Network Programming

Overview

This note introduces remote programming of Cryo-con instruments using the Ethernet LAN remote interface. Examples are given for TCP and UDP communication using the C++ language. If you are using LabView™, please refer to documentation provided in the LabView™ driver package.

Familiarity with network structures, connections, protocols and C++ programming is assumed.

Communication between a host computer (client) and a Cryo-con instrument (server) is done using an ASCII based text language. From the host perspective, the sequence is always to write a command and then read back the response.

For a detailed description of Cryo-con's implementation of the IEEE-488.2 SCPI remote programming language, please refer to Application Brief AB-015. A summary of remote commands is included here.

Remote Programming Tips

1. To view the last command that the instrument received and the last response it generated, press the System key and then select the Network Configuration Menu. The last two lines of this menu show > and < characters. These two lines show the last command received by the instrument and the last response generated.
2. Since the Cryo-con instrument acts as a server, it should use a static IP. DHCP is not recommended.
3. All commands written to the instrument must be terminated by a \n character. Nothing will be parsed until this terminator is received. All \r characters are ignored. Command and response strings are limited to 80 characters.
4. Commands and queries written to the instrument always generate a response. The user should follow every data write with a data read operation. A data read completion indicates that the instrument has completed processing and is ready for more commands. The minimum response is a \n character. If a syntax error is detected, the instrument will respond with a NAK\n sequence.
5. Commands are parsed left to right. If a syntax error is detected, all commands preceding the error are executed.
6. It is often easiest to test commands by using the Cryo-con utility software. Run the program, connect to the instrument and use the Interact function to send commands and view the response. Some communications programs like Windows Hyperterminal can be used to interact with the instrument via the TCP port. Both TCP and UDP commands may be used with the TCPclient.exe or UDPclient.exe Windows program included with this note.
7. Commands written to the instrument are not case-sensitive but the instrument's response is always upper-case. Floating-point values are returned in their full precision. Large values are returned in C language notation. For example, 1.23 times 10 to the -12th power is returned as 1.23e-12.
8. For ease of software development, keywords in all SCPI commands may be shortened. The short form of a keyword is the first four characters of the word, except if the last character is a vowel. If so, the truncated form is the first three characters of the word. Some examples are: inp for input, syst for system alar for alarm etc.

Using Cryo-con's eNetDLL

Cryo-con's eNetDLL program is a Windows based dynamic library that simplifies TCP communications. Programming is reduced to simple connect, disconnect, send and read functions. The default port of 5000 is assumed.

EnetDLL.h

The TCP programming functions provided by eNetDLL are given here.

```
//-----------------------------------------------------------------------------
// DESC: This function first disconnects the current TCP connection if connected.
//       and then, establishes a new connection to the instrument with the IP
//       address string
// PARAMS:     char iIPStr[] - IP address string buffer
// RETURNS:    true - succeeded, otherwise, failed.
// NOTES:      If failed, this function will pop a message box with the Windows
//             error code. The IP address string must be in the format
//             "BYTE1.BYTE2.BYTE3.BYTE4"
//-----------------------------------------------------------------------------
bool ENET_API Connect( char iIPStr[] );

//-----------------------------------------------------------------------------
// DESC: This function disconnects the current connection if connected.
// PARAMS:     None
// RETURNS:    true - succeeded, otherwise, failed.
// NOTES:      None
//-----------------------------------------------------------------------------
bool ENET_API Disconnect();

//-----------------------------------------------------------------------------
// DESC: This function sends a text command/query to the instrument through an
//       established TCP connection.
// PARAMS:     char iStr - input command/query text string buffer
//             int iLen - length of the command/query string to send
// RETURNS:    true - succeeded, otherwise, failed.
// NOTES:      The Connect() function must be called and succeeded before
//             sending.
//-----------------------------------------------------------------------------
bool ENET_API Send( char iStr[], int iLen );

//-----------------------------------------------------------------------------
// DESC: This function reads the response from the instrument after sending a
//       query through an established TCP connection.
// PARAMS:     char oStr[] - output string buffer for the returned response string
//             int iLen - length of the expected response string
// RETURNS:    true - succeeded, otherwise, failed.
// NOTES:      Call this function only after A query has been sent.
//             The response string is truncated to iLen if longer than iLen.
//-----------------------------------------------------------------------------
bool ENET_API Read( char oStr[], int iLen );

//-----------------------------------------------------------------------------
// DESC: This function reads the response from the instrument after sending a
//       query through an established TCP connection.
// PARAMS:     None
// RETURNS:    Pointer to a return string. NULL if failed.
// NOTES:      Call this function only after A query has been sent.
//             The max length of the return string is MAX_PATH.
//-----------------------------------------------------------------------------
ENET_API char* ReadByPtr();
```

TCPIPdrv.h

TCPIPdrv is a class definition for eNetDLL.

```
class TCPIPdrv  {
  private:
    bool TCPopen;

  public:
    TCPIPdrv();
    virtual ~TCPIPdrv();

  //-------------------------------------------------------------------------
  // DESC: This function first disconnects the current TCP connection if
  //       connected and then establishes a new connection to the instrument
  //       with the IP address string
  // PARAMS:     char iIPStr[] - IP address string buffer
  // RETURNS:    true - succeeded, otherwise, failed.
  // NOTES:      If failed, this function will pop a message box with the Windows
  //             error code. The IP address string must be in the format
  //             "BYTE1.BYTE2.BYTE3.BYTE4"
  //-------------------------------------------------------------------------
  bool open( char* );

  //-------------------------------------------------------------------------
  // DESC: This function disconnects the current connection if connected.
  // PARAMS:     None
  // RETURNS:    true - succeeded, otherwise, failed.
  // NOTES:      None
  //-------------------------------------------------------------------------
  bool close();

  //-------------------------------------------------------------------------
  // DESC: This function sends a text command/query to the instrument through an
  //       established TCP connection.
  // PARAMS:     char * - input command /query text string buffer
  //
  // RETURNS:    true - succeeded, otherwise, failed.
  // NOTES:      The Connect() function must be called and succeeded before
  //             sending.
  //-------------------------------------------------------------------------
  bool write(char *);

  //-------------------------------------------------------------------------
  // DESC: This function reads the response from the instrument after sending a
  //       query through an established TCP connection.
  // PARAMS:   char oStr[] - output string buffer for the returned response string
  //             int iLen - length of the expected response string
  // RETURNS:  true - succeeded, otherwise, failed.
  // NOTES:    Call this function only after A query has been sent.
  //             The response string is truncated to iLen if longer than iLen.
  //-------------------------------------------------------------------------
  bool  read( char *, int  );
```

```
  /*-------------------------------------------------------
    Function to send a query and read the response.
       inputs are:
         char *   string to send.
         char *   input buffer
         int      Number of chars in input buffer.
       return is true for success.
  -------------------------------------------------------*/
  bool IO(char *, char *, int);

  /*-------------------------------------------------------
    Function to send a command.
       inputs are:
         char *   string to send.
       return is true for success.
  -------------------------------------------------------*/
  bool TCPIPdrv::IO(char *);

  /*-------------------------------------------------------
    Function to send a query and read the response.
       inputs are:
         char *   string to send.
         char *   input buffer
       return is true for success.
  -------------------------------------------------------*/
  bool TCPIPdrv::IO(char *, char *);

  /*-------------------------------------------------------
    Function to send a command, read the response and
       then compare the response to a desired response.
       inputs are:
         char *   string to send.
         char *   desired response
         int                 number of characters to compare
       return is true for success.
  -------------------------------------------------------*/
  bool IOcmp(char *, char *, int );

  /*-------------------------------------------------------
    Function to send a command, read the response and
       then compare the response to a desired response.
       inputs are:
         char *   string to send.
         char *   desired response
       return is true for success.
  -------------------------------------------------------*/
  bool IOcmp(char *, char * );
};
```

TCP example using eNetDLL and TCPIPdrv

```
// -----  Example TCP program using Cryo-con's EZNET dll and C++  -----
// TCPIP declarations
#include "TCPIPdrv.h"

TCPIPdrv LAN; //Define global LAN object
char IPA[ ] = "192.168.1.5"; //Instrument's IP address on the LAN
char tempstr[257]; //temporary character string

//Open the instrument.
If(!LAN.open(IPA)){
   //can't connect...
   LAN.close();
   throw ("Can't talk to instrument");
};

//read the IDN string
LAN.IO("*IDN?",tempstr,256);
printf("IDN is %s\n",tempstr); //Print IDN

//read the MAC address
LAN.IO("net:mac?",tempstr,256);
printf("MAC is: %s\n",tempstr);

//Start temperature control
LAN.IO("control");

//Stop temperature control
LAN.IO("stop");

//Read channel B input
LAN.IO("input? B",tempstr,256);
printf("Channel B temperature is: %s\n",tempstr);

//send compound command to input channel A.
LAN.IO("INPUT A:UNIT S;SENSOR 33;:*OPC?",tempstr,256);

//close the instrument
LAN.close();
```

User Datagram Protocol (UDP)

UDP is a simple connection-less protocol to use when communicating with Cryo-con instruments. The user binds a UDP socket and communicates with the instrument in a fashion similar to the older RS-232 style communications.

Before you can bind a UDP socket, you will need to know the instrument's IP address and port number. Both the IP and port number may be obtained from the front panel of the instrument or by using the discovery function of Cryo-con's utility software.

> ⓘ **Note:** The UDP port number is always the TCP port number plus one.

```
/*****************************************************************************
Write console commands and read a response using UDP.
Uses winsock2 for Windows to implement a Berkeley socket which  should be
compatible with most operating systems.
*****************************************************************************/
#defineWIN32_LEAN_AND_MEAN
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <conio.h>
#include <winbase.h>
#include <malloc.h>
int sockfd;
#define MAX_BUFF_SIZE 1000
void main(void){
   //IP address of the Cryo-con instrument. Default is 192.168.1.5
   const char serverip[] = "192.168.1.5";

   //UDP port number is TCP port plus 1. Default is 5001
   unsigned short port = 5001;

   //Initialize UDP socket
   struct sockaddr_in server,local;
   int retval,fromlen;
   char *Buffer;
   WSADATA wsaData;
   fromlen = sizeof(local);
   memset(&server,0,sizeof(server));
   Buffer = (char *)malloc(MAX_BUFF_SIZE);
   server.sin_family = AF_INET;
   server.sin_addr.S_un.S_addr = inet_addr(serverip);
   server.sin_port = htons(port);
   if (WSAStartup(0x202,&wsaData) == SOCKET_ERROR){
      printf("WSAStartup failed with error %d\n",WSAGetLastError());
      WSACleanup();
      exit(1);
   }

   //Establish a UDP socket
   sockfd    = socket(AF_INET,SOCK_DGRAM,0);
   retval =0;

   //loop reading the console, writting data and reading the response.
   printf("Type x to exit\n");
   while(1){
      //get operator input
      printf("\n:");
      gets(Buffer);
      if(Buffer[0] == 'x')break;//Exit on an x character
      strcat(Buffer,"\r\n");//cat required terminator
```

```
      //Send command
      retval = sendto(sockfd, Buffer, strlen(Buffer), 0,
             (struct sockaddr *)&server, sizeof(struct sockaddr));

      if (retval <0){
        fprintf(stderr,"sendto() failed: error %d\n",WSAGetLastError());
        WSACleanup();
        closesocket(sockfd);
        exit(1);
      }

      //Read response
      fromlen = sizeof(struct sockaddr);
      retval = recvfrom(sockfd, Buffer, MAX_BUFF_SIZE, 0,
             (struct sockaddr *)&local, &fromlen);

      if (retval <0){
        fprintf(stderr,"recvfrom() failed: error %d\n",WSAGetLastError());
        WSACleanup();
        closesocket(sockfd);
        exit(1);
      }

      //null terminate response string
      Buffer[retval]=0;
      printf("%d>%s",retval,Buffer);

    };//end while 1
    closesocket(sockfd);
}
```

Using the Terminal Control Protocol (TCP)

TCP is a connection orientated protocol that is more complex than UDP. The user must bind a TCP socket and negotiate a connection before communicating with an instrument.

> ⓘ **Note:** Cryo-con instruments support up to five simultaneous TCP connections. If a connection is inactive for five minutes, it is automatically closed.

Before you can bind a TCP socket, you will need to know the instrument's IP address and port number. Both the IP and port number may be obtained from the front panel of the instrument or by using the discovery function of Cryo-con's utility software.

```
/*****************************************************************************
Write console commands and read a response using TCP.
Uses winsock2 for Windows to implement a Berkeley socket which should be
compatible with most operating systems.
*****************************************************************************
/
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>
#define MAX_BUF_SIZE 4000
char Buffer[MAX_BUF_SIZE];

void main(void){
   //IP address of the Cryo-con instrument. Default is 192.168.1.5
   const char serverip[] = "192.168.1.5";

   //TCP port number. Default is 5000
   unsigned short port = 5000;

   //Initialize TCP socket
   int socket_type = SOCK_STREAM;
   struct sockaddr_in server;
   int ret,retval=0,temp =0;
   unsigned int rd = 0;
   WSADATA wsaData;
   SOCKET  conn_socket;
   memset(&server,0,sizeof(server));
   server.sin_addr.s_addr=inet_addr(serverip);
   server.sin_family = AF_INET;
   server.sin_port = htons(port);
   if (WSAStartup(0x202,&wsaData) == SOCKET_ERROR){
     printf("\nWSAStartup failed with error %d",WSAGetLastError());
     WSACleanup();
     exit(1);
   }

   //Establish a TCP socket
   conn_socket = socket(AF_INET,socket_type,0);
   if (conn_socket <0 ){
     printf("\nClient: Error Opening socket: Error %d\n",WSAGetLastError());
     WSACleanup();
     exit(1);
   }
```

```
   //Establish a TCP connection
   if (connect(conn_socket,(struct sockaddr*)&server,
        sizeof(server))== SOCKET_ERROR) {
      printf("\nConnect() failed: %d",WSAGetLastError());
      WSACleanup();
      exit(1);
   }

   //loop reading the console, writting data and reading the response.
   printf("Type x to exit\n");
   while(1){
      //get operator input
      printf("\n:");
      gets(Buffer);
      if(Buffer[0] == 'x')break;//Exit on an x character
      strcat(Buffer,"\r\n");//cat required terminator

      //Send command
      retval = send(conn_socket,Buffer,strlen(Buffer),0);
      if (retval == SOCKET_ERROR){
         fprintf(stderr,"send() failed: error %d\n",WSAGetLastError());
         WSACleanup();
         exit(1);
      }

      //Read response
      retval = recv(conn_socket,Buffer,MAX_BUF_SIZE,0);
      if (retval == SOCKET_ERROR){
         fprintf(stderr,"send() failed: error %d\n",WSAGetLastError());
         WSACleanup();
         exit(1);
      }

      //null terminate response string
      Buffer[retval]=0;

      printf("%d>%s",retval,Buffer);
   };//end while 1
   closesocket(conn_socket);
}//end main
```

Remote Command Reference

Although the remote programming language is common to all Cryo-con instruments, inappropriate commands are not implemented on a given instrument. Futher, instruments may have different selection parameters.

Control Loop Start /Stop commands
STOP CONTrol CONTrol?

SYSTEM commands
SYSTem:LOCKout {ON | OFF} SYSTem:NVSave SYSTem:REMLed {ON | OFF} SYSTem:BEEP <seconds> SYSTem:DISTc {0.5 | 1 | 2 | 4 | 8 | 16 | 32 | 64} SYSTem:ADRes <address> SYSTem:RESeed SYSTem:HOMe SYSTem:SYNCtaps <taps> SYSTEM:NAME "name" SYSTem:HWRev? SYSTem:FWREV? SYSTem:LINefreq **{**60 **|** 50**}** SYSTem:DRES {FULL | 1 | 2 | 3} SYSTem:PUControl {ON | OFF} SYSTem:BAUD {9600 | 19200 | 38400 | 57200} SYSTem:DATe "mm/dd/yyyy" SYSTem:TIMe "hh:mm:ss"

Configuration Commands
CONFig <ix>:NAMe "name" CONFig <ix>:SAVe CONFig <ix>:RESTore

Input Commands

INPut? {A | B | C | D} or INPut {A | B | C | D}:TEMPerature?
INPut {A | B | C | D}:UNITs {K | C | F | S}
INPut {A | B | C | D}:NAMe Instrument Name"
INPut {A | B | C | D}:SENPr?
INPut {A | B | C | D}:VBIas {100MV | 10MV | 1.0MV}
INPut {A | B | C | D}:BRANge {Auto | 2 | 20 | 200 | 2.0K | 20K}
INPut {A | B | C | D}:SENsorix <ix>
INPut {A | B | C | D}:BRUNlock?
INPut {A | B | C | D}:POWer?
INPut {A | B | C | D}:ALARm?
INPut {A | B | C | D}:ALARm:HIGHest <setpt>
INPut {A | B | C | D}:ALARm:LOWEst <setpt>
INPut {A | B | C | D}:ALARm:DEAdband <setpt>
INPut {A | B | C | D}:ALARm:HIENa { YES | NO }
INPut {A | B | C | D}:ALARm:LOENa { YES | NO }
INPut {A | B | C | D}:Clear
INPut {A | B | C | D}:LTEna { YES | NO }
INPut {A | B | C | D}:AUDio { YES | NO }
INPut {A | B | C | D}:MINimum?
INPut {A | B | C | D}:MAXimum?
INPut {A | B | C | D}:VARiance?
INPut {A | B | C | D}:SLOpe?
INPut {A | B | C | D}:OFFSet?
INPut:STAts:TIMe?
INPut:STAts:RESet

Loop Commands

LOOP {1 | 2 | 3 | 4}:SOURce {A | B | C | D}
LOOP {1 | 2 | 3 | 4}:SETPt <setpt>
LOOP {1 | 2 | 3 | 4}:
    TYPe { OFF | PID | MAN | TABLE | RAMPP | RAMPT}
LOOP {1 | 2 | 3 | 4}:TABelix <ix>
LOOP {1 | 2 | 3 | 4}:RANGe { HI | MID | LOW}
LOOP {1 | 2 | 3 | 4}:RAMP?
LOOP {1 | 2 | 3 | 4}:RATe <rate>
LOOP {1 | 2 | 3 | 4}:PGAin <gain>
LOOP {1 | 2 | 3 | 4}:IGAin <gain>
LOOP {1 | 2 | 3 | 4}:DGAin <gain>
LOOP {1 | 2 | 3 | 4}:PMAnual <pman>
LOOP {1 | 2 | 3 | 4}:OUTPwr?
LOOP {1 | 2 | 3 | 4}:HTRRead?
LOOP {1 | 2 | 3 | 4}:HTRHst?
LOOP {1}:LOAD {50 | 25}
LOOP {1 | 2 | 3 | 4}:MAXPwr <maxpwr>
LOOP {1 | 2 | 3 | 4}:MAXSet <maxset>

Autotune Commands

LOOP {1 | 2 | 3 | 4}:AUTotune:STARt
LOOP {1 | 2 | 3 | 4}:AUTotune:EXIT
LOOP {1 | 2 | 3 | 4}:AUTotune:SAVE
LOOP {1 | 2 | 3 | 4}:AUTotune:MODe {P | PI | PID}
LOOP {1 | 2 | 3 | 4}:AUTotune:DELTap <num>
LOOP {1 | 2 | 3 | 4}:AUTotune:TIMeout <num>
LOOP {1 | 2 | 3 | 4}:AUTotune:PGAin?
LOOP {1 | 2 | 3 | 4}:AUTotune:PGAin?
LOOP {1 | 2 | 3 | 4}:AUTotune:DGAin?
LOOP {1 | 2 | 3 | 4}:AUTotune:STATus?

OVERTEMP commands

OVERtemp:ENABle {ON | OFF}
OVERtemp:SOURce {A | B | C | D}
OVERtemp:TEMPerature <temp>

Sensor Calibration Curve Commands

CALcur
SENSor <index>:NAMe name string"
SENSor <index>:NENTry?
SENSor <index>:UNITs {VOLTS | OHMS | LOGOHM}
SENSor <index>:
    TYPe { DIODE | ACR | PTC100 | PTC1K |
    PTC10K | PTC10K | NTC10UA }
SENSor <index>:MULTiply <multiplier>

Relay Commands

RELays? {0 | 1}
RELays {0 | 1} :SOURce {A | B | C | D}
RELays {0 | 1} :MODe {auto | control | on | off}
RELays {0 | 1} :HIGHest <setpt>
RELays {0 | 1} :LOWEST <setpt>
RELays {0 | 1} :DEADband <deadband>
RELays {0 | 1} :HIENa { YES | NO }
RELays {0 | 1} :LOENa { YES | NO }

PID Table Commands

PIDTable? <num>
PIDTable <num>:NAMe Name String"
PIDTable <num>:NENTry?
PIDTable <num>
PIDTable <num>:TABLe

Network Commands

NETWork:IPADdress
NETWork:MACaddress

Mail Commands

MAIL {A | B | C | D} :ADDR "IPA"
MAIL {A | B | C | D}:FROM "from e-mail address"
MAIL {A | B | C | D}:DEST to e-mail address"
MAIL {A | B | C | D}:PORT <port number>
MAIL {A | B | C | D}:STATE {ON | OFF}

IEEE Common Commands

*CLS
*ESE
*ESR
*OPC
*IDN?
*RST
*SRE
*STB

Data-logging Commands

DLOG:STATe {ON | OFF}
DLOG:INTerval <Seconds>
DLOG:COUNt?
DLOG?
DLOG:READ?

DLOG:RESEt
DLOG:CLEAr

Cryogenic Control Systems, Inc.
PO Box 7012
Rancho Santa Fe, CA 92067-7012

Telephone: 858 756 3900
FAX: 858 759 3515

CCTechSupport@cryocon.com
www.cryocon.com

Updated: 2/12/11