
USING APOGEE LABS NETWORK NATIVE ASCII TRANSFER COMMUNICATIONS

AP30

-Andy Grebe

ABSTRACT

The purpose of this document is to provide information to assist the software engineer who is responsible for developing applications which communicate with Apogee Labs, Inc. Network Native ASCII Transfer (NNAT). NNAT facilitates communication with Apogee Labs products using native ASCII commands. The user is afforded the opportunity to control the target unit using setup commands, read the status and, where appropriate, read processed information. Since the communication interface is a network connection, this information exchange may take place from any location, whether it's in the same room, the same building, or on the other side of the globe.

KEYWORDS

NNAT - Network Native ASCII Transfer

TCP/IP

SOFTWARE

The NNAT uses TCP/IP packets to receive commands from and send replies to the user application. The software needed to communicate with NNAT is a simple TCP/IP client. This client may be written in a variety of programming languages and reside on an equally diverse list of operating systems. The examples provided in this paper are written in C++ for Linux, UNIX and MS Windows.

Software written in C or C++ uses the socket library, which is included in Linux, UNIX, and Windows compilers.

Linux/UNIX software uses:

```
#include <sys/types.h>
#include <sys/socket.h>
```

Windows will use

```
#include <winsock.h>
```

Software written for Windows needs the WSASStartup to start socket operations and WSACleanup to close the socket operations. Other than these two function calls, the client programs are identical.

```
WORD wVersionRequested;
WSADATA wsaData;
int err;
```

```
wVersionRequested = MAKEWORD( 2, 2 );
err = WSASStartup( wVersionRequested, &wsaData );
```

First, create a socket for a connection. This is done by calling the socket function:

```
int clientFd;
clientFd = socket(AF_INET, SOCK_STREAM, 0);
```

The first argument lets your system know that you wish to create a socket that will communicate with the internet using IPV4. `SOCK_STREAM` tells the system to use TCP/IP packets. Since only one protocol is available for NNAT setup, the protocol is set to 0.

Now that the socket has been created, it needs to be named.

```
struct sockaddr_in address;
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("192.168.0.10");
address.sin_port = htons(1501);
```

The function `inet_addr` converts the character string into binary data in network byte order. The function `htons`, converts the port to network byte order. After the `sockaddr_in` structure has been filled, connect with the server by using the `connect` function.

```
int len = sizeof(address);
result = connect(clientFd, (struct sockaddr *)&address, len);
```

At this point, packets may be sent to and received from the server using `send` and `recv`.

```
const short PACKET_SIZE = 1000;
char nnatPacket[PACKET_SIZE];

send(clientFd, nnatPacket, PACKET_SIZE, 0);
recv(clientFd, nnatPacket, PACKET_SIZE, 0);
```

Both `send` and `recv` use the `clientFd` to send and receive data. The `send` function takes `PACKET_SIZE` bytes from `nnatPacket` and places them into a TCP/IP packet. The `recv` instruction places the data from the incoming TCP/IP packet into `nnatPacket`, filling up to `PACKET_SIZE` bytes.

PROGRAMMING HINTS

Each of the Network Native ASCII Commands ends with either an ack, '>', or a nack, '?'. By using this, software can be written to send and receive multiple commands which will stop when either a command file is finished or a nack is received. The second software example uses this method to output a file of commands for the 4800 recorder.

For a clean disconnect from the NNAT server, send an "exit", after which the server will disconnect the client.

EXAMPLES

The first example is a command line-type client. The user writes a native ASCII command and the client software sends this to the NNAT server. Once the NNAT server responds to the command, the client software outputs the response to the screen.

```
// client1.cpp
//
// Written for a Linux or UNIX O.S.
//
// 6/13/03 Andy Grebe
```

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

using std::cin;
using std::cout;
using std::endl;

const unsigned short PACKET_SIZE = 1000;

// usage
void usage() {
    printf("Usage: client -i <IP Address>\n\n");
}

int main(int argc, char **argv) {
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char buffer[PACKET_SIZE];
    char addr[20];

    if (argc != 3) {
        usage();
        return 0;
    }
    strcpy(addr, argv[2]);

    // Create a socket for the client.

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // Name the socket, as agreed with the server.

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(addr);
    address.sin_port = htons(1501);
    len = sizeof(address);

    // Now connect our socket to the server's socket.

    result = connect(sockfd, (struct sockaddr *)&address, len);
```

```
if(result == -1) {
    perror("oops");
    return 1;
}

// Give the user a nice little prompt
printf(">");
// We can now read/write via sockfd.

while (strcmp(buffer, "exit")) {
    // In case someone is writing a script-file
    // and the file does not end with exit
    if (cin.eof()) {
        strcpy(buffer, "exit");
    }
    else {
        cin.getline(buffer, PACKET_SIZE);
    }
    send(sockfd, buffer, PACKET_SIZE, 0);
    recv(sockfd, buffer, PACKET_SIZE, 0);
    cout << buffer;
}
cout << endl;
close(sockfd);
return 0;
}
```

This second example, written for Windows, opens a file with commands, and sends these commands until the file is finished or an error response is detected.

```
// client2.cpp
//
// Written for Windows
//
// 6/13/03 Andy Grebe

#include <winsock.h>
#include <iostream>
#include <fstream>
#include <sys/types.h>
#include <stdio.h>

using std::cin;
using std::cout;
using std::cerr;
using std::endl;
using std::ifstream;

const unsigned short PACKET_SIZE = 1000;
// usage void usage() {
```

```
    printf
("Usage: client -i <IP Address> -f <script file>\n\n");
}

int main(int argc, char **argv) {
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char buffer[PACKET_SIZE], bufferCopy[PACKET_SIZE];
    char addr[20];
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    ifstream is;

    wVersionRequested = MAKEWORD( 2, 2 );

    err = WSStartup( wVersionRequested, &wsaData );
    if (err != 0) {
        cout << "Could not find proper Winsock DLL" << endl;
        return 0;
    }

    if (argc != 5) {
        usage();
        return 0;
    }
    strcpy(addr, argv[2]);

    is.open(argv[4]);
    if (is.fail()) {
        cerr << "Could not open " << argv[4] << endl;
        return 0;
    }

    // Create a socket for the client.

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // Name the socket, as agreed with the server.

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(addr);
    address.sin_port = htons(1500);
    len = sizeof(address);

    // Now connect our socket to the server's socket.
```

```

result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
    perror("oops");
    return 1;
}

// Give the user a nice little prompt
printf(">");
// We can now read/write via sockfd
// After an "exit" has been written to the NNAT server, exit the program
while (strcmp(bufferCopy, "exit")) {
    // In case someone is writing a script-file
    // and the file does not end with exit
    if (is.eof()) {
        strcpy(buffer, "exit");
    }
    else {
        is.getline(buffer, PACKET_SIZE);
    }
    cout << buffer << endl;
    strcpy(bufferCopy, buffer);
    send(sockfd, buffer, PACKET_SIZE, 0);

    recv(sockfd, buffer, PACKET_SIZE, 0);
    if (buffer[strlen(buffer) - 1] != '>' && strcmp(bufferCopy, "exit")) {
        cout << "Error at command: " << bufferCopy << ", " << buffer << endl;
        WSACleanup();
        return 0;
    }
    cout << buffer;
}
cout << endl;
is.close();
WSACleanup();
return 0;
}

```

This command file, which may be used with the 4800 recorder, sets up the name of the file to record to, the notes for the file, and starts the recording process.

```

control recorder
set record filename=NNATD
set record notes=NNATD-controlled recording test
set record enable=on
set operation=record
exit

```