the receive request is RECEIVE AND TRUNCATE, ITP/VM throws away any remaining received data that didn't fit into the supplied receive buffer. If the receive request is RECEIVE AND HOLD, ITP/VM will hold on to any remaining received data. Your program can obtain this data by issuing another receive request, specifying either TRUNCATE or HOLD.

   You can use combinations of these requests to implement any receive buffering strategy. For example, your program could specify HOLD to receive a packet header, followed by TRUNCATE with a zero length buffer to throw away the packet data. Your program could find out the length of the next received packet by specifying HOLD with a zero length buffer.

   With the exception of SPP RECEIVE AND HOLD requests, ITP/VM will never return more than one packet in response to a single receive request. If you supply a buffer which is larger than the next arriving packet (or the rest of a partially consumed packet from a previous HOLD request), the request is completed when the packet is returned in the buffer. ITP/VM will not wait for further received data to fill the buffer. Because of this, you are assured that packets aren't arbitrarily placed in receive buffers. The beginning of a received packet will always be returned at the beginning of a supplied receive buffer.

A different strategy is used for SPP RECEIVE AND HOLD: the data portions of multiple packets are considered a unstructured byte stream unless EOM or ATTN are set. When the request is made and several packets have arrived, data is copied into the user buffer until either the buffer is filled or the data is exhausted (see also section 4.6.22). Thus, the sender's SPP packet structure is invisible to the receiver.

4.5.5 Multicast Addressing

ITP/VM allows you to specify as the destination network address parameter of many requests a 48-bit Ethernet multicast (sometimes called group or logical) host number. Each transmitted Ethernet packet that specifies a particular multicast destination address will be received only by network hosts that have enabled that multicast address.

When ITP/VM begins operation, it will only receive packets addressed to its host's physical address or the broadcast address. If your application requires reception of multicast packets, it can enable one or more multicast addresses to be recognized for the duration of an ITP/VM access. Your application should specify the required multicast address(es) in the load multicast call (loadmcast()).

4.6 DESCRIPTION OF ITP/VM PROGRAM CALLS

This section describes each ITP/VM request in detail. The descriptions are listed in alphabetical order.

Each request to ITP/VM involves a call to one of the subroutines in the SNITPUSR TXTLIB. Each of these subroutines, in turn, builds a block of information about the client process. ITP/VM will use this information to process the user request. This block of data is delivered to ITPACP via IUCV[1:110–176] and.[9]

Your program must include the file "snitpusr.h" to make use of the subroutines in the SNITPUSR TXTLIB and the definition of structures such as statsconn, statsgen, statnet, statssock, rcv_spp, and rcv_itp.

Note that all the necessary extern definitions are included in "snitpusr.h", and in consequence need not be put in your program. Note also that all these subroutines are macro definitions which the C compiler will convert to a special call (to one of a much smaller set of routines). Therefore you must ensure that you put no spaces between the routine name and the opening parenthesis of the routine name.

4.6.1 deaccess()

Dissolve an ITP access.

Synopsis:
```
unsign32 deaccess( a_id );
unsign16 a_id;
```

Description:
This request causes the access with access-ID of a_id to be terminated. Any outstanding requests on this access are finished with an error return of XN_ABORT. Any related socket and/or Sequenced Packet Protocol connection will be destroyed. The access-ID is now meaningless.

If your VM is re-ipled with any accesses still active, this operation will be implicitly performed by ITP/VM on all active accesses owned by your VM.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id;
unsign32 msgcode;

if ( ( msgcode = deaccess( a_id ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.2 echacc()

Access as a Client of the ECHO Protocol.

Synopsis:
```
unsign32 echacc( &a_id, sock_num );
unsign16 a_id, sock_num;
```
Description:
This request identifies the issuing program as a client of the Echo Protocol. In a_id, the call returns the access-ID to be used in subsequent Echo Requests (echreq()).

A local socket with number <sock_num> is created. If <sock_num> is zero, the socket will be given a dynamically allocated socket number from 3001 to 65534 decimal.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id, sock_num;
unsign32 msgcode;

if ( ( msgcode = echacc( &a_id, sock_num ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.3 echreq()

Echo Request.

Synopsis:
```
   unsign32 echreq( &nbytes,a_id,dna,timeout,buff,buffsize );
   char     buff[];
   unsign16 a_id, buffsize, dna[6], nbytes, timeout;
```
Description:
This request is useful for verifying that a path exists through the internet to and from <dna[6]>.

The request causes the contents of a user-supplied buffer to be sent as an Echo Request packet. The packet is sent from the socket obtained in a previous Echo Access request (echacc()). The packet is sent to the destination address specified in <dna[6]>. The request then waits at the local socket for an Echo Protocol reply.

The argument <a_id> is an Access-Identifier that was obtained in a previous Echo Protocol Access (echacc()).

The request will timeout if no PEP request packet is received within the <timeout> interval.

You should take care when specifying a multicast or broadcast host number in <dna[6]>. A large number of Echo Response packets could appear at the local socket causing buffers internal to ITP/VM to be consumed. These buffers are unavailable until the local socket is destroyed as a result of a Deaccess request (deaccess()).

The number of bytes transmitted is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
   #include <snitpusr.h>

   extern char      *msgCnvrt();

   char     buff[] = "test echo\n", msgbuf[512];
   unsign16 a_id,
            dna[6] = {0x0000,0x0000,0xAA00,0x0400,0x09A4,0x0040},
            nbytes;
   unsign32 msgcode;

   echacc( &a_id, 0 );
   if ( ( msgcode = echreq( &nbytes, a_id, dna, 600, buff,
                            sizeof( buff ) ) ) != XN_SUCCESS )
          printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.4 idpacc()

Access as Client of IDP.

Synopsis:

```
unsign32 idpacc( &a_id, sock_num );
unsign16 a_id, socknum;
```

Description:

This request identifies the issuing program as a client of the Internet Datagram Protocol (IDP). A local socket with socket number <sock_num> is created. The call returns the access-ID to be used for all other further requests on sock_num.

If <sock_num> is UNKNOWN (zero), the socket number will be assigned dynamically. You have no control over what socket number is assigned if <sock_num> is UNKNOWN (except that it will be in the range of dynamic socket numbers: decimal 3001 to 65534).

After successful completion of this operation, you can send and/or receive IDP packets from the local socket and/or obtain socket specific statistics.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id, sock_num;
unsign32 msgcode;

if ( ( msgcode = idpacc( &a_id, sock_num ) ) != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.5 idphrcv()

Receive IDP Packet and Hold.

Synopsis:
```
unsign32 idphrcv( &rcv_itp,a_id,timeout,buff,buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:
This request causes the next available IDP packet from the local socket (created at IDP Access time) to be returned in <buff>. The packet is not de-capsulated; the 30-byte IDP header is also available. All packets arriving at the socket (including Error Protocol packets) are available.

If the length of the data to be returned is greater than the size of the user buffer, the remainder will be held and can be obtained with another Receive Internet Datagram (Hold or Truncate) request.

The variable "rcv_itp.nbytes" contains the number of bytes remaining in the packet. The variable "rcv_itp.rcv_bytes" contains the number of bytes actually placed in the user buffer. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

The request will timeout if no packet is received within the specified <timeout> interval.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct   rcv_itp rcv_itp;
unsign16 a_id;
unsign32 msgcode;

idpacc( &a_id, 0 );
if ( ( msgcode = idphrcv( &rcv_itp, a_id, 600, buff,
                          sizeof( buff ) ) ) != XN_SUCCESS )
      printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.6 idptrcv()

Receive IDP Packet and Truncate.

Synopsis:
```
unsign32 idptrcv( &rcv_itp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:
This request causes the next available Internet Datagram Protocol packet from the local socket (created at IDP Access time) to be returned in the user buffer. The packet is not de-capsulated; the 30-byte IDP header is also available. All packets arriving at the socket (including Error Protocol packets) are available.

If the length of the data available is larger than the user buffer, the remainder will be thrown away. The variable "rcv_itp.nbytes" will contain the number of bytes that were thrown away. The variable "rcv_itp.rcv_bytes" contains the number of bytes actually placed in your buffer. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

The request will timeout if no PEP request packet is received within the <timeout> interval.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct   rcv_itp rcv_itp;
unsign16 a_id;
unsign32 msgcode;

idpacc( &a_id, 0 );
if ( ( msgcode = idptrcv( &rcv_itp, a_id, 600, buff,
                          sizeof( buff ) ) ) != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.7 idpxmt()

Transmit an IDP Packet.

Synopsis:

```
unsign32 idpxmt( &nbytes, a_id, dna, type, check, buff,
                    buffsize );
BOOLEAN  check;
char     buff[], type;
unsign16 a_id, buffsize, dna[6], nbytes;
```

Description:
This request encapsulates the data in the buffer into an Internet Datagram and transmits it.

Arguments:
The argument <a_id> is the Access-ID returned by a previous IDP Access operation (idpacc()).

The argument <dna[6]> is the address of the destination socket (network, host, and socket number).

A destination network number of UNKNOWN (0) causes the data to be sent on all directly connected networks and is less efficient than a specific (non-zero) destination network number. The destination network number can be ALL (ones complement of zero) or a multicast host number only if the destination network supports broadcast and multicast.

The argument <type> is the IDP Packet Type byte to be placed in the packet. This value should be used to indicate to the receiving program how the data portion contents of the packet should be interpreted. You can obtain from Xerox unique values for the Internet Packet Type.

The argument <check_flag> should be TRUE (1) if you want the IDP packet to carry a software checksum within it; otherwise, it will be transmitted unchecksummed. You could specify a value of FALSE for <check_flag> if you knew that the underlying transmission medium provided an acceptable level of packet data reliability. For example, each Ethernet packet contains a 32-bit CRC, guaranteeing a high level of packet data reliability.

Note that the supplied data buffer should contain only the data portion of the IDP packet. ITP/VM will encapsulate the data into an IDP packet.

The actual number of data bytes transmitted is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[] = "test idp\n", msgbuf[512];
unsign16 a_id,
         dna[6] = {0x0000,0x0000,0xAA00,0x0400,0x09A4,0x0040},
         nbytes;
unsign32 msgcode;

idpacc( &a_id, 0 );
if ( ( msgcode = idpxmt( &nbytes, a_id, dna, 0, FALSE, buff,
                         sizeof( buff ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.8 itpclose()

Close the ITP User Interface.

Synopsis:
```
   unsign32 itpclose();
```
Description:
The itpclose() call causes ITP to deaccess all active sockets belonging to the calling program. The call also severs the IUCV path to ITPACP[1:156–157] and.[9]

Errors: See Appendix A.

Example:
```
   #include <snitpusr.h>

   extern char     *msgCnvrt();

   char     msgbuf[512];
   unsign32 msgcode;

   if ( ( msgcode = itpclose() ) != XN_SUCCESS )
          printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.9 itpini()

Initialize the IUCV path to the ITPACP VM.

Synopsis:
```
   unsign32 itpini();
```

Description:
This call connects an IUCV path to the ITPACP VM[1:143–144] [9].

Errors: See Appendix A.

Example:
```
   #include <snitpusr.h>

   extern char     *msgCnvrt();

   char     msgbuf[512];
   unsign32 msgcode;

   if ( ( msgcode = itpini() ) != XN_SUCCESS )
          printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.10 loadmcast()

Load Multicast Address(es).

Synopsis:
```
unsign32 loadmcast( &nbytes, a_id, buff, buffsize );
char     buff[];
unsign16 a_id, buffsize, nbytes;
```

Description:

The loadmcast() call enables the multicast addresses in <buff> for the duration of the access defined by <a_id>.

<buff> points to a list of multicast addresses.

<buffsize> is the length of this list, in bytes, and it must be a multiple of six bytes.

The actual number of bytes transmitted is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

The specified multicast addresses are actually loaded into the NI1010A Controller. The NI1010A filters all received multicast packets.

Errors: See Appendix A.

Example:
```
/* enable two multicast addresses */

#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id,
         mcast[] = {0x0307,0x0100,0x0100,0x0307,0x0100,0x0101};
unsign32 msgcode;

if ( ( msgcode = loadmcast( NIL,a_id,mcast,sizeof( mcast ) ) )
       != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.11 pepareq()

Access as a PEP Requester.

Synopsis:

```
pepareq( &a_id, sock_num );
unsign16 a_id, sock_num;
```

Description:

Creates a local socket with number <sock_num> which the issuing program can use for issuing PEP requests. The Access-ID returned in <a_id> can be used in subsequent Transmit PEP Request (pepxreq()) and Receive Response (pephreq(), peptreq()) calls.

If <sock_num> is UNKNOWN (0), a socket number between 3001 and 65534 (decimal) will be dynamically allocated.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id, sock_num;
unsign32 msgcode;

if ( ( msgcode = pepareq( &a_id, 0 ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.12 pepares()

Access as PEP Responder.

Synopsis:
```
   unsign32 pepares( &a_id, sock_num );
   unsign16 a_id, sock_num;
```

Description:
Your program can issue this request to obtain a socket so that it can receive PEP Requests and transmit PEP Responses.

<a_id> will contain an Access_ID to be used in subsequent Send PEP Response (pepxres()) and Receive PEP Request (pephres(), peptres()) calls.

<sock_num> is the number of the (local) socket. A value of UNKNOWN (0) means that ITP/VM is to dynamically allocate a socket number from the range 3001 to 65534 (decimal). You will probably want to specify a "well-known" socket number here unless you have made arrangements for remote PEP Requesters to know the socket number.

Errors: See Appendix A.

Example:
```
   #include <snitpusr.h>

   extern char     *msgCnvrt();

   char     msgbuf[512];
   unsign16 a_id;
   unsign32 msgcode;

   if ( ( msgcode = pepares( &a_id, 100 ) ) != XN_SUCCESS )
         printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.13 pephreq()

Receive PEP Response and Hold.

Synopsis:
```
unsign32 pephreq( &rcv_itp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:
After a PEP Request operation (pepxreq()) completes, your program can invoke this function to obtain PEP Response data. <a_id> is the Access-ID returned from a prior PEP Requester Access (pepareq()).

If the supplied buffer is smaller than the remaining PEP Response data, as much of the PEP Response that will fit is moved into the buffer, and the remainder is kept. The number of bytes remaining is returned in <rcv_itp.nbytes>. The remainder of the packet can be obtained with another Receive PEP Response and Truncate (peptreq()) or Hold (pephreq()) call. The number of bytes actually placed in the buffer is returned in <rcv_itp.rcv_bytes>. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

The value of <timeout> is not used by this call since, by definition, a response must be waiting to be read.

Note that the entire PEP Response packet is available. This includes the IDP header, PEP ID and Client Type as well as any data.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct   rcv_itp rcv_itp;
unsign16 a_id;
unsign32 msgcode;

pepareq( &a_id, 0 );
if ( ( msgcode = pephreq( &rcv_itp, a_id, 1, sizeof( buff ) ) )
       != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.14 pephres()

Receive PEP Request and Hold.

Synopsis:

```
unsign32 pephres( &rcv_itp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:

Your program can invoke this operation to wait for the arrival of a PEP Request packet. The PEP Request packet is returned in the supplied buffer.

<a_id> is the Access_ID returned by a previous Access for PEP Responder request (pepares()).

The request will timeout if no PEP request packet is received during a <timeout> interval.

If the remaining PEP Request packet data is larger than the size of the supplied buffer, the remainder will be kept. The remainder can be obtained with additional Receive PEP Request Packet operations (pephres(), peptres()). <rcv_itp.nbytes> will contain the number of bytes remaining, and <rcv_itp.rcv_bytes> will contain the number actually transferred into <buff>. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

Note that the entire PEP Request packet is available. This includes the 30-byte IDP header as well as the PEP ID and Client Type.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct rcv_itp rcv_itp;
unsign16 a_id;
unsign32 msgcode;

pepares( &a_id, 100 );
if ( ( msgcode = pephres( &rcv_itp,a_id,1000,sizeof( buff ) ) )
        != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.15 peptreq()

Receive PEP Response and Truncate.

Synopsis:
```
unsign32 peptreq( &rcv_itp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:
After a Transmit PEP Request operation (pepxreq()) completes, your program can invoke this function to obtain the PEP Response data.

<a_id> is the Access-ID returned from a previous PEP Requester Access (pepareq()).

If the supplied buffer is smaller than the remaining PEP Response data, as much of the PEP Response that will fit is moved into the buffer, and the remainder is lost. The number of bytes lost is returned in <rcv_itp.nbytes>. The number of bytes actually placed in buff is returned in <rcv_itp.rcv_bytes>. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

The value of <timeout> is not used by this call since, by definition, a response must be waiting to be read.

Note that the entire PEP Response packet is available. This includes the IDP header, PEP ID and Client Type as well as any data.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char    *msgCnvrt();

char     buff[512], msgbuf[512], rq[] = "this is a request";
struct   rcv_itp rcv_itp;
unsign16 a_id,
         dna[6] = {0x0000,0x0000,0xAA00,0x0400,0x09A4,0x0040},
         nbytes;
unsign32 msgcode;

pepareq( &a_id, 0 );
pepxreq( &nbytes, a_id, dna, 1, 1000, 100, FALSE, rq,
         sizeof( rq ) );
if ( ( msgcode = peptreq( &rcv_itp, a_id, 1, buff,
                 sizeof( buff ) ) ) != XN_SUCCESS )
      printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.16 peptres()

Receive PEP Request and Truncate.

Synopsis:
```
unsign32 peptres( &rcv_itp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_itp rcv_itp;
unsign16 a_id, buffsize, timeout;
```

Description:
Your program can invoke this operation to wait for the arrival of a PEP Request packet. The PEP Request packet will be returned in the supplied buffer.

<a_id> is the Access_ID returned by a previous Access for PEP Responder operation (pepares()).

The request will timeout if no PEP request packet is received within a <timeout> interval.

If the remaining PEP Request packet data is larger than the size of the supplied buffer, the remainder will be thrown away. Upon success, <rcv_itp.nbytes> will return the number of bytes lost and <rcv_itp.rcv_bytes> will return the number of bytes actually placed in <buff>. If you are not interested in these numbers, specify a NIL pointer instead of <&rcv_itp>.

Note that the entire PEP Request packet is available. This includes the 30-byte IDP header as well as the PEP ID and Client Type.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct   rcv_itp rcv_itp;
unsign16 a_id;
unsign32 msgcode;

pepares( &a_id, 100 );
if ( ( msgcode = peptres( &rcv_itp, a_id, 1000, buff,
                          sizeof( buff ) ) ) != XN_SUCCESS )
      printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.17 pepxreq()

Transmit PEP Request and Wait for PEP Response.

Synopsis:

```
unsign32 pepxreq( &nbytes,a_id,dna,ctype,timeout,response,
                   check, buff, buffsize );
BOOLEAN  check;
char     buff[];
unsign16 a_id, buffsize, ctype, dna[6], nbytes, response,
         timeout;
```

Description:
This request causes a PEP Request packet to be sent from the local socket acquired by a previous PEP Requester Access request (pepareq()).

The packet's destination is given by <dna[6]>. The packet carries a client type of <ctype>. If <check> is TRUE, the PEP packet also carries an Internet Datagram Protocol checksum.

<response> is an estimate of the delay time (in 10ms units) at the remote node. This is the time that the remote node would take to process an incoming PEP Request, formulate the PEP response, and transmit the response.

The request waits for a PEP Reply to arrive at the local socket. The Reply must match the Request. If no Reply is received in a <timeout> interval, the request is finished with a timeout error. ITP/VM may repeatedly transmit the PEP Request packet, depending on <response>, <timeout>, and an internally generated network "round-trip delay" estimate.

After a PEP Request has been successfully completed, you can obtain the PEP Response packet by issuing a Receive PEP Response (pephreq(), peptreq()) call.

Note that the buffer should contain only the data portion (if any) of the PEP Request. ITP/VM will encapsulate the data and provide the IDP header and PEP ID.

The number of bytes transmitted from the buffer <buff> is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512], request[] = "this is a request";
unsign16 a_id,
         dna[6] = {0x0000,0x0000,0xAA00,0x0400,0x09A4,0x0040},
         nbytes;
unsign32 msgcode;

pepareq( &a_id, 0 );
if ( ( msgcode = pepxreq( &nbytes, a_id, dna, 0, 1000, 0, FALSE,
                 request, sizeof( request ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.18 pepxres()

Send PEP Response.

Synopsis:
```
unsign32 pepxres( &nbytes,a_id,ctype,check,buff,buffsize );
BOOLEAN  check;
char     buff[];
unsign16 a_id, buffsize, ctype, nbytes;
```

Description:
After a PEP Request packet (pephres(), peptres()) is received, your program can invoke this operation to send a PEP Response packet.

<a_id> is the Access-ID returned by a previous Access for PEP Responder operation (pepares()).

<ctype> is the Client Type that ITP/VM will use when it constructs the Response packet.

<check_flag> should be TRUE (1) if the Response packet is to carry an IDP checksum with it.

Note that the buffer should contain only the data portion of the PEP Response packet. ITP/VM will encapsulate the data with an IDP header, PEP ID and the specified Client Type. ITP/VM will send the PEP Response packet to the socket that originated the PEP Request packet.

The number of bytes transmitted from the buffer <buff> is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512], req[512], resp[] = "this is the response";
struct   rcv_itp rcv_itp;
unsign16 a_id, nbytes;
unsign32 msgcode;

pepares( &a_id, 100 );
peptres( &rcv_itp, a_id, 1000, req, sizeof( req ) );
if ( ( msgcode = pepxres( &nbytes, a_id, 0, FALSE, resp,
                          sizeof( resp ) ) ) != XN_SUCCESS )
      printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.19 sppacc()

Access as a Client of SPP.

Synopsis:
```
unsign32 sppacc( &a_id, sock_num );
unsign16 a_id, sock_num;
```

Description:

This request allocates a socket with number <sock_num> for purposes of sending and receiving Sequenced Packet Protocol (SPP) packets. The sppacc() call does NOT open a connection. To open an SPP connection, use the sppopen() call.

Upon successful completion, <a_id> contains the Access-ID which must be specified in subsequent Sequenced Packet Open (sppopen()), Send (sppxmt()), Receive (spptrcv() and spphrcv()), Force(sppforce()), Close (sppclose()), and Deaccess (deaccess()) operations.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id;
unsign32 msgcode;

if ( ( msgcode = sppacc( &a_id, 0 ) ) != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.20 sppclose()

Close an SPP Connection.

Synopsis:
```
unsign32 sppclose( a_id );
unsign16 a_id;
```

Description:
This request causes an open connection to be dissolved. The remote end of the connection is not informed. The local socket still exists. The Access-ID <a_id> is still valid.

All unconsumed received packets are thrown away. All transmitted but unacknowledged packets are thrown away.

No packets are transmitted or received as a result of this request.

Because the local socket still exists, any further packets that arrive at the socket are retained.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id;
unsign32 msgcode;

if ( ( msgcode = sppclose( &a_id ) ) != XN_SUCCESS )
      printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.21 sppforce()

Force Sequenced Packet.

Synopsis:

```
   unsign32 sppforce( &nbytes, a_id, buff, buffsize );
   char     buff[];
   unsign16 a_id, buffsize, nbytes;
```

Description:

This request is used by a service-listener program to pass a previously received service-request packet (SPP) to a service-supplier program. Your service-listener program is responsible for starting the service-supplier process and coordinating the service-supplier's socket.

<a_id> is the Access_ID returned by a previous Access for Sequenced Packet Protocol request (sppacc()).

The packet pass-off goes as follows (ITPACP has been modified to permit this type of pass-off):

1. Your program accesses a "well-known" service-listener socket as a client of the IDP (idpacc()).
2. Your program issues an IDP receive request (idptrcv()) on the returned Access-ID.
3. A service-request packet (SPP) arrives at the well-known socket and the receive request issued in step 2 completes.
4. Your program accesses an UNKNOWN socket as a client of the SPP (sppacc()), causing a new socket to be dynamically allocated.
5. Your program issues the Force Sequenced Packet request (sppforce()) specifying the Access-ID returned by ITP/VM in step 4. ITP/VM will change the destination socket number in the packet to be that of the socket obtained in step 4. ITP/VM then sends the packet to that socket as if it had been received from the network and routed to that socket.
6. Your listener program spawns a supplier process passing it the Access-ID obtained in step 4.
7. Using the passed-off Access-ID, the supplier program executes a passive sppopen() and proceeds.

The number of bytes forced on the SPP socket is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>
#define PRI_supplier   5    /* priority of supplier process */
#define SIZ_supplier   8192 /* stack size of above          */

extern char     *msgCnvrt();
extern struct pcb *c_fork();

unsign16 dna0[] = {0, 0, 0, 0, 0, 0};

PROCESS listener()
{
  extern PROCESS supplier();
  char      buff[512], msgbuf[512];
  struct    rcv_itp rcv_itp;
  unsign16 idp_aid, nbytes, spp_aid;
  unsign32 msgcode;

  idpacc( &idp_aid, 0 );
  while ( ( msgcode = idptrcv( &rcv_itp, &idp_aid, 0, buff,
                              sizeof( buff ) ) ) == XN_SUCCESS &&
          ( msgcode = sppacc( &spp_aid, 0 ) ) == XN_SUCCESS &&
          ( msgcode = sppforce( &nbytes, &spp_aid, buff,
                            rcv_itp.rcv_bytes ) ) == XN_SUCCESS )
       c_fork( supplier, spp_aid, PRI_supplier, SIZ_supplier );

  printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
}

PROCESS supplier( a_id )
unsign16 a_id;
{
     sppopen( a_id, dna0, 600, FALSE, FALSE, FALSE );
     .
     .
     deaccess( a_id );
}
```

4.6.22 spphrcv()

Receive SPP Data and Hold.

Synopsis:
```
unsign32 spphrcv( &rcv_spp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_spp rcv_spp;
unsign16 a_id, buffsize, timeout;
```

Description:
Returns in a user-supplied buffer pointed to by <buffp> at most <buffsize> bytes of Sequenced Packet data that has not been accepted by the program. Only the data field of Sequence Packets is returned.

Note that, unlike all other receive requests, the size of the buffer is not restricted to 1500 bytes. As SPP packets arrive, their data will be appended into the buffer. The request will successfully complete on the following conditions:

1. The buffer is filled.
2. The buffer contains data from previous packet(s) and a just-arrived packet has a different datastream type.
3. The buffer contains data from previous packet(s) and a just-arrived packet is an Attention packet.
4. The buffer contains data from previous packet(s) and no new data is received in the timeout period.
5. The last data byte of a packet with the EOM bit set has been placed in the buffer.
6. The buffer contains data byte from an Attention packet.

In all cases, the remaining packet data will be held and can be obtained with another Receive Sequenced Packet (spphrcv(), spptrcv()) call.

<a_id> is the Access-Identifier returned by a previous Sequenced Packet Protocol Access(sppacc()).

The request will timeout if no data is available in a <timeout> interval.

Upon success the call returns the following items:

1. <rcv_spp.rcv_bytes> contains the number of bytes placed in the user buffer <buff>.
2. <rcv_spp.nbytes> contains the number of bytes not transferred to the user.
3. <rcv_spp.type> is the datastream type from the packet that contained the returned data.

4. <rcv_spp.eom> is TRUE (1) if the eom bit was on in the packet that contained the received data.
5. <rcv_spp.attn> is TRUE if the Attention bit was on in the packet that contained the received data. As per the Xerox ITP Specification, attention packets are delivered twice, immediately upon reception and again in the order of packets transmitted.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct rcv_spp rcv_spp;
unsign16 a_id;
unsign32 msgcode;

sppacc( &a_id, 0 );
if ( ( msgcode = spphrcv( &rcv_spp, a_id, 1000, buff,
                          sizeof( buff ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.23 sppopen()

Open an SPP Connection.

Synopsis:

```
unsign32 sppopen(a_id, dna, timeout, active, check, relp);
BOOLEAN  active, check, relp;
unsign16 a_id, dna[6], timeout;
```

Description:
This request opens a Sequenced Packet Protocol connection between the local socket obtained in a previous Sequenced Packet Access request (sppacc()) and the destination network address specified in <dna[6]>.

The request will timeout if no connection is made within a <timeout> interval.

If <active> is TRUE, the connection will be initiated from the local socket (active open) and a probe will be sent to <dna[6]>. Otherwise, the request will wait at the local socket for an initiation from the destination network address (passive open). In the case of a passive open, any or all <dna[6]> fields may be specified as UNKNOWN (0), allowing "don't care" address matches of remote active open attempts.

If <check> is TRUE, all outbound packets will carry a software (IDP) checksum. In any case, all inbound packets containing a software checksum will be checked.

If <relp> is TRUE, the local end of the connection will receive packets in reliable packet mode, that is, exactly once, but not necessarily in the order sent by the other end of the connection. Reliable packet mode decreases the received packet buffering load placed on ITP/VM. Reliable packet mode should be used by applications (such as real-time or voice applications) that require once-only delivery, but not packet ordering.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
unsign16 a_id,
         dna[6] = {0x0000,0x0000,0xAA00,0x0400,0x09A4,0x0040};
unsign32 msgcode;

sppacc( &a_id, 0 );
if ( ( msgcode = sppopen( a_id,dna,1000,TRUE,FALSE,FALSE ) )
       != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.24 spptrcv()

Receive SPP Data and Truncate.

Synopsis:

```
unsign32 spptrcv( &rcv_spp, a_id, timeout, buff, buffsize );
char     buff[];
struct   rcv_spp rcv_spp;
unsign16 a_id, buffsize, timeout;
```

Description:

Returns in a user-supplied buffer pointed to by <buff> at most <buffsize> bytes of Sequenced Packet data that has not been accepted by the program.

Only the data field of Sequence Packets is returned. If the length of the data remaining in the packet is greater than the size of the buffer, the remaining packet data will be lost.

<a_id> is the Access-Identifier returned by a previous Sequenced Packet Protocol Access (sppacc()) request.

The request will timeout if no data is available in a <timeout> interval.

Upon success the call returns the following items:

1. <rcv_spp.rcv_bytes> contains the number of bytes placed in the user buffer <buffp>.
2. <rcv_spp.nbytes> contains the number of bytes not transferred to the user. These bytes are now lost.
3. <rcv_spp.type> is the datastream type from the packet that contained the returned data.
4. <rcv_spp.eom> is TRUE (1) if the eom bit was on in the packet that contained the received data.
5. <rcv_spp.attn> is TRUE if the Attention bit was on in the packet that contained the received data. As per the Xerox ITP Specification, attention packets are delivered twice, immediately upon reception and again in the order of packets transmitted.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
struct rcv_spp rcv_spp;
unsign16 a_id;
unsign32 msgcode;

sppacc( &a_id, 0 );
if ( ( msgcode = spptrcv( &rcv_spp, a_id, 1000, buff,
                       sizeof( buff ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.25 sppxmt()

Transmit SPP Data.

Synopsis:
```
unsign32 sppxmt( &nbytes, a_id, type, attn_flag, eom_flag,
                  buff,  buffsize );
BOOLEAN  attn_flag, eom_flag;
char     buff[], type;
unsign16 a_id, buffsize, nbytes;
```

Description:
This request causes the contents of a user-supplied buffer to be accepted for transmission by the Sequenced Packet Protocol.

Successful completion of this request does not necessarily mean that the data has already passed to the remote process, received by the remote node, or sent by the local node. If necessary, a higher level protocol can be used to determine if a specific packet has been accepted by the remote process.

<a_id> is the Access-Identifier that was returned by a previous Sequenced Protocol Access Request (sppacc()).

<type> is a user-supplied datastream-type. Its value is passed uninterpreted to the remote process. <type> should be used to indicate to the remote process how to interpret the data. You could, for example, specify different <type> values to multiplex several data stream formats onto a single connection.

<attn_flag> can be set to TRUE (1) to cause the packet to be marked as an Attention Packet. If <attn_flag> is true, only one byte of data is allowed in the buffer. As per the Xerox ITP Specification the packet will be sent to the remote node regardless of the receivers flow control state.

<eom_flag> can be set to TRUE (1) to cause the Sequenced Packet carrying the supplied data to have its eom flag set. You could set <eom_flag> to TRUE, for example, to indicate the end of a logical message that is being sent as more than one SPP packet.

Note that the buffer should contain only the data to be sent. ITP/VM will provide the IDP and SPP headers.

The number of bytes accepted for transmission by ITP/VM is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
unsign16 a_id, nbytes;
unsign32 msgcode;

sppacc( &a_id, 0 );
if ( ( msgcode = sppxmt( &nbytes, a_id, 0, FALSE, FALSE, buff,
                      sizeof( buff ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.26 stataconn()

Report Statistics On All Local Connections.

Synopsis:
```
unsign32 stataconn( &nbytes, buff, buffsize );
char     buff[];
unsign16 buffsize, nbytes;
```

Description:
This request returns, in a user supplied buffer, information about all (any) connections.

The first value returned is a 32-bit unsigned binary number of currently open connections.

Then follows zero or more blocks of per connection statistics. Each block follows the format defined in the Get Specific Connection Statistics request (statsconn()).

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes transferred to the user buffer is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
unsign16 nbytes;
unsign32 msgcode;

if ( ( msgcode = stataconn( &nbytes, buff, sizeof( buffp ) ) )
        != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.27 statanet()

Report Statistics On All Level 0 Network Interfaces.

Synopsis:
```
unsign32 statanet( &nbytes, buff, buffsize );
char     buff[];
unsign16 buffsize, nbytes;
```

Description:

This request will return, in a user-supplied buffer, statistics of all (any) Level 0 network interfaces.

The first value returned is a 32-bit unsigned binary number of network drivers found.

Then follows zero or more blocks of per network statistics. Each block follows the format defined in the Report Specific Network Statistics request (statnet()) except that each block is preceded by the 32-bit network number of that network.

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The statistics block corresponding to a Level 0 network interface not provided by INTERLAN will be filled with zeroes.

The number of bytes placed in the user buffer <buff> is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
unsign16 nbytes;
unsign32 msgcode;

if ( ( msgcode = statanet( &nbytes, buff, sizeof( buffp ) )
        != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.28 statasock()

Report Statistics on all Local Sockets.

Synopsis:

```
unsign32 statasock( &nbytes, buffp, buffsize );
char     buff[];
unsign16 buffsize, nbytes;
```

Description:

Returns in a user-supplied buffer, statistics about all existing (local) sockets.

The first value returned is a 32-bit unsigned binary number of existing sockets.

Then follows zero or more blocks of per-socket statistics. Each block follows the format described in the statssock() request, which is used to report statistics on a specific socket.

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes transferred to the user buffer <buff> is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     buff[512], msgbuf[512];
unsign16 nbytes;
unsign32 msgcode;

if ( ( msgcode = statasock( &nbytes, buff, sizeof( buff ) )
        != XN_SUCCESS )
        printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.29 statnet()

Report Statistics on a Specific Level 0 Network Interface.

Synopsis:
```
   unsign32 statnet( &nbytes, net_num, buffp, buffsize);
   unsign16 buffsize, nbytes, net_num[2];
   struct   statnet *buffp;
```

Description:
This request will place, in a user-supplied buffer, network interface statistics from the INTERLAN controller interfacing to network <net_num>. This request will fail if the network interface is not an INTERLAN controller.

This request returns 16-bit unsigned binary values and character arrays as follows:
```
 struct statnet {
  unsign16 zero,         /* always zero */
           sixtytwo,     /* always sixty-two */
           phys_add[3],  /* physical address
                            (eg 0702 0001 6c03 hex) */
           nframes,      /* number of frames received */
           rcv_frames,   /* number of frames in receive fifo */
           xmt_frames,   /* number of frames transmitted */
           excess,       /* number of excess collisions */
           coll_frags,   /* number of collision fragments
                            received */
           nlost,        /* number of times 1 or more frames
                            lost */
           rcv_mcast,    /* number of multicast frames accepted */
           lost_mcast,   /* number of multicast frames rejected */
           crc_frames,   /* number of frames received with crc
                            error */
           align_frames,/* number of frames received with
                            alignment error */
           collisions,   /* number of collisions */
           owind_coll,   /* number of out-of-window collisions */
           reserved[8];  /* reserved */
  char     mod_id[6],    /* up to 5 ASCII module ID bytes
                            plus a null terminator byte */
           firm_id[6];   /* up to 5 ASCII firmware ID bytes
                            plus a null terminator byte */
  };
```

These values are obtained from the INTERLAN Ethernet controller board. The controller zeroes the counters after they are read. Refer to the appropriate User Guide (NI1010A/NI2010A) for a complete description of these values.

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes placed in the user buffer is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
#include <snitpusr.h>

extern char     *msgCnvrt();

char      msgbuf[512];
struct    statnet stats;
unsign16 nbytes, net_num[2];
unsign32 msgcode;

if ( ( msgcode = statnet( &nbytes, net_num, &stats,
                         sizeof( stats ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.30 statsconn()

Report Statistics on a Specific SPP Connection.

Synopsis:

```
unsign32 statsconn( &nbytes, a_id, buffp, buffsize );
unsign16 a_id, buffsize, nbytes;
struct   statconn *buffp;
```

Description:

This request returns in a user-supplied buffer, statistics about a specific connection.

<a_id> is the Access-Identifier returned by a previous "Access for Sequenced Packet Protocol" request (sppacc()).

The buffer will contain 32-bit unsigned binary values:

```
struct statconn {
 unsign32 sock_num,     /* 16-bit local socket number */
          rcv_packts,   /* number of packets delivered to the
                           associated socket by the router */
          xmt_packts,   /* number of packets sent from the
                           associated socket */
          conn_id,      /* 16-bit local connection id */
          rem_host[3],  /* remote host number
                           (eg 0207 0100 0781 hex) */
          rem_sock,     /* 16-bit remote socket number */
          rem_conn_id,  /* 16-bit remote connection id */
          rt_delay,     /* round trip delay to remote host
                           (in 10 ms units) */
          rcv_usr,      /* number of received packets taken by
                           user program */
          xmt_acked,    /* number of packets sent and acked */
          rcv_probes,   /* number of probes received */
          xmt_probes,   /* number of probes sent */
          rcv_acks,     /* number of non-probe system packets
                           (acks) received */
          xmt_acks,     /* number of non-probe system packets
                           (acks) sent */
          rcv_dups,     /* number of duplicate packets received */
          rxmt_packs,   /* number of packets retransmitted */
 };
```

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes placed in the user buffer is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:

```
#include <snitpusr.h>

extern char     *msgCnvrt();

char     msgbuf[512];
struct   statconn stat_buff;
unsign16 nbytes;
unsign32 msgcode;

sppacc( &a_id, 0 );
if ( ( msgcode = statasock( &nbytes, a_id, &stat_buff,
                        sizeof( stat_buff ) ) ) != XN_SUCCESS )
       printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.31 statsgen()

Report General Statistics.

Synopsis:

```
   unsign32 statsgen( &nbytes, buffp, buffsize );
   struct   statgen *buffp;
   unsign16 buffsize, nbytes;
```

Description:

This request places general statistics in a user-supplied buffer. Each statistic is returned as a 32-bit long binary value.

The statistics are returned in the following format:

```
 struct statgen {
  char     pname[8];    /* up to seven protocol name ASCII
                           characters immediately followed by a
                           null byte */
  char     version[8];  /* up to seven ITP/VM version ASCII
                           characters immediately followed by a
                           null byte */
  unsign32 host_numb[3],/* local host number
                           (eg 0207 0100 03bd hex) */
           rte_packets, /* number of packets routed between
                           (locally connected) networks */
           rcv_packets, /* number of idp packets received
                           from network drivers */
           xmt_packets, /* number of idp packets given to network
                           drivers for transmission */
           rte_bytes,   /* number of bytes routed between
                           (locally connected) networks */
           rcv_bytes,   /* number of idp packet bytes received
                           from network drivers */
           xmt_bytes,   /* number of idp packet bytes given to
                           network drivers for transmission */
           err_packets, /* number of error packets given to router
                           for transmission */
           lost_packets,/* number of received packets router threw
                           away due to software checksum error or
                           non-existent destination socket */
           ech_packets, /* number of echo responses given by
                           internal echo request listener to
                           router for transmission */
           rip_req,     /* number of rip requests received
                           at Socket 1 */
           rip_res,     /* number of rip responses received
                           at Socket 1 */
           rip_xres,    /* number of rip responses given by
                           routing information listener to a
                           router for transmission */
           spp_xmts,    /* total spp send requests from user
                           programs */
```

```
          spp_rcvs,     /* total spp receive requests from user
                           programs */
          pep_reqs,     /* total pep requester requests from user
                           programs */
          pep_res,      /* total pep responder requests from user
                           pgms and internal statistics server */
          rte_entries,  /* number of routing table entries in
                           use */
          freebytes;    /* number of free bytes remaining
                           (dummy) */
};
```

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes placed in the user buffer <buffp> is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:

```
   #include <snitpusr.h>

   extern char     *msgCnvrt();

   char      msgbuf[512];
   struct    statgen stats;
   unsign16  nbytes;
   unsign32  msgcode;

   if ( ( msgcode = statsgen( &nbytes, &stats, sizeof( stats ) ) )
          != XN_SUCCESS )
          printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

4.6.32 statssock()

Report Statistics on a Specific Socket.

Synopsis:
```
   unsign32 statssock( &nbytes, a_id, buffp, buffsize );
   unsign16 a_id, buffsize, nbytes;
   struct   statsock *buffp;
```

Description:
This request returns, in a user-supplied buffer, statistics about the socket obtained by a previous "Access for Internet Datagram", Access for Sequenced Packet", "Access for Echo Protocol", or Access for Packet Exchange" request.

<a_id> is the Access-ID returned by the previous access request.

The information is returned as three 32-bit unsigned binary values:
```
 struct statsock {
  unsign32 sock_num,    /* the 16-bit socket number */
           rcv_packets, /* the number of packets delivered to the
                           socket by the router */
           xmt_packets; /* the number of packets sent from the
                           socket */
  };
```

Note that if the supplied buffer is not large enough to hold the above statistics, no error will occur.

The number of bytes placed in the user buffer is returned in <nbytes>. If you are not interested in this number, specify a NIL pointer instead of <&nbytes>.

Errors: See Appendix A.

Example:
```
   #include <snitpusr.h>

   extern char     *msgCnvrt();

   char     msgbuf[512];
   struct   statsock stats;
   unsign16 a_id, nbytes;
   unsign32 msgcode;

   sppacc( &a_id, 0 );
   if ( ( msgcode = statssock( &nbytes, &stats, sizeof( stats ) ) )
          != XN_SUCCESS )
          printf( "%s\n", msgCnvrt( msgbuf, msgcode, 0 ) );
```

66

A. Appendix: ITP errors

This Appendix describes all the possible function values returned by ITP. These return values are all defined in the include files "IUCVMSG H" and "XNMSG H" on the SLACNET disk.

XN_SUCCESS	0x0C598001	Normal successful completion
XN_BADHST	0x0C59800A	something wrong with host number
XN_NOSOK	0x0C598012	local socket doesn't exist
XN_SOKFUL	0x0C59801A	no more local sockets available
XN_BADNET	0x0C598022	something wrong with network number
XN_BADSOK	0x0C59802A	something wrong with socket
XN_NETOS	0x0C598032	interface error - OS related
XN_NETHW	0x0C59803A	interface error - hardware related
XN_UXXXX	0x0C598042	error number reserved for future use
XN_ABORT	0x0C59804A	signaled to abort
XN_TIMOUT	0x0C598052	Network timeout occurred
XN_SOKEXI	0x0C59805A	local socket already in
XN_CONOPE	0x0C598062	connection already open
XN_NONET	0x0C59806A	unknown network number
XN_NOTBRO	0x0C598072	dest network does not support broadcast
XN_NOMEM	0x0C59807A	out of dynamic memory
XN_CONNS	0x0C598082	no more ccbs available
XN_NONUM	0x0C59808A	unknown locally connected network number
XN_ETNADR	0x0C598092	Ethernet addresses conflict at init
XN_NORTE	0x0C59809A	out of rte's on initialization
XN_KILL	0x0C5980A2	signalled to die
XN_NOTOPE	0x0C5980AA	connection not open
XN_WINDOW	0x0C5980B2	client's receive window too large
XN_HIMDED	0x0C5980BA	breath-of-life failure
XN_NOISMI	0x0C5980C2	not an Interlan controller
XN_NOPEPR	0x0C5980CA	PEP response not allowed without request
XN_TOOBIG	0x0C5980D2	data (packet) too large
XN_NOAID	0x0C5980DA	no ACBs for external client access
XN_DIFFER	0x0C5980EA	echoed packet is different
XN_XXXX	0x0C5980F2	unspecified error at destination
XN_BADAID	0x0C5980E2	access-ID not in use or bad
XN_CSUM	0x0C5980FA	checksum or pkt fmt error at destination
XN_SOCK	0x0C598102	no such socket at destination
XN_BUFF	0x0C59810A	resource limitation at destination
XN_RXXX	0x0C598112	unspecified error at router
XN_RSUM	0x0C59811A	checksum or pkt fmt error at router
XN_PATH	0x0C598122	no path to network from router
XN_HOPS	0x0C59812A	pkt routed too many times (router loop)
XN_SIZE	0x0C598132	packet too large for router
XN_ERR	0x0C59813A	unknown error packet returned
XN_NOSTAT	0x0C598142	no statistics available

XN_NONCB	0x0C59814A	no more ncb's available
XN_NETNUM	0x0C598152	network number already defined
IUCV_SUCCESS	0x0C5D8001	Normal successful IUCV return
IUCV_IPATHID	0x0C5D800C	Invalid path id
IUCV_QUIESCED	0x0C5D8014	Path quiesced - no sends allowed
IUCV_EX_MSGLIM	0x0C5D801C	Message limit exceeded
IUCV_NPRI_MSG	0x0C5D8024	Priority messages not allowed on this path
IUCV_SHORT_BUF	0x0C5D802C	Receive/answer buffer too short
IUCV_PROT_EXCP	0x0C5D8034	Protection exception on buffer
IUCV_ADDR_EXCP	0x0C5D803C	Addressing exception on buffer
IUCV_I_CLSPATH	0x0C5D8044	MSGID found but MSGCLS or PATHID invalid
IUCV_PURGD_MSG	0x0C5D804C	Message has been purged
IUCV_NEG_MSGLN	0x0C5D8054	Message length is negative
IUCV_LOGOFF_TG	0x0C5D805C	Target communicator not logged on
IUCV_NOBUFF_TG	0x0C5D8064	Target communicator has not declared buffer
IUCV_COMCO_MAX	0x0C5D806C	Maximum number of connections for this communicator exceeded
IUCV_TGTCO_MAX	0x0C5D8074	Maximum number of connections for target exceeded
IUCV_UNAUTHRZD	0x0C5D807C	No authorization found
IUCV_ICPSERVIC	0x0C5D8084	Invalid CP system service name
IUCV_INVCPFUNC	0x0C5D808C	Invalid CP function code
IUCV_MSGLIM255	0x0C5D8094	Message limit greater than 255
IUCV_DECL_BUFF	0x0C5D809C	A buffer has been previously declared
IUCV_ORG_SEVER	0x0C5D80A4	Originator has invoked SEVER
IUCV_PARM_N_AL	0x0C5D80AC	Parm list data not allowed on this path
IUCV_DIR_ERR	0x0C5D818C	CP directory error
IUCV_NO_MSGFND	0x0C5D8194	Specified message not found
IUCV_NMSGBLK	0x0C5D819C	No free MSGBLK available
IUCV_NOSERVICE	0x0C5D81A4	This service was never offered
IUCV_REFUSED	0x0C5D81AC	IUCV connection refused by target
IUCV_CLOSED	0x0C5D81B4	IUCV connection to ITPACP not open
IUCV_AUDIT	0x0C5D81BC	IUCV error recorded in audit word
IUCV_TIMEOUT	0x0C5D81C4	IUCVWAIT timeout

B. Appendix: THE INTERLAN REMOTE STATISTICS SERVER

Your program can obtain ITP statistics from remote INTERLAN ITP nodes.

ITP/VM contains a Remote Statistics Server process. This process (internal to ITP) waits at well-known socket 040b (hex) and responds to Packet Exchange Protocol (PEP) request packets.

To interface to the Remote Statistics Server, your program should issue an Access as PEP Requester (pepareq()) request. After this request completes, your program should issue a Transmit PEP Request (pepxreq()) call. In the transmit call, your program should specify a PEP Client Type of 8002 (hex). This specifies "Remote Statistics Request". The first (and only) word of the PEP Request data should specify the type of remote statistics desired, as shown in Table 1.

First Buffer Word	Meaning
0	Request General Statistics
1	Request All Socket Statistics
2	Request All Connection Statistics
3	Request All Link Statistics

Table 1. Remote Statistics Request

After successful completion of the Transmit PEP Request call, your program should issue a Receive PEP Response (pephreq(), peptreq()) call. This call will return in the specified buffer an IDP packet that contains the PEP Response from the Remote Statistics Server. The PEP Client Type of this packet will be 8003 (hex). This specifies "Remote Statistics Response". The first data word of this packet will be as shown in Table 2.

First Data Word (hex)	Meaning
0100	General Statistics
0101	All Socket Statistics
0102	All Connection Statistics
0103	All Link Statistics

Table 2. Remote Statistics Response

The remaining data will be formatted as described in the appropriate Get All Statistics request (stataconn(), statanet(), statasock(), statsgen()).

67

C. Appendix: PROGRAMMING EXAMPLES FOR ITP/VM

The following two programs are included in the ITP/VM distribution kit.

The two sample programs are written in the C language. They are "source.c" and "sink.c".

You can run these programs to create a Sequenced Packet Protocol (SPP) connection. These programs will run on the same host or on different hosts. The programs illustrate the use of the SNITPUSR TXTLIB subroutines and the manipulation of access-id's.

C.1 SOURCE.C

```
/* SOURCE:
   This program listens for the connection then sends the data */

#include "snitpusr.h"

#define MY_SOCKET 0x28

extern char    *msgCnvrt();

extern int      printf(), sscanf();

extern exit();

main( argc, argv )
     int argc;
     char *argv[];
 {
     BOOLEAN exec();          /* forward declaration of exec */
     char *buff, *malloc();
     static unsign16 aid, bytes, dna[6];

     int nprt, npkt = 100, nskp = 0, bufsz = 512, i;
     unsigned sock_num = MY_SOCKET;

     if ( argc >= 2 ) sscanf( argv[1], "%d", &npkt );
     if ( argc >= 3 ) sscanf( argv[2], "%d", &nskp );
     if ( argc >= 4 ) sscanf( argv[3], "%d", &bufsz );
     buff = malloc( bufsz );
     for ( i = 0; i < bufsz; i++ ) buff[i] = i;
     printf( "Source: no. packets = %d, packet size = %d\n",
                      npkt,             bufsz );

     if ( exec( itpini(), "itpini" ) &&
          exec( sppacc( &aid, sock_num ), "sppacc" ) &&
          exec( sppopen( aid, dna, 6000, FALSE, FALSE, FALSE ),
                "sppopen" ) ) {
          nprt = 1;
          if ( nskp <= 0 ) nprt = 0;
          for ( i = 0 ; i < npkt ; i++ ) {
               if ( exec( sppxmt( &bytes,aid,0,0,0,buff,bufsz ),
                          "sppxmt" ) == FALSE ) break;
               nprt--;
               if ( nprt == 0 ) {
                    printf("Source: buffer no. %d sent.\n", i);
                    nprt = nskp;
               }
          }
          exec( deaccess( aid ), "deaccess" );
     }
     exec( itpclose(), "itpclose" );
     free( buff );
     exit( 0 );
 }
```

```
BOOLEAN exec( funcval, funcname )
     char *funcname;
     unsign32 funcval;
 {
     static char *lastname = 0;

     char buffer[512];

     if ( funcval == XN_SUCCESS ) {
          if ( lastname != funcname )
               printf( "Source: %s successful.\n", funcname );
          lastname = funcname; return( TRUE );
     }
     printf( "Source: %s failed: %s\n",
                  funcname,  msgCnvrt( buffer,funcval,0 ) );
     return( FALSE );
 }
```

C.2 SINK.C

```
/* SINK:
   This program initiates the connection and receives the data */

#include "snitpusr.h"

extern char    *msgCnvrt();

extern int      printf(), sscanf();

extern exit();

#define MY_SOCKET 0x29
#define HIS_SOCKET 0x28
#define ALL 0xffff

main( argc, argv )
     int argc;
     char *argv[];
 {
     BOOLEAN exec();           /* forward declaration of exec */
     int nprt, npkt = 100, nskp = 0, i, bufsz = 512;
     struct rcv_spp rcv_spp;

     static unsign16 aid,
                     dna[6] = { 0,0,ALL,ALL,ALL,HIS_SOCKET };
     char *buff, *malloc();

     if ( argc >= 2 ) sscanf( argv[1], "%d", &npkt );
     if ( argc >= 3 ) sscanf( argv[2], "%d", &nskp );
     if ( argc >= 4 ) sscanf( argv[3], "%d", &bufsz );
     buff = malloc( bufsz );
     printf( "Sink: no. buffers = %d, buffer size = %d\n",
                    npkt,               bufsz );

     if ( exec( itpini(), "itpini" ) &&
          exec( sppacc( &aid, MY_SOCKET ), "sppacc" ) &&
          exec( sppopen( aid, dna, 6000, TRUE, FALSE, FALSE ),
               "sppopen" ) ) {
          nprt = 1;
          if ( nskp <= 0 ) nprt = 0;
          for ( i = 0 ; i < npkt ; i++ ) {
               if ( exec( spphrcv(&rcv_spp,aid,20000,buff,bufsz),
                   "spphrcv" ) == FALSE ) break;
              nprt--;
               if ( nprt == 0 ) {
                    printf("Sink: buffer no. %d received.\n",i);
                    nprt = nskp;
               }
          }
          exec( deaccess( aid ), "deaccess" );
     }
     exec( itpclose(), "itpclose" );
     free( buff );
     exit( 0 );
 }
BOOLEAN exec( funcval, funcname )
     char *funcname;
     unsign32 funcval;
 {
     static char *lastname = 0;

     char buffer[512];

     if ( funcval == XN_SUCCESS ) {
          if ( lastname != funcname )
               printf( "Sink: %s successful.\n", funcname );
          lastname = funcname; return( TRUE );
     }
     printf( "Sink: %s failed: %s\n",
                    funcname, msgCnvrt( buffer,funcval,0 ) );
     return( FALSE );
 }
```

D. Appendix: NETMON - NETWORK MONITORING UTILITY

D.1 OVERVIEW

NETMON helps you to verify that ITP/VM is operating correctly and helps you to understand the functions of Xerox NS ITP-based computer networks. It is a screen-oriented menu driven program designed to run on 327x type terminals.

D.2 MAIN MENU DISPLAY

Entering NETMON<cr> will display the main menu.

```
* * * * * * * * * * *   N E T M O N   * * * * * * * * * * *

Use Tabkey to move cursor to desired display and enter Host
Name or Address Number (#### #### ####), defaults to local
address.

Press ENTER to go into Statistics or PF03 to EXIT.

                                         Host Name/Address

            General ITP Statistics

            ITP Network Statistics

            ITP Socket Statistics

            ITP Connection Statistics
```

D.3 GENERAL ITP STATISTICS DISPLAY

```
                                                 12:51:14
    General ITP Statistics     Host:  0207 0100 27DA

XNS ITP Protocol               V01-003 Version
      0 IDP Packets Routed           0 IDP Bytes Routed
   3903 IDP Packets Received 1785583 IDP Bytes Received
   4055 IDP Packets Trans.   1675324 IDP Bytes Transmitted
      2 Error Packets Sent           0 Packets Disc. by Router
      0 Echo Responses Sent          0 Router Responses Sent
     41 Router Requests Rcvd.        0 Router Responses Rcvd.
   2401 SPP Send Requests         2504 SPP Receive Requests
    192 Packet Exchange Req.        36 Packet Exchange Resp.
      0 Routing Table Entries        0 Free Bytes Remaining


To EXIT: <cr>, <cr>
====>
```

Host: The host number is a twelve hex digit value. The host number is either hardwired into an Ethernet controller board or entered at NETGEN time. Note that host numbers are unique and that there is only one host number per host, independent of the number of Ethernet controllers on that host.

Protocol: Currently ITP/VM supports the Xerox NS ITP Protocol.

Version: This is the current version of ITP/VM.

IDP Packets Routed: Internet datagrams routed between network drivers on the host.

IDP Bytes Routed (Internet): Bytes routed between network drivers on the host.

IDP Packets Received: Internet datagrams given to the router by network drivers on the host.

IDP Bytes Received: Bytes given to the router by network drivers on the host.

IDP Packets Transmitted: Internet datagrams given to network drivers on the host.

IDP Bytes Transmitted: Bytes given to network drivers on the host.

Error Packets Sent: ITP error packets given to the router for transmission. Some possible errors are datagram checksum errors, non-existant destination socket and resource limitation problems.

Packets Discarded By Router: Received datagrams discarded by the internet router. Some possible errors are datagram checksum errors, non-existant destination socket, datagram in an infinite loop (the IDP specification has defined this to be a datagram routed through 15 internet routers), or the packet is too large to be routed to another network driver.

Echo Responses Sent: Echo response packets given to the router by the well known echo socket for transmission. The Echo Protocol is used to verify that a remote host may be reached and that the Internet Datagram Protocol is functioning correctly.

Router Responses Sent: This is the total number of router responses given to the router for transmission either gratuitously or in reponse to an external request.

Router Requests Received: Requests for routing information the router has received.

Router Responses Received: Router response packets the router has received because it requested another host's routing information or it received a gratuitous routing response.

SPP Send Requests: Sequenced Packet Protocol send requests made by user programs.

SPP Receive Requests: Sequenced Packet Protocol receive requests made by user programs.

Packet Exchange Requests: Packet Exchange requests made by the user programs.

Packet Exchange Responses: Packet Exchange responses made by the user programs or the internal remote statistics server.

Routing Table Entries: Networks the host is aware of (either directly connected or accessible through one or more internet routers). The upper bound on the number is selected during NETGEN time.

Free Bytes Remaining: Free bytes available in the dynamic memory region. A certain amount of the dynamic memory region is used by ITP/VM in maintaining internal tables; the remainer is used for data buffering. The upper bound on this value is selected during NETGEN time.

D.4 ITP NETWORK STATISTICS DISPLAY

```
                                                              12:51:48

  ITP Network Statistics      Host Number:  0207 0100 27DA
  Module ID:   NM10A             Firmware Version:  V05.00

   38719 Frames Received
       0 Frames in Receive FIFO
     178 Frames Transmitted
       0 Excess Collisions
      18 Collision Fragments Received
       0 Frames Lost
       6 Multicast Frames Accepted
   38482 Multicast Frames Rejected
       3 Frames Received with CRC Error
       0 Frames Received with Alignment Error
       0 Collisions on Transmit
       0 Out-of-window Collisions

 To EXIT: <cr>, <cr>

 ====>
```

This display shows the statistics maintained by an INTERLAN NI1010 Ethernet Communications Controller. Refer to the NI1010 manual for a detailed description of these values. While ITP/VM statistics are maintained on a cumulative basis, the network driver statistics are reset each time the Network display is called by NETMON. (In ITP/VM frames are equivalent to "packets".)

Host Number: The INTERLAN contoller's physical address; a twelve hex digit value.

Module ID: The identifier of the Interlan Ethernet Protocol Module. (e.g. NM10 or NM10A).

Firmware Version: The firmware version number of the Interlan Ethernet Protocol Module (e.g. V5.00).

Frames Received: The cumulative number of packets (including collision fragments and multicast packets) that were received by the controller.

Frames in Receive FIFO: Number of packets waiting in the controller's received packet buffer.

Frames Transmitted: The number of packets transmitted onto the Ethernet.

Excess Collisions: The cumulative number of times a transmit packet incurred 16 successive collisions when attempting to gain access to the network.

Collision Fragments Received: Number of collision fragments that the controller received.

Frames Lost: Frames lost.

Multicast Frames Accepted: Packets received having a multicast address matching one of the multicast address assigned to the controller.

Multicast Frames Rejected: Packets received having a multicast address not matching any of the multicast addresses assigned to the controller.

Frames Received With CRC Error: Number of packets received having a CRC error.

Frames Received With Alignment Error: Packets received with an alignment error (i.e. the frame size was not an integral multiple of 8 bits).

Collisions on Transmit: The cumulative number of collisions incurred by the controller in attempting to transmit a packet.

Out-of-window Collisions: The number of out-of-window (i.e. beyond the 51.2 uSec slot time) incurred by the controller in attempting to transmit a packet.

D.5 ITP SOCKET STATISTICS DISPLAY

```
                                                          12:51:35

   ITP Socket Statistics              Host:  0207 0100 27DA
Socket Num. Socket Description    Inbound Pkts  Outbound Pkts

     3        Router Error             0              0
   40B        INTERLAN Stat Server    36             36
     2        Echo                     0              0
     1        Routing Information     42              1
     5        Courier Listener        44              0



To EXIT: <cr>, <cr>
====>
```

This display shows the number of inbound and outbound packets through every allocated socket. There are four permanently allocated sockets in ITP/VM:

SOCKET NUMBER	SOCKET DESCRIPTION
1	Routing Information
2	Echo
3	Router Error
40B	INTERLAN Statistics Server

D.6 ITP CONNECTION STATISTICS DISPLAY

```
                                                          12:52:55
                        ITP Connection Statistics

      Remote Host
  Socket  Conn_Id  Pkts_Rcvd  Pkts_Xmtd     Delay User_Rcvd..
Rem_Sock Rem_Conn Probe_Rcvd Probe_Xmtd Acks_Rcvd Acks_Xmtd..
-------------------------------------------------------------

   0207 0100 27DA
     BCA     52D5         13          11      266          7
     BCB     52D4          1           4        1          0



To EXIT: <cr>, <cr>
====>
```

This display provides a complete picture of up to four sequence packet protocol connections.

Remote Host Number: The INTERLAN contoller's physical address; a twelve hex digit value.

Socket: Currently allocated socket for the connection.

Connection ID: Local Connection Identification.

Packets Received: Packets delivered to the associated socket by the router.

Packets Transmitted: Packets sent from the associated socket.

Delay: Round trip delay to remote host (in 10 ms units).

User Received: Received packets taken by user program.

Sent/Ackd: Packets transmitted and acknowledged.

Remote Socket: Socket number of the other end of the SPP connection.

Remote Connection ID: Connection Identification of the other end of the SPP connection.

Probe Received: Number of probes (a request for allocation or acknowledgement) received.

Probe Transmitted: Probes transmitted.

Acks Received: Non-probe system packets (acks) received.

Acks Transmitted: Non-probe system packets (acks) transmitted.

Duplicates Received: Duplicate packets received.

Retransmitted: Packets retransmitted due to delayed acknowledgment by the receiver.

REFERENCES

1. IBM Virtual Machine/System Product *System Programmer's Guide, Release 3* Endicott, New York: August, 1983; edition SC19-6203-2
2. IBM Virtual Machine/System Product *CMS Command and Macro Reference, Release 3* Endicott, New York: September 1983; edition SC19-6209-2
3. IBM Virtual Machine/System Product *CMS User's Guide, Release 3* Endicott, New York: September 1983; edition SC19-6210-2
4. Interlan, Inc. *HOW TO USE THE INTERLAN NS4240 XEROX ITP NETWORK SOFTWARE (ITP/UNIX)* Westford, MA 01886: December, 1983; Documentation Part Number 950-1015-AA
5. Interlan, Inc. *Network Communications Executive Programmer's Manual* Westford, MA 01886: October, 1984; Documentation Part Number 950-1045-00
6. Xerox System Integration Standard *Internet Transport Protocols* Stamford, Connecticut 06904: December, 1981; XSIS 028112.
7. Xerox System Integration Standard *Level 0 Point-to-Point Protocol/Product Specification T33-2.0* Stamford, Connecticut 06904: January, 1982; XSIS 018201.
8. Xerox System Integration Standard *Courier: The Remote Procedure Call Protocol* Stamford, Connecticut 06904: December, 1981; XSIS 038112.
9. Hans Frese *SNET010: The Cometized IUCV Subroutine Package* SLAC: 1985
10. Dave Wiser *SNET013: The Message Facility* SLAC: (in writing)