Operating on the Ethernet

All LSA1000 functions are controlled via Ethernet, the LAN (Local Area Network) software standard. The instrument uses Ethernet's TCP/IP network protocol, accessed by the BSD Sockets API. ***For connecting to PC or network over the Ethernet, see also Chapter 4 of the accompanying Operator's Manual.***

This API (Applications Programming Interface) sits above the TCP/IP protocol stack in all UNIX systems and is also available in Windows 95 and Windows NT. It is for the most part platform-independent, and should allow the same source code to compile and run on each of the supported systems.

WinSock

The commonly known *WinSock* API, derived from the original BSD Sockets API, may also be used to communicate with BSD Sockets-based systems. *WinSock*, for *Windows Sockets,* is used by the leading Internet servers.

TCP (Stream Socket)

Of the two types of connections supported by BSD Sockets, UDP and TCP, the LSA1000 uses TCP — also called *stream socket* — as its underlying protocol. This is a 'reliable' protocol, which ensures that packets are in the correct sequence and that none are missing.

VICP

The Versatile Instrument Control Protocol (VICP) is the LSA1000 protocol for Ethernet operation. The connection established between the controlling device, or *client*, and the LSA1000, or *server*, is made using a known port number. Each of the common Internet protocols uses a predefined port number — FTP, for example, uses 21, and HTTP 80. **The VICP port number is 1861.**

Note: A USB (Universal Serial Bus) port is located on the instrument's rear panel even though USB communication is not supported at this time. It is intended that this communication protocol will be supported in future LeCroy software releases.

The client sends standard ASCII remote commands through the Ethernet socket, just as they would be sent via GPIB, but with an 8-byte header at the start of each transfer. This header contains information about the type of block and its length. Block types include 'Data with/without EOI', and Device Clear, and allow GPIB behavior to be emulated.

Addressing

Every Ethernet device has an IP address designated by four numbers between 0 and 255, separated by periods — for example, 12.34.56.78. Your LSA1000's address is set to 172.25.1.2 at the factory but can be changed using the COMM_NET command.

Standard Messages

The following are IEEE 488.1 standard messages that go beyond mere reconfiguration of the bus and that have an effect on the operation of the instrument. All except GET are executed immediately upon reception — not in chronological order.

- In response to a universal **Device CLear** (DCL) or a Selected Device Clear message (SDC), the LSA1000 clears the input or output buffers, aborts the interpretation of the current command (if any) and clears any pending commands. Status registers and status-enable registers are not cleared. Although DCL has an immediate effect it can take several seconds to execute this command if the instrument is busy.
- The **Group Execute Trigger** message (GET) causes the LSA1000 to arm the trigger system. It is functionally identical to the "*TRG" command.

Programming Ethernet Transfers

Data Transfer Header

The format of the header sent before each data block, both to and from the LSA1000, is set out in the following table:

Byte #	Purpose
0	Operation
1	Header Version
2	Spare (reserved for future expansion)
3	Spare (reserved for future expansion)
4	Block Length, (bytes of data), MSB
5	Block Length (bytes of data)
6	Block Length (bytes of data)
7	Block Length, (bytes of data), LSB

The 'Operation' bits and meanings are:

D7	D6	D5	D4	D3	D2	D1	D0
DATA	REMOTE	LOCKOUT	CLEAR	SRQ	Reserved	Reserved	EOI

Data Bit	Mnemonic	Purpose
D7	DATA	Data block (D0 indicates termination with/without EOI)
D6	REMOTE	Remote Mode
D5	LOCKOUT	Local Lockout (Lockout front-panel)
D4	CLEAR	Device Clear (if sent with data, clear occurs before data block is passed to parser)
D3	SRQ	SRQ (Device to PC only)
D2..D1	Reserved	Reserved for future expansion
D0	EOI	Block terminated in EOI Logic "1" = use $\rightarrow$ EOI terminator Logic "0" = no EOI terminator

It is possible that the LSA1000 and the controlling application will get out of sync with each other. For this, a recovery mechanism has been defined, and the controller at the end of the connection that detects the problem is responsible for closing the socket and re-opening it.

Problem Solving

The TCP 'NAGLE' algorithm: One of the algorithms used in the TCP layer of the TCP/IP stack is the cause of important remote control performance problems. This algorithm has the function of buffering up small packets and sending them only when a 'large' packet has been filled or a time limit of 200 ms has expired. Even a simple query is bound by these limitations.

However, when NAGLE is turned off, this 'round-trip' time is reduced by approximately one hundred. The following function call disables the algorithm when using the standard BSD Sockets API of the 'C' language (equivalent function calls may exist in other environments).

```
const int disable = 1;

if   (0   !=   setsockopt(socket,   IPPROTO_TCP,
TCP_NODELAY, (char*)&disable, sizeof(disable)))

{

    … failed …

}
```

Multiple Client Support: The current design of network remote control allows support of only one client at a time. This applies equally to the operation of the LSA1000. And because of this the number of simultaneous connections that can be made with the instrument has been restricted to one.

This can cause problems if a remote client disconnects or hangs without closing its connection (socket). Unfortunately there is no 'clean' way for the server to know when this has happened. If the LSA1000 seems to be refusing connections then a reboot may be required.

This problem is due to be addressed in a future revision of the protocol.

Problem Solving

C' Language: The following sample 'C' code allows a simple dialog to be established with the LSA1000. The sockets are used in a blocking mode (processing is suspended while a response is awaited). Non-blocking operation is beyond the scope of this manual, but is covered in almost any BSD sockets reference.

```
/*------------------------------------------------------------
------------------

       LeCroy LSA1000 BSD Sockets Remote Control Example


       Overview:
              This example shows how to send a remote query
to a LSA1000
              and read it's response. It should be used as
a model for more
              complex remote control systems.

       Requirements:
              Microsoft Visual C++ 4.x, 5.0 compiler
              Windows 95/NT host

       Version: 1.0, August 14th

       Notes:
              Ensure  that  the  SERVER_ADDRESS  correctly
reflects the address of
              the device under control.

  ----------------------------------------------------------
------------------*/

#include "windows.h"
#include <stdio.h>

#define SERVER_PORT            1861
#define SERVER_ADDRESS         "172.25.1.2"
#define HEADER_LENGTH          8

#define FLAG_EOI               0x80 + 0x01
#define FLAG_NO_EOI            0x80

int socketFd;                  /* client socket handle */

/* function prototypes */
BOOL connectToScope();
void disconnectFromScope();
int readString(char *replyBuf, int userBufferSize);
BOOL  sendString(char  *message,  int  bytesToSend,  BOOL
eoiTermination);

/* main: program entry point */
int main()
```

```
{
    char replyBuf[81];

    connectToScope();
    sendString("*idn?\n", 6, TRUE);
    readString(replyBuf, 80);
    disconnectFromScope();

    printf("Scope's reply: [%s]\n", replyBuf);

    return(0);
}

/* connectToScope: connect to a network device */
BOOL connectToScope()
{
    SOCKADDR_IN       serverAddr;                /* server's
socket address */
    int sockAddrSize = sizeof (SOCKADDR);         /* size of
socket address structures */

    /* one-time initialization of WinSock
       (not required on UNIX platforms) */
       int err;
       WORD wVersionRequested = MAKEWORD(1, 1);
       WSADATA wsaData;

       err = WSAStartup(wVersionRequested, &wsaData);
       if (err != 0)
       {
           printf("ERROR: could not initialize WinSock\n");
           return(FALSE);
       }

    /* build server socket address */
       serverAddr.sin_family = AF_INET;
       serverAddr.sin_port = htons (SERVER_PORT);

       if           ((serverAddr.sin_addr.s_addr           =
inet_addr(SERVER_ADDRESS)) == -1)
       {
           printf("ERROR: Bad server address\n");
           return(FALSE);
       }

    /* create client's socket */
       socketFd = socket(AF_INET, SOCK_STREAM, 0);
       if (socketFd == INVALID_SOCKET)
       {
             printf("ERROR:  socket()  failed,  error  code  =
%d\n", WSAGetLastError());
           return(FALSE);
       }

    /* connect to server (scope) */
```

```
       if ((connect(socketFd, (SOCKADDR FAR *) &serverAddr,
sockAddrSize)) == SOCKET_ERROR)
       {
             printf("ERROR: socket() failed, error code =
%d\n", WSAGetLastError());
           return(FALSE);
       }

    /* success */
       return(TRUE);
}

/* disconnectFromScope: disconnect from a network device */
void disconnectFromScope()
{
    closesocket(socketFd);
}

/* sendString: send a string to the device, with or without
EOI termination */
BOOL  sendString(char  *message,  int  bytesToSend,  BOOL
eoiTermination)
{
    static unsigned char headerBuf[HEADER_LENGTH];
    int bytesSent;

    /* send header */
       if(eoiTermination)
           headerBuf[0] = FLAG_EOI;
       else
           headerBuf[0] = FLAG_NO_EOI;
       headerBuf[1] = 1;                    /*       header
version 1 */
       headerBuf[2] = 0x00;          /* unused */
       headerBuf[3] = 0x00;          /* unused */
       *((unsigned     long     *)     &headerBuf[4])     =
htonl(bytesToSend);   /* message size */

       if     (send(socketFd,     (char     *)     headerBuf,
HEADER_LENGTH, 0) != HEADER_LENGTH)
       {
           printf("ERROR: could not send header\n");
           return(FALSE);
       }

    /* send contents of message */
       bytesSent = send(socketFd, message, bytesToSend, 0);
       if   ((bytesSent   ==   ERROR)   ||   (bytesSent   !=
bytesToSend))
       {
           printf("ERROR: 'send' failed\n");
           return(FALSE);
       }

   return(TRUE);
}
```

```
/* readString: read a string from the device into a user-
supplied buffer */
int readString(char *replyBuf, int userBufferSize)
{
        int blockSize = 0, thisBlockSize, bytesReceived;
        BOOL blockEOITerminated = FALSE;
        unsigned char headerBuf[HEADER_LENGTH];

        /* read the header */
                if(recv(socketFd,    (char    *)    headerBuf,
HEADER_LENGTH, 0) == 8)
                {
                    /* extract the number of bytes contained
in this packet */
                        blockSize = ntohl(*((unsigned long *)
&headerBuf[4]));

                    /* check the integrity of the header */
                         if(!((headerBuf[0]  ==  FLAG_EOI  ||
headerBuf[0] == FLAG_NO_EOI) &&
                                headerBuf[1] == 0x01))
                        {
                            /* error state, cannot recognise
                header since we
                                    are  out  of  sync,  need  to
close & reopen the socket */
                                disconnectFromScope();
                                connectToScope();
                                return(0);
                        }

                    /* inform the caller of the EOI state */
                        if(headerBuf[0] == 0xaa)
                            blockEOITerminated = TRUE;
                }

        /* read the data block */
                thisBlockSize      =      min(userBufferSize,
blockSize);

                bytesReceived   =   recv(socketFd,   replyBuf,
thisBlockSize, 0);

                if(bytesReceived != thisBlockSize)
                    printf("ERROR: truncated read\n");
                else
                    replyBuf[bytesReceived] = '\0';   /*
ensure string termination */

        return(bytesReceived);
}
```

3 The LSA1000 and the Ethernet