LANTRONIX®

Encryption DLL User Guide

Part Number 900-372
Revision C June 2005

Copyright & Trademark

Ethernet is a trademark of XEROX Corporation. UNIX is a registered trademark of The Open Group. Windows 95, Windows 98, Windows 2000, and Windows NT are trademarks of Microsoft Corp. Netscape is a trademark of Netscape Communications Corporation.

Contacts

Lantronix Corporate Headquarters
15353 Barranca Parkway
Irvine, CA 92618, USA
Phone: 949-453-3990
Fax: 949-453-3995

Technical Support
Phone: 800-422-7044 or 949-453-7198
Fax: 949-450-7226
Online: www.lantronix.com/support

Sales Offices
For a current list of our domestic and international sales offices, go to the Lantronix web site at www.lantronix.com/about/contact.

Disclaimer & Revisions

The information in this guide may change without notice. The manufacturer assumes no responsibility for any errors that may appear in this guide.

Date	Rev.	Comments
	A	Initial Document.
3/30/05	B	Updated and added more content.
6/15/05	C	Removed out of date references.

Contents

1: Using This Guide

Purpose and Audience

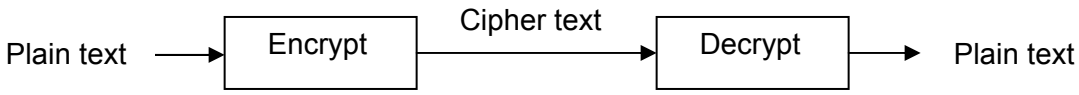
This guide is intended for the engineers responsible for incorporating DLLs into their software. It provides an overview of encryption and networking and the function calls contained within the 2.0 version of the Lantronix Encryption DLL. In addition, it provides several examples to support Rijndael AES 128,192 and 256 bit encryption and decryption methods, two-Fish encryption and decryption methods, Cipher Block Chaining (CBC) mode, and Cipher Feedback 128 bit (CFB128) mode, and networking.

Version 2.0 of the Encryption DLL now has the CBC and CFB code built in. The examples in this User Guide, which use the original VC_blockEncrypt in algorithms to perform CBC and CFB, are now accomplished by calling VC_Init, VC_Encrypt, VC_Decrypt and VC_Cleanup. The VC_blockEncrypt function is still available in the 2.0 Encryption DLL.

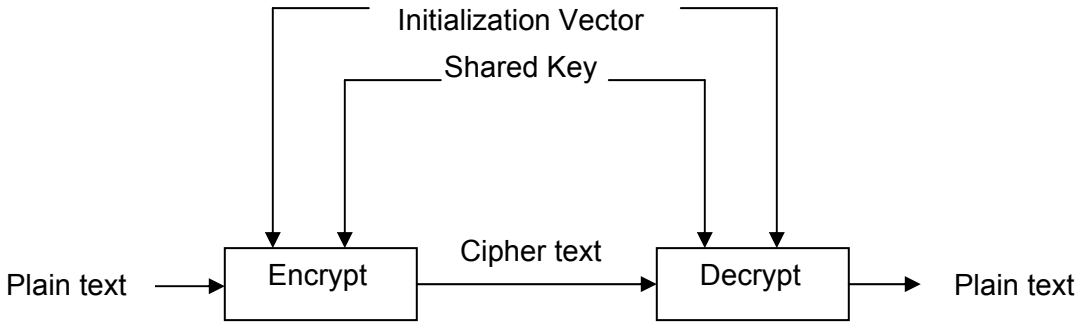
Also include in the 2.0 Encryption DLL are functions to open “encryption” sockets for UDP and TCP to communicate over the network. The sockets can be used in either Client or Server mode. In this way, a user can write applications that communicate over the wire to a Device Server like the SDS1100, SDS2100 or WiPort WPA which support AES encryption of 128, 192 and 256 bits.

2: Encryption Overview

Data encryption is a process of data translation into another format or into an alternate language that only a matching process understands. In its simplest form, it is possible to encrypt data and then decrypt it to yield the original data.



In order to perform the encryption and decryption, two additional pieces of information are required along with the plain text. The first piece of information is the encryption key. The key may come from an independent source or it can be a shared secret between the encryption and decryption processes. The second piece of information is the Initialization Vector (IV). The IV is a series of bits representing the starting place in the encryption and decryption process.

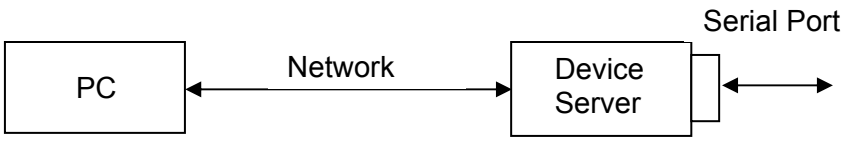


The encryption process combines these three pieces of data to an output known as ciphertext. As such, plain text goes in and ciphertext comes out. The decryption process does the exact opposite: ciphertext goes in and plain text comes out.

When using the Cipher Block Chaining (CBC) mode, initialize the IV for each call to the encryption function. When using Cipher Feedback (CFB128) mode, initialize the IV once as all successive calls rely on previously encrypted data.

3: Communicating with Device Servers

Data can be sent and received between a PC and a Device Server via a network. Data sent to the Device Server over a network is received on the network side of the Device Server and then forwarded out the serial port side of the Device Server. Data received from the Device Server's serial port can be sent over the network to a PC. Communication with a Device Server can be accomplished either through TCP or UDP, depending on the configuration of the Device Server.



The purpose of the encryption DLL is to provide a developer with the capability of sending and receiving encrypted data between a Device Server over a network. Only Device Servers, such as the Lantronix SDS1100, SDS2100, or WiPort WPA, which have the capability to process encrypted data, can participate in this communication. These Device Servers can send and receive encrypted data using AES encryption at 128, 192, and 256 bits.

When a Device Server communicates over the network, it can accomplish this either as a client or a server. If the Device Server is configured as a client, then the PC would initiate the connection. If the Device Server is configured as a server then the Device Server would initiate the connection.

A program running on a PC would perform different operations depending on the mode of the program. Will it be a server or a client? Here is a brief and simple call order that a PC would typically perform and the configuration a Device Server would need, if the PC will be a TCP client, send and receive data, then close the connection.

Device Server:

Port No:	10001	
Connect Mode:	C0	*(Wait for TCP connection. (UDP would be CC))*

PC:

VC_encOpenSocket	*(using TYPE_CLIENT)*
VC_encSendData	
VC_encRecvData	
VC_encShutDown	
VC_encCloseSocket	

Here is a brief and simple call order that a PC would typically perform and the configuration a Device Server would need, if the PC is will be a TCP server, send and

receive data, then close the connection. *(Notice the addition of the VC_encListen and VC_encAccept function calls.)*

Device Server:

Port No:	10001	
Connect Mode:	C5	*(Initiate TCP connection with "169.1.1.1:10001")*
Remote Port No:	10001	
Remote IP Address:	"169.1.1.1"	

PC (IP Address of "169.1.1.1"):

VC_encOpenSocket	*(using TYPE_SERVER)*
VC_encListen	*(Wait for connection)*
VC_encAccept	*(returns socket when Device Server connects)*
VC_encSendData	
VC_encRecvData	
VC_encShutDown	
VC_encCloseSocket	

Descriptions of each of the above functions can be found in the Function Prototypes section of this document.

The parameters passed in the VC_encOpenSocket function determine the characteristics of the encrypted socket. The mode can be either CBC or CFB. The encryption type can be either TwoFish or AES. The key length can be either128, 192 or 256. *(It should be noted that the TwoFish algorithm only supports 128 bit encryption.)*

The best way to learn how to use the functions in this document is to analyze real working code. A few examples have been included on the product CDs of Device Servers that support encryption. You will need Visual Studio for the 'C' examples and Visual Basic 6.0 for the Visual Basic examples. If you do not have access to the product CD then contact your Lantronix representative or contact Lantronix Technical Support at www.lantronix.com.

The examples that have been created to demonstrate the use of these functions are:

- **encsocktest**
  - A project written in 'C' that demonstrates the use of the cipher socket functions in Client mode. The program accesses the encryption DLL at run time using the LoadLibrary system call.

- **encsocktest_server**
  - A project written in 'C' that demonstrates the use of the cipher socket functions in Server mode. The program accesses the encryption DLL at run time using the LoadLibrary system call.

- **Vctest**
  - A project written in 'C' that demonstrates the use of the cipher functions. The program accesses the encryption DLL at run time using the LoadLibrary system call.

- **vctest_lib**
  - A project written in 'C' that demonstrates the use of the cipher functions. The program demonstrates the use of statically linking with the encryption entry points using the cbx_enc_2_0.lib file. This causes the system to load the encryption DLL at program load time.

- **vb_cob**
  - A project written in Visual Basic that demonstrates the use of the VC_blockEncrypt function. The CBC and CFB algorithms are implemented in the project code.

- **vb_cob_enc**
  - A project written in Visual Basic that demonstrates the use of the CBC and CFB functions implemented in the DLL.

4: Encryption DLL

The Lantronix 2.0 Encryption DLL (cbx_enc_2_0.dll) now contains many function calls. The original function, VC_blockEncryption, is still supported. These functions support:

- Rijndael AES 128,192 and 256 bit encryption and decryption methods
- Two-Fish encryption and decryption methods
- Cipher Block Chaining (CBC) mode
- Cipher Feedback 128 bit (CFB128) mode
- Windows Sockets

These functions have bindings for Visual Basic, C, and Java.

The prototypes for these functions can be found in the **Function Prototypes** section of this document.

C binding

void VC_blockEncrypt(char *iv, char *key, int length, char *text, int RDkeyLen)

iv: the address of the 16 byte initialization vector.

key: the address of 32 characters depicting a 16 byte hex key.

Example:

hex_key[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };

Would be defined as:

static char *key = "000102030405060718191A1B1C1D1E1F";

length: the number of data bytes (including padding).

Must be a multiple of 16 for CBC mode.

If length is negative, then decrypt according to CBC (block mode).

If length is zero, then encrypt according to ECB mode which is used by CFB128.

text: the array of bytes to encrypt or decrypt.

The results are returned in text.

RDkeyLen: 0 for Two Fish, 128, 192, or 256 for Rijndael Encryption

CBC example:

To encrypt 100 plaintext bytes contained in buf using CBC mode and Rijndael (AES128):

```
static char *key = "000102030405060718191A1B1C1D1E1F";
int i;
char iv[16];
for (i = 0; i < 16; i++)       /* create random initialization vector */
    iv[i] = rand();
VC_blockEncrypt(iv, key, ((100 + 15) / 16) * 16, buf, 128);
```

Upon return, buf will contain the ciphertext.

Note: *Buf must be large enough to hold the number of bytes to encrypt rounded up to a multiple of 16. (In this example, buf holds 112 bytes.)*

Decryption is the same call, with buf containing ciphertext, and the length negated:

```
/* iv & key must be known */
VC_blockEncrypt(iv, key, -((100 + 15) / 16) * 16, buf, 128);
```

Upon return, buf will contain the plaintext bytes.

Note: *That buf must be large enough to hold the number of bytes to decrypt rounded up to a multiple of 16. (In this example, 112 bytes.)*

CFB128 example:

To encrypt 100 plaintext bytes contained in 'buf' using CFB128 mode and Rijndael:

```
static char *key = "000102030405060718191A1B1C1D1E1F";
static char ive[16];          /* Initialize only once */
static int pstate = 0;        /* Unused initialization vector cipher bytes */
int i;
for (i = 0; i < 100; i++) {
    if (pstate == 0) {
        VC_blockEncrypt(ive, key, 0, ive, 128);
        pstate = 16;      /* 128 bits, or 16 bytes */
    }
        ive[16 - pstate] ^= buf[i];
        buf[i] = ive[16 - pstate];
        pstate--;
    }
```

Buf now contains the ciphertext.

Note: *Since CFB mode depends on previously encrypted data, ive and pstate must be static.*

Decryption is the same call, but uses the decrypt state machine and ivd:

```
static char *key = "000102030405060718191A1B1C1D1E1F";
static char ivd[16];          /* Initialized with the IV used for encryption */
static int pstatd = 0;        /* Unused initialization vector cipher bytes */
int i;
for (i = 0; i < 100; i++) {
    if (pstatd == 0) {
        VC_blockEncrypt(ivd, key, 0, ivd, 128);
        pstatd = 16;      /* 128 bits, or 16 bytes */
    }
    plaintext[i] = ivd[16 - pstatd] ^ ciphertext[i];
```

```
            ivd[16 - pstatd] = ciphertext[i];
            pstatd--;
        }
```

Buf now contains the plaintext.

Note: *Since CFB mode depends on previously decrypted data, ivd and pstatd must be static*.

Basic binding:

Private Declare Sub VB_blockEncrypt Lib "cbx_enc_2_0" _

VB_blockEncrypt(iv As Byte, ByVal key As String, ByVal length As Long, text As Byte, ByVal RDkeyLen As Long)

iv: the address of the 16 byte initialization vector.

key: the address of 32 characters depicting a 16 byte hex key.

Example:

hex_key(16) bytes 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f

Would be defined as:

key = "000102030405060718191A1B1C1D1E1F"

length: the number of data bytes (including padding).

Must be a multiple of 16 for CBC mode.

If length is negative, then decrypt according to CBC (block mode).

If length is zero, then encrypt according to ECB mode which is used by CFB128.

text: the array of bytes to encrypt or decrypt.

The results are returned in text.

RDkeyLen: 0 for Two Fish, 128, 192, or 256 for Rijndael Encryption

CBC example:

To encrypt 100 plaintext bytes contained in 'buf' using CBC mode and Rijndael (AES128):

```
Private key As String * 32
Private iv(16) As Byte
Dim bytes As Integer
key = "000102030405060718191A1B1C1D1E1F"
bytes = 100
For i = 0 To 15
    iv(i) = 255 * Rnd
    Next i
VB_blockEncrypt(iv(0), key, length + (16 - (length Mod 16)), buf(0), 28);
```

Upon return, buf contains the ciphertext.

Note: *That buf must be large enough to hold the number of bytes to encrypt rounded up to a multiple of 16. (In this example, 112 bytes.)*

Decryption is the same call, with buf containing ciphertext, and the length negated:

```
Private key As String * 32

Dim bytes As Integer

key = "000102030405060718191A1B1C1D1E1F"

bytes = 100

VB_blockEncrypt(iv(0), key(0),

length + (16 - (length Mod 16)), buf(0), 128);
```

Upon return, buf will contain the plaintext bytes.

Note: *That buf must be large enough to hold the number of bytes to decrypt rounded up to a multiple of 16. (In this example, 112 bytes.)*

CFB128 example:

To encrypt 100 plaintext bytes contained in 'buf' using CFB128 mode and Rijndael (AES128):

```
Private key As String * 32

Private pstate As Integer

Private ive(16) As Byte

key = "000102030405060718191A1B1C1D1E1F"

For i = 1 To 100

    If pstate = 0 Then

        Call VB_blockEncrypt(ive(0), key, 0, ive(0), 128)

        pstate = 16

        End If

    ive(16 - pstate) = ByteXOR(ive(16 - pstate), Asc(Mid(buf, i, 1)))

    buf(i) = ive(16 - pstate)

    pstate = pstate - 1

    Next i
```

Buf now contains the ciphertext.

Note: *Since CFB mode depends on previously encrypted data, ive and pstate must be declared static (global).*

To decrypt 100 ciphertext bytes contained in buf using CFB128 mode and Rijndael (AES128):

```
Private key As String * 32
Private pstatd As Integer
Private ivd(16) As Byte
key = "000102030405060718191A1B1C1D1E1F"
For i = 1 To 100
    If pstatd = 0 Then
        Call VB_blockEncrypt(ivd(0), key, 0, ivd(0), 128)
        pstatd = 16
    End If
    ivd(16 - pstatd) = ByteXOR(ivd(16 - pstatd), Asc(Mid(buf, i, 1)))
    buf(i) = ivd(16 - pstatd)
    pstatd = pstatd - 1
    Next i
```

Buf now contains the plaintext.

Note: *Since CFB mode depends on previously decrypted data, ivd and pstatd must be declared static (global).*

Java binding:

private static native void

VJ_blockEncrypt(byte[] iv, String key, int length, byte[] text, int RDkeyLen);

iv: the array of the 16 byte initialization vector.

key: the array of 32 characters depicting a 16 byte hex key.

Example:

hex_key[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };

Would be defined as:

private String keyStr = new
String("000102030405060718191A1B1C1D1E1F ");

length: the number of data bytes (including padding).

Must be a multiple of 16 for CBC mode.

If length is negative, then decrypt according to CBC (block mode).

If length is zero, then encrypt according to ECB mode which is used by CFB128.

text: the array of bytes to encrypt or decrypt.

The results are returned in text.

RDkeyLen: 0 for Two Fish, 128, 192, or 256 for Rijndael Encryption

CBC example:

To encrypt 100 plaintext bytes contained in 'buf' using CBC mode and Rijndael (AES128):

```
private String keyStr =
new String("000102030405060718191A1B1C1D1E1F ");
private int i, newlen;
private char iv[16];
        /* Seed random number generator */
Random rnd = new Random();
        /* Fill Initialization Vector with random data */
rnd.nextBytes(iv);
        /* Round up length to a multiple of 16 (bytes per block) */
newlen = ((100 + 15) / 16) * 16;
```

```
byte pt[] = new byte[newlen];
        /* Fill pt with Random data */
rnd.nextBytes(pt);
        /* Copy the plaintext into the work buffer (pt) because the encoder will
        perform the transformation in place */
for (i = 0; i < len; i++)
    pt[i] = buf[i];
        /* CBC encrypt */
VJ_blockEncrypt(iv, keyStr, newlen, pt, 128);
```

Upon return, pt contains the ciphertext.

Note: *That pt must be large enough to hold the number of bytes to encrypt rounded up to a multiple of 16. (In this example, pt is 112 bytes.)*

Decryption is the same call, with buf containing ciphertext, and the length negated:

```
private String keyStr =
new String("000102030405060718191A1B1C1D1E1F ");
private int i, newlen;
private char iv[16];
VJ_blockEncrypt(iv, key, -((100 + 15) / 16) * 16, buf, 128);
        /* Initialization vector must be known!! */
        /* Calculate the length in multiples of blocks */
newlen = ((100 + 15) / 16) * 16;
        /* Extract the ciphertext into the work array because the decoder will
        perform the transformation in place */
byte pt[] = new byte[newlen];
        /* Copy the ciphertext into the pt work buffer */
for (i = 0; i < newlen; i++)
    pt[i] = buf[i];
        /* Decrypt it by negating the encrypted data length */
VJ_blockEncrypt(ivd, keyStr, -newlen, pt, 1);
        /* Copy the pt into the plaintext buffer */
```

Upon return, pt contains the plaintext bytes.

Note: *That pt must be large enough to hold the number of bytes to decrypt rounded up to a multiple of 16. (In this example, pt is 112 bytes.)*

CFB128 example:

To encrypt 100 plaintext bytes contained in 'buf' using CFB128 mode and Rijndael (AES128):

```
private String keyStr = new String("000102030405060718191A1B1C1D1E1F ");
 private char ive[16];
 private int pstate = 0;      /* Unused initialization vector cipher bytes */
 private int i;
 byte ct[] = new byte[100];
 for (i = 0; i < 100; i++) {
      if (pstate == 0) {
           VJ_blockEncrypt(ive, keyStr, 0, ive, 128);
            pstate = 16;
      }
      ive[16 - pstate] ^= buf[i];
      ct[i] = ive[16 - pstate];
      pstate--;
}
```

ct now contains the ciphertext.

Note: *Since CFB mode depends on previously encrypted data, ive and pstate must be static.*

Decryption is the same call, but uses the decrypt state machine and ivd.

```
private String keyStr =
new String("000102030405060718191A1B1C1D1E1F ");
private char ivd[16];
private int pstatd = 0;      /* Unused initialization vector cipher bytes */
private int i;
byte pt[] = new byte[100];
for (i = 0; i < 100; i++) {
     if (pstatd == 0) {
          VJ_blockEncrypt(ivd, key, 0, ivd, 128);
          pstatd = 16;        /* 128 bits, or 16 bytes */
     }
```

```
            pt[i] = ivd[16 - pstatd] ^ ciphertext[i];
            ivd[16 - pstatd] = ciphertext[i];
            pstatd--;
        }
```

pt now contains the plaintext.

Note: *Since CFB mode depends on previously decrypted data, ivd and pstatd must be static.*

5: Function Prototypes

The prefix of the functions in the Cobox encryption DLL depends on the language used. The prefix VC_ is used for Visual C++, VB_ for Visual Basic and VJ_ for Java. This document will describe the VC_ functions. The parameters and calling order are the same for all three languages. Four examples of using some of the functions in C and two examples of using some of the functions in Visual Basic can be found on the CD.

The functions return either ENC_SUCCESS, which is zero, or an error, which is always negative. When dealing with the encryption sockets, SOCKET_ERROR can be returned when an error in Winsock occurs. A specific error code can be retrieved by calling WSAGetLastError.

The VC_blockEncrypt function is the only function available in the original form of the DLL, which is cbx_enc.dll. The new DLL is called cbx_enc_2_0.dll. The new functions will use a handle that is created either through the VC_Init or VC_encOpenSocket function. This handle is opaque to the user. It is known to the user as a PVOID.

The values of a few constants and error codes that are returned from functions in the encryption DLL can be found in the Error Code Values and Other Constants section

The prototype of the original cipher function, VC_blockEncrypt, can be found in the Existing Function section.

The prototypes for cipher functions that have incorporated the CBC and CFB algorithms in them, but do not perform any socket functions, can be found in the Cipher Functions section.

Finally, the cipher functions that have CBC, CFB, and Windows Sockets can be found in the Cipher Socket Functions section.

Error Code Values and Other Constants

```
#define     MODE_ECB                1     // Use ECB mode
#define     MODE_CBC                2     // Use CBC mode
#define     MODE_CFB1               3     // Use CFB1 mode
#define     ENC_TYPE_NONE           0     // No cipher used
#define     ENC_TYPE_TWOFISH        1     // Use TwoFish cipher
#define     ENC_TYPE_AES            2     // Use AES cipher
#define     MAX_KEY_SIZE            64    // Max Key size
#define     MAX_IV_SIZE             16    // Init Vector Size
#define     TYPE_SERVER             1     // Create Server socket
#define     TYPE_CLIENT             2     // Create Client socket

// Error Defines:

#define     ENC_SUCCESS             0     // Function returns OK
#define     BAD_KEY_MAT             -2    // Key Length error
#define     BAD_KEY_INSTANCE        -3    // TwoFish key error
#define     BAD_CIPHER_MODE         -4    // Unsupported cipher
#define     BAD_CIPHER_STATE        -5    // Invalid cipher params
#define     BAD_INPUT_LEN           -6    // Invalid array lengths
#define     BAD_PARAMS              -7    // Invalid parameter
#define     BAD_IV_MAT              -8    // Invalid Init vector
#define     BAD_ENDIAN              -9    // Endian error
#define     BAD_ALIGN32             -10   // Alignment error
#define     BAD_KEY_DIR             -11   // TwoFish key direction
#define     ENC_ERR_NOMEMORY        -12   // No memory available
#define     ENC_ENCRYPTION_ERR      -13   // An encryption error
#define     ENC_DECRYPTION_ERR      -14   // A decryption error
#define     ENC_ACCESS_WINSOCK_ERR  -15   // Access Winsock error
#define     ENC_NO_RECV_INIT_VECTOR -16   // Did not receive the
                                          // init vector from a
                                          // client.
#define     ENC_FATAL_ERROR         -17   // A Fatal,
                                          // unrecoverable error
                                          // has occurred.

// Error Defined in WinSock.h:

#define     SOCKET_ERROR            (-1)
```

Existing Function

There is one function in the first version of the cbx_enc.dll file, VC_blockEncrypt.

VC_blockEncrypt

Encrypts or Decrypts data in a buffer. This is the original function, unmodified. There are no sockets associated with this function. The cipher parameter contains the initialization vector. This needs to be passed in each time. This vector contains random numbers. When encrypting, the text parameter contains the plain, or clear, text. Upon return, the text parameter will contain the encrypted text. When decrypting, the text parameter contains the encrypted text. Upon return, the text parameter will contain the decrypted text. The description of each parameter can be found under the "Parameters" section below.

Prototype

```
void __cdecl
VC_blockEncrypt(
      char *cipher,
      char *key,
      int  bytes,
      char *text,
      int  RijndaelKeyLen
);
```

Parameters

cipher

[in] Initialization Vector: For TCP, encrypted/decrypted IV. For UDP, iv32.

key

[in] A Character string containing the key.

bytes

[in] # of bytes - for TCP = 0, for UDP > 0 (encrypt) or < 0 (decrypt)

text

[in/out] encrypted/decrypted text (must be aliquot by 16 for UDP)

RijndaelKeyLen

[in] 0 if TwoFish, >0 if Rijndael (128, 192 or 256)

Return Values

Nothing is returned. "text" parameter contains encrypted or decrypted text.

Cipher Functions

These functions retain state information in the form an opaque handle returned to the user. The state information consists of the encryption mode, encryption type, key, keyLength, initVector when processing using CFB, and other pieces of data. Also, CBC and CFB have now been incorporated into the encryption DLL.

Use the functions in this section, together as a group, to encrypt and decrypt buffers using CBC and CFB. No networking is involved in these functions. A developer can perform encryption using these functions and develop his own networking code to send and receive the encrypted buffers. If a developer wishes to use the networking portion of the encryption DLL then the functions in the Cipher Functions section need to be used.

VC_Init

VC_Init initializes an encryption session. The parameters are saved in the opaque handle and used by the other cipher functions. This is accomplished by issuing the user an opaque handle that will be passed to the other functions by the user. The user would first call VC_Init and use the returned handle parameter in calls to VC_Encrypt and VC_Decrypt. When the user is finished encrypting and decrypting data, the handle needs to be passed to VC_Cleanup. It is important that the user calls VC_Cleanup as this releases memory allocated for the handle. A return value ENC_SUCCESS indicates a successful initialization.

Prototype

```
int __cdecl
VC_Init(
      int   mode,
      char  *initVector,
      char  *key,
      int   keyLength,
      int   encryptionType,
      PVOID *ptrHandle
);
```

Parameters

mode

[in] Encryption mode: Either CBC=2, CFB1=3

initVector

[in] 16 byte initialization vector

key

[in] key string (example: “000102030405060708091a1b1c1d1e1f” for 128 bits)

KeyLength

[in] Length of key in bits (128, 192, or 256)

encryptionType

[in] ENC_TYPE_TWOFISH, ENC_TYPE_AES

ptrHandle

[out] Opaque handle returned to user and used in subsequent calls to the other functions in this section.

Return Values

```
BAD_KEY_MAT
BAD_PARAMS
ENC_ERR_NOMEMORY
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_Encrypt

This function will encrypt the data that is contained in the plainText parameter. The encrypted data will be returned in the cipherText parameter. The handle parameter can be obtained by calling VC_Init. The length of the data to be encrypted is contained in the textLen parameter.

If the CBC mode is used, then a new initVector needs to be created before each call to VC_Encrypt and passed to it. Also, the plainText buffer and consequently the value of textLen need to be a multiple of 16 bytes. The cipherText buffer needs to be large enough to hold the encrypted data.

Prototype

```
int __cdecl
VC_Encrypt(
      PVOID handle
      char  *initVector,
      char  *plainText,
      char  *cipherText,
      int   *textLen
);
```

Parameters

handle

[in] Opaque handle

initVector

[in] Only used for CBC. The initVector needs be “refreshed” each call. This is not used for CFB and can be NULL if the current mode is MODE_CFB1.

plainText

[in] Buffer that holds the plain text

cipherText

[out] Buffer to hold cipher text

textLen

[in/out] Length of text to be encrypted. Length of data encrypted upon returned.

Return Values

```
BAD_PARAMS
BAD_INPUT_LEN
BAD_CIPHER_STATE
BAD_KEY_INSTANCE
BAD_ALIGN32
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_Decrypt

This function will decrypt the data that is contained in the cipherText parameter. The decrypted data will be returned in the plainText parameter. The handle parameter can be obtained by calling VC_Init. The length of the data to be decrypted is contained in the textLen parameter.

If the CBC mode is used, then the initVector used to Encrypt the data needs to be passed to the VC_Decrypt function. Also, the size of the cipherText buffer will be a multiple of 16 bytes. The plainText buffer needs to be large enough to hold the decrypted data.

Prototype

```
int __cdecl
VC_Decrypt(
      PVOID handle
      char  *initVector,
      char  *cipherText,
      char  *plainText,
      int   *textLen
);
```

Parameters

handle

[in] Opaque handle

initVector

[in] Only used for CBC. The initVector needs be “refreshed” each call. This is not used for CFB and can be NULL if the current mode is MODE_CFB1.

cipherText

[in] Buffer to hold cipher text

plainText

[out] Buffer that holds the plain text

textLen

[in/out] Length of text to be decrypted. Length of data decrypted returned.

Return Values

```
BAD_PARAMS
BAD_INPUT_LEN
BAD_CIPHER_STATE
BAD_KEY_INSTANCE
BAD_ALIGN32
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_Cleanup

Clean up memory allocated for handle.

Prototype

```
int __cdecl
VC_Cleanup(
      PVOID handle
);
```

Parameters

handle

[in] Opaque Handle

Return Values

```
BAD_PARAMS
ENC_FATAL_ERROR
ENC_SUCCESS
```

Cipher Socket Functions

The following functions act as wrappers for the Winsock 2 functions with the added functionality to encrypt and decrypt data. Successful calls will return ENC_SUCCESS. If an error occurs, then the return value will be a negative number. Either the number defined above in the Defines section, or SOCKET_ERROR. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Use these functions, together as a group, when using the encrypted sockets capability of the encryption DLL.

VC_encOpenSocket

This function will open a Windows socket. It will also setup and remember the encryption parameters. If the protocol is TCP, it will also send a random initialization vector over the network connection. TCP will use CFB and UDP will use CBC. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Prototype

```
int __cdecl
VC_encOpenSocket(
      int   type,
      char  *remoteHost,
      int   remotePort,
      int   protocol,
      char  *initVector,
      char  *key,
      int   keyLength,
      int   encryptionType,
      PVOID *ptrHandle
 );
```

Parameters

type

[in] Server=1, Client=2

remoteHost

[in] remote host

remotePort

[in] remote port

protocol

[in] IPPROTO_UDP or IPPROTO_TCP

initVector

[in] 16 byte initialization vector

key

[in] key string

keyLength

[in] Length of key in bits

encryptionType

[in] None=0, twoFish=1, AES=2

ptrHandle

[out] Pointer to Opaque handle returned to user

Return Values

```
BAD_KEY_MAT
BAD_PARAMS
ENC_ERR_NOMEMORY
ENC_FATAL_ERROR
ENC_SUCCESS
SOCKET_ERROR
```

VC_encGetWinSocket

This function returns the socket used by the lower-layer Winsock functions. The user can use this value to call native Winsock functions. An example would be to place the socket in Non-Blocking mode.

Prototype

```
int __cdecl
VC_encGetWinSocket(
      PVOID  handle,
      SOCKET *socket
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

socket

[out] Pointer to a Winsock SOCKET.

Return Values

```
BAD_PARAMS
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_encGetSocketAddr

This function returns a pointer to the SOCKADDR_IN structure which is defined in Winsock.h. The user can use this structure to obtain the IP address and port number used for the socket.

Prototype

```
int __cdecl
VC_encGetSocketAddr(
      PVOID handle,
      SOCKADDR_IN **sockAddr
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

sockAddr

[out] Pointer to a pointer to a structure of type SOCKADDR_IN

Return Values

```
BAD_PARAMS
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_encGetWinSockDLL

This function will return the HANDLE to the Winsock DLL. The user can use this to obtain Winsock functions. For example, the user can call the WSAGetLastError function which is contained in the Winsock DLL.

Prototype

```
int __cdecl
VC_encGetWinSockDLL(
      HINSTANCE *hDLL
);
```

Parameters

hDLL

[in/out] Pointer to a pointer to an HINSTANCE structure. This is the handle to the Winsock functions

Return Values

```
ENC_ACCESS_WINSOCK_ERR
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_encListen

For type = Server, Listen for a connection. If the socket was opened as a Server socket, then a call to this function will place the socket in a state in which it is listening for an incoming connection. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Prototype

```
int __cdecl
VC_encListen(
      PVOID handle
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

Return Values

```
BAD_PARAMS
ENC_ACCESS_WINSOCK_ERR
ENC_FATAL_ERROR
ENC_SUCCESS
SOCKET_ERROR
```

VC_encAccept

For type = Server, Accept a connection. If the socket was opened as a Server socket, and the VC_encListen function was called, then a call to this function will permit an incoming connection attempt on a socket. The accepted connection's socket will be returned in the clientHandle parameter. This handle can be used to send and receive data using VC_encSendData and VC_encRecvData, respectively. This handle needs be close using VC_encCloseSocket when use of the handle is not longer desired. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Prototype

```
int __cdecl
VC_encAccept(
      PVOID handle,
      PVOID *clientHandle
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

clientHandle

[out] pointer to an opaque handle that defines the socket that was accepted. This handle can be used in function calls that require a handle parameter. If clientHandle is returned as INVALID_SOCKET then the return value of VC_encAccept will be SOCKET_ERROR.

Return Values

```
BAD_PARAMS
ENC_ACCESS_WINSOCK_ERR
ENC_ERR_NOMEMORY
ENC_FATAL_ERROR
ENC_SUCCESS
SOCKET_ERROR
```

VC_encSendData

This function will take the input buffer, encrypt it, and send it out the socket that is described in the handle parameter. On success, the number of bytes sent is returned in the bufSize parameter. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Prototype

```
int __cdecl
VC_encSendData(
      PVOID handle,
      char  *buf,
      int   *bufSize
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

buf

[in] Pointer to a buffer than contains the plain text data.

bufSize

[in/out] Size of the data in the buf parameter. The size of the data actually encrypted and sent is returned in this parameter.

Return Values

```
BAD_PARAMS
ENC_ACCESS_WINSOCK_ERR
ENC_ERR_NOMEMORY
ENC_FATAL_ERROR
ENC_SUCCESS
SOCKET_ERROR
```

VC_encRecvData

This function will read data from a socket and return. The bufSize parameter is the length of the data in buf, in bytes. On success, the number of bytes received is returned in bufSize. If SOCKET_ERROR is returned, a specific error code can be retrieved by calling the WSAGetLastError Winsock function.

Prototype

```
int __cdecl
VC_encRecvData(
      PVOID handle,
      char  *buf,
      int   *bufSize
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

buf

[out] Pointer to a buffer than will receive the plain text data.

bufSize

[in/out] Size of the buf parameter, in bytes. The size of the data actually decrypted and received is returned in this parameter.

Return Values

```
BAD_PARAMS
ENC_ACCESS_WINSOCK_ERR
ENC_ERR_NOMEMORY
ENC_FATAL_ERROR
ENC_SUCCESS
SOCKET_ERROR
```

VC_encShutDown

Gracefully disables all send and receives on a socket.

Prototype

```
int __cdecl
VC_encShutDown(
      PVOID handle,
      int how
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

how

[in] Flag that describes what types of operation will no longer be allowed:

SD_RECEIVE, SD_SEND, or SD_BOTH

Return Values

```
ENC_FATAL_ERROR
ENC_SUCCESS
```

VC_encCloseSocket

Close the Windows socket and cleans up memory.

Prototype

```
void __cdecl
VC_encCloseSocket(
      PVOID handle
);
```

Parameters

handle

[in] Opaque handle obtained by calling VC_encOpenSocket

Return Values

```
ENC_FATAL_ERROR
ENC_SUCCESS
```