COPYRIGHT

Firmas Digitales S.R.L.
Leandro N. Alem 1080 2º C
(1001) Buenos Aires
ARGENTINA

Tel. : (5411) 4314-8972
Cel. : (5411) 4970-6665
(5411) 4403-4230
(5411) 4474-2064
Fax : (5411) 4314-8973

E-mail : info@fd.com.ar
Web : www.fd.com.ar

TRADEMARKS

CONTENTS

PREFACE

FDCrypt is a set of high performance routines that provides the designer with the possibility to add cryptography to their applications. These same applications can make use of the FDCrypt functions without the necessity of knowing its specific implementation. These routines were developed with multi-platform support and comply to current international standards.

This manual was written for the designer who would like to incorporate FDCrypt to their applications. Although it includes a section which explains techniques and cryptographic algorithms, it does not constitute a complete guide on the subject. Previous experience in cryptography is convenient but not necessary. For the reader who would like to learn more about the subjects presented here, a list of study material is available for consultation in Appendix E. A glossary of terms is also available in Appendix F. Furthermore, the examples presented in this manual require the knowledge of diverse programming languages.

MANUAL TYPEFACES AND SYMBOLS

In this document, you will find the following guides:

Italic	Identifies function headers
`Courier`	Program examples
Dotted Line	Its definition is described in the glossary
Underlined	Cross referenced to another section of the manual

Section 1

Introduction to Cryptography

This chapter presents an introduction to basic concepts of modern cryptography. Appendix-E contains a list of books and articles where you can learn more about these subjects.

OVERVIEW

Cryptography provides a combination of techniques to code messages in such a way as to be able to store and transmit said messages in a secure manner. For example, cryptography can be used to store confidential information so that an intruder cannot read it, or to transmit messages by insecure or unreliable channels in a totally secure manner. Besides maintaining confidentiality, cryptography can be used to assure the integrity of data to be stored or transmitted, that is, that this data cannot be modified and that these changes go unperceived. It can also verify the authenticity of a message and using digital signatures does not permit these messages to be repudiated, that is, that the person who sent the message not recognize its origin. In conclusion, cryptography provides the following services:

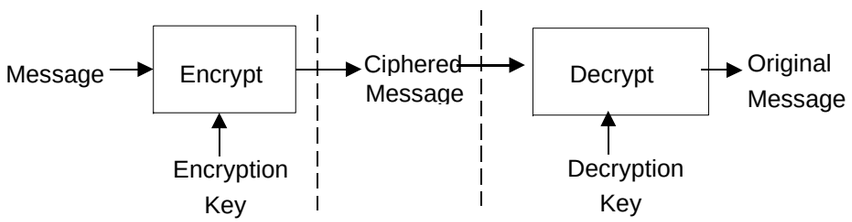
- ✓ Confidentiality
- ✓ Integrity
- ✓ Authentication
- ✓ Non-repudiation

Traditionally, cryptography was essentially restricted to military and diplomatic applications (whose origins go back to Julius Caesar), by means of symmetric algorithms, in which the same key is used to encrypt and decrypt. These types of algorithms have the disadvantage of having to resolve the key distribution problem through secure channels. In 1976, Diffie and Hellman published a fundamental work about how to distribute keys through insecure channels. The simple fact is that each user possesses two keys: a private one, kept secret and a public one, accessible to anyone.
The existence of this technology is what makes electronic commerce and the authentication of transactions via networks like Internet possible, and moreover, there is currently an increasing demand for secure cryptographic products.

Cryptographic algorithms use keys. Only the key should be secret. Knowledge of the algorithms involved does not permit access to protected information if the password utilized is unknown. Therefore, an emphasis should be placed on the correct storage, control, lifetime and destruction of generated and used keys. The design of a security system should be in the hands of experts.

ENCRYPTION

Using cryptography, an original message is coded so that it looks like a message made up of random bits and is impossible to convert again to the original message without a secret key. When we refer to a “message”, we refer to any information, be it a file or a string of characters.
An non-encrypted message is known as a plaintext, while an encrypted message is called a ciphertext. Once encrypted, a message can be stored or transmitted and remains secret until it is decoded to its original state. A secret key is used in order to encrypt or decrypt a message. Only persons possessing this key can read the message.
There exist two types of encryption algorithms: symmetric, where the key for encrypting is the same as the key for decrypting, and public key algorithms (also known as asymmetric algorithms) where the keys for encrypting and decrypting are different.



Symmetric Algorithms

Symmetric algorithms use the same key to encrypt and decrypt. These keys change often and therefore, are also known as session keys when they are generated randomly. Compared with public key algorithms, they are much faster and therefore used to encrypt large quantities of data. The available symmetric algorithms in FDCrypt are: DES, Triple DES and IDEA, as defined by international standards.

Public Key Algorithms

Public key algorithms (asymmetric) use two different and related keys. One is called the “public” key and the other is called the “private” key. The private key is kept in secret by the owner of the key and the public key is distributed to all who require it. When one key is used to encrypt, the other key must be used to decrypt the message.
Public key algorithms are much slower than symmetric algorithms. As a result, they are used in combination with symmetric algorithms to encrypt a session key together with the public key of the receiver of the message. These systems are known as hybrid systems. They are also used to digitally sign an encrypted message together with the private key of the sender. Thus, anybody can verify the origin of the message. The public key algorithm implemented in FDCrypt is RSA, de facto international standard in finance, government and administration applications, whenever maximum reliability and security is required.

HASHING

Hashing algorithms (also called fingerprints, checksums or message digests) allows one to verify that a message has not been modified, since a message of arbitrary size produces an output of fixed size. These types of functions are known as one-way functions due to the fact that it is very easy to calculate a hash for a message, but very difficult to find a message that produces a particular hash value. Given that the cardinality of the space of all possible messages is much greater than the number of different combinations for a determined size of hash, there necessarily exist various messages that produce the same result, although it is computationally impossible to find them.
Hashing algorithms implemented, according to international norms, in FDCrypt are: MD4, MD5, SHA, SHA-1, RIPEMD-128 AND RIPEMD-160.

MAC AUTHENTICATION

A MAC is the result of applying a key dependent hash function to a message. Although they do not provide confidentiality, they do allow one to verify the integrity and authenticity of a message. FDCrypt implements generation and verification of MAC utilizing any symmetric algorithm.

DIGITAL SIGNATURES

Digital signatures are used to verify the integrity and authenticity of a message. The latter can also be achieved using conventional cryptographic algorithms. The digital signature guarantees as well, the non-repudiation of a message and thus has the same legal value as a traditional holographic signature (in countries that possess a digital signature law). In Argentina, through a presidential decree in May of 1998, that status is granted to the aforementioned technique for the entire national public administration.
Digital signatures are generated using a public key algorithm. For this, a hash is encrypted with the private key of the sender in the message to be signed. Anyone can verify the validity of the digital signature from the message using the public key of the sender.
It is worth mentioning that the use of digital signatures usually requires the presence of a Certifying Authority (CA) of renowned prestige, which guarantees the origin of each active public key in the system, and is also responsible for divulging those keys that are no longer in use, what are known as revocation lists.
FDCrypt implements digital signatures utilizing any hashing algorithm with RSA.

DIGITAL ENVELOPES

Digital envelopes are used to solve the key distribution problem when symmetric algorithms are used. Digital envelopes are generated using a public key algorithm. For this, a session key is encrypted with the public key of the recipient of the message. Only the recipient can open the envelope and recover the session key necessary to decrypt the encrypted message. FDCrypt implements digital envelopes combining any symmetric algorithm with RSA.

COMPRESSION

One characteristic that differentiates a plaintext from a ciphertext is that encrypted messages seem to be composed of random bits, while the former generally contain a certain order or redundancy. This redundancy can be measured and is proportional to the inverse of the entropy. The security of a cryptographic system does not solely depend on the encryption algorithm used, but also depends on the quantity of different messages that can be generated. The security can be increased by reducing the redundancy of plaintexts. This is achieved by using compression algorithms that eliminate the redundancy from the original message. After being encrypted-transmitted-decrypted, the message is expanded to recover its original format. FDCrypt offers adaptive lossless compression based on the LZW algorithm.

SYMMETRIC ENCRYPTION

Symmetric key algorithms encrypt messages in units called blocks. The usual size of a block is 64 bits. The specification of an algorithm is limited to the encryption/decryption process of a unitary block. The implementation details of how a message is padded to block size and how the encryption of multiple blocks is performed, is specified independently from the algorithm.

Operation Modes

Normally a message is composed of more than one block. The operation mode indicates how the blocks should be linked. FDCrypt implements the following operation modes: ECB, CBC, CFB, OFB, BCF y BOF. All of which are defined according to international standards.

Padding

The size of most messages is not a multiple of a block's length. Normally, the last block is shorter and must be treated in a special way or padded until it reaches the size of a block. This padding is removed during decryption. FDCrypt offers different forms of padding all according to international standards.

Initialization Vectors

With the exception of the ECB operation mode, all the rest use feedback. These operation modes require an initialization vector of random bits in such a way that the same message is encrypted in different forms even though the same key is utilized. Each message must be encrypted with a different initialization vector. This vector need not remain secret and can be stored along with the ciphered message.

Section 2

Fundamental Concepts

This chapter presents the general philosophy with which the functions that comprise FDCrypt were developed, as well as the available algorithms and supported platforms.

OVERVIEW

To perform cryptographic operations it is necessary to combine different functions. For example, if you want to send a encrypted message with a DES session key, you must follow the subsequent steps: (a) generate a DES session key, (b) encrypt the message with the key, (c) encrypt the DES session key with the public key of the recipient, (d) destroy the session key.
In this way we can see that the session key behaves as an object that has a determined lifetime. This object is created, used and destroyed. Given that cryptographic objects are very sensitive, they are administered transparently. The user refers to these objects through a single handle generated by FDCrypt at the moment the object is created. The successive operations are carried out passing only the required handle. Finally, all objects must be destroyed. This philosophy assures that the user cannot perform invalid cryptographic operations. Existing cryptographic objects in FDCrypt are:

- Hash
- Symmetric Keys
- Public and Private Keys
- Digital Signatures
- Digital Envelopes

All objects can be created, updated, exported, imported and destroyed. Normally these objects are known as context areas.

ERROR HANDLING

All FDCrypt functions return a status code indicating if the function was executed in a satisfactory manner or not. If the function is executed correctly the constant FD_OK (zero) is returned; on the contrary, the error cause is returned expressed as a negative number. Error codes can be due to operational errors, like for example, trying to create a hash type object using as argument a symmetric encryption algorithm. In general these errors are due to the incorrect use of the provided functions. A second type of error are those related to security. For example, when verifying the integrity of a message, the comparison of two hash returns that they are not equal. Consult appendix C for a list of all possible errors and their causes.

COMBINING OPERATION MODES WITH PADDING

The way in which a buffer is encrypted depends on the combination of parameters used. To better understand how the encryption is performed, we can define the additive property when several successive encryptions are equivalent to a single encryption:

E (“abc”) ≡ E (“a”) + E (“b”) + E (“c”)

The only mode without feedback is ECB. In this mode it is not necessary to indicate an initialization vector. It is normally preferable to use CBC mode, which offers greater security. ECB mode operates on 64 bit blocks. If the length of the buffer to encrypt is a multiple of 8 and a padding method is not specified, the buffer is encrypted block by block, and the result is an encrypted buffer with the same length as the original buffer. Under these conditions the additive property is preserved. If a padding method is specified, then the buffer to encrypt is padded to a multiple of 8 according to the specified method (padding is always added, even when the length is already a multiple of 8). Then the buffer is encrypted block by block as in the previous case. Finally, if a padding method is not specified and the buffer is not a multiple of 8, then a special truncation technique is applied to the last block. The result has the same length as the original buffer. These last two cases do not preserve the additive property. The truncation technique consists of encrypting the last entire block a second time and performing a XOR operation with the last partial block. Since ECB isn’t a feedback mode, this technique requires the length of the buffer be greater than the size of a block. This is the only combination of parameters not allowed in the *fdEncryptBuffer* function.

In feedback operation modes, an initialization vector must be supplied. This vector must be transmitted together with the encrypted data. Initialization vectors can be generated randomly and shouldn't be reused.

CBC mode is the safest operation mode and operates on 64 bit blocks. If the length of the buffer to encrypt is a multiple of 8 and a padding method is not specified, the encrypted buffer will be the same length as the original buffer. Under these conditions the additive property is preserved. If a padding method is specified, then the buffer to encrypt is padded to a multiple of 8 according to the specified method. Then the buffer is encrypted as in the previous case. Finally, if a padding method is not specified and the buffer is not a multiple of 8, then a special truncation technique is applied to the last block. The result has the same length as the original buffer. These last two cases do not preserve the additive property. The truncation technique consists of encrypting the entire last block a second time and performing a XOR operation with the last partial block. Unlike the ECB mode, it is possible to operate with sizes less than a block (although in that case it is indistinguishable from CFB mode).

The rest of the feedback modes (CFB, OFB, BCF y BOF) are generally used without padding, since they have the particularity of conserving the additive property regardless of the length of the buffer to encrypt. They are ideal for encrypting a stream where the length is always variable. Although not common, they can be used with padding. In this case the buffer is padded as in the ECB and CBC modes and the additive property is lost. CFB and OFB modes feedback every 64 bits and are faster than BCF and BOF modes. BOF mode should be avoided since in some conditions the effectiveness of the key is reduced by half, making it insecure.

FORMAT MASKS

In order to facilitate the integration of other products, all the functions that export and import buffers have an associated parameter called a format mask. This parameter indicates how the output should be coded when exporting a buffer, or how an input buffer should be parsed. This coding affects the size of the buffer. Since input/output buffers are normally predicted by the calling routine, it is important that the required size can be calculated correctly. For all exported objects there exist functions that inform the size of said objects. These sizes are always expressed in binary format. Hexadecimal codification (generic, ASCII or EBCDIC) implies that all buffers should double in size. If separation spaces and/or a null termination string character are specified then the actual buffer size should be large enough to accommodate these extra characters. Format masks can be combined (as long as they are not

exclusive). Consult appendix B for a list of all available format masks and their associated constants.

DESCRIPTION OF ALGORITHMS

FDCrypt implements the following algorithms:

- ✓ Hash calculation: MD4, MD5, SHA, SHA-1, RIPEMD-128, RIPEMD-160
- ✓ Symmetric key generation: DES (56 bits), Triple DES (112 y 168 bits), IDEA (128 bits)
- ✓ Encryption and decryption of symmetric algorithms
- ✓ Symmetric algorithms operation modes: ECB, CBC, CFB, OFB, BCF, BOF
- ✓ Parametric generation of public keys : RSA (from 512 to 8192 bits)
- ✓ Adaptive lossless compression: LZW
- ✓ MAC calculation and verification using symmetric algorithms
- ✓ Digital signature combining hashing algorithms with RSA.
- ✓ Digital envelope combining symmetric algorithms with RSA.

TYPES OF DATA

The function arguments of FDCrypt use the following types of data:

Type of Data	**Description**
int	Signed Integer, 16 / 32 bits long
unsigned int	Unsigned Integer, 16 / 32 bits long
long	Signed Integer, 32 / 64 bits long
unsigned long	Unsigned Integer, 32 / 64 bits long
char *	Character string

Equivalencies between types of data:

'C' Language	**Visual Basic 16 bits**	**Visual Basic 32 bits**	**Visual Fox**	**Delphi**

short	Integer	Integer	Short	SmallInt
int	Integer	Long	Integer	Integer
long	Long	Long	Long	Longint
unsigned short	Integer [1]	Integer [1]	Short [1]	Word
unsigned int	Integer [1]	Long [1]	Integer [1]	Cardinal
unsigned long	Long [1]	Long [1]	Long [1]	LongInt [2]
char	String * 1	String * 1	String * 1	Char
unsigned char	String * 1	String * 1	String * 1	Byte
char *	String	String	String	Pchar

(1) Visual Basic and Visual FoxPro do not support unsigned integers. The closest type of data is used.
(2) Delphi does not support unsigned long integers in 16 bits. In 32 bits it can be replaced by the Cardinal type of data.

SUPPORTED PLATFORMS

The complete set of routines that composes FDCrypt is available in 16, 32 and 64 bits for diverse architectures and operating systems. The functions are written in ANSI C and are provided as dynamic and/or static link libraries. All functions are reentrant and support multithreading. In the PC environment they should be linked specifying ‘large model’. Likewise, interfaces have been developed for diverse programming languages. Those available are summarized in the following tables:

Operating System	**Version**	**Available Variables**
DOS		16-bit static link library
WIN16	Windows 3.1 WFW 3.11	16-bit static link library 16-bit dynamic link library
WIN32	Windows 95 Windows 98 Windows NT 3.51 Windows NT 4.0	32-bit static link library 32-bit dynamic link library
UNIX	SCO UNIX HP-UX SUN SOLARIS IBM AIX	32-bit static link library 32-bit shared library

	Open BSD LINUX Sequent DYNIX	
	Digital Alpha	64-bit static link library 64-bit shared library
IBM / MVS		32-bit load module library

Operating System	**Supported Languages**	**Available Libraries**
DOS	Borland C++ 4.x, 5.x	static link library (LIB)
	Visual C++ 1.5x	static link library (LIB)
WIN16	Borland C++ 4.x, 5.x	static link library (LIB)
	Visual C++ 1.5x	static link library (LIB)
	Borland Delphi 1.0 Visual Basic 3.0, 4.0	dynamic link library (DLL)
WIN32	Borland C++ 5.x Borland C++ Builder 1.0, 3.0	static link library (LIB)
	Visual C++ 2.0, 4.0, 5.0	static link library (LIB)
	Borland Delphi 2.0, 3.0, 4.0 Visual Basic 4.0, 5.0 Visual FoxPro 5.0	dynamic link library (DLL)

Futhermore, FDCrypt can be used from any language that can access a dynamic link library (DLL). This includes: Microsoft Access, CA Clipper, Borland Visual dBASE, Borland Paradox, Powersoft PowerBuilder, etc.

Section 3

FDCrypt Functions Library

This chapter contains all the functions with their specific syntax corresponding to each program language. Appendix A contains a summary of each one.

FUNCTION DETAILS

The following guide is used to discriminate among the different program languages :

C/C++ *Prototypes for C / C++*
PAS *Prototypes for Pascal / Delphi - 16 / 32 bits*
VB3 *Prototypes for Visual Basic - 16 bits*
VB5 *Prototypes for Visual Basic - 32 bits*
FOX *Prototypes for Visual FoxPro - 32 bits*

For clarity, the prototypes are presented in reduced form, emphasizing the parameters and types of data, not the particularities of the language. For an exact reference you should consult the files provided for each program language. As an example, one function is shown with the differences between the file and manual.

Prototypes for C / C++

File: fdapi.h

FILE *int FDAPI fdGenerateKey (long *kid, int algorithm);*

C/C++ *int fdGenerateKey (long *kid, int algorithm);*

Prototypes for Pascal / Delphi

File: fdapi.pas

FILE *Function fdGenerateKey (Var kid: Longint; algorithm: Integer): Integer; stdcall; External 'fdapi32.dll'*

FILE *Function fdGenerateKey (Var kid: Longint; algorithm: Integer): Integer; far; pascal; External 'fdapi16.dll'*

PAS *fdGenerateKey (Var kid: Longint; algorithm: Integer): Integer;*

Prototypes for Visual Basic 16 bits

File: fdapi16.bas

FILE *Declare Function fdGenerateKey% Lib "fdapi16.dll" (kid As Long, ByVal algorithm As Integer)*

VB3 *fdGenerateKey (kid As Long, ByVal algorithm As Integer) As Integer*

Prototypes for Visual Basic 32 bits

File: fdapi32.bas

FILE *Declare Function fdGenerateKey& Lib "fdapi32.dll" (kid As Long, ByVal algorithm As Long)*

VB5 *fdGenerateKey (kid As Long, ByVal algorithm As Long) As Long*

Prototypes for Visual FoxPro 32 bits

File: fdapi32.prg

FILE *Declare Integer fdGenerateKey In fdapi32.dll Integer @kid, Integer algorithm*

FOX *Integer fdGenerateKey Integer @kid, Integer algorithm*

Hash Calculation and Verification Functions

To calculate a hash, first you must initialize a context area specifying the algorithm that you want to utilize with the *fdCreateHash* function. Next, the data should be processed performing successive calls to the functions *fdHashBuffer*, *fdHashNumber* and *fdHashFile*. In order to obtain the hash value the *fdExportHash* function is invoked, with which the

context area cannot be updated anymore. Finally, you must destroy the context area with the *fdDestroyHash* function. The *fdInfoHash* function allows you to dynamically obtain data from a context area. The *fdImportHash* function allows you to generate a context area with a fixed value. This is useful to continue performing other operations, like digital signature. The *fdCompareHash* function allows you to compare two hash context areas without the need to previously export each value.

fdCreateHash Function

C/C++ *int fdCreateHash (long *hid, int algorithm);*
PAS *fdCreateHash (Var hid: Longint; algorithm: Integer): Integer;*
VB3 *fdCreateHash (hid As Long, ByVal algorithm As Integer) As Integer*
VB5 *fdCreateHash (hid As Long, ByVal algorithm As Long) As Long*
FOX *Integer fdCreateHash Integer @hid, Integer algorithm*

Initializes a hash calculation. It creates a blank hash context area for a hashing algorithm. It returns a handle to the created hash context area. The context area is available to be updated with the *fdHashBuffer, fdHashNumber, fdHashFile* functions.

Inputs
algorithm: Constant that identifies the hashing algorithm of the created context area. Consult Appendix B for a list of all available algorithms and their associated constants.

Outputs
hid: Handle of the created hash context area.

fdDestroyHash Function

C/C++ *int fdDestroyHash (long hid);*
PAS *fdDestroyHash (Var hid: Longint): Integer;*
VB3 *fdDestroyHash (ByVal hid As Long) As Integer*
VB5 *fdDestroyHash (ByVal hid As Long) As Long*
FOX *Integer fdDestroyHash Integer hid*

Destroys a hash context area, freeing occupied resources.

Inputs
hid: Handle to the hash context area that you want to destroy.

fdInfoHash Function

C/C++ *int fdInfoHash (long hid, int *algorithm, int *digestSize);*
PAS *fdInfoHash (hid: Longint; Var algorithm: Integer; Var digestSize: Integer): Integer;*
VB3 *fdInfoHash (ByVal hid As Long, algorithm As Integer, digestSize As Integer) As Integer*
VB5 *fdInfoHash (ByVal hid As Long, algorithm As Longr, digestSize As Long) As Long*
FOX *Integer fdInfoHash Integer hid, Integer @algorithm, Integer @digestSize*

Returns information about a hash context area. It informs the type of hash algorithm used and the size of the digest that is produced.

Inputs
hid: Handle to the hash context area from which data will be obtained.

Outputs
algorithm: Hashing algorithm associated to that context area. Consult Appendix B for a list of all available algorithms and their associated constants.
digestSize: Size in bytes occupied by the final digest of the associated algorithm to that context area. The size refers to the stored digest in binary format. They are normally 16 or 20 bytes.

fdImportHash Function

C/C++ *int fdImportHash (long *hid, int algorithm, char *digest, int cod);*
PAS *fdImportHash (Var hid: Longint; algorithm: Integer; digest: PChar; cod: Integer): Integer;*
VB3 *fdImportHash (hid As Long, ByVal algorithm As Integer, ByVal digest As String, ByVal cod As Integer) As Integer*
VB5 *fdImportHash (hid As Long, ByVal algorithm As Long, ByVal digest As String, ByVal cod As Long) As Long*
FOX *Integer fdImportHash Integer hid, Integer algorithm, String @digest, Integer cod*

Initializes a hash. It creates a fixed hash context area for a hashing algorithm. It returns a handle to the created hash context area. The context area is initiated with a digest and cannot be updated, but can be signed or compared to other areas.

Inputs

algorithm: Constant that identifies the hashing algorithm of the created context area. Consult Appendix B for a list of all available algorithms and their associated constants.
digest: Pointer to a buffer that contains the digest with which the created context area is initialized. The buffer size will depend on the specified algorithm and format mask.
cod: Constant that identifies how a digest is coded. Consult Appendix B for a list of all available format masks and their associated constants.

Outputs
hid: Handle to the created hash context area.

fdExportHash Function

C/C++ *int fdExportHash (long hid, char *digest, int cod);*
PAS *fdExportHash (hid: Longint; digest: PChar; cod: Integer): Integer;*
VB3 *fdExportHash (ByVal hid As Long, ByVal digest As String, ByVal cod As Integer) As Integer*
VB5 *fdExportHash (ByVal hid As Long, ByVal digest As String, ByVal cod As Long) As Long*
FOX *Integer fdExportHash Integer hid, String @digest, Integer cod*

Calculates and returns the corresponding digest to a hash context area. The context area remains fixed and cannot continue to be updated.

Inputs
hid: Handle to a hash context area to be exported.
cod: Constant that identifies how a digest is coded. Consult Appendix B for a list of all available format masks and their associated constants.

Outputs
digest: Pointer to a buffer that returns the calculated digest. The buffer should have an adequate size in order to be able to contain the digest. The buffer size will depend on the algorithm and conversion format. The size can be obtained by using the *fdInfoHash* function.

fdCompareHash Function

C/C++ *int fdCompareHash (long hid1, long hid2);*
PAS *fdCompareHash (hid1: Longint; hid2: Longint): Integer;*
VB3 *fdCompareHash (ByVal hid1 As Long, ByVal hid2 As Long) As Integer*
VB5 *fdCompareHash (ByVal hid1 As Long, ByVal hid2 As Long) As Long*
FOX *Integer fdCompareHash Integer hid1, Integer hid2*

Calculates and compares digests from two hash context areas. The context areas remain fixed and cannot continue to be updated. If the digests are different, the status code **FD_ERR_HASHNOTEQUAL** is returned.

Inputs
hid1: Handle to the first hash context area to be compared.
hid2: Handle to the second hash context area to be compared.

fdHashBuffer Function

C/C++ *int fdHashBuffer (long hid, char *buf, unsigned int len);*
PAS *fdHashBuffer (hid: Longint; buf: PChar; len: Cardinal): Integer;*
VB3 *fdHashBuffer (ByVal hid As Long, ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdHashBuffer (ByVal hid As Long, ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdHashBuffer Integer hid, String @buf, Integer _len*

Updates a hash context area with a variable length buffer. The context area must be available in order to be updated.

Inputs
hid: Handle to the hash context area to be updated.
buf: Pointer to a variable length buffer.
len: Size of specified buffer.

fdHashNumber Function

C/C++ *int fdHashNumber (long hid, long num);*
PAS *fdHashNumber (hid: Longint; num: Longint): Integer;*
VB3 *fdHashNumber (ByVal hid As Long, ByVal num As Long) As Integer*
VB5 *fdHashNumber (ByVal hid As Long, ByVal num As Long) As Long*
FOX *Integer fdHashNumber Integer hid, Long num*

Updates a context area with a number. The context area must be available to be updated. This function is provided to guarantee that the hash of a number is not affected by the underlying architecture (big-endian / little-endian and processor word size).

Inputs
hid: Handle to the hash context area to be updated.
num: Number used to feed hash.

Generating Symmetric Keys Functions

Symmetric keys can be created in three different ways. The *fdGenerateKey* function allows you to generate random session keys. The *fdDeriveKey* function allows you to create keys whose function is derived by calculating the hash to a buffer. And the *fdImportKey* function allows you to create keys specified by the user. In the following section the functions are specified for encrypting and decrypting. Some operation modes require a initialization vector. The following section describes the necessary functions to associate initialization vectors to symmetric keys. Once used, the symmetric keys are destroyed with the *fdDestroyKey* function. The value of a key can also be recovered with the *fdExportKey* function. The *fdInfoKey* function allows you to dynamically obtain data from a context area and the *fdCompareKey* function allows you to verify if two keys are the same.

fdGenerateKey Function

C/C++ *int fdGenerateKey (long *kid, int algorithm);*
PAS *fdGenerateKey (Var kid: Longint; algorithm: Integer): Integer;*
VB3 *fdGenerateKey (kid As Long, ByVal algorithm As Integer) As Integer*
VB5 *fdGenerateKey (kid As Long, ByVal algorithm As Long) As Long*
FOX *Integer fdGenerateKey Integer @kid, Integer algorithm*

Generates a random symmetric key. It creates a key context area for a symmetric encryption algorithm. It returns a handle to the created key context area. The key is generated randomly and is guaranteed not to be a weak key. The generated key also has an associated initialization vector that is generated randomly.

Inputs
algorithm: Constant that identifies the symmetric encryption algorithm of the created context area. Consult Appendix B for a list of all available algorithms and their associated constants.

Outputs
kid: Handle to the created key context area.

fdDeriveKey Function

C/C++ *int fdDeriveKey (long *kid, int algorithm, char *buf, unsigned int len, int hash);*
PAS *fdDeriveKey (Var kid: Longint; algorithm: Integer; buf: PChar; len: Cardinal; hash: Integer): Integer;*
VB3 *fdDeriveKey (kid As Long, ByVal algorithm As Integer, ByVal buf As String, ByVal _len As Integer, ByVal hash As Integer) As Integer*
VB5 *fdDeriveKey (kid As Long, ByVal algorithm As Long, ByVal buf As String, ByVal _len As Long, ByVal hash As Long) As Long*
FOX *Integer fdDeriveKey Integer @kid, Integer algorithm, String @buf, Integer _len, Integer hash*

Generates a symmetric key from a passphrase. It creates a key context area for a symmetric encryption algorithm. It returns a handle to the created key context area. The key is generated by calculating a hash of a buffer. The size of the hash must be greater or equal to the size of the key to be generated. If the size of the hash exceeds the size of the key, the remaining bits are used to define an initialization vector.

Inputs

algorithm: Constant that identifies the symmetric encryption algorithm of the created context area. Consult Appendix B for a list of all available algorithms and their associated constants.
buf: Pointer to a variable length buffer.
len: Size of specified buffer.
hash: Constant that identifies the hashing algorithm that will be used to derive the symmetric key. Consult Appendix B for a list of all available algorithms and their associated constants.

Outputs
kid: Handle to the created key context area.

fdDestroyKey Function

C/C++	*int fdDestroyKey (long kid);*
PAS	*fdDestroyKey (kid: Longint): Integer;*
VB3	*fdDestroyKey (ByVal kid As Long) As Integer*
VB5	*fdDestroyKey (ByVal kid As Long) As Long*
FOX	*Integer fdDestroyKey Integer kid*

Destroys a key context area, freeing occupied resources.

Inputs
kid: Handle to the key context area that you want to destroy.

fdInfoKey Function

C/C++	*int fdInfoKey (long kid, int *algorithm, int *keySize);*
PAS	*fdInfoKey (kid: Longint; Var algorithm: Integer; Var keySize: Integer): Integer;*
VB3	*fdInfoKey (ByVal kid As Long, algorithm As Integer, keySize As Integer) As Integer*
VB5	*fdInfoKey (ByVal kid As Long, algorithm As Long, keySize As Long) As Long*
FOX	*Integer fdInfoKey Integer kid, Integer @algorithm, Integer @keySize*

Returns information about a key context area. It informs the type of symmetric algorithm used and the size of the key.

Inputs
kid: Handle to the key context area from where data will be obtained.

Outputs
algorithm: Symmetric encryption algorithm associated to that context area. Consult appendix B for a list of all available algorithms and their associated constants.
keySize: Size in bytes occupied by the symmetric key of the algorithm associated to that context area. The size refers to the stored key in binary format. They are normally 8, 16 or 24 bytes.

fdImportKey Function

C/C++ *int fdImportKey (long *kid, int algorithm, char *key, int cod);*
PAS *fdImportKey (Var kid: Longint; algorithm: Integer; key: PChar; cod: Integer): Integer;*
VB3 *fdImportKey (kid As Long, ByVal algorithm As Integer, ByVal key As String, ByVal cod As Integer) As Integer*
VB5 *fdImportKey (kid As Long, ByVal algorithm As Long, ByVal key As String, ByVal cod As Long) As Long*
FOX *Integer fdImportKey Integer @kid, Integer algorithm, String @key, Integer cod*

Loads a symmetric key. It creates a key context area for a symmetric encryption algorithm. It returns a handle to the created key context area. The context area is initialized with the specified key by the user. This key does not possess an associated initialization vector.

Inputs
algorithm: Constant that identifies the symmetric encryption algorithm of the created context area. Consult appendix B for a list of all available algorithms and their associated constants.
key: Pointer to a buffer that contains the symmetric key with which the created context area will initialize. The size of the buffer will depend on the specified algorithm and format mask.
cod: Constant that identifies how the symmetric key is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
kid: Handle to the created key context area.

fdExportKey Function

C/C++ *int fdExportKey (long kid, char *key, int cod);*
PAS *fdExportKey (kid: Longint; key: PChar; cod: Integer): Integer;*
VB3 *fdExportKey (ByVal kid As Long, ByVal key As String, ByVal cod As Integer) As Integer*
VB5 *fdExportKey (ByVal kid As Long, ByVal key As String, ByVal cod As Longr) As Long*
FOX *Integer fdExportKey Integer kid, String @key, Integer cod*

Returns the symmetric key that corresponds to a key context area.

Inputs
kid: Handle to the key context area to be exported.
cod: Constant that identifies how the symmetric key is coded. Consult appendix B for a list of all available format masks and their associated constants.

Output
key: Pointer to a buffer where the symmetric key is returned. The buffer should have an adequate size in order to contain that key. The size of the buffer will depend on the specified algorithm and format mask. The size can be obtained using the *fdInfoKey* function.

fdCompareKey Function

C/C++ *int fdCompareKey (long kid1, long kid2);*
PAS *fdCompareKey (kid1: Longint; kid2:Longint): Integer;*
VB3 *fdCompareKey (ByVal kid1 As Long, ByVal kid2 As Long) As Integer*
VB5 *fdCompareKey (ByVal kid1 As Long, ByVal kid2 As Long) As Long*
FOX *Integer fdCompareKey Integer kid1, Integer kid2*

Compares the symmetric keys of two key context areas. If the keys are different the status code **FD_ERR_KEYNOTEQUAL** is returned.

Inputs
kid1: Handle to the first key context area to be compared.
kid2: Handle to the second key context area to be compared.

Encryption with Symmetric Keys Functions

To encrypt or decrypt with symmetric keys, three types of functions are provided. The *fdEncryptBlock* and *fdDecryptBlock* functions allow you to encrypt single blocks in memory. The *fdEncyrptBuffer* and *fdDecryptBuffer* functions allow you to encrypt arbitrary sized buffers using different operation modes and padding methods. For these functions it may be necessary to associate an initialization vector to the key used. In the following section the necessary functions are described. Finally, the *fdEncryptFile* and *fdDecryptFile* functions allow you to encrypt files (see section on file functions).

fdEncryptBlock Function

C/C++ *int fdEncryptBlock (long kid, char *out, char *in);*
PAS *fdEncryptBlock (kid: Longint; _out: PChar; _in: PChar): Integer;*
VB3 *fdEncryptBlock (ByVal kid As Long, ByVal _out As String, ByVal _in As String) As Integer*
VB5 *fdEncryptBlock (ByVal kid As Long, ByVal _out As String, ByVal _in As String) As Long*
FOX *Integer fdEncryptBlock Integer kid, String @out, String @ByVal in*

Encrypts a single block with a symmetric key. It returns an encrypted block of the same length. Being a single block, it is not necessary to specify operation mode nor padding method.

Inputs
kid: Handle to the symmetric key context area.
in: Pointer to a buffer containing a single block (8 bytes) to be encrypted.

Outputs
out: Pointer to a buffer where an encrypted block is returned.

fdDecryptBlock Function

C/C++ *int fdDecryptBlock (long kid, char *out, char *in);*
PAS *fdDecryptBlock (kid: Longint; _out: PChar; _in: PChar): Integer;*
VB3 *fdDecryptBlock (ByVal kid As Long, ByVal _out As String, ByVal _in As String) As Integer*
VB5 *fdDecryptBlock (ByVal kid As Long, ByVal _out As String, ByVal _in As String) As Long*
FOX *Integer fdDecryptBlock Integer kid, String @out, String @ByVal in*

Decrypts a single block with a symmetric key. It returns a decrypted block of the same length. Being a single block, it is not necessary to specify operation mode nor padding method.

Inputs
kid: Handle to the symmetric key context area.
in: Pointer to a buffer containing a single block (8 bytes) to be decrypted.

Outputs
out: Pointer to a buffer where an decrypted block is returned.

fdEncryptBuffer Function

C/C++ *int fdEncryptBuffer (long kid, char *out, unsigned int *olen, char *in, unsigned int ilen, int mode, int pad);*
PAS *fdEncryptBuffer (kid: Longint; _out: PChar; Var olen: Cardinal; _in: PChar; ilen: Cardinal; mode: Integer; pad: Integer): Integer;*
VB3 *fdEncryptBuffer (ByVal kid As Long, ByVal _out As String, olen As Integer, ByVal _in As String, ByVal ilen As Integer, ByVal mode As Integer, ByVal pad As Integer) As Integer*
VB5 *fdEncryptBuffer (ByVal kid As Long, ByVal _out As String, olen As Long, ByVal _in As String, ByVal ilen As Long, ByVal mode As Long, ByVal pad As Long) As Long*
FOX *Integer fdEncryptBuffer Integer kid, String @out, Integer @olen, String @in, Integer ilen, Integer mode, Integer pad*

Encrypts an arbitrary sized buffer with a symmetric key. Returns an encrypted buffer together with its corresponding length. For a description of how the different operation modes and padding methods operate, consult Combining Operation Modes with Padding in Section 2.

Inputs
kid: Handle to the symmetric key context area.
in: Pointer to a variable length buffer with the data to be encrypted.
ilen: Size of the specified input buffer.
mode: Constant that identifies the operation mode to be used. Consult appendix B for a list of all available operation modes and their associated constants.
pad: Constant that identifies the padding method to be used. Consult appendix B for a list of all available padding methods and their associated constants.

Outputs
out: Pointer to a buffer where the encrypted buffer is returned.
olen: Length of the encrypted buffer

fdDecryptBuffer Function

C/C++ *int fdDecryptBuffer (long kid, char *out, unsigned int *olen, char *in, unsigned int ilen, int mode, int pad);*
PAS *fdDecryptBuffer (kid: Longint; _out: PChar; Var olen: Cardinal; _in: PChar; ilen: Cardinal; mode: Integer; pad: Integer): Integer;*
VB3 *fdDecryptBuffer (ByVal kid As Long, ByVal _out As String, olen As Integer, ByVal _in As String, ByVal ilen As Integer, ByVal mode As Integer, ByVal pad As Integer) As Integer*
VB5 *fdDecryptBuffer (ByVal kid As Long, ByVal _out As String, olen As Long, ByVal _in As String, ByVal ilen As Long, ByVal mode As Long, ByVal pad As Long) As Long*
FOX *Integer fdDecryptBuffer Integer kid, String @out, Integer @olen, String @in, Integer ilen, Integer mode, Integer pad*

Decrypts an arbitrary sized buffer with a symmetric key. It returns a decrypted buffer together with its corresponding length. For a description of how the different operation modes and padding methods operate, consult Combining Operation Modes with Padding in Section 2.

Inputs
kid: Handle to the symmetric key context area.
in: Pointer to a variable length buffer with the data to be decrypted.
ilen: Size of the specified input buffer.
mode: Constant that identifies the operation mode to be used. Consult appendix B for a list of all available operation modes and their associated constants.

pad: Constant that identifies the padding method to be used. Consult appendix B for a list of all available padding methods and their associated constants.

Outputs
out: Pointer to a buffer where the decrypted buffer is returned.
olen: Length of the original buffer

Initialization Vectors Management Functions

For all feedback modes it is necessary to specify an initialization vector associated to the symmetric key to be used. The *fdGenerateIV* function generates a random initialization vector. The *fdImportIV* and *fdExportIV* functions permit you to specify or obtain an initialization vector. Since initialization vectors are modified during encryption, the *fdResetIV* function allows you to return an initialization vector to its original state.

fdGenerateIV Function

C/C++ *int fdGenerateIV (long kid);*
PAS *fdGenerateIV (kid: Longint): Integer;*
VB3 *fdGenerateIV (ByVal kid As Long) As Integer*
VB5 *fdGenerateIV (ByVal kid As Long) As Long*
FOX *Integer fdGenerateIV Integer kid*

Generates a random initialization vector for a symmetric key.

Inputs
kid: Handle to the key context area to which an initialization vector will be affixed.

fdResetIV Function

C/C++ *int fdResetIV (long kid);*
PAS *fdResetIV (kid: Longint): Integer;*
VB3 *fdResetIV (ByVal kid As Long) As Integer*
VB5 *fdResetIV (ByVal kid As Long) As Long*
FOX *Integer fdResetIV Integer kid*

Synchronizes the initialization vector of a symmetric key to its initial position.

Inputs
kid: Handle to the symmetric key context area.

fdImportIV Function

C/C++ *int fdImportIV (long kid, char *iv, int cod);*
PAS *fdImportIV (kid: Longint; iv: PChar; cod: Integer): Integer;*
VB3 *fdImportIV (ByVal kid As Long, ByVal iv As String, ByVal cod As Integer) As Integer*
VB5 *fdImportIV (ByVal kid As Long, ByVal iv As String, ByVal cod As Long) As Long*
FOX *Integer fdImportIV Integer kid, String @iv, Integer cod*

Specifies an initialization vector for a symmetric key. Initialization vectors have the same size as the block (8 bytes).

Inputs
kid: Handle to the key context area.
iv: Pointer to a buffer that contains the initialization vector. The size of the buffer will depend on the specified format mask.
cod: Constant that identifies how the initialization vector is coded. Consult appendix B for a list of all available format masks and their associated constants.

fdExportIV Function

C/C++ *int fdExportIV (long kid, char *iv, int cod);*
PAS *fdExportIV (kid: Longint; iv: PChar; cod: Integer): Integer;*
VB3 *fdExportIV (ByVal kid As Long, ByVal iv As String, ByVal cod As Integer) As Integer*
VB5 *fdExportIV (ByVal kid As Long, ByVal iv As String, ByVal cod As Long) As Long*
FOX *Integer fdExportIV Integer kid, String @iv, Integer cod*

Returns the initialization vector of a symmetric key. The initialization vectors have the same size as the block (8 bytes). Although the successive encryptions modify the value of the initialization vector, this function always returns the original initialization vector value.

Inputs
kid: Handle to the key context area.
cod: Constant that identifies how the initialization vector is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
iv: Pointer to a buffer where the initialization vector is returned. The size of the buffer will depend on the specified format mask.

MAC Generation and Verification Functions

The *fdGenerateMAC* and *fdVerifyMAC* functions allow you calculate and verify a MAC of a buffer. They are useful when only authentication and high processing speed are required. In order to calculate a MAC it is required to specify a feedback operation mode. The operation modes allowed are: CBC, CFB and BCF. As a result, the symmetric keys must be associated to an initialization vector.

fdGenerateMAC Function

C/C++ *int fdGenerateMAC (long kid, char *mac, char *buf, unsigned int len, int mode, int size, int cod);*

PAS *fdGenerateMAC (kid: Longint; mac: PChar; buf : PChar; len: Cardinal; mode: Integer; size: Integer; cod: Integer): Integer;*

VB3 *fdGenerateMAC (kid As Long, ByVal mac As String, ByVal buf As String, ByVal _len as Integer, ByVal mode As Integer, ByVal size As Integer, ByVal cod As Integer) As Integer*

VB5 *fdGenerateMAC (kid As Long, ByVal mac As String, ByVal buf As String, ByVal _len as Long, ByVal mode As Long, ByVal size As Long, ByVal cod As Long) As Long*

FOX *Integer fdGenerateMAC Integer kid, String @mac, String @buf, Integer _len, Integer mode, Integer size, Integer cod*

Calculates the MAC of an arbitrary sized buffer using a symmetric key.

Inputs

kid: Handle to the symmetric key context area to be used.
buf: Pointer to a variable length buffer.
len: Size of the specified buffer.
mode: Constant that identifies the operation mode to be used. Only modes CBC, CFB and BCF are allowed. Consult appendix B for the operation modes and their associated constants.
size: Size of the MAC to be calculated. Consult appendix B for a list of available constants of MAC sizes.
cod: Constant that identifies how the MAC is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs

mac: Pointer to a buffer where the calculated MAC is returned. Buffer size will depend on MAC size and the specified format mask.

fdVerifyMAC Function

C/C++ *int fdVerifyMAC (long kid, char *mac, char *buf, unsigned int len, int mode, int size, int cod);*

PAS *fdVerifyMAC (kid: Longint; mac: PChar; buf : PChar; len: Cardinal; mode: Integer; size: Integer; cod: Integer): Integer;*

VB3 *fdVerifyMAC (kid As Long, ByVal mac As String, ByVal buf As String, ByVal _len as Integer, ByVal mode As Integer, ByVal size As Integer, ByVal cod As Integer) As Integer*

VB5 *fdVerifyMAC (kid As Long, ByVal mac As String, ByVal buf As String, ByVal _len as Long, ByVal mode As Long, ByVal size As Long, ByVal cod As Long) As Long*

FOX *Integer fdVerifyMAC Integer kid, String @mac, String @buf, Integer _len, Integer mode, Integer size, Integer cod*

Verifies the MAC of a arbitrary sized buffer using a symmetric key. If the MAC is not verified the status code **FD_ERR_MACNOTEQUAL** is returned.

Inputs
kid: Handle to the symmetric key context area to be used.
mac: Pointer to a buffer that contains a MAC to be verified. Buffer size will depend on MAC size and the specified format mask.
buf: Pointer to a variable length buffer.
len: Size of the specified buffer.
mode: Constant that identifies the operation mode to be used. Only modes CBC, CFB and BCF are allowed. Consult appendix B for the operation modes and their associated constants.
size: Size of the MAC to be verified. Consult appendix B for a list of available constants of MAC sizes.
cod: Constant that identifies how the MAC is coded. Consult appendix B for a list of all available format masks and their associated constants.

Generating Public and Private Keys Functions

To generate a pair of RSA keys, first you must invoke the *fdGeneratePrivateKey* function and then the public key is derived with the *fdDerivePublicKey* function. Keys can be exported or imported using the *fdImportRSAKey* and *fdExportRSAKey* functions. Once used,

RSA keys are destroyed with the *fdDestroyRSAKey* function. The *fdInfoRSA* function dynamically obtains data from a RSA Key.

fdGeneratePrivateKey Function

C/C++ *int fdGeneratePrivateKey (long *prv, int bits, unsigned long pubexp);*
PAS *fdGeneratePrivateKey (Var prv: Longint; bits: Integer; pubexp: Longint): Integer;*
VB3 *fdGeneratePrivateKey (prv As Long, ByVal bits As Integer, ByVal pubexp As Long) As Integer*
VB5 *fdGeneratePrivateKey (prv As Long, ByVal bits As Long, ByVal pubexp As Long) As Long*
FOX *Integer fdGeneratePrivateKey Long @prv, Integer bits, Long pubexp*

Generates a RSA private key. It creates a private key context area and returns a handle to the created context area. The key is generated randomly. Before invoking this function it is convenient to register a random buffer using *fdRegisterRandomBuffer*.

Inputs
bits: Key size in bits. It should be in the range of 512-8192 and be an even number.
pubexp: Determines the public exponent. You can use some of the defined constants or an arbitrary value. In that case it must be a number greater than or equal to 3 and not divisible by 2. Consult appendix B for the list of available constants.

Outputs
prv: Handle to the created private key context area.

fdDerivePublicKey Function

C/C++ *int fdDerivePublicKey (long *pub, long prv);*
PAS *fdDerivePublicKey (Var pub: Longint; prv: Longint): Integer;*
VB3 *fdDerivePublicKey (pub As Long, ByVal prv As Long) As Integer*
VB5 *fdDerivePublicKey (pub As Long, ByVal prv As Long) As Long*
FOX *Integer fdDerivePublicKey Long @pub, Long prv*

Derives the corresponding RSA public key to a private key. It creates a public key context area and returns a handle to the created context area.

Inputs

prv: Handle to the created private key context area.

Outputs
pub: Handle to the created public key context area.

fdDestroyRSAKey Function

C/C++ *int fdDestroyRSAKey (long pid);*
PAS *fdDestroyRSAKey (pid: Longint): Integer;*
VB3 *fdDestroyRSAKey (ByVal pid As Long) As Integer*
VB5 *fdDestroyRSAKey (ByVal pid As Long) As Long*
FOX *Integer fdDestroyRSAKey Long pid*

Destroys a key context area, public or private, freeing occupied resources.

Inputs
pid: Handle to the key context area that is to be destroyed.

fdInfoRSA Function

C/C++ *int fdInfoRSA (long pid, int *bits, int *type, int *size);*
PAS *fdInfoRSA (pid: Longint; Var bits: Integer; Var priv: Integer; Var size: Integer): Integer;*
VB3 *fdInfoRSA (ByVal pid As Long, bits As Integer, type As Integer, size As Integer) As Integer*
VB5 *fdInfoRSA (ByVal pid As Long, bits As Long, type As Long, size As Long) As Long*
FOX *Integer fdInfoRSA Long pid, Integer @bits, Integer @priv, Integer @size*

Returns information about a RSA key context area. It informs the length, key type and size in bytes necessary to store it.

Inputs
pid: Handle to the key context area from which data will be obtained.

Outputs
bits: Key size in bits.
type: If the key is public or private. Consult appendix B for the associated constants.

size: Size in bytes that the key occupies when exporting it. This value depends on the type and size of the key. The value corresponds to its storage in binary format.

fdExportRSAKey Function

C/C++ *int fdExportRSAKey (long pid, char *buf, int cod);*
PAS *fdExportRSAKey (pid: Longint; buf: PChar; cod: Integer): Integer;*
VB3 *fdExportRSAKey (ByVal pid As Long, ByVal buf As String, ByVal cod As Integer) As Integer*
VB5 *fdExportRSAKey (ByVal pid As Long, ByVal buf As String, ByVal cod As Long) As Long*
FOX *Integer fdExportRSAKey Long pid, String @buf, Integer cod*

Returns the corresponding RSA key to a key context area.

Inputs
pid: Handle to the RSA key context area to be exported.
cod: Constant that identifies how the key is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
buf: Pointer to a buffer where the RSA key is returned. The buffer must be of adequate size in order to be able to contain this key. The size of the buffer will depend on the key and format mask. The size can be obtained using the *fdInfoRSA* function.

fdImportRSAKey Function

C/C++ *int fdImportRSAKey (long *pid, char *buf, int cod);*
PAS *fdImportRSAKey (*pid: Longint; buf: PChar; cod: Integer): Integer;*
VB3 *fdImportRSAKey (pid As Long, ByVal buf As String, ByVal cod As Integer) As Integer*
VB5 *fdImportRSAKey (pid As Long, ByVal buf As String, ByVal cod As Long) As Long*
FOX *Integer fdImportRSAKey Long @pid, String @buf, Integer cod*

Loads a RSA key. It creates a key context area and returns a handle to the created context area.

Inputs
buf: Pointer to a buffer that contains the RSA key.

cod: Constant that identifies how the RSA key is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
pid: Handle to the created key context area.

fdSetRSAParam Function

C/C++ *int fdSetRSAParam (int n);*
PAS *fdSetRSAParam (n: Integer): Integer;*
VB3 *fdSetRSAParam (ByVal n As Integer) As Integer*
VB5 ***f****dSetRSAParam (ByVal n As Integer) As Long*
FOX *Integer fdSetRSAParam Integer n*

Allows you to fix RSA key generation system parameters.

Inputs
n: Error probability in the generation of prime numbers. The error probability will be less than 1 in 10^n. The value of n must be in the range of 14-50.

Generating Digital Signatures Functions

This section contains the encryption functions that require the use of a RSA private key. The procedure used can be consulted in the document PKCS #1. Three encryption functions are provided. The *fdSignBuffer* function allows you to encrypt an arbitrary buffer. The *fdSignHash* and *fdSignHashExtra* functions digitally sign a previously calculated hash. The result of these three functions is a digital signature. This signature can be exported with the *fdExportSignature* function and imported again with the *fdImportSignature* function. To obtain the size of the digital signature the *fdInfoSignature* function is provided. The signatures must be destroyed with the *fdDestroySignature* function.

fdSignBuffer Function

C/C++ *int fdSignBuffer (long *sid, char *buf, int len, long pid);*
PAS *fdSignBuffer (Var sid: Longint; buf: PChar; len: Integer; pid: Longint): Integer;*
VB3 *fdSignBuffer (sid As Long, ByVal buf As String, ByVal _len As Integer, ByVal pid As Long) As Integer*
VB5 *fdSignBuffer (sid As Long, ByVal buf As String, ByVal _len As Long, ByVal pid As Long) As Long*
FOX *Integer fdSignBuffer Long @sid, String @buf, Integer _len, Long pid*

Encrypts an arbitrary buffer with a private key generating a context area to a digital signature. The buffer length should be such that it can be encrypted with a single modular exponentiation.

Inputs
buf: Pointer to a variable length buffer with the data to be encrypted.
len: Size of specified buffer. It cannot exceed the key size minus 11 bytes.
pid: Handle to the created digital signature context area.

fdSignHash Function

C/C++ *int fdSignHash (long *sid, long hid, long pid);*
PAS *fdSignHash (Var sid: Longint; hid: Longint; pid: Longint): Integer;*
VB3 *fdSignHash (sid As Long, ByVal hid As Long, ByVal pid As Long) As Integer*
VB5 *fdSignHash (sid As Long, ByVal hid As Long, ByVal pid As Long) As Long*
FOX *Integer fdSignHash Long @sid, Long hid, Long pid*

Signs a hash with a private key generating a context area to a digital signature. The hash context area can no longer continue to be updated.

Inputs
hid: Handle to the hash context area to be signed.
pid: Handle to a private key context area.

Outputs
sid: Handle to the created digital signature context area.

fdSignHashExtra Function

C/C++ *int fdSignHashExtra (long *sid, long hid, long pid, char *buf, int len);*
PAS *fdSignHashExtra (Var sid: Longint; hid: Longint; pid: Longint; buf: PChar; len: Integer): Integer;*
VB3 *fdSignHashExtra (sid As Long, ByVal hid As Long, ByVal pid As Long, ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdSignHashExtra (sid As Long, ByVal hid As Long, ByVal pid As Long, ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdSignHashExtra Long @sid, Long hid, Long pid, String @buf, Integer _len*

Signs a hash with a private key generating a context area to a digital signature. The hash context area can no longer continue to be updated. Optionally, you can specify a buffer with data of the user. The sum of the hash and the user data must be such that it can be encrypted with a single modular exponentiation.

Inputs
hid: Handle to the hash context area to be signed.
pid: Handle to a private key context area.
buf: Pointer to a variable length buffer.
len: Size of specified buffer.

Outputs
sid: Handle to the created digital signature context area.

fdDestroySignature Function

C/C++ *int fdDestroySignature (long sid);*
PAS *fdDestroySignature (sid: Longint): Integer;*
VB3 *fdDestroySignature (ByVal sid As Long) As Integer*
VB5 *fdDestroySignature (ByVal sid As Long) As Long*
FOX *Integer fdDestroySignature Long sid*

Destroys a digital signature context area, freeing occupied resources.

Inputs
sid: Handle to the digital signature context area that is to be destroyed.

fdInfoSignature Function

C/C++ *int fdInfoSignature (long sid, int *size);*
PAS *fdInfoSignature (sid: Longint; Var size: Integer): Integer;*
VB3 *fdInfoSignature (ByVal sid As Long, size As Integer) As Integer*
VB5 *fdInfoSignature (ByVal sid As Long, size As Long) As Long*
FOX *Integer fdInfoSignature Long sid, Integer @size*

Returns information about a digital signature context area. It informs the size in bytes necessary to store it.

Inputs
sid: Handle to the digital signature context area from which data is to be obtained.

Outputs
size: Size in bytes that the signature occupies when exporting it. This value depends on the size of the private key used to produce the signature. The value corresponds to its storage in binary format.

fdExportSignature Function

C/C++ *int fdExportSignature (long sid, char *signature, int cod);*
PAS *fdExportSignature (sid: Longint; signature: PChar; cod: Integer): Integer;*
VB3 *fdExportSignature (ByVal sid As Long, ByVal signature As String, ByVal cod As Integer) As Integer*
VB5 *fdExportSignature (ByVal sid As Long, ByVal signature As String, ByVal cod As Long) As Long*
FOX *Integer fdExportSignature Long sid, String @signature, Integer cod*

Returns the corresponding digital signature to a context area.

Inputs
sid: Handle to the digital signature context area to be exported.
cod: Constant that identifies how the signature is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
signature: Pointer to a buffer where the digital signature is returned. The buffer must be of adequate size in order to contain this signature. The size of the buffer will depend on the

private key used to produce the signature and the format mask. The size can be obtained using the *fdInfoSignature* function.

fdImportSignature Function

C/C++ *int fdImportSignature (long *sid, char *signature, int cod);*
PAS *fdImportSignature (Var sid: Longint; signature: PChar; cod: Integer): Integer;*
VB3 *fdImportSignature (sid As Long, ByVal signature As String, ByVal cod As Integer) As Integer*
VB5 *fdImportSignature (sid As Long, ByVal signature As String, ByVal cod As Long) As Long*
FOX *Integer fdImportSignature Long @sid, String @signature, Integer cod*

Loads a digital signature. It creates a digital signature context area and returns a handle to the created context area.

Inputs
signature: Pointer to a buffer that contains the digital signature.
cod: Constant that identifies how the signature is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
sid: Handle to the created digital signature context area.

Generating Digital Envelopes Functions

This section contains the encryption functions that require the use of a RSA public key. The procedure used can be consulted in the document PKCS #1. Four encryption functions are provided. The *fdEnvelopBuffer* function allows you to encrypt an arbitrary buffer. The *fdEnvelopKey* and *fdEvelopKeyExtra* digitally envelop a symmetric key. The *fdEnvelopHash* function allows you to digitally envelop a previously calculated hash. The result of these functions is a digital envelope. This envelope can be exported with the *fdExportEnvelope* function and imported again with the *fdImportEnvelope* function. To obtain the size of a digital envelope the *fdInfoEnvelope* is provided. Digital Envelopes must be destroyed with the *fdDestroyEnvelope* function.

fdEnvelopBuffer Function

C/C++ *int fdEnvelopBuffer (long *sid, char *buf, int len, long pid);*
PAS *fdEnvelopBuffer (Var sid: Longint; buf: PChar; len: Integer; pid: Longint): Integer;*
VB3 *fdEnvelopBuffer (sid As Long, ByVal buf As String, ByVal _len As Integer, ByVal pid As Long) As Integer*
VB5 *fdEnvelopBuffer (sid As Long, ByVal buf As String, ByVal _len As Long, ByVal pid As Long) As Long*
FOX *Integer fdEnvelopBuffer Long @sid, String @buf, Integer _len, Long pid*

Encrypts an arbitrary buffer with a public key generating a digital envelope context area. The length of the buffer should be such that it can be encrypted with a singular modular exponentiation.

Inputs
buf: Pointer to a variable length buffer with the data to be encrypted.
len: Size of specified buffer. It cannot exceed the key size minus 11 bytes.
pid: Handle to a public key context area.

Outputs
sid: Handle to the created digital envelope context area.

fdEnvelopKey Function

C/C++ *int fdEnvelopKey (long *sid, long kid, long pid);*
PAS *fdEnvelopKey (Var sid: Longint; kid: Longint; pid: Longint): Integer;*
VB3 *fdEnvelopKey (sid As Long, ByVal kid As Long, ByVal pid As Long) As Integer*
VB5 *fdEnvelopKey (sid As Long, ByVal kid As Long, ByVal pid As Long) As Long*
FOX *Integer fdEnvelopKey Long @sid, Long kid, Long pid*

Encrypts a symmetric key with a public key generating a context area to a digital envelope. If the symmetric key has an associated initialization vector, it is also stored within the same digital envelope.

Inputs
kid: Handle to the key context area to be encrypted.
pid: Handle to the public key context area.

Outputs

sid: Handle to the created digital envelope context area.

fdEnvelopKeyExtra Function

C/C++ *int fdEnvelopKeyExtra (long *sid, long kid, long pid, char *buf, int len);*
PAS *fdEnvelopKeyExtra (Var sid: Longint; kid: Longint; pid: Longint; buf: PChar; len: Integer): Integer;*
VB3 *fdEnvelopKeyExtra (sid As Long, ByVal kid As Long, ByVal pid As Long, ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdEnvelopKeyExtra (sid As Long, ByVal kid As Long, ByVal pid As Long, ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdEnvelopKeyExtra Long @sid, Long kid, Long pid, String @buf, Integer _len*

Encrypts a symmetric key with a public key generating a context area to a digital envelope. Optionally, a buffer with data of the user can be specified. The sum of the key and the user data should be such that it can be encrypted with a single modular exponentiation.

Inputs
kid: Handle to the key context area to be encrypted.
pid: Handle to the public key context area.
buf: Pointer to a variable length buffer.
len: Size of the specified buffer.

Outputs
sid: Handle to the created digital envelope context area.

fdEnvelopHash Function

C/C++ *int fdEnvelopHash (long *sid, long hid, long pid);*
PAS *fdEnvelopHash (Var sid: Longint; hid: Longint; pid: Longint): Integer;*
VB3 *fdEnvelopHash (sid As Long, ByVal hid As Long, ByVal pid As Long) As Integer*
VB5 *fdEnvelopHash (sid As Long, ByVal hid As Long, ByVal pid As Long) As Long*
FOX *Integer fdEnvelopHash Long @sid, Long hid, Long pid*

Encrypts a hash with a public key generating a context area to a digital envelope. The hash context area can no longer continue to be updated.

Inputs

hid: Handle to the hash context area to be encrypted.
pid: Handle to the public key context area.

Outputs
sid: Handle to the created digital envelope context area.

fdDestroyEnvelope Function

C/C++ *int fdDestroyEnvelope (long sid);*
PAS *fdDestroyEnvelope (sid: Longint): Integer;*
VB3 *fdDestroyEnvelope (ByVal sid As Long) As Integer*
VB5 *fdDestroyEnvelope (ByVal sid As Long) As Long*
FOX *Integer fdDestroyEnvelope Long sid*

Destroys a digital envelope contest area, freeing occupied resources.

Inputs
sid: Handle to the created digital envelope context area that is to be destroyed.

fdInfoEnvelope Function

C/C++ *int fdInfoEnvelope (long sid, int *size);*
PAS *fdInfoEnvelope (sid: Longint; Var size: Integer): Integer;*
VB3 *fdInfoEnvelope (ByVal sid As Long, size As Integer) As Integer*
VB5 *fdInfoEnvelope (ByVal sid As Long, size As Long) As Long*
FOX *Integer fdInfoEnvelope Long sid, Integer @size*

Returns information about a digital envelope context area. It informs the size in bytes necessary to store it.

Inputs
sid: Handle to the created digital envelope context area from where the data will be obtained.

Outputs
size: Size in bytes that the exported envelope occupies. This value depends on the size of the public key used to produce the envelope. The value corresponds to its storage in binary format.

fdExportEnvelope Function

C/C++ *int fdExportEnvelope (long sid, char *envelope, int cod);*
PAS *fdExportEnvelope (sid: Longint; envelope: PChar; cod: Integer): Integer;*
VB3 *fdExportEnvelope (ByVal sid As Long, ByVal envelope As String, ByVal cod As Integer) As Integer*
VB5 *fdExportEnvelope (ByVal sid As Long, ByVal envelope As String, ByVal cod As Long) As Long*
FOX *Integer fdExportEnvelope Long sid, String @envelope, Integer cod*

Returns the corresponding digital envelope to a context area.

Inputs
sid: Handle to the digital envelope context area to be exported.
cod: Constant that identifies how the envelope is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
envelope: Pointer to a buffer where the digital envelope is returned. The buffer must be of adequate size in order to contain this envelope. The size of the buffer will depend on the public key used to produce the envelope and the format mask. The size can be obtained using the *fdInfoEnvelope* function.

fdImportEnvelope Function

C/C++ *int fdImportEnvelope (long *sid, char *envelope, int cod);*
PAS *fdImportEnvelope (*sid: Longint; envelope: PChar; cod: Integer): Integer;*
VB3 *fdImportEnvelope (sid As Long, ByVal envelope As String, ByVal cod As Integer) As Integer*
VB5 *fdImportEnvelope (sid As Long, ByVal envelope As String, ByVal cod As Long) As Long*
FOX *Integer fdImportEnvelope Long @sid, String @envelope, Integer cod*

Loads a digital envelope. It creates a digital envelope context area and returns a handle to the created context area.

Inputs
envelope: Pointer to a buffer that contains the digital envelope

cod: Constant that identifies how the envelope is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
sid: Handle to the created digital envelope context area.

Digital Signatures and Envelopes Verification Functions

This section contains the decryption functions that require the use of RSA public and private keys. The procedure used can be consulted in the document PKCS #1. The *fdRecoverBuffer* function allows you to decrypt an encrypted buffer with a public or private key. The *fdRecoverHash* and *fdRecoverHashExtra* functions recover a hash while the *fdRecoverKey* and *fdRecoverKeyExtra* functions recover a symmetric key.

fdRecoverBuffer Function

C/C++ *int fdRecoverBuffer (char *buf, int *len, long sid, long pid);*
PAS *fdRecoverBuffer (buf: PChar; Var len: Integer; sid: Longint; pid: Longint): Integer;*
VB3 *fdRecoverBuffer (ByVal buf As String, _len As Integer, ByVal sid As Long, ByVal pid As Long) As Integer*
VB5 *fdRecoverBuffer (ByVal buf As String, _len As Long, ByVal sid As Long, ByVal pid As Long) As Long*
FOX *Integer fdRecoverBuffer String @buf, Integer @len, Long sid, Long pid*

Decrypts a digital signature or envelope using a public or private key, respectively. It returns the original buffer with its corresponding length.

Inputs
sid: Handle to the digital signature or envelope context area to be decrypted.
pid: Handle to the RSA key context area.

Outputs
buf: Pointer to a buffer where the decrypted data is returned.
len: Size of the original buffer.

fdRecoverHash Function

C/C++ *int fdRecoverHash (long *hid, long sid, long pid);*
PAS *fdRecoverHash (Var hid: Longint; sid: Longint; pid: Longint): Integer;*
VB3 *fdRecoverHash (hid As Long, ByVal sid As Long, ByVal pid As Long) As Integer*
VB5 *fdRecoverHash (hid As Long, ByVal sid As Long, ByVal pid As Long) As Long*
FOX *Integer fdRecoverHash Long @hid, Long sid, Long pid*

Recovers a hash from a digital signature or envelope using a corresponding public or private key. It returns a handle to the created hash context area. The context area is initialized with a digest and cannot be updated, but can be signed or compared to other areas.

Inputs
sid: Handle to the digital signature or envelope context area to be decrypted.
pid: Handle to the RSA key context area.

Outputs
hid: Handle to the created hash context area.

fdRecoverHashExtra Function

C/C++ *int fdRecoverHashExtra (long *hid, char *buf, int *len, long sid, long pid);*
PAS *fdRecoverHashExtra (Var hid: Longint; buf: PChar; Var len: Integer; sid: Longint; pid: Longint): Integer;*
VB3 *fdRecoverHashExtra (hid As Long, ByVal buf As String, _len As Integer, ByVal sid As Long, ByVal pid As Long) As Integer*
VB5 *fdRecoverHashExtra (hid As Long, ByVal buf As String, _len As Long, ByVal sid As Long, ByVal pid As Long) As Long*
FOX *Integer fdRecoverHashExtra Long @hid, String @buf, Integer @len, Long sid, Long pid*

Recovers a hash from a digital signature or envelope using a corresponding public or private key. It returns a handle to the created hash context area. The context area is initialized with a digest and cannot be updated, but can be signed or compared to other areas. Moreover, it returns a buffer with user data in the case that some data exists.

Inputs
sid: Handle to the digital signature or envelope context area to be decrypted.
pid: Handle to the RSA key context area.

Outputs
hid: Handle to the created hash context area.
buf: Pointer to a buffer where the user data is returned.
len: Size of the buffer with user data. In the case that there is none, zero length is returned.

fdRecoverKey Function

C/C++ *int fdRecoverKey (long *kid, long sid, long pid);*
PAS *fdRecoverKey (Var kid: Longint; sid: Longint; pid: Longint): Integer;*
VB3 *fdRecoverKey (kid As Long, ByVal sid As Long, ByVal pid As Long) As Integer*
VB5 *fdRecoverKey (kid As Long, ByVal sid As Long, ByVal pid As Long) As Long*
FOX *Integer fdRecoverKey Long @kid, Long sid, Long pid*

Recovers a symmetric key from a digital envelope using a private key. It returns a handle to the created symmetric key context area. If the digital envelope contains an initialization vector, then this is recovered and associated to the symmetric key.

Inputs
sid: Handle to the digital envelope context area to be decrypted.
pid: Handle to the private key context area.

Outputs
kid: Handle to the created symmetric key context area.

fdRecoverKeyExtra Functions

C/C++ *int fdRecoverKeyExtra (long *kid, char *buf, int *len, long sid, long pid);*
PAS *fdRecoverKeyExtra (Var kid: Longint; buf: PChar; Var len: Integer; sid: Longint; pid: Longint): Integer;*
VB3 *fdRecoverKeyExtra (kid As Long, ByVal buf As String, _len As Integer, ByVal sid As Long, ByVal pid As Long) As Integer*
VB5 *fdRecoverKeyExtra (kid As Long, ByVal buf As String, _len As Long, ByVal sid As Long, ByVal pid As Long) As Long*
FOX *Integer fdRecoverKeyExtra Long @kid, String @buf, Integer @len, Long sid, Long pid*

Recovers a symmetric key from a digital envelope using a private key. It returns a handle to the created symmetric key context area. Moreover, returns a buffer with user data in the case that there is user data.

Inputs
sid: Handle to the digital envelope context area to be decrypted.
pid: Handle to the private key context area.

Outputs
kid: Handle to the created symmetric key context area.
buf: Pointer to a buffer where the user data is returned.
len: Size of the buffer with user data. In the case that there is none, zero length is returned.

Data Compression Functions

The *fdCompressBuffer* and *fdExpandBuffer* functions allow you to compress and expand a buffer in memory. If the original text does not possess the necessary redundancy in order to be reduced, it is returned without changes. In either case a header is added in order to be able to distinguish the situation. The *fdInfoCompress* function obtains data from a compressed buffer.

fdCompressBuffer Function

C/C++ *int fdCompressBuffer (char *out, unsigned int *olen, char *in, unsigned int ilen, int algorithm);*
PAS *fdCompressBuffer (_out: PChar; Var olen: Cardinal; _in: PChar; ilen: Cardinal; algorithm: Integer): Integer;*
VB3 *fdCompressBuffer (ByVal obuf As String, olen As Integer, ByVal ibuf As String, ByVal ilen As Integer, ByVal algorithm As Integer) As Integer*
VB5 *fdCompressBuffer (ByVal obuf As String, olen As Long, ByVal ibuf As String, ByVal ilen As Long, ByVal algorithm As Long) As Long*
FOX *Integer fdCompressBuffer String @obuf, Integer @olen, String @ibuf, Integer ilen, Integer algorithm*

Compresses an arbitrary sized buffer. It returns a compressed buffer together with its corresponding length. If the buffer cannot be compressed the original buffer is returned with a four byte header. This is the maximum possible size of the output buffer.

Inputs
in: Pointer to a variable length buffer with data to be compressed.
ilen: Size of the specified input buffer.
algorithm: Constant that identifies the compression algorithm to be used. Consult appendix B for a list of all available algorithms and their associated constants.

Outputs
out: Pointer to a buffer where the compressed buffer is returned.
olen: Length of the compressed buffer.

fdExpandBuffer Function

C/C++ *int fdExpandBuffer (char *out, unsigned int *olen, char *in, unsigned int ilen, int algorithm);*
PAS *fdExpandBuffer (_out: PChar; Var olen: Cardinal; _in: PChar; ilen: Cardinal; algorithm: Integer): Integer;*
VB3 *fdExpandBuffer (ByVal obuf As String, olen As Integer, ByVal ibuf As String, ByVal ilen As Integer, ByVal algorithm As Integer) As Integer*
VB5 *fdExpandBuffer (ByVal obuf As String, olen As Long, ByVal ibuf As String, ByVal ilen As Long, ByVal algorithm As Long) As Long*
FOX *Integer fdExpandBuffer String @obuf, Long @olen, String @ibuf, Integer ilen, Integer algorithm*

Expands a compressed buffer to its original size. It returns the original buffer and its length. The *fdInfoCompress* function allows you to know the original size of a compressed buffer.

Inputs
in: Pointer to a compressed buffer.
ilen: Size of the compressed buffer.
algorithm: Constant that identifies the compression algorithm to be used. Consult appendix B for a list of all available algorithms and their associated constants.

Outputs
out: Pointer to a buffer where the original buffer is returned.
olen: Length of the original buffer.

fdInfoCompress Function

C/C++ *int fdInfoCompress (unsigned int *size, int *compress, char *buf, unsigned int len);*
PAS *fdInfoCompress (Var size: Cardinal; Var compress: Integer; buf: PChar; len: Cardinal): Integer;*
VB3 *fdInfoCompress (size As Integer, compress As Integer, ByVal buf As String, ByVal_ len As Integer) As Integer*
VB5 *fdInfoCompress (size As Long, compress As Long, ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdInfoCompress String Integer @size, Integer @compress, String @buf, Integer_ len*

Returns information about a compressed buffer. It informs the original size and data about its compression.

Inputs
buf: Pointer to a compressed buffer.
ilen: Size of the compressed buffer.

Outputs
size: Size that the original buffer occupies.
compress: Returns “true” if the buffer is truly compressed.

Miscellaneous Functions

This section contains diverse functions that allow you to perform conversions or administer memory in a secure manner.

fdRegisterRandomBuffer Function

C/C++ *int fdRegisterRandomBuffer (char *buf, unsigned int len);*
PAS *fdRegisterRandomBuffer (buf: PChar; len: Cardinal): Integer;*
VB3 *fdRegisterRandomBuffer (ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdRegisterRandomBuffer (ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdRegisterRandomBuffer String @buf, Integer _len*

Registers a buffer with random bits generated externally. These bits will combine with random number generation routines to produce random numbers cryptographically secure. This function should be invoked especially before generating RSA keys.

Inputs
buf: Pointer to a buffer containing random bits.
len: Size of supplied buffer.

fdGenerateRandom Function

C/C++ *int fdGenerateRandom (char *buf, unsigned int len);*
PAS *fdGenerateRandom (buf: PChar; len: Cardinal): Integer;*
VB3 *fdGenerateRandom (ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdGenerateRandom (ByVal buf As String, ByVal _len As long) As Long*
FOX *Integer fdGenerateRandom String @buf, Integer _len*

Fills a buffer with random bits.

Inputs
len: Size of buffer to be padded.

Outputs
buf: Pointer to a buffer to be padded.

fdWipeBuffer Function

C/C++ *int fdWipeBuffer (char *buf, unsigned int len);*
PAS *fdWipeBuffer (buf: PChar; len: Cardinal): Integer;*
VB3 *fdWipeBuffer (ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdWipeBuffer (ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdWipeBuffer String @buf, Integer _len*

Overwrites a buffer with pseudorandom bits, destroying its content.

Inputs
len: Size of buffer to be overwritten.

Outputs
buf: Pointer to a buffer to be overwritten.

fdConvertBuffer Function

C/C++ *int fdConvertBuffer (char *out, char *in, unsigned int len, int ocod, int icod);*
PAS *fdConvertBuffer (_out: PChar; _in: PChar; len: Cardinal; ocod: Integer; icod: Integer): Integer;*
VB3 *fdConvertBuffer (ByVal obuf As String, ByVal ibuf As String, ByVal _len As Integer, ByVal ocod As Integer, ByVal icod As Integer) As Integer*
VB5 *fdConvertBuffer (ByVal obuf As String, ByVal ibuf As String, ByVal _len As Long, ByVal ocod As Long, ByVal icod As Long) As Long*
FOX *Integer fdConvertBuffer String @obuf, String @ibuf, Integer _len, Integer ocod, Integer icod*

Converts a buffer by importing it in one format and reexporting it in another.

Inputs
in: Pointer to a buffer containing data to be converted.
len: Size of the data to convert to binary format.
ocod: Constant that identifies how the output buffer will be coded.
icod: Constant that identifies how the input buffer is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
out: Pointer to a buffer of adequate size to contain the converted buffer.

fdInfoLength Function

C/C++ *int fdInfoLength (unsigned int *size, unsigned int len, int cod);*
PAS *fdInfoLength (Var size: Cardinal; len: Cardinal; cod: Integer): Integer;*
VB3 *fdInfoLength (size As Integer, ByVal _len As Integer, ByVal cod As Integer) As Integer*
VB5 *fdInfoLength (size As Long, ByVal _len As Long, ByVal cod As Long) As Long*
FOX *Integer fdInfoLength String Integer @size, Integer _len, Integer cod*

Returns the size a buffer occupies, coded with a format mask.

Inputs
len: Size of the data in binary format.
cod: Constant that identifies how the data will be coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
size: Size that the coded data occupies with the specified mask.

fdExportNumber Function

C/C++ *int fdExportNumber (char *out, long num, int cod);*
PAS *fdExportNumber (_out: PChar; num: Longint; cod: Integer): Integer;*
VB3 *fdExportNumber (ByVal obuf As String, ByVal num As Long, ByVal cod As Integer) As Integer*
VB5 *fdExportNumber (ByVal obuf As String, ByVal num As Long, ByVal cod As Long) As Long*
FOX *Integer fdExportNumber String @obuf, Long num, Integer cod*

Converts a number to hexadecimal format. This function is independent of the endianess of the processor.

Inputs
num: Number to be converted.
cod: Constant that identifies how the data will be coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
out: Pointer to a buffer that will contain the coded number according to the specified mask.

fdImportNumber Function

C/C++ *int fdImportNumber (long *num, char *in, int cod);*
PAS *fdImportNumber (Var num: Longint; _in: PChar; cod: Integer): Integer;*
VB3 *fdImportNumber (num As Long, ByVal ibuf As String, ByVal cod As Integer) As Integer*
VB5 *fdImportNumber (num As Long, ByVal ibuf As String, ByVal cod As Long) As Long*
FOX *Integer fdImportNumber Long @num, String @ibuf, Integer cod*

Converts a hexadecimal buffer to a number. This function is independent of the endianess of the processor.

Inputs
in: Pointer to a buffer that contains a coded number.
cod: Constant that identifies how the number is coded. Consult appendix B for a list of all available format masks and their associated constants.

Outputs
num: Converted Number.

fdAlloc Function

C/C++ *int fdAlloc (char **buf, unsigned int size);*
PAS *fdAlloc (Var buf: PChar; size: Cardinal): Integer;*

Reserves a secure block of memory of specified size.
Note: This function is not available from Visual Basic.

fdFree Function

C/C++ *int fdFree (char *buf);*
PAS *fdFree (buf: PChar): Integer;*

Frees a block of memory destroying its contents.

Note: This function is not available from Visual Basic.

fdVersion Function

C/C++ *int fdVersion (char *id, char *ver);*
PAS *fdVersion (id: PChar; ver: PChar): Integer;*
VB3 *fdVersion (ByVal id As String, ByVal ver As String) As Integer*
VB5 *fdVersion (ByVal id As String, ByVal ver As String) As Long*
FOX *Integer fdVersion String @id, String @ver*

Returns two strings identifying the version of the routines installed.

fdEnableModule Function

C/C++ *int fdEnableModule (int algorithm, char *code);*
PAS *fdEnableModule (algorithm: Integer; code: PChar): Integer;*
VB3 *fdEnableModule (ByVal algorithm As Integer, ByVal code As String) As Integer*
VB5 *fdEnableModule (ByVal algorithm As Long, ByVal code As String) As Long*
FOX *Integer fdEnableModule Integer algorithm, String @code*

Enables the use of a cryptographic module for users who utilize that form of commercialization.

Inputs
algorithm: Constant that identifies the algorithm that is to be enabled. Consult appendix B for a list of all available algorithms and their associated constants.
code: Authorization code provided by Firmas Digitales S.R.L.

File Management Functions

In this section, routines that perform simple cryptographic operations on files are described.

fdWipeFile Function

C/C++ *int fdWipeFile (char *file, int count);*
PAS *fdWipeFile (_file: PChar; count: Integer): Integer;*
VB3 *fdWipeFile (ByVal file As String, ByVal count As Integer) As Integer*
VB5 *fdWipeFile (ByVal file As String, ByVal count As Long) As Long*
FOX *Integer fdWipeFile String @file, Integer count*

Overwrites and erases a file, destroying its content with pseudorandom bits.

Inputs
file: Name of the file to be eliminated.
count: Number of times a file is overwritten with pseudorandom bits before it is erased.

fdHashFile Function

C/C++ *int fdHashFile (long hid, char *file);*
PAS *fdHashFile (hid: Longint; _file: PChar): Integer;*
VB3 *fdHashFile (ByVal hid As Long, ByVal file As String) As Integer*
VB5 *fdHashFile (ByVal hid As Long, ByVal file As String) As Long*
FOX *Integer fdHashFile Long hid, String @file*

Updates a hash context area with the contents of a file. The context area must be available in order to be updated.

Inputs
hid: Handle to the hash context area to be updated.
file: Name of the file to be digested.

fdEncriptFile Function

C/C++ *int fdEncryptFile (long kid, char *out, char *in, int mode);*
PAS *fdEncryptFile (kid: Longint; _out: PChar; _in: PChar; mode: Integer): Integer;*
VB3 *fdEncryptFile (ByVal kid As Long, ByVal ofile As String, ByVal ifile As String, ByVal mode As Integer) As Integer*
VB5 *fdEncryptFile (ByVal kid As Long, ByVal ofile As String, ByVal ifile As String, ByVal mode As Long) As Long*
FOX *Integer fdEncryptFile Long kid, String @ofile, String @ifile, Integer mode*

Encrypts a file with a symmetric key. If necessary it generates at random an initialization vector that is stored together with the encrypted file. The size of the output file is slightly greater than the original file.

Inputs
kid: Handle to the symmetric key context area.
in: Name of the file to be encrypted.
out: Name of the encrypted file.
mode: Constant that identifies the operation mode that is to be used. Consult appendix B for a list of all available operation modes and their associated constants.

fdDecryptFile Function

C/C++ *int fdDecryptFile (long kid, char *out, char *in, int mode);*
PAS *fdDecryptFile (kid: Longint; _out: PChar; _in: PChar; mode: Integer): Integer;*
VB3 *fdDecryptFile (ByVal kid As Long, ByVal ofile As String, ByVal ifile As String, ByVal mode As Integer) As Integer*
VB5 *fdDecryptFile (ByVal kid As Long, ByVal ofile As String, ByVal ifile As String, ByVal mode As Long) As Long*
FOX *Integer fdDecryptFile Long kid, String @ofile, String @ifile, Integer mode*

Decrypts a file with a symmetric key. It recovers the file with its original length, removing the padding and the initialization vector added during encryption.

Inputs
kid: Handle to the symmetric key context area.
in: Name of the file to be decrypted.
out: Name of the decrypted file.
mode: Constant that identifies the operation mode that is to be used. Consult appendix B for a list of all available operation modes and their associated constants.

fdCompressFile Function

C/C++ *int fdCompressFile (char *out, char *in, int algorithm);*
PAS *fdCompressFile (_out: PChar; _in: PChar; algorithm: Integer): Integer;*
VB3 *fdCompressFile (ByVal ofile As String, ByVal ifile As String, ByVal algorithm As Integer) As Integer*
VB5 *fdCompressFile (ByVal ofile As String, ByVal ifile As String, ByVal algorithm As Long) As Long*
FOX *Integer fdCompressFile String @ofile, String @ifile, Integer algorithm*

Compresses a file. The effectiveness depends on the redundancy of the original file. In extreme cases the size of the compressed file can be greater than the original file.

Inputs
in: Name of the file to be compressed.
out: Name of the compressed file.
algorithm: Constant that identifies the compression algorithm that is to be used. Consult appendix B for a list of all available algorithms and their associated constants.

fdExpandFile Function

C/C++ *int fdExpandFile (char *out, char *in, int algorithm);*
PAS *fdExpandFile (_out: PChar; _in: PChar; algorithm: Integer): Integer;*
VB3 *fdExpandFile (ByVal ofile As String, ByVal ifile As String, ByVal algorithm As Integer) As Integer*
VB5 *fdExpandFile (ByVal ofile As String, ByVal ifile As String, ByVal algorithm As Long) As Long*
FOX *Integer fdExpandFile In Lib fdapi32.dll String @ofile, String @ifile, Integer algorithm*

Expands a compressed file to its original size.

Inputs
in: Name of the compressed file.
out: Name of the original file.
algorithm: Constant that identifies the compression algorithm that is to be used. Consult appendix B for a list of all available algorithms and their associated constants.

fdSaveBuffer Function

C/C++ *int fdSaveBuffer (char *file, char *buf, unsigned int len);*
PAS *fdSaveBuffer (_file: PChar; buf: PChar; len: Cardinal): Integer;*
VB3 *fdSaveBuffer (ByVal file As String, ByVal buf As String, ByVal _len As Integer) As Integer*
VB5 *fdSaveBuffer (ByVal file As String, ByVal buf As String, ByVal _len As Long) As Long*
FOX *Integer fdSaveBuffer String @file, String @buf, Integer _len*

Saves the content of a memory buffer to a file.

Inputs
file: Name of the file to be saved.
buf: Pointer to a buffer with data to be saved.
len: Size of the specified buffer.

fdLoadBuffer Function

C/C++ *int fdLoadBuffer (char *buf, unsigned int *len, char *file);*
PAS *fdLoadBuffer (buf: PChar; var len: Cardinal; _file: PChar): Integer;*
VB3 *fdLoadBuffer (ByVal buf As String, _len As Integer, ByVal file As String) As Integer*
VB5 *fdLoadBuffer (ByVal buf As String, _len As Long, ByVal file As String) As Long*
FOX *Integer fdLoadBuffer String @buf, Integer @_len, String @file*

Reads the content of a file to a buffer in memory.

Inputs
file: Name of the file to be read.

Outputs
buf: Pointer to a buffer with the read data.
len: Size of the read buffer.

fdInfoFile Function

C/C++ *int fdInfoFile (unsigned int len, char *file);*
PAS *fdInfoFile (var len: Cardinal; _file: PChar): Integer;*
VB3 *fdInfoFile (_len As Integer, ByVal file As String) As Integer*
VB5 *fdInfoFile (_len As Long, ByVal file As String) As Long*
FOX *Integer fdInfoFile Integer @_len, String @file*

Obtains the size of a file.

Inputs
file: Name of the file to be read.

Outputs
len: Size of the read buffer.

Section 4

Program Examples

Hashing Functions

This program calculates the SHA-1 hash of the ‘abc’ string.

```
void ejemplo1 (void)
{
    long hid;
    char digest[50];
    int r;

    r = fdCreateHash (&hid, ALG_SHA1);
    if (r != FD_OK) { printf ("Error %i in fdCreateHash\n", r); exit (1); }

    r = fdHashBuffer (hid, "abc", 3);
    if (r != FD_OK) { printf ("Error %i in fdHashBuffer\n", r); exit (1); }

    r = fdExportHash (hid, digest, COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdExportHash\n", r); exit (1); }

    r = fdDestroyHash (hid);
    if (r != FD_OK) { printf ("Error %i in fdDestroyHash\n", r); exit (1); }

    printf ("The message digest is %s.\n", digest);
}
```

Symmetric Encryption

This program generates a 3xDES session key and a random initialization vector. The ‘abc’ string is encrypted with these elements en CBC mode using PKCS #5 padding.

```
void ejemplo2 (void)
{
    long kid;
    char cipher[50], plan[50], key[50], iv[50];
    int r, szcipher, szplain;

    r = fdGenerateKey (&kid, ALG_DES3);
    if (r != FD_OK) { printf ("Error %i in fdGenerateKey\n", r); exit (1); }

    r = fdGenerateIV (kid);
    if (r != FD_OK) { printf ("Error %i in fdGenerateIV\n", r); exit (1); }
```

```
    r = fdEncryptBuffer (kid, cipher, &szcipher, "abc", 3, MODE_CBC, PAD_PKCS);
    if (r != FD_OK) { printf ("Error %i in fdEncryptBuffer\n", r); exit (1); }

    r = fdResetIV (kid);
    if (r != FD_OK) { printf ("Error %i in fdResetIV\n", r); exit (1); }

    r = fdDecryptBuffer (kid, plain, &szplain, cipher, szcipher, MODE_CBC, PAD_PKCS);
    if (r != FD_OK) { printf ("Error %i in fdDecryptBuffer\n", r); exit (1); }

    r = fdExportKey (kid, key, COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdExportKey\n", r); exit (1); }

    r = fdExportIV (kid, iv, COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdExportIV\n", r); exit (1); }

    r = fdDestroyKey (kid);
    if (r != FD_OK) { printf ("Error %i in fdDestroyKey\n", r); exit (1); }

    printf ("The session key was %s.\n", key);
    printf ("The initialization vector was %s.\n", iv);
}
```

MAC Authentication

This program calculates a 32 bit MAC of an ‘abc’ string en CBC mode, using a DES key derived from the word ‘password’.

```
void ejemplo3 (void)
{
    long kid;
    char mac[20];
    int r, szcipher, szplain;

    r = fdDeriveKey (&kid, ALG_DES, "password", 8, ALG_MD5);
    if (r != FD_OK) { printf ("Error %i in fdDeriveKey\n", r); exit (1); }

    r = fdImportIV (kid, "0000 0000 0000 0000", COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdImportKey\n", r); exit (1); }

    r = fdGenerateMAC (kid, mac, "mensaje", 7,  MODE_CBC, MAC_SIZE4, COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdGenerateMAC\n", r); exit (1); }

    r = fdResetIV (kid);
    if (r != FD_OK) { printf ("Error %i in fdResetIV\n", r); exit (1); }

    r = fdVerifyMAC (kid, mac, "mensaje", 7,  MODE_CBC, MAC_SIZE4, COD_HEXZ);
    if (r != FD_OK) { printf ("Error %i in fdVerifyMAC\n", r); exit (1); }

    r = fdDestroyKey (kid);
    if (r != FD_OK) { printf ("Error %i in fdDestroyKey\n", r); exit (1); }

    printf ("The calculated MAC is %s.\n", mac);
}
```

Generating Public Keys

This program generates and prints a RSA private key of 1024 bits.

```
Sub ejemplo4 ()
    Dim prv As Long
    Dim r as Integer
    Dim bits As Integer
    Dim ktype As Integer
    Dim ksize As Integer
    Dim buf As String
    Dim sz As Integer

    r = fdGeneratePrivateKey(prv, 1024, PUB_FERMAT4)
    If r <> FD_OK Then Debug.Print "Error " & r & " in fdGeneratePrivateKey" : Stop

    r = fdInfoRSA(prv, bits, ktype, ksize)
    If r <> FD_OK Then Debug.Print "Error " & r & " in fdInfoRSA" : Stop

    sz = 2 * ksize + 1
    buf = Space$(sz)

    r = fdExportRSAKey(prv, buf, COD_HEXZ Or COD_LOWER))
    If r <> FD_OK Then Debug.Print "Error " & r & " in fdExportRSAKey" : Stop
    Debug.Print "PRIVATE KEY="; buf

    r = fdWipeBuffer(buf, sz)
    If r <> FD_OK Then Debug.Print "Error " & r & " in fdWipeBuffer" : Stop

    r = fdDestroyRSAKey(prv)
    If r <> FD_OK Then Debug.Print "Error " & r & " in fdDestroyRSAKey" : Stop
End Sub
```

Digital Signatures

This program generates a pair of RSA keys, calculates a hash of a string, signs it with a private key and verifies the digital signature with the corresponding public key.

```
void ejemplo5 (void)
{
    long pub, pri, hid, sid, hi2;
    char *str = "abc", buf[20];
    int r = 0;

    r = fdGeneratePrivateKey (&pri, 1024, PUB_FERMAT4);
    if (r != FD_OK) { printf ("Error %i in fdGeneratePrivateKey\n", r); exit (1); }

    r = fdDerivePublicKey (&pub, pri);
    if (r != FD_OK) { printf ("Error %i in fdDerivePublicKey\n", r); exit (1); }

    r = fdCreateHash (&hid, ALG_MD5);
    if (r != FD_OK) { printf ("Error %i in fdCreateHash\n", r); exit (1); }
```

```
    r = fdHashBuffer (hid, str, strlen (str));
    if (r != FD_OK) { printf ("Error %i in fdHashBuffer\n", r); exit (1); }

    r = fdSignHash (&sid, hid, pri);
    if (r != FD_OK) { printf ("Error %i in fdSignHash\n", r); exit (1); }

    r = fdRecoverHash (&hid2, sid, pub);
    if (r != FD_OK) { printf ("Error %i in fdRecoverHash\n", r); exit (1); }

    r = fdCompareHash (hid, hid2);
    if (r != FD_OK) { printf ("Error %i in fdCompareHash\n", r); exit (1); }

    r = fdDestroySignature (sid);
    if (r != FD_OK) { printf ("Error %i in fdDestroySignature\n", r); exit (1); }

    r = fdDestroyHash (hid);
    if (r != FD_OK) { printf ("Error %i in fdDestroyHash\n", r); exit (1); }

    r = fdDestroyHash (hid2);
    if (r != FD_OK) { printf ("Error %i in fdDestroyHash\n", r); exit (1); }

    r = fdDestroyRSAKey (pub);
    if (r != FD_OK) { printf ("Error %i in fdDestroyRSAKey\n", r); exit (1);

    r = fdDestroyRSAKey (pri);
    if (r != FD_OK) { printf ("Error %i in fdDestroyRSAKey\n", r); exit (1);
}
```

Section 5

Appendixes

APPENDIX A : FUNCTION SUMMARIES

Hash Calculation and Verification

fdCreateHash	Initializes an ‘empty’ hash context area
fdDestroyHash	Destroys a hash context area
fdInfoHash	Obtains information about a hash context area
fdImportHash	Initializes a hash context area with a ‘fixed’ area
fdExportHash	Obtains a hash value from a context area
fdCompareHash	Compares two hash for equality
fdHashBuffer	Updates a hash with a buffer
fdHashNumber	Updates a hash with a number

Generating Symmetric Keys

fdGenerateKey	Creates a symmetric key generated randomly
fdDeriveKey	Creates a symmetric key calculating a hash of a passphrase provided by the user
fdDestroyKey	Destroys a symmetric key
fdInfoKey	Obtains information about a symmetric key
fdImportKey	Creates a symmetric key based on an external key
fdExportKey	Obtains the symmetric key from a context area
fdCompareKey	Compares two symmetric keys for equality

Encryption with Symmetric Keys

fdEncryptBlock	Encrypts a block of data with a symmetric key
fdDecryptBlock	Decrypts a block of data with a symmetric key
fdEncryptBuffer	Encrypts an arbitrary sized buffer with a symmetric key
fdDecryptBuffer	Decrypts a buffer with a symmetric key

Initialization Vector Management

fdGenerateIV	Generates a random initialization vector for a key
fdResetIV	Synchronizes an initialization vector for a symmetric key
fdImportIV	Specifies an initialization vector for a symmetric key
fdExportIV	Obtains the initialization vector from a symmetric key

MAC Generation and Verification

fdGenerateMAC	Generates a MAC for a message
fdVerifyMAC	Verifies a MAC for a message

Generating Public and Private Keys

fdGeneratePrivateKey	Generates randomly a private key
fdDerivePublicKey	Obtains the corresponding public key to a private key
fdDestroyRSAKey	Destroys a public or private key
fdInfoRSA	Obtains information about a public or private key
fdImportRSAKey	Creates a context area based on an external RSA key
fdExportRSAKey	Obtains the corresponding RSA key to a context area
fdSetRSAParam	Allows you to specify parameters of the RSA algorithm

Generating Digital Signatures

fdSignBuffer	Encrypts a buffer with a private key
fdSignHash	Encrypts a hash with a private key
fdSignHashExtra	Encrypts a hash and a buffer with a private key
fdDestroySignature	Destroys a digital signature
fdInfoSignature	Obtains information from a digital signature
fdImportSignature	Creates a context area based on an external digital signature
fdExportSignature	Obtains a digital signature from a context area

Generating Digital Envelopes

fdEnvelopBuffer	Encrypts a buffer with a public key
fdEnvelopKey	Encrypts a symmetric key with a public key
fdEnvelopKeyExtra	Encrypts a symmetric key and a buffer with a public key
fdEnvelopHash	Encrypts a hash with a public key
fdDestroyEnvelope	Destroys a digital envelope
fdInfoEnvelope	Obtains information from a digital envelope
fdImportEnvelope	Creates a context area based on an external digital envelope
fdExportEnvelope	Obtains a digital envelope from a context area

Digital Signatures and Envelopes Verification

fdRecoverBuffer	Recovers a buffer from a digital signature or envelope
fdRecoverHash	Recovers a hash from a digital signature or envelope
fdRecoverHashExtra	Recovers a hash and a buffer from a digital signature or envelope
fdRecoverKey	Recovers a symmetric key from a digital signature
fdRecoverKeyExtra	Recovers a symmetric key and a buffer from a digital signature

Adaptive Compression and Expansion

fdCompressBuffer	Compresses a buffer
fdExpandBuffer	Expands a buffer to its original size
fdInfoCompress	Obtains the original size from a compressed buffer

Miscellaneous Functions

fdRegisterRandomBuffer	Registers a buffer of random bits generated externally
fdGenerateRandom	Generates a vector of random bits
fdWipeBuffer	Destroys a buffer with a pseudorandom sequence
fdConvertBuffer	Converts a buffer from one format to another
fdInfoLength	Obtains the size that a buffer occupies in a format
fdImportNumber	Imports a hexadecimal format number
fdExportNumber	Exports a hexadecimal format number
fdAlloc	Reserves a secure block of memory
fdFree	Frees a block of memory, destroying its content
fdVersion	Obtains information about the version of the installed routine
fdEnableModule	Enables the use of a cryptographic module

File Management

fdWipeFile	Destroys the content of a file
fdHashFile	Updates a hash with the content of a file
fdEncryptFile	Encrypts a file with a symmetric key
fdDecryptFile	Decrypts a file with a symmetric key
fdCompressFile	Compresses a file
fdExpandFile	Expands a file to its original size
fdSaveBuffer	Stores a buffer in a file
fdLoadBuffer	Recovers a buffer from a file

APPENDIX B : DEFINED CONSTANTS

There exist diverse parameters that are coded. Here, the symbolic constants and their meanings are presented:

Symmetric Algorithms

ALG_DES1	DES algorithm with 56 bit key
ALG_DES2	3xDES algorithm with 112 bit key
ALG_DES3	3xDES algorithm with 168 bit key
ALG_IDEA	IDEA algorithm with 128 bit key

Hashing Algorithms

ALG_MD4	MD4 algorithm, produces a hash of 128 bits
ALG_MD5	MD5 algorithm, produces a hash of 128 bits
ALG_SHA0	SHA algorithm, produces a hash of 160 bits
ALG_SHA1	SHA algorithm, revision 1, produces a hash of 160 bits
ALG_RIPEMD128	RIPE-MD 128 algorithm, produces a hash of 128 bits
ALG_RIPEMD160	RIPE-MD 160 algorithm, produces a hash of 160 bits

Compression Algorithms

ALG_LZW	LZW adaptive compression algorithm

Operation Modes

MODE_ECB	Electronic Codebook Mode
MODE_CBC	Cipher Block Chaining Mode
MODE_CFB	Cipher Feedback Mode
MODE_OFB	Output Feedback Mode
MODE_BCF	Byte Cipher Feedback Mode

MODE_BOF	Byte Output Feedback Mode
MODE_CFB_64	Equivalent to MODE_CFB
MODE_OFB_64	Equivalent to MODE_OFB
MODE_CFB_8	Equivalent to MODE_BCF
MODE_OFB_8	Equivalent to MODE_BOF

Padding Methods

PAD_NONE	No padding
PAD_PKCS	Padding according to standard PKCS #5 section 6.2
PAD_X923	Padding according to standard X9.23

RSA Key Type

RSA_PRIVATE	RSA key is private
RSA_PUBLIC	RSA key is public

Public Key Exponents

PUB_RANDOM	The public exponent is chosen randomly
PUB_FERMAT4	The public exponent is the fourth Fermat number (65537)

MAC Sizes

MAC_SIZE4	Produces a MAC of 4 bytes according to ANSI X9.9
MAC_SIZE6	Produces a MAC of 6 bytes
MAC_SIZE8	Produces a MAC of 8 bytes

Format Masks

COD_BIN	Binary Format
COD_HEX	Generic hexadecimal codification

COD_ASCII	ASCII hexadecimal codification
COD_EBCDIC	EBCDIC hexadecimal codification
COD_GROUP16	Separation with spaces every 16 bits
COD_GROUP32	Separation with spaces every 32 bits
COD_GROUP64	Separation with spaces every 64 bits
COD_STRINGZ	The string with the null character
COD_UPPER	Hexadecimal characters in upper case letters
COD_LOWER	Hexadecimal characters in lower case letters
COD_HIGHLOW	Bytes coded normally
COD_LOWHIGH	Bytes coded in inverse form
COD_HEXZ	Equivalent to COD_HEX y COD_STRINGZ
COD_ASCIIZ	Equivalent to COD_ASCII y COD_STRINGZ
COD_EBCDICZ	Equivalent to COD_EBCDIC y COD_STRINGZ

APPENDIX C : ERROR CODES

All functions return an number indicating whether the operation was satisfactory or not.

Code	Description
0	**FD_OK** The function ended correctly.
-101	**FD_ERR_PROTECTION** A protected version is being used and does not possess a correctly programmed hardlock. If it possesses the key, then the drivers are incorrectly installed. Contact Firmas Digitales S.R.L. to enable the functions that you need.
-102	**FD_ERR_NOINIT** The FDCrypt library is not initialized. The initialization of FDCrypt is performed automatically. Contact Firmas Digitales S.R.L. to inform them about this problem.
-103	**FD_ERR_INVALIDALGORITHM** The algorithm parameter is invalid or the selected operation is not coherent with the type of algorithm specified.
-104	**FD_ERR_INVALIDCODE** The code to unblock an algorithm is invalid. Contact Firmas Digitales S.R.L. in order to obtain a valid code, a HardLock or an unlimited version of the routines.
-105	**FD_ERR_OUTOFMEMORY** The invoked routine could not obtain the dynamic memory necessary in order to perform the selected operation. Given the reduced demands on memory of FDCrypt, it is likely that this is due to an error in a program that obtains memory and doesn't return it when closed. Since this situation is critical, the program should be closed immediately.
-106	**FD_ERR_UNDEFINEDERROR** Non-specific or internal error.
-201	**FD_ERR_INVALIDPOINTER** One of the specified parameters is a pointer which points to a position of invalid

memory.

-202 **FD_ERR_INVALIDLENGTH**
One numeric parameter contains an invalid value or is out of range. Verify the documentation of the routine and what are the restrictions for the values of the parameters.

-203 **FD_ERR_INVALIDMASK**
The format mask to import or export binary values is invalid. Verify the combinations used.

-204 **FD_ERR_INVALIDINPUT**
The content of the parameter does not coincide with the value of the associated format mask.

-205 **FD_ERR_INVALIDHANDLE**
The handle to a context area is invalid or does not correspond to the type of object expected.

-301 **FD_ERR_INVALIDFILENAME**
The file name is invalid. Verify the specified name.

-302 **FD_ERR_FILENOTFOUND**
The file does not exist. Verify that it does exist and that the path is correct.

-303 **FD_ERR_WRITEFAILED**
The file could not be written for one of the following reasons: the file is read only; you do not have the proper licenses; or there was an error in the physical device.

-304 **FD_ERR_OUTPUTCANNOTBEINPUT**
The output file cannot be the input file. Specify an output file different from that of the input file.

-305 **FD_ERR_INVALIDFORMAT**
The input file does not correspond to the type of file expected. It is probably corrupted.

-306 **FD_ERR_READFAILED**
The file could not be read for one of the following reasons: you do not have the

proper licenses; or there was an error in the physical device.

-401 FD_ERR_HASHCLOSED
You are trying to update a context area of an already closed hash. Once a hash is closed it cannot continue to be modified.

-402 FD_ERR_ALGORITHMNOTEQUAL
You are trying to perform an operation using two different algorithms. Verify that the algorithms are the same.

-403 FD_ERR_HASHNOTEQUAL
The compared hash values are different. It is likely that there has been an alteration in the original data from which the hash is derived.

-501 FD_ERR_WEAKKEY
The specified symmetric key is weak. Specify a correct key or use the random generation of symmetric keys.

-502 FD_ERR_INVALIDMODE
The operation mode is invalid. Verify the documentation concerning available operation modes.

-503 FD_ERR_INVALIDPAD
The padding method is invalid. Verify the documentation concerning available padding methods.

-504 FD_ERR_KEYNOTEQUAL
The two symmetric keys compared are different. Verify how the keys were obtained.

-505 FD_ERR_IVNOTSET
You are trying to use a symmetric key in a feedback mode without having previously specified an initialization vector.

-506 FD_ERR_INVALIDMAC
The type of MAC is invalid. Consult the available constants.

-507 FD_ERR_MACNOTEQUAL
The calculated MAC does not coincide with the MAC of reference.

-601 **FD_ERR_BITSOUTOFRANGE**
The specified length is out of range. The value should be in the 512-8192 range and should be an even number.

-602 **FD_ERR_INVALIDEXPONENT**
The specified public exponent is invalid. Consult the documentation for what are the allowed values of public exponents.

-603 **FD_ERR_NOPASSPHRASE**
A passphrase must be specified for a private key.

-604 **FD_ERR_INVALIDKEY**
The specified passphrase is incorrect or the key is corrupted.

-605 **FD_ERR_NOTPRIVATEKEY**
A private key must be specified and a public key is being specified.

-606 **FD_ERR_NOTPUBLICKEY**
A public key must be specified and a private key is being specified.

-607 **FD_ERR_DATATOOLONG**
The specified data cannot be encrypted using a single modular exponentiation. Reduce the size of the data or else use longer public keys.

-608 **FD_ERR_INTEGRITYCHECK**
The digital signature or envelope does not have the correct format. Most likely the signature is corrupted or the correct RSA key is not being used.

APPENDIX D : COMPILING IN ‘C’

This appendix provides additional information for getting started with FDCrypt.

Static Link Library (LIB) Usage

A LIB is a executable library that contains functions to be used by other applications and are linked in load time. This is known as a static link library. Internally, it is divided into modules. When linking the library with an application only the referenced modules are incorporated to the resulting executable code.

Given that the library utilizes floating point operations, the standard floating point library should also be linked. The file libfdapi.a is provided. The parameter -L is used to indicate to the compiler where the library is found.

Makefile example

```
.c.o:
	cc -Xc -v -c -O $*.c

prog: prog.o
	cc prog.o -L. -lfdcapi -lm -o prog
```

Dynamic Link Library (DLL) Usage

A DLL is an executable library that contains functions or resources to be used by other applications and/or DLL’s. Some advantages to using them are:

- Reduces the size of the executable file.
- Permits better utilization of the system’s memory.
- Provides greater flexibility when faced with changes in the application that uses it.
- They can be shared.

There are two types of usage for DLL’s that are described in the next section.

Load-Time Dynamic Linking

Load time dynamic linking occurs when the code of an application calls explicitly to the code of a function that resides in a DLL. When the source code is compiled, the DLL function generates an external reference to the function in the object code. In order to resolve this external reference, the application must be linked with the importation library (.LIB) for the DLL.

One external function in an importation library informs the linker that the code for that function is in a DLL. In order to resolve the external references, the linker simply adds information to the executable file that tells the system where it can find the DLL code when the process begins.

When the system starts up a program that contains dynamic link references, it uses the information that is found in the file of the executable program to be able to locate the required DLL's. The system then searches for the DLL's in the following sequence:

1. The directory where the executable program is found (for the current process).
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH.

If the system cannot locate the specified DLL, then it terminates the process and opens a window that reports the error. On the contrary, if the system localizes the DLL, then it maps the modules of the DLL in the address space of the process. Finally, the system modifies the executable code of the process in order to provide the start-up addresses of the DLL functions.

Like the rest of a program code, the DLL code is mapped in the address space of a process when it starts up and is only loaded into the memory when necessary.

Example

```
/* Archivo : loadtime.c
   Ejemplo de un simple programa que usa la función
```

```
   miEntrada() de entradas.dll
*/
#include <windows.h>
void miEntrada(LPTSTR); /* Función de la DLL */
void main(void)
{
    miEntrada("Probando el uso de una función de la DLL\n");
}
```

Since “loadtime.c” explicitly calls a DLL function, the executable module of the application must be linked as the importation library (in this case denominated “entradas.lib”).

Next, some typical command lines are included to compile and link a program that uses a DLL with load-time linking.

```
# Compilar loadtime.c para crear loadtime.obj.
loadtime.obj: loadtime.c
    $(cc)  $(cflags) $(cvars) loadtime.c

# Linkear loadtime.obj y entradas.lib para crear loadtime.exe.
loadtime.exe: loadtime.obj entradas.lib
    $(link) $(conflags) \
    -out:loadtime.exe loadtime.obj entradas.lib $(conlibs)
```

Run-Time Dynamic Linking

Run-time dynamic linking occurs when the process calls the LoadLibrary function (to specify the name of the DLL) and the GetProcAddress function (to obtain the start-up address of a function in the DLL). Run-time dynamic linking eliminates the need to link the process with the importation library. Since the process does not explicitly call the DLL functions, then it is not necessary to generate external references to them.

If the call to LoadLibrary specifies an already mapped DLL module in the address space of the process that invokes it, the function simply returns a handle of the DLL and increases the module references counter. On the contrary, if there doesn’t exist such reference to a DLL module, the LoadLibrary function tries to locate the DLL using the same search sequence used for load-time dynamic linking. If the search is successful, the system maps the DLL module in the address space of the process.

If the system cannot find the DLL, the LoadLibrary function returns NULL. If LoadLibrary is successful, it returns the handle of the DLL module. The process can use this handle to identify the DLL in the call to the GetProcAddress or FreeLibrary functions.

The GetModuleHandle function also returns the handle used in GetProcAddress or FreeLibrary. GetProcAddress is only successful if the DLL module is found already mapped in the address space of the process, be it by means of a load-time dynamic linking or a previous call to LoadLibrary. Unlike LoadLibrary, the GetModuleHandle function does not increase the module reference counter. The GetModuleFileName function returns the complete path of the module associated with the handle returned by GetModuleHandle or FreeLibrary.

The process can use GetProcAddress to obtain the initial handle returned by LoadLibrary or the address of the function in the DLL (that is using a DLL handle returned by LoadLibrary or GetModuleHandle).

When the DLL module is no longer necessary, the process can call the FreeLibrary function. This function decreases the module reference counter and unmaps it from the address space of the process (if the reference counter is equal to zero).

Run-time dynamic linking allows a processor to continue its execution even when the DLL is not available. The process can then use an alternative method to fulfill its objective. For example, if a process cannot locate a DLL, it can try to use another or else inform the user of the error produced. If the user can provide the complete path of the lacking DLL, the process can use this information to load the library although it is not found in the normal search path. This situation is opposite that of load-time dynamic linking, since in that case, the system terminates the process if it cannot find the desired DLL.

Example

The same DLL can be used for both types of linking. If the system can find the specified DLL, the code that is then included produces the same output as the example given in the load-time dynamic linking section. The program uses the LoadLibrary function to obtain the handle from “entradas.dll”. If LoadLibrary is successful, the program uses the returned handle in the GetProcAddress function in order to then obtain the start-up address of the miEntrada function of the DLL. After calling the DLL function, the program calls the FreeLibrary function to free the process from the DLL.

```
/*
   Archivo : loadtime.c
   Ejemplo de un simple programa que usa las funciones
   LoadLibrary y GetProcAddress para acceder a la función
   miEntrada() de entradas.dll
*/
```

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

typedef VOID (*MYPROC)(LPTSTR);

VOID main(VOID)
{
    HINSTANCE hinstLib;
    MYPROC ProcAdd;
    BOOL fFreeResult, fRunTimeLinkSuccess = false;

    /* Obtener el handle de la DLL. */
    hinstLib = LoadLibrary("entradas");

    /* Si el handle es válido, tratar de obtener la
       dirección de la función. */
    if (hinstLib != NULL) {
        ProcAdd =(MYPROC) GetProcAddress(hinstLib, "miEntrada");

        /* Si la dirección de la función es válida,
           llamar a la función. */
        if (fRunTimeLinkSuccess = (ProcAdd != NULL))
            (ProcAdd)("Mensaje mediante una función de DLL \n");

        /* Liberar la DLL. */
        fFreeResult = FreeLibrary(hinstLib);
    }

    /* Si no se puedo realizar el llamado a la función,
       usar otra alternativa. */
    if (! fRunTimeLinkSuccess)
        printf("Mensaje a través de otra alternativa\n");
}
```

Since the program uses run-time dynamic linking, then it is not necessary to link with the importation libraries of the DLL when the executable program is being created. Next, some typical command lines are included to compile and link a program that uses a DLL with run-time linking.

```
# Compile runtime.c to create runtime.obj.
runtime.obj: runtime.c
    $(cc)  $(cflags) $(cvars) runtime.c

# Link runtime.obj to create loadtime.exe.
runtime.exe: runtime.obj
    $(link) $(conflags) \
    -out:runtime.exe runtime.obj $(conlibs)
```

APPENDIX E : BIBLIOGRAPHY AND DOCUMENTATION

These are some of the books and documents that you may find useful in understanding the present manual:

- ✓ Dorothy E. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.
- ✓ Friedrich L. Bauer, *Decrypted Secrets*, Springer-Verlag, 1997
- ✓ W.Diffie and M.Hellman, New Directions in cryptography, IEEE Transactions on Information Theory, IT-22, 1976, pp.644-654.
- ✓ Alfred J.Menezes, Paul C.van Oorschot and Scott A.Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
- ✓ Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, 1996.
- ✓ National Institute of Standards and Technology, *FIPS PUB 46-2, 81, 180, 180-1*
- ✓ IETF, Request for Comments, *RFC's 1319, 1320, 1321*, April 1992
- ✓ RSA Laboratories, *Public-Key Cryptography Standards*, RSA Data Security, Nov. 1993.
- ✓ Douglas R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.
- ✓ Gus Simmons, *Contemporary Cryptology*, IEEE Press, 1992.
- ✓ Michael Luby, *Pseudorandomness and Cryptographic Applications*, Princeton University Press, 1996.
- ✓ U. M. Maurer (ed), *Eurocrypt '96, Advances in Cryptology - Lecture Notes in Computer Science*, Springer-Verlag, 1996.

APPENDIX F : GLOSSARY

Asymmetric Algorithm *Algorithm that requires two different keys, one to encrypt and one to decrypt. One is called the public key and the other private key. An example is the RSA algorithm.*

BCF *Byte Cipher Feedback Mode. Similar to the CFB mode except that instead of shifting in groups of 64 bits, it shifts in groups of 8 bits.*

Block *Basic unit of information that can be encrypted or decrypted. In symmetric algorithms, the usual size of a block is 64 bits.*

Block Cipher Algorithms *Are those algorithms that encrypt messages of fixed size called blocks. The usual size of a block is 64 bits. Examples of these algorithms are: DES and IDEA.*

BOF *Byte Output Feedback Mode. Similar to OFB mode except that instead of shifting in groups of 64 bits, it shifts in groups of 8 bits. This mode is unsafe and not recommended.*

CBC *Cipher Block Chaining Mode. Operation mode that consists of combining each block of plaintext with the anteriorly ciphered block using XOR. This method requires an initialization vector.*

CFB *Cipher Feedback Mode. Operation mode that consists in utilizing a shift register that is encrypted and combined with the plaintext using XOR..The ciphertext is shifted within the register. The shift register must be initialized with an initialization vector.*

Checksum *See Message Digest.*

Ciphertext *Encrypted message.*

Compression *Process by which redundancy is eliminated from a plaintext.*

Compression Function *See Hashing Algorithm.*

Context Area	*Refers to an area of memory reserved for FDCrypt, capable of storing all the data associated to a cryptographic object, like keys or digital signatures. Once finalized the operations, the context areas must be destroyed*
Cryptanalysis	*Branch of the science that studies the techniques by which one can neutralize or break cryptographic algorithms. In particular, it is applied to the obtaining of plaintexts from the interception of ciphertexts.*
Cryptography	*Branch of the science that studies the techniques by which two entities can communicate through an insecure channel in a secure manner.*
Cryptology	*Branch of the science that includes Cryptography and Cryptanalysis.*
Digital Envelope	*Mechanism for the distribution of keys of a symmetric algorithm consisting of encrypting these keys with the public key of the recipient, in such a way that only the recipient can recover them with their private key.*
Digital Signature	*Is the result of encrypting the message digest with the private key of the sender of said message. Anyone can verify the validity of the digital signature, using the public key of the signer.*
ECB	*Electronic Code Book Mode. Operation mode that consists in encrypting each block individually.*
Encryption	*Act by which a message is coded in order to transform it into a ciphertext.*
Fingerprint	*See Message Digest.*
Hashing Algorithms	*Are algorithms that permit you to verify that a message has not been modified (integrity), given that an arbitrary sized message produces an output of fixed size. Examples of this type of algorithm are: MD5 and SHA.*

Initialization Vector *Group of random bits used in feedback operation modes in such a way that a single message is always encrypted differently. These vectors do not have to remain secret and can be transmitted along with the ciphertext.*

Message *Any information, be it a file or a chain of characters.*

Message Digest *Output produced by a Hashing algorithm.*

OFB *Output FeedBack Mode. Operation mode that consists in utilizing a shifted register that is encrypted and combined with the plaintext using XOR. The result of the encryption is shifted within the register. The shifted register must be initialized with an initialization vector.*

Operation Mode *Normally, a message is divided into blocks. The operation mode indicates how the blocks should be processed. Some options are: ECB, CBC or CFB.*

Padding *The majority of messages do not consist of an even number of blocks. Normally, the last block is shorter and must be treated in a special way or padded to reach the size of a block. When padding, it is done according to a standard in which this padding can be removed during decryption.*

PKCS *Public Key Cryptography Standards. De-facto set of standards about the implementation of public key algorithms.*

Plaintext *Non-encrypted message or cleartext.*

Public Key Algorithm *See asymmetric algorithm.*

Repudiate *Refers to the case when the presumed author of a message fails to recognize the origin of a message.*

Symmetric Algorithm *Encryption algorithm that is characterized by the use of the same key to encrypt and decrypt. Examples of this type of algorithm are: DES and IDEA.*

XOR	*Logical operation between two bits, such that if both are equal the result is 0, otherwise.1.*

Notes :

Notes :