Encryption Enabling Your Serial Device

09/23/2003

Overview

With the technological advances of recent years, it is now possible to interact with serial devices from a remote location. By using low cost hardware, like the Lantronix CoBox Family of Device Servers, you can quickly convert the RS-232, RS-422, or RS-485 serial port into an Ethernet interface. This interface can be accessed by any IP-based application over an IP network from any place in the world. The serial data is simply encapsulated into TCP or UDP packets, which can travel through any IP based network.

Getting application access to the serial port over a network is called serial tunneling. The serial data is simply encapsulated into TCP or UDP packets, which can travel through any IP based network. The hardware performing this function is called a Device Server.

Since Device Servers are inherently network aware, the ability to add functionality like web services, e-mail, network diagnostics are easily within reach. In this paper we'll cover the detailed requirements of data encryption, as it pertains to serial tunneling with the Lantronix CoBox family.

Encryption Overview

Often Device Servers are connected through the Internet, which exposes the serial device data to security risks. Data encryption is a means of data translation into another format, or alternate language which only the two end-nodes understand. Lantronix Device Servers support either Rijndael (AES 128) Encryption, or Two Fish Encryption (from Counterpane).

In the simplest connection scheme of two Lantronix Device Servers set up as a serial tunnel, no encryption application programming is required since both Device Servers can perform the encryption automatically. However, in the case where a host-based application is interacting with the serial device through it’s own network connection, modification of the application is required to support data encryption. The application code and supporting library functionality in this paper support both Two Fish and Rijndael Encryption.

Sample Functions

Communication with the Device Server can be accomplished by using UDP (User Datagram Protocol) or TCP (Transmission Control Protocol). This paper will document and present application code which can be easily moved into your application by showing examples of both UDP and TCP protocols. It is assumed the application development environment is to be running on a Microsoft Windows based PC, and a C development environment is installed. It is also assumed the reader understands how to use the development tools.

First we need to get the serial device connected to the network. For this task you'll need a Device Server such as the Lantronix UDS-100 and the Lantronix Device Installer application to ease the configuration requirements. Follow the documentation to configure the network and serial parameters of the device server. Connect the serial port of the Device Server to your serial equipment using the appropriate cable and connectors, and connect the Device Server to the Ethernet network using an appropriate cable. Refer to figure 1. Please configure the Device Server to factory defaults, configure Interface Mode, Speed, and Flow Control as required. Data encryption should be disabled.

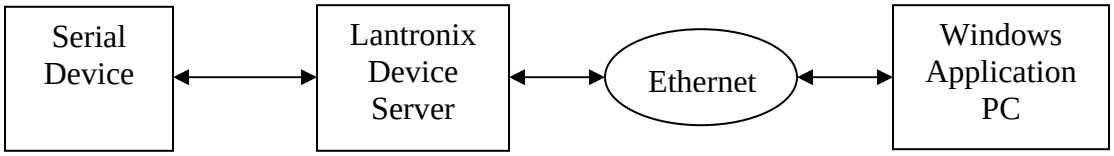


Figure 1

For test purposes only, you can connect your equipment as shown in figure 2.

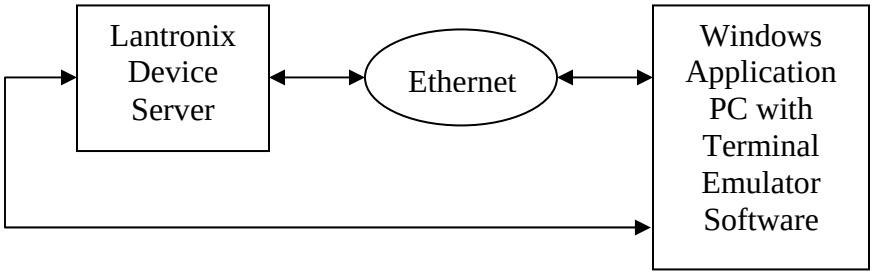


Figure 2

From a software standpoint, we need a program, which can:

1. Open a socket
2. Send and receive TCP or UDP packets
3. Interact with the user
4. Perform encryption and decryption of the data
5. Close and shutdown the socket

Step 1: Open the socket

To perform any network socket function in Windows, you must first initialize the Window's Socket API.

```
WSADATA info;
WSAStartup(MAKEWORD(1,1), &info);
```

Then you can open the UDP socket.

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
```

or, the TCP socket

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
```

Step 2: Send and receive TCP or UDP packets

Writing to a socket, requires that you pass the writing routine the IP address and protocol port number of the destination. This will be the Device Server's IP Address and the port number is 10001 (by default). This information is passed via the sockaddr structure (peer). (On a connected TCP socket, as an alternative, send() instead of sendto() may be used without the sockaddr structure. For simplicity of documentation, this document will only use the sendto() function, since it will work for both TCP and UDP.)

```
sendto(sock, (char *) sbuffer, LenBytesToSend, 0, (struct sockaddr *) peer,
sizeof (struct sockaddr));
```

When receiving data, the function will also retrieve the senders information in the sockaddr structure (from). (On a connected TCP socket, as an alternative, recv() instead of recvfrom() may be used without the sockaddr structure. For simplicity of documentation, this document will only use the recvfrom() function, since it will work for both TCP and UDP.)

```
recvfrom(sock, buf, toget, 0, (struct sockaddr *) from, &fromlen))
```

Step 3: Interact with the user

Interaction with the user is simply printf statements and getch() to gather keyboard data. Of course, your application may be generating data without keyboard interaction.

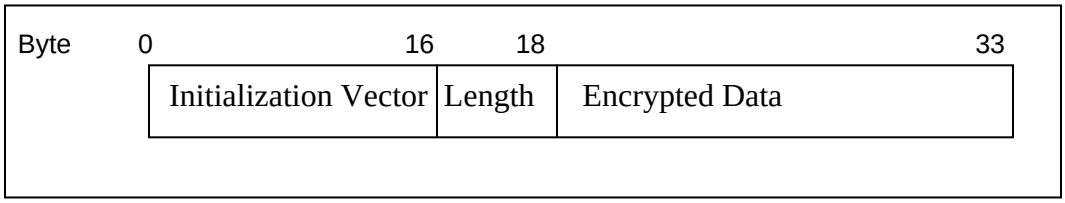
```
for (i = 0; i < len; i++) {
      printf("%c", plaintext[i]);
}

while (_kbhit() != 0) {
      plaintext[in++] = _getch();
}
```

Step 4: Perform encryption and decryption of the data

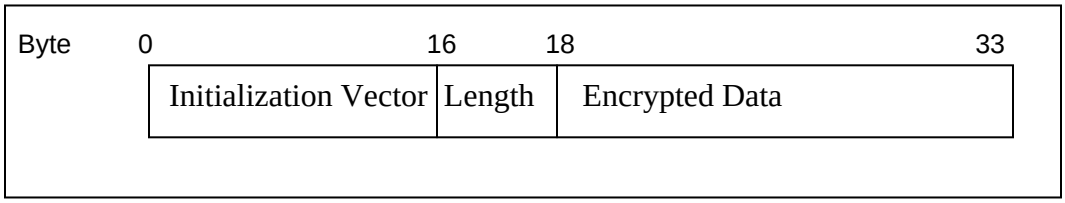
The encryption and decryption modes used depend upon the socket type. In order for the network peers to be in-sync, the Initialization Vector needs to be the same so both peers start at the same point. Since TCP is a connection-oriented protocol, the Initialization Vector only needs to be sent once at connection startup time. However, UDP does not have acknowledgements, so the Initialization Vector must be sent in every packet, in each direction.

Lantronix employs Cipher Block Chaining (CBC) mode for encrypting data on UDP sockets. In block mode, the UDP packet payload must contain the Initialization Vector, the data length (big endian format), and then the encrypted data. Since this is a block method, the data is padded into a number of 16 byte blocks.

Packet 1:



Packet 2:



An example of a UDP CBC encryption routine might look like this:

```
/* Seed random number generator */
```

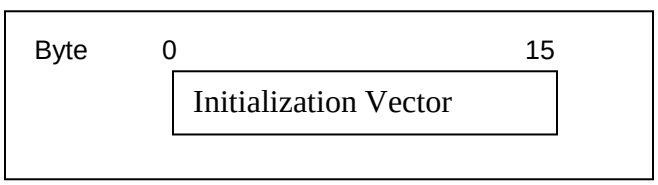
```
srand( (unsigned)time(NULL));
        /* Fill Initialization Vector with random data and copy
                it into the UDP payload */
for (i = 0; i < 16; i++)
   ive[i] = ciphertext[i] = rand();
        /* Insert the data length into the payload */
ciphertext[16] = len >> 8;
ciphertext[17] = len & 0xff;
        /* Copy the plaintext into the payload because the
             encoder will perform the transformation in place */
memcpy(&ciphertext[18], plaintext, len);
        /* Round up the length to a multiple of 16 (bytes per block) */
newlen = ((len + 15) / 16) * 16;
        /* Pad blocks with random data */
for (i = len; i < newlen; i++)
     ciphertext[i+18] = rand();
        /* CBC encrypt */
VC_blockEncrypt(ive, shared_key, newlen, &ciphertext[18], method);
        /* Return the payload length */
     return(newlen + 18);
```

Decryption for UDP is the reverse process:

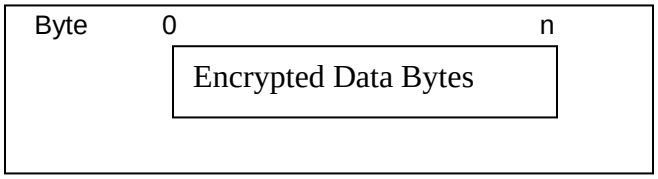
```
        /* Extract the payload Initialization Vector */
memcpy(ivd, ciphertext, 16);
        /* Extract the payload real data length */
len = (ciphertext[16] << 8) + ciphertext[17];
        /* Calculate the payload block data length */
blen = ((len+ 15) / 16) * 16;
        /* Extract the ciphertext into the destination array because the
             decoder will perform the transformation in place */
memcpy(plaintext, &ciphertext[18], blen);
        /* Decrypt it by negating the encrypted data length */
VC_blockEncrypt(ivd, shared_key, -blen, plaintext, method);
        /* Return the length of the real decrypted data */
return(len);
```

Lantronix employs Cipher Feedback 128 bit (CFB128) mode for encrypting data on TCP sockets. In this mode, the first TCP packet payload sent must contain the Initialization Vector. This packet is sent only by the active peer calling connect() on the socket. Since this is a connection-oriented protocol, we only need to send encrypted data bytes in successive packets. Also note that Lantronix employs CFB128 bit mode, which implies the encryption function is only called once every 128 encrypted bits (or 16 bytes).

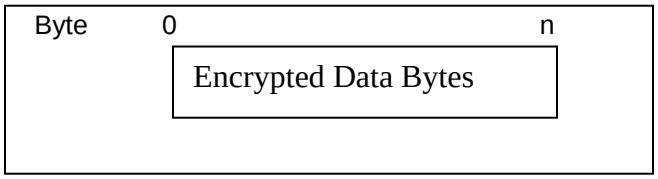
Packet 1



Packet 2



Packet 3



Example TCP code follows:

First you must initialize the state machines, the Initialization Vectors, and initialize the network peer.

```
pstate = pstatd = 0;
    /* Initialize the encode and decode Initialization Vectors */
srand( (unsigned)time(NULL));
for (i = 0; i < 16; i++)
    ive[i] = ivd[i] = rand();
    /* Send IV */
sendto (sock, ive, sizeof(ive), 0, (struct sockaddr_in *) &peer,
                                                    sizeof (struct sockaddr));
```

An example of a TCP CFB128 encryption routine might look like this:

```
    for (i = 0; i < len; i++) {
            if (pstate == 0) {
            blockEncrypt(ive, shared_key, 0, ive, (method - 1));
            pstate = 16;
        }
        ive[16 - pstate] ^= plaintext[i];
            ciphertext[i] = ive[16 - pstate];
            pstate--;
    }
```

Decryption for TCP is the reverse process:

```
    for (i = 0; i < len; i++) {
    if (pstatd == 0) {
        blockEncrypt(ivd, shared_key, 0, ivd, (method - 1));
        pstatd = 16;
    }
    plaintext[i] = ivd[16 - pstatd] ^ ciphertext[i];
        ivd[16 - pstatd] = ciphertext[i];
        pstatd--;
}
```

```
return(len);
```

Step 5: Close and shutdown the down socket

Shutting down is an important step in gracefully closing the socket.

```
shutdown(sock, 2);
closesocket(sock);
```

Finally release the Window's Sockets API resources.

```
WSACleanup();
```

The Lantronix Encryption DLL (cbx_enc.dll) contains one API call but can be called from Visual Basic, C or Java.

```
void __stdcall    VB_blockEncrypt(char *iv, char *key, int bytes, char *text, char method)
void __cdecl      VC_blockEncrypt(char *iv, char *key, int bytes, char *text, char method)
void __cdecl      VJ_blockEncrypt(char *iv, char *key, int bytes, char *text, char method)
```

where:

iv: is a pointer to the 16 byte initialization vector.
key: points to 32 characters depicting a 16 byte hex key.
For example:

```
hex_key[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
```

would be defined as:

```
key[32] = { '0','0','0','1','0','2','0','3','0','4','0','5','0','6','0','7','0','8',
'0','9','0','A','0','B','0','C','0','D','0','E','0','F' };
```

bytes: is the number of data bytes (including padding) to encrypt using CBC.
If bytes is negative, then decrypt according to CBC (block mode).
If bytes is zero, then encrypt according to ECB mode which is used by CFB128 mode.
text: is the array of bytes to encrypt or decrypt. The results are returned in text.
method: 1 for Rijndael Encryption, 0 for Two Fish

Now we need to put all this information into a program, which can handle TCP & UDP, CBC encryption & decryption, CFB128 encryption & decryption, and Rijndael or Two Fish encryption methods. The sample application follows.

```
Usage: IP_Address_or_hostname_of_peer portnumber TCP|UDP [0=None 1=Two Fish
2=Rijndael]
```

After compilation, you should be able to successfully transfer data to and from your serial device with either TCP or UDP (remember to configure the Device Server accordingly) with and without encryption.

Note: the encryption key in this program is:

0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F

the Device Server must be configured with the same key, entered as:

00-01-02-03-04-05-06-07-08-09-0A-0B-0C-0D-0E-0F

NOTE:
Contact Lantronix Customer Support (1-800-422-7044) for firmware, which supports encryption if you do not have it already. CoBox Mini rev 2, Micro, UDS-10, UDS-100, and Din Rail products support Two Fish, while the XPort SE supports Rijndael. Customer support can also provide the cbx_enc.dll library required for use with this program.

```
/*********************************************************************
 *
 *  The intention of this program is to show by example one way to
 *  create a socket and transfer data to and from a peer device on the
 *  network.  This program should work for both TCP (SOCK_STREAM) and UDP
 *  (SOCK_DGRAM) sockets.  This program will also work with Two Fish and
 *  Rijndael encryption.
 *  This sample will:
 *          1. open the socket
 *          2. connect to the peer (for TCP) if the IP address is valid
 *               or listen for incoming connection if the IP address is 0.0.0.0
 *          3. transfer data to and from the console.
 *  It writes via printf(), and reads via getch().
 *
 *  SYNTAX: usage: ip_address_of_peer port_number_of_peer socket_type encryption_method
 *
 *  For use with WIN32 systems.
 *  Author: Garry Morris garrym@lantronix.com
 *********************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <sys/types.h>
#include <errno.h>
#include <time.h>
#include <winsock2.h>
#include <ws2tcpip.h>

typedef SOCKET LTX_SOCKET;
#define SOCKBUFSIZE 256

LTX_SOCKET open_socket(char *hostname, int port, int type, struct sockaddr_in *peer);
void ltx_closesocket(LTX_SOCKET sock);
int main_loop(LTX_SOCKET sock, struct sockaddr_in *from, int encryption);
int read_socket(LTX_SOCKET sock, char *rbuf, int len, struct sockaddr_in *from);
int write_socket(LTX_SOCKET sock, char *sbuf, int len, struct sockaddr_in *to);

int tcp_encrypt(char *plaintext, int len, char *ciphertext, int method);
int udp_encrypt(char *plaintext, int len, char *ciphertext, int method);
int tcp_decrypt(char *ciphertext, int len, char *plaintext, int method);
int udp_decrypt(char *ciphertext, int len, char *plaintext, int method);

int  serial_read (char *plaintext, int len);
void serial_write(char *plaintext, int len);
void trace(char *txt);

unsigned char key[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f }; /* Encryption key */
int (*blockEncrypt)();              /* Pointer to VC_blockEncrypt() function */
int (*ltx_encrypt)();               /* Pointer to encryption method routine () */
int (*ltx_decrypt)();               /* Pointer to decryption method routine () */
char ive[16], ivd[16];              /* Pointer to initialization vectors */
```

```
char pstate, pstatd;                /* Encryption and decryption state variable */
char shared_key[33];                /* 2 byte version of key */

/*****************************************************************************
 *
 *  FUNCTION: Perform IO operation to/from socket & serial interface
 *          int main(int argc, char **argv)
 *  PARAMETERS:
 *          argc: command line argument count
 *          argv: array of pointers to command line arguments
 *  RETURN VALUE:
 *          Return less than 0 upon error, 0 for normal termination.
 *
 *****************************************************************************/
int main(int argc, char **argv)
{
     LTX_SOCKET sock, lsock;        /* Socket ids */
     int i, j, type;                /* temp variables */
     struct sockaddr_in peer;       /* Network peer socket information */
     WSADATA  info;                 /* WinSock info handle */
     HINSTANCE hDLL;                /* cbx_enc.dll handle id */
     int encryption;                /* Encryption method (None, TwoFish or Rijndael) */

     if (argc < 4) {
          fprintf(stderr, "Usage: %s host_ip port TCP|UDP [0=No Encryption | 1=TwoFish |
2=Rijndael]\n", argv[0]);
          fprintf(stderr, "\t host_ip: IP address of peer \n\t\t(for TCP listen, set to 0.0.0.0)\n");
          fprintf(stderr, "\t port: UDP or TCP port number of peer \n");
          fprintf(stderr, "\t\t(for TCP or UDP server, this is the local listen port)\n");
          fprintf(stderr, "\t TCP|UDP: TCP for TCP transport, UDP for UDP transport\n");
          fprintf(stderr, "\t optional encryption argument: \n\t\t0=No Encryption, 1=TwoFish,
2=Rijndael (default=0)\n");
          return(0);
     }

                    /*  start up winsock */
     if ((i = WSAStartup(MAKEWORD(1,1), &info)) < 0 ) {
          fprintf(stderr, "Error on WSAStartup(), error (%d)\n", i);
          return(i);
     }
                    /* Determine the socket type / protocol */
     if ((argv[3][0] == 'T') || (argv[3][0] == 't')) {
          type = SOCK_STREAM;
     }
     else {
          type = SOCK_DGRAM;
     }
          /* Open the socket */
          /* Hostname or ip address of peer */
          /* Port number of UDP or TCP listener on the peer */
          /* Type: SOCK_STREAM or SOCK_DGRAM */
          /* Peer: pointer to sockaddr_in structure */
     if ((sock = open_socket(argv[1], atoi(argv[2]), type, &peer)) <= 0) {
          WSACleanup();
          return(-1);
     }
```

```
	hDLL = 0;
	encryption = 0;
		/* Parse encryption method - default is none */
	if (argc > 4)
		encryption = atoi(argv[4]);
		/* Initialize all the encryption requirements */
	if (encryption) {
			/* Load the DLL */
		hDLL = LoadLibrary("cbx_enc");
		if (hDLL != NULL) {
				/* Find the VC_blockEncrypt() function */
			blockEncrypt = (int (*)()) GetProcAddress(hDLL, "VC_blockEncrypt");
			if (!blockEncrypt) {
				FreeLibrary(hDLL);
				return(-1);
			}
		}
			/* Convert HEX key to 2 byte hex sequence */
		for (i = 0, j = 0; i < sizeof(key); i++, j+= 2) {
			sprintf(&shared_key[j], "%.2X", key[i]);
		}

		if (type == SOCK_STREAM) {
				/* Initialize the encode and decode states */
			pstate = pstatd = 0;
			if (peer.sin_addr.S_un.S_addr != 0) { /* active connect */
					/* Initialize the encode and decode Initialization Vectors */
				srand( (unsigned)time(NULL));
				for (i = 0; i < 16; i++)
					ive[i] = ivd[i] = rand();
					/* Send IV */
				if ((write_socket(sock, ive, sizeof(ive), (struct sockaddr_in *) &peer)) !=
sizeof(ive)) {
					trace("Writing IV failed\n");
				}
			}
				/* Set the encrypt and decrypt TCP routines */
			ltx_encrypt = tcp_encrypt;
			ltx_decrypt = tcp_decrypt;
		}
		else {
				/* Set the encrypt and decrypt UDP routines */
			ltx_encrypt = udp_encrypt;
			ltx_decrypt = udp_decrypt;
		}
	}

	if ( (peer.sin_addr.S_un.S_addr == 0) && (type == SOCK_STREAM) ){ /* passive listen */
		do {	/* wait for connect */
			i = sizeof(peer);
			if ((lsock = accept(sock, (struct sockaddr *) &peer, &i)) < 0) {
				trace("accept failed\n");
				peer.sin_addr.S_un.S_addr = 0;
			}
			else {
```

```
                    if (encryption) {
                            /* Read the IV from the peer */
                        if ((read_socket(lsock, ivd, 16, &peer)) != 16) {
                            trace("Reading IV failed\n");
                        }
                        memcpy(ive, ivd, 16);
                    }
                        /* Main loop dance */
                    while ((main_loop(lsock, &peer, encryption)) > 0);
                    ltx_closesocket(lsock);
                }
            } while (peer.sin_addr.S_un.S_addr != 0);
        }
        else {
                /* Main loop dance */
            while ((main_loop(sock, &peer, encryption)) > 0);
        }
            /* Shutdown and close the socket */
        ltx_closesocket(sock);
            /* Clean up the WinSock resources */
        WSACleanup();
            /* Free the DLL */
        if (encryption && hDLL)
            FreeLibrary(hDLL);
            /* Done */
        return(0);
}

/*****************************************************************************
 *
 *  FUNCTION: Perform IO operation to/from socket & serial interface
 *          int main_loop(LTX_SOCKET sock, struct sockaddr_in *from, int encryption)
 *  PARAMETERS:
 *          sock: socket id
 *          from: pointer to sockaddr_in structure to hold network peer information
 *          encryption: encryption method
 *  RETURN VALUE:
 *          Return less than 0 upon error.
 *
 *****************************************************************************/
int
main_loop(LTX_SOCKET sock, struct sockaddr_in *from, int encryption)
{
        int timeout, in, out, toget;
        fd_set rfds, efds;
        struct timeval timev;
        char *buf;
        char plaintext[SOCKBUFSIZE];        /* Plain text */
        char ciphertext[SOCKBUFSIZE+34];/* Encrypted text. Round up to hold blocks(16), IV(16)
and length(2) */

        if (encryption) {
            buf = ciphertext;
            toget = sizeof(ciphertext);
        }
        else {
```

```
        buf = plaintext;
        toget = sizeof(plaintext);
    }

            /* main loop */
    while (1) {
        FD_ZERO(&rfds);
        FD_ZERO(&efds);
        FD_SET(sock, &rfds);
        FD_SET(sock, &efds);
                /* Set timeout to 100mS.  This timeout will affect serial polling frequency */
        timev.tv_sec = 0;
        timev.tv_usec = 100000;
        timeout = select(FD_SETSIZE, &rfds, (fd_set *) NULL, &efds, &timev);
        if (timeout == -1) {
                /* OS interrupt during select - should be ignored */
            // return(timeout);
        }
        else if (timeout == 0) {
                /* Real timeout - do nothing */
            // return(0);
        }
        else if (FD_ISSET(sock, &efds)) {
                /* Bad news, exception on socket - needs to be closed and reopened */
            return(-2);
        }
        else if (FD_ISSET(sock, &rfds)) {
                /* Received socket data input, decrypt it if required, and write it to serial output */
            if ((in = read_socket(sock, buf, toget, from)) > 0) {
                if (encryption)
                    in = ltx_decrypt(ciphertext, in, plaintext, encryption);
                if (in)
                    serial_write(plaintext, in);
            }
            else if (in < 0) {
                    /* Error reading from socket */
                return(-3);
            }
            else
                return(0);
        }
            /* Check for serial input, encrypt it if required, and send it */
        if ((in = serial_read(plaintext, sizeof(plaintext))) > 0) {
            if (encryption)
                in = ltx_encrypt(plaintext, in, ciphertext, encryption);
            if ((out =  write_socket(sock, buf, in, (struct sockaddr_in *) from)) < in) {
                    /* Socket write error */
                return(-4);
            }
        }
    }
    return(0);
}

/*****************************************************************************
 *
```

```
 *  FUNCTION: Open a socket, and connect to peer if TCP
 *          SOCKET open_socket(char *hostname, int port, int type, struct sockaddr_in *peer)
 *  PARAMETERS:
 *          hostname: pointer to string containing network peer's IP address or hostname
 *          port: port number of network peer to communicate with
 *          type: SOCK_STREAM (tcp) or SOCK_DGRAM (udp)
 *          peer: pointer to sockaddr_in structure to hold network peer information
 *  RETURN VALUE:
 *          Socket id or 0 upon error.
 *
 ******************************************************************************/
LTX_SOCKET
open_socket(char *hostname, int port, int type, struct sockaddr_in *peer)
{
     LTX_SOCKET              sock;
     struct hostent          *hdata;
     unsigned long           ipaddr;

                    /*  create the socket */
     if ((sock = socket(AF_INET, type, 0)) < 0) {
          trace("Error creating socket\n");
          return(0);
     }
                    /* Find the IP address and convert to 4 byte int */
     if (isdigit(hostname[0])) { /* dot decimal notation? */
          if ((ipaddr = inet_addr(hostname)) == INADDR_NONE ) {
               /* inet_addr failed */
               trace("inet_addr failed\n");
               ltx_closesocket(sock);
               return(0);
          }
               /* We have a valid address. */
          memcpy (&peer->sin_addr, &ipaddr, 4);
     }
     else {
          if ((hdata = gethostbyname (hostname)) == (struct hostent *) NULL) {
               /* Can't resolve hostname */
               trace("Can't resolve hostname\n");
               ltx_closesocket(sock);
               return(0);
          }
        memcpy (&peer->sin_addr, *(hdata->h_addr_list), hdata->h_length);
     }
                    /* Complete initialization of peer data structure */
     peer->sin_family = AF_INET;
     peer->sin_port = htons((unsigned short) port);

     if (peer->sin_addr.S_un.S_addr == 0) { /* Wait for incoming */
                    /* bind struct the port */
          if ((bind(sock, (struct sockaddr *) peer, sizeof(struct sockaddr_in))) < 0) {
               trace("Bind error\n");
               closesocket(sock);
               return(-1);
          }
     }
```

```
    if (type == SOCK_DGRAM) /* (UDP) we're done */
        return(sock);

                /* Must be TCP */
    if (peer->sin_addr.S_un.S_addr != 0) { /* active connect */
                /* Let's connect to the peer */
        if ((connect(sock, (struct sockaddr *) peer, sizeof(struct sockaddr_in))) != 0) {
            /* Connection failed */
            trace("Connection failed\n");
            shutdown(sock, 2);
            ltx_closesocket(sock);
            return(0);
        }
                /* Return socket id */
        return(sock);
    }
                /* Must be passive listen */
                /* Setup the listen queue */
    if ((listen(sock, 5)) != 0) {
        trace("listen failure");
        shutdown(sock, 2);
        closesocket(sock);
        return(-1);
    }

    return(sock);
}

/*****************************************************************************
 *
 *  FUNCTION: Close a socket
 *        SOCKET ltx_closesocket(LTX_SOCKET sock)
 *  PARAMETERS:
 *        sock: socket id
 *  RETURN VALUE:
 *        None.
 *
 *****************************************************************************/
void
ltx_closesocket(LTX_SOCKET sock)
{
        /* Disable further reads and writes */
    shutdown(sock, 2);
        /* Close it */
    closesocket(sock);
}

/*****************************************************************************
 *
 *  FUNCTION: Read from a socket
 *        int read_socket(LTX_SOCKET sock, char *rbuf, int len, struct sockaddr_in *to)
 *  PARAMETERS:
 *        sock: socket id
 *        rbuf: pointer to data buffer
 *        len: length of data buffer (max bytes to read)
 *        to: pointer to sockaddr_in structure that holds the peer information
```

```
 *  RETURN VALUE:
 *          Number of bytes read
 *
 *  WARNING: this is a blocking function
 *
 ****************************************************************************/
int
read_socket(LTX_SOCKET sock, char *rbuf, int len, struct sockaddr_in *from)
{
      int in, fromlen;

      fromlen = sizeof(struct sockaddr_in);
      if ((in = recvfrom(sock, rbuf, len, 0, (struct sockaddr *) from, &fromlen)) < 0) {
                trace("recvfrom failed\n");
      }
      return(in);
}

/****************************************************************************
 *
 *  FUNCTION: Write to a socket
 *          int write_socket(LTX_SOCKET sock, char *sbuf, int len, struct sockaddr_in *to)
 *  PARAMETERS:
 *          sock: socket id
 *          sbuf: pointer to data payload
 *          len: length of data payload
 *          to: pointer to sockaddr_in structure that holds the peer information
 *  RETURN VALUE:
 *          Number of bytes sent
 *
 ****************************************************************************/
int
write_socket(LTX_SOCKET sock, char *sbuf, int len, struct sockaddr_in *to)
{
      int n, sent, tosend;

      if (to->sin_addr.S_un.S_addr == 0)
            return(0);
      sent = 0;
      tosend = len;
                /* Loop until all data is sent, or error condition */
      while(tosend){
            n = sendto(sock, (char *) &sbuf[sent], tosend, 0, (struct sockaddr *) to, sizeof (struct
sockaddr));
            if (n <= 0) {
                  trace("sendto failed\n");
                  return(sent);
            }
            else {
                  sent += n;
                  tosend -= n;
            }
      }
      return(sent);
}
```

```
/*****************************************************************************
 *
 *  FUNCTION: Encrypt data using CBC
 *          int udp_encrypt(char *plaintext, int len, char *ciphertext, int method)
 *  PARAMETERS:
 *          plaintext: pointer to source array containing data to be encrypted
 *          len: length of source data in bytes
 *          ciphertext: pointer to destination array, sink for encrypted data
 *          method: encryption method (1 = Two Fish, 2 = Rijndael)
 *  RETURN VALUE:
 *          Length of the UDP payload (IV[16], WORD length, encrypted data)
 *
 *****************************************************************************/
int udp_encrypt(char *plaintext, int len, char *ciphertext, int method)
{
      int i, newlen;

                /* Seed random number generator */
      srand( (unsigned)time(NULL));
                /* Fill Initialization Vector with random data and copy
                        it into the UDP payload */
      for (i = 0; i < 16; i++)
         ive[i] = ciphertext[i] = rand();
                /* Add the data length to the payload */
      ciphertext[16] = len >> 8;
      ciphertext[17] = len & 0xff;
                /* Copy the plaintext into the payload because the
                     encoder will perform the transformation in place */
      memcpy(&ciphertext[18], plaintext, len);
                /* Round up the length to a multiple of 16 (bytes per block) */
      newlen = ((len + 15) / 16) * 16;
                /* Pad blocks with random data */
      for (i = len; i < newlen; i++)
           ciphertext[i+18] = rand();
                /* CBC encrypt */
      blockEncrypt(ive, shared_key, newlen, &ciphertext[18], (method - 1));
                /* Return the payload length */
      return(newlen + 18);
}

/*****************************************************************************
 *
 *  FUNCTION: Decrypt CBC data
 *          int udp_decrypt(char *ciphertext, int len, char *plaintext, int method)
 *  PARAMETERS:
 *          ciphertext: pointer to source array containing encrypted data
 *          len: length of source data in bytes
 *          plaintext: pointer to destination array, sink for decrypted data
 *          method: encryption method (1 = Two Fish, 2 = Rijndael)
 *  RETURN VALUE:
 *          Length of the decrypted data
 *
 *****************************************************************************/
int udp_decrypt(char *ciphertext, int len, char *plaintext, int method)
{
      int dlen;
```

```
        /* Do we have the Initialization Vector and length? */
    if (len < 18) return (0);
        /* Extract the payload Initialization Vector */
    memcpy(ivd, ciphertext, 16);
        /* Extract the payload data length */
    dlen = (((ciphertext[16] << 8) + ciphertext[17] + 15) / 16) * 16;
        /* Do we have a complete payload? */
    if (len < (dlen + 18)) return (0);
        /* Extract the payload real data length */
    len = (ciphertext[16] << 8) + ciphertext[17];
        /* Extract the ciphertext into the destination array because the
            decoder will perform the transformation in place */
    memcpy(plaintext, &ciphertext[18], dlen);
        /* Decrypt it by negating the encrypted data length */
    blockEncrypt(ivd, shared_key, -dlen, plaintext, (method - 1));
        /* Return the length of the real decrypted data */
    return(len);
}


/*****************************************************************************
 *
 *  FUNCTION: Encrypt data using CFB128
 *      int tcp_encrypt(char *plaintext, int len, char *ciphertext, int method)
 *  PARAMETERS:
 *      plaintext: pointer to source array containing data to be encrypted
 *      len: length of source data in bytes
 *      ciphertext: pointer to destination array, sink for encrypted data
 *      method: encryption method (1 = Two Fish, 2 = Rijndael)
 *  RETURN VALUE:
 *      Length of the TCP payload (encrypted data)
 *
 *****************************************************************************/
int tcp_encrypt(char *plaintext, int len, char *ciphertext, int method)
{
        int i;

        for (i = 0; i < len; i++) {
            if (pstate == 0) {
                blockEncrypt(ive, shared_key, 0, ive, (method - 1));
                pstate = 16;
            }
            ive[16 - pstate] ^= plaintext[i];
        ciphertext[i] = ive[16 - pstate];
        pstate--;
        }
        return(len);
}

/*****************************************************************************
 *
 *  FUNCTION: Decrypt CFB128 data
 *      int tcp_decrypt(char *ciphertext, int len, char *plaintext, int method)
 *  PARAMETERS:
 *      ciphertext: pointer to source array containing encrypted data
```

```
 *          len: length of source data in bytes
 *          plaintext: pointer to destination array, sink for decrypted data
 *          method: encryption method (1 = Two Fish, 2 = Rijndael)
 *  RETURN VALUE:
 *          Length of the decrypted data
 *
 ******************************************************************************/
int tcp_decrypt(char *ciphertext, int len, char *plaintext, int method)
{
          int i;

          for (i = 0; i < len; i++) {
               if (pstatd == 0) {
                    blockEncrypt(ivd, shared_key, 0, ivd, (method - 1));
                    pstatd = 16;
               }
               plaintext[i] = ivd[16 - pstatd] ^ ciphertext[i];
          ivd[16 - pstatd] = ciphertext[i];
          pstatd--;
          }
          return(len);
}

/******************************************************************************
 *
 *  --- Modify this routine to write data to your serial device ---
 *
 *  FUNCTION: Send data to serial device
 *          void serial_write(char *plaintext, int len)
 *  PARAMETERS:
 *          plaintext: pointer to data array, to send to serial device
 *          len: length of data in bytes
 *  RETURN VALUE:
 *          None.
 *
 ******************************************************************************/
void serial_write(char *plaintext, int len)
{
          int i;

          for (i = 0; i < len; i++) {
               printf("%c", plaintext[i]);
          }
}

/******************************************************************************
 *
 *  --- Modify this routine to read data from your serial device ---
 *
 *  FUNCTION: Receive data from serial device
 *          void serial_read(char *plaintext, int len)
 *  PARAMETERS:
 *          plaintext: pointer to destination data array to sink serial data
 *          len: length of array in bytes
 *  RETURN VALUE:
 *          Number of bytes recieved from serial device.
```

```
 *
 ****************************************************************************/
int serial_read(char *plaintext, int len)
{
      int in;

      in = 0;
      while ( (_kbhit() != 0) && (in < len) ) {
            plaintext[in++] = _getch();
      }

      return(in);
}

/* Error prints */
void trace(char *txt)
{
      fprintf(stderr, "%s", txt);
}
```

To modify the program to work with your serial device, modify the serial_read() and serial_write() functions.

Summary

With the help of this paper, you can now implement encrypted and non-encrypted data streams to your serial device. This paper also explains the fundamental requirements of encryption implementation on both TCP and UDP transports, as well as read and writing to network sockets.