

The industrial strength public and private key data encryption toolbox



LockBoxTM 2

The all-in-one library of powerful routines you'll use every day to insure the security of your data and messages.

- *RSA Cipher for secure encryption/decryption*
- *Hashing components quickly create hash messages*
- *Digital Signatures assure file authenticity*
- *Rijndael encryption component for NISD encryption*
- *Easy to use Cryptographic components*
- *Full source code, no royalties*



TURBOPOWER[®]
Software Company

LockBox 2TM

TurboPower Software Company
Colorado Springs, CO

Order line (U.S. and Canada): 800/333.4160

Elsewhere: 719/471.3898

Fax: 719/471.9091

www.turbopower.com

© 1997-2001 TurboPower Software Company. All rights reserved.

First Edition April 1997

License Agreement

This software and its documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

At the time of this writing this software and accompanying documentation (the "Software") is subject to restrictions and controls imposed by the International Traffic in Arms Regulations and Arms Export Control Act (the "Acts"). Neither the Software nor any direct product thereof may be acquired, shipped, United States or Canada or may be used for any purpose prohibited by the Acts. However, U.S. citizens and U.S. permanent resident aliens may travel to countries not prohibited by the Acts with the Software when it is installed on their personal computer and not otherwise used or transferred in violation of the Acts. Please check the TurboPower Software Web site and the LICENSE.TXT file for the latest information on LockBox availability outside the U.S. and Canada.

Copyright © 1997-2001 by TurboPower Software Company. All rights reserved.

TurboPower Software Company authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without run-time fees or further licenses, your own compiled programs based on any of the source code of OfficePartner. You may not distribute any of the OfficePartner source code, compiled units, or compiled example programs without written permission from TurboPower Software Company. You may not use OfficePartner to create components or controls to be used by other developers without written approval from TurboPower Software Company. OfficePartner is licensed for use solely on Microsoft Windows platforms.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend upon OfficePartner. However, others who receive your source code or units need to purchase their own copies of OfficePartner in order to compile the source code or to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. Group programming projects making use of this software must purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software Company for volume discounts and site licensing agreements.

This software and accompanying documentation is deemed to be "commercial software" and "commercial computer software documentation," respectively, pursuant to DFAR Section 227.7202 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display or disclosure of the Software by the US Government or any of its agencies shall be governed solely by the terms of this agreement and shall be prohibited except to the extent expressly permitted by the terms of this agreement. TurboPower Software Company, 15 North Nevada, Colorado Springs, CO 80903-1708.

With respect to the physical media and documentation provided with OfficePartner, TurboPower Software Company warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software Company will replace the defective media or documentation at no cost to you.

TurboPower Software Company warrants that the software will function as described in this documentation for a period of 60 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software Company will update the defective software at no cost to you.

TurboPower Software Company further warrants that the purchaser will remain fully satisfied with the product for a period of 60 days from receipt. If you are dissatisfied for any reason, and TurboPower Software Company cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software Company, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program media and documentation in undamaged condition. TurboPower Software Company honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase a product directly from us.

TURBOPOWER SOFTWARE COMPANY DOES NOT ASSUME ANY LIABILITY FOR THE USE OF OFFICEPARTNER BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE COMPANY BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section and to any additional licensing terms contained in the DEPLOY.HLP file. If you do not agree, you should immediately return the entire OfficePartner package for a refund.

All TurboPower product names are trademarks or registered trademarks of TurboPower Software Company. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Chapter 1: Introduction	1
LockBox Functionality	2
What's New in This Release?	3
System Requirements	4
Installation	5
Organization of this Manual	6
Technical Support	9
Unified Modeling Language (UML)	10
Suggested Reading	14
Chapter 2: Overview	15
Encryption and Decryption	16
Hashing and Digital Signing	22
Using LockBox	24
Other Algorithms	29
Chapter 3: Low-Level Cryptographic Routines	31
Blowfish Cipher	32
DES Cipher	41
ELF Hash	49
LockBox Block Cipher	51
LockBox Key Generation Routines	57
LockBox Message Digest	59
LockBox Quick Cipher	62
LockBox Random Number Ciphers	66
LockBox Stream Cipher	70
MD5 Hash	73
Mix128 Hash	76
Rijndael Cipher	78
RSA Cipher	87
Secure Hashing Algorithm (SHA-1)	93
Triple-DES Cipher	97
Chapter 4: Encryption Components	105
TLbBaseComponent Class	107
TLbCipher Class	109
TLbSymmetricCipher Class	114
TLbBlowfish Component	117

TLbDES Component	119
TLb3DES Component	121
TLbRijndael Component	123
TLbAsymmetricCipher Class	126
TLbRSA Class	129
Chapter 5: Hash Components	133
TLbHash Class	135
TLbMD5 Component	138
TLbSHA1 Component	140
Chapter 6: Digital Signature Components	143
TLbSignature Class	147
TLbDSA Component	153
TLbRSASSA Component	154
Chapter 7: Asymmetric Key Class	155
TLbAsymmetricKey Class	156
Chapter 8: Stream Classes	161
TLbSCStream Class	163
TLbSCFileStream Class	166
TLbRNG32Stream Class	169
TLbRNG32FileStream Class	172
TLbRNG64Stream Class	175
TLbRNG64FileStream Class	178
Identifier Index	i
Subject Index	v

Chapter 1: Introduction

Cryptography, to most people, is concerned with keeping communications private. Indeed, the hiding of the content of sensitive messages has been the emphasis of cryptography throughout much of its history.

Encryption is the process that converts data into some form that is not immediately understandable. Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who can see and read the encrypted data. Decryption is the reverse of encryption; it is the transformation of encrypted data back into some comprehensible form.

Encryption and decryption require the use of some secret, private information, usually referred to as a key. Depending on the encryption mechanism used, the same key might be used for both encryption and decryption, or a different key might be used in the decryption step than that used in the encryption process.

LockBox provides services that enable you to add cryptography to your applications. The functions in LockBox can be used in your applications without any knowledge of the underlying encryption algorithm or implementation, in the same way that you can use a graphics library for your programs without knowing anything about the particular graphics hardware configuration.

No previous experience is required with cryptography or other security related subjects in order to use LockBox, but it can be helpful to understand the terminology and the cryptographic process. For that reason a discussion is included on the background to cryptography in Chapter 2, followed by a section that details how LockBox can help fulfill your cryptographic requirements.

LockBox Functionality

LockBox is a cross-platform library and can be used with Borland Delphi or C++Builder applications under Windows and with Kylix under Linux. The following section describes some of the functionality provided by LockBox.

RSA

To help deal with public key requirements, LockBox includes an implementation of the RSA algorithm. Using this standard, you can publish public keys with which your correspondents can encrypt and send data to you. The data can then be decrypted with your private key. LockBox can, of course, generate key-pairs for this algorithm. Keys are usually 512 bits in length.

MD5 and SHA-1

These two acronyms are cryptographically secure hashing algorithms. Hashes for files, messages, or other data can be easily calculated using LockBox.

Digital Signatures

LockBox provides the facilities for you to easily digitally sign messages and documents using the standard Digital Signature Algorithm (DSA). This algorithm builds upon the LockBox's secure hashing and public key algorithms.

Data Encryption Standard (DES)

LockBox provides a full implementation of the Data Encryption Standard (DES) symmetric algorithm. Keys are, by definition, 56 bits in length. For even greater security, you can use LockBox's version of triple-DES, the algorithm that encrypts by encoding your data three times with DES.

Advanced Encryption Standard (AES)

To maintain its lead at the forefront of encryption algorithms, LockBox includes an encapsulation of the Rijndael algorithm, the standard that will eventually replace DES. Key lengths vary but can be up to 256 bits.

Blowfish

Blowfish is a well-regarded, patent-free, public domain algorithm. LockBox provides facilities to encrypt and decrypt using this popular algorithm. Key lengths are 128 bits.

What's New in This Release?

This section describes some of the highlights of this major release of LockBox and is for users who are upgrading from an earlier version of LockBox. For more information on these features you should read the relevant sections of this documentation.

Asymmetric (public key) encryption

In response to customer requests for public key encryption, LockBox includes the RSA algorithm. You can generate private and public key pairs, encrypt and decrypt data with those key pairs.

Extra secure hashing

LockBox now supports the SHA-1 secure hashing algorithm to supplement the previously supported MD5 hashing algorithm,

Digital signatures

Once RSA and SHA-1 was available, the ability to generate and verify digital signatures for documents, messages and arbitrary data was added. LockBox supports two signatures: RSASSA and DSA.

The Advanced Encryption Standard

The National Institute of Standards and Technology (NIST) announced in October 2000 that the Rijndael algorithm would be the replacement for DES. LockBox provides an implementation of this algorithm, which is of great future importance.

Ease of use

Research showed that although LockBox provided working implementations of important encryption algorithms, using them was not always intuitive. For this release easy-to-use components have been added that encapsulate the important algorithms and that enable you to get your job done quicker without having to worry about esoteric subjects like contexts, blocks and the like.

String encryption

One of the most popular requests for LockBox was the ability to take an arbitrary string and encrypt it (and, once encrypted, to be able to decrypt it, of course). LockBox supports this important functionality by encrypting a string and Base64 encoding the result.

System Requirements

To use LockBox in Windows, you must have the following hardware and software:

- A computer capable of running Windows 9x, Me, NT, or Windows 2000.
- Delphi version 3 or later, or C++Builder version 3 or later. LockBox does not support Delphi 1, Delphi 2, or C++Builder 1.
- A hard disk with at least 10 MB of free space is strongly recommended. To install all LockBox files and compile the example programs requires about 3 MB of disk space.

To use LockBox in Linux, you must have the following hardware and software:

- A computer capable of running one of the supported distributions of Linux.
- Kylix version 1 or later.
- A hard disk with at least 10 MB of free space is strongly recommended. To install all LockBox files and compile the example programs requires about 3 MB of disk space.

Installation

Install LockBox directly from the TurboPower Product Suite CD. Simply insert the CD into your CD-ROM drive, select LockBox 2 from the list of products, click "Install", and follow the instructions. If the TurboPower introductory splash screen does not appear automatically upon insertion of the CD, run `X:\CDROM.EXE` where *X* is the letter of your CD-ROM drive.

Organization of this Manual

The manual is organized as follows:

- Chapter 1 introduces LockBox and discusses its installation and this documentation. Chapter 1 also includes an overview of the Unified Modeling Language used in TurboPower Software documentation.
- Chapter 2 provides some basic cryptography concepts and a glossary of terms and acronyms. It then explains how to use LockBox in order to implement some of these topics.
- Chapter 3 discusses the low-level routines in LockBox. These routines provide encapsulations of all the encryption and hashing algorithms in LockBox.
- Chapter 4 introduces the encryption components. These components encapsulate the various strong block ciphers in LockBox. Using them, you can easily encrypt and decrypt files, streams, buffers, and strings.
- Chapter 5 explains the hashing components. These components simplify the use of the strong cryptographic hashing algorithms in LockBox.
- Chapter 6 supplies information about the digital signature algorithms implemented in LockBox and then introduces the components that facilitate the use of such algorithms.
- Chapter 7 presents the asymmetric key manager class. This class makes the use of RSA and the Digital Signature components easier to use by encapsulating the functionality required to manage asymmetric keys.
- Chapter 8 explains the stream classes. These are simple encapsulations of the stream ciphers in LockBox to act on streams and are only present for backwards compatibility.

Each reference chapter starts with an overview of the routines and classes discussed in that chapter. Each class or collection of routines (usually contained in a unit) are then documented individually, in the following format:

Overview

A description of the class or unit.

Class hierarchy

Shows the ancestors of the class being described. The hierarchy also lists the unit in which each class is declared. Some classes in the hierarchy are identified with a number in a bullet: ❶. This indicates that some of the methods listed for the class being described are inherited from this ancestor and documented in the ancestor class.

Properties

Lists all the properties in the class. Some properties may be identified with a number in a bullet: ❶. These properties are documented in the ancestor class from which they are inherited.

Methods

A list of all the methods in the class. Some methods may be identified with a number in a bullet: ❶. These methods are documented in the ancestor class from which they are inherited.

Procedures/Functions

A list of all the procedures and functions in the class.

Reference Section

Detailed documentation for the properties, methods, procedures, and functions implemented at this level in the class hierarchy or implemented in this unit. These descriptions are in alphabetical order. They have the following format:

- Declaration of the method, procedure, or function.
- A short, one-sentence purpose. The ⚡ symbol is used to mark the purpose to make it easy to skim through these descriptions.
- Description of the property, method, procedure, or function. Parameters are also described here.
- Examples are provided in many cases.
- The *See also* section lists other properties, methods, procedures, or functions that are pertinent to this item.

Throughout the manual, the ⚡ symbol is used to mark a caution. Please pay special attention to these items.

Naming conventions

To avoid class name conflicts with components and classes included with Delphi or from other third party suppliers, all LockBox class names begin with “TLb”. The “Lb” stands for LockBox.

On-line help

Although this manual provides a complete discussion of LockBox, keep in mind that there is an alternative source of information available. Once properly installed, help is available from within the IDE. Pressing <F1> with the caret or focus on a LockBox property, routine or component displays the help for that item.

Technical Support

The best way to get an answer to your technical support question is to post it in the LockBox newsgroup on our news server (news.turbopower.com). Many of our customers find the newsgroups an invaluable resource where they can learn from the experiences of others and share ideas, in addition to getting quick answers to questions.

To get the most from the newsgroups, we recommend you use dedicated newsreader software.

Newsgroups are public, so please do not post your product serial number, product unlocking code, or any other private numbers (such as credit card numbers) in your messages.

TurboPower's KnowledgeBase is another excellent support option. It has hundreds of articles about TurboPower products accessible through an easy-to-use search engine www.turbopower.com/search. The KnowledgeBase is open 24 hours a day, 7 days a week, so you'll have another way to find answers to your questions even when we're not available.

Other support options are described in the support brochure included with LockBox. You can also read about our support options at www.turbopower.com/support.

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a visual modeling language intended to depict the design of object-oriented systems. UML consists of assorted graphical notations and a specification for combining these notations to describe the various aspects of an object-oriented system. It can be used to capture and portray both the static structure of a system and its dynamic behavior. Classes, objects, states, use cases, and components, (among other things) can all be graphically modeled with the UML.

At TurboPower Software, the UML is used to describe the design and requirements for a given system. Since UML is such a standardized way of showing relationships between aspects of a system, it makes sense for our manuals to use UML to show the associations and interactions between the various classes and components in the product.

This section provides a small primer on basic UML, enough to read the figures in this manual. If you want more information, please see *UML Distilled* by Martin Fowler (ISBN 0-201-32562-2).

Figure 1.1 shows the UML representation of a typical class. A standard class is a rectangle, divided into three compartments: the class name, the property listing, and the method listing. Both the property and method listings can be suppressed for clarity. (Sometimes the property compartment is known as the attribute compartment, but since the VCL name for these entities is “property”, we shall not use the word “attribute”.)

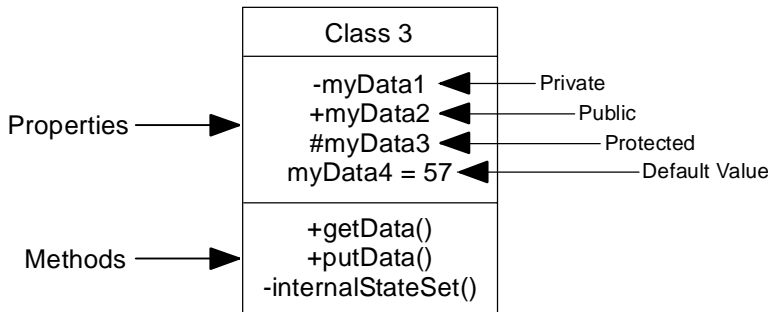


Figure 1.1: An example of a class in UML

Figure 1.1 also shows whether properties and methods are private, protected or public. A special character prior to the name shows the visibility of the identifier. If there is no such special character, no assumption about visibility can be made. Table 1.1 shows the visibility characters and their meaning.

Table 1.1: *Table of visibility characters*

Character	Meaning
-	Private
#	Protected
+	Public
	Not shown

The UML specification allows for additional compartments within a class notation. Events are shown as a fourth compartment as shown in Figure 1.2. The standard compartments of class name, properties, and methods are not required by UML to have a name, but additional compartments are. Figure 1.2 shows this by giving the bottom compartment a name of Events. Since events are always public, the visibility character is omitted. Notice that in Figure 1.2 the property compartment is suppressed.

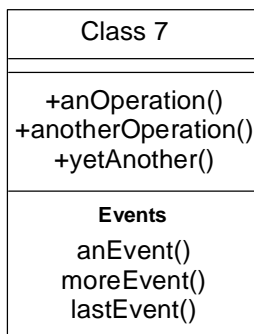


Figure 1.2: A component with events in UML

Classes are combined in class diagrams. These diagrams show the relationships between the classes. In TurboPower Software products, *inheritance* and *containment* are the main types of static relationships that can be shown.

Inheritance using UML is shown in Figure 1.3. In this diagram class 2 inherits from class 1; in other words, the arrow points to the ancestor.

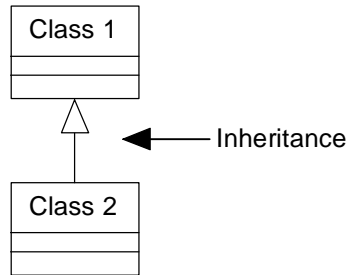


Figure 1.3: Class inheritance in UML

Containment can be shown in one of two ways: composition (containment by value) and aggregation (containment by reference). The two different kinds of containment cause some confusion. Typically, aggregation implies a set of one or more other objects used by the container object. For example, a polygon object would have a list of point objects describing the shape of the polygon. We can alter this list by adding more points or changing their values. The list of points is an *aggregation*.

The polygon might also have an object within it that described how the polygon is to be drawn: the boundary color, the fill color, whether a shadow has to be shown, etc. This object is totally contained within the polygon object, and although you can change the properties of the object, you cannot replace it with another. This state of affairs is known as *composition*. The contained object is viewed as an indelible part of the container.

Composition is shown with a black diamond and aggregation by a white diamond. In Figure 1.4, class 4 contains class 6 through composition and class 4 contains class 5 by aggregation.

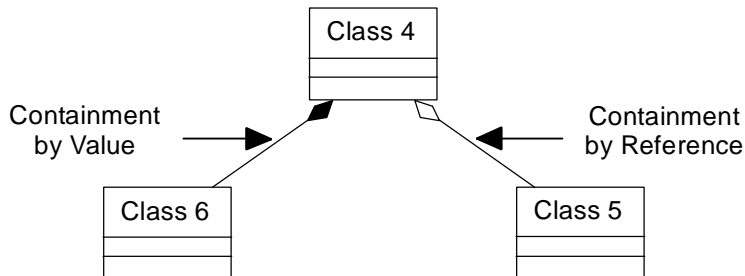


Figure 1.4: Class continuation in UML

A UML sequence diagram shows the dynamic interactions between classes and *actors* along a time line. (An actor is an entity that lies outside the system being depicted. The actor may be a person but equally well may be a server, a different system, or anything else.) A sequence diagram shows class instances as boxes at the top of vertical lines called lifelines. Time is assumed to flow from top to bottom. The sequence diagram helps with the visualization of the order of events happening in the system.

A close-ended arrow between two lifelines represents a method call or a message being sent; an a dashed line with a closed arrow heas shows a return value. Method calls are nearly always labeled with the method names. Return values typically list the data being returned. *Self-delegation* is when an instance calls a method on itself and is shown by a close-ended arrow looping back to the same lifeline.

In a sequence diagram, instances are shown as the instance name followed by a colon and then the class name. Class names are always shown, but instance names are optional.

Figure 1.5 shows an example sequence diagram. Here the user calls the Start method of Class 1. This in turn causes Class 1 to call the getData method of Class 3. Class 3 then calls its own internalSetState method, followed by a call to the Connect method of the database. It returns data to Class 1.

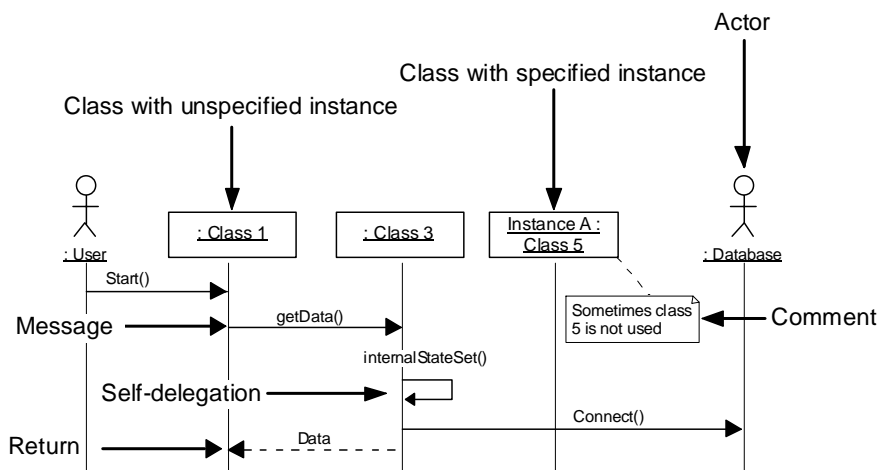


Figure 1.5: Sequence diagram in UML

Comments can be attached to all UML diagrams. They are shown as a box with a folded top left corner containing the comment text. Comments can be attached to items for clarity via a dashed line.

Suggested Reading

- Schneier, Bruce. *Applied Cryptography*. New York: John Wiley & Sons, 1996.
- Stinson, D. R. *Cryptography: Theory and Practice*. Boca Raton: CRC Press, 1995.
- RSA Laboratories. *Public-Key Cryptography Standards*. Bedford: RSA Data Security, November 1993.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge: MIT Press, 1990.
- Davies, D.W. and W.L. Price. *Security for Computer Networks*. New York: John Wiley & Sons, 1989.
- Denning, Dorothy E. *Cryptography and Data Security*. New York: Addison-Wesley, 1982.

Chapter 2: Overview

This chapter takes a look at what cryptography is, discusses some of the common techniques and methods used in dealing with secure data and communications, and explains how LockBox helps achieve cryptographic security within applications. It is not possible to provide a complete discussion of cryptography in this chapter; after all, entire books have been written on the subject. Instead, this chapter reviews some basic terminology and looks at some of the techniques that cryptography and LockBox provide. For more information about cryptography—the pros and cons, the techniques, the algorithms, the attacks—please refer to the “Suggested Reading” section in chapter 1 on page 17.

One topic not covered in this chapter is security, especially with regard to cryptanalytic attacks and risk. This area of cryptography (*cryptanalysis*) is large and the techniques can be esoteric and academic. Again, for more information, refer to the general cryptography books listed in the “Suggested Reading” section of Chapter 1. *Applied Cryptography* (Schneier, 1996) is an excellent reference.

Encryption and Decryption

Cryptography is that branch of computer science that deals with securing data and communications. The most familiar area in cryptography is *encryption*: transforming understandable data into a form that is incomprehensible and that looks like random noise. The converse operation is known as *decryption*: the process whereby encrypted data is converted back to something understandable and usable.

In cryptography, the data to encrypt and decrypt has some special names. The original, understandable data is known as *plaintext*, whereas the encrypted form is *ciphertext* (*cipher* being an alternative name for an encryption algorithm). The data is usually known as a *message*, because generally the objective is to send the data to someone else without any eavesdroppers or hackers being able to understand it. The recipient of the message does not have to be another person; it may well be a case in which the message is a file that is encrypted on the user's hard disk.

The general encryption/decryption process is shown in Figure 2.1. The plaintext message is encrypted with the encryption algorithm. The algorithm uses a secret key to perform the encryption. After this step, the message is a ciphertext message, which can be safely stored on an insecure disk or sent to someone over an insecure transmission medium. The recipient of the ciphertext then decrypts it with the corresponding decryption algorithm, using the decryption key, to produce the original plaintext message. It goes without saying that the decryption algorithm used is dependent on and derived from the encryption algorithm—indeed, sometimes the algorithms are one and the same.

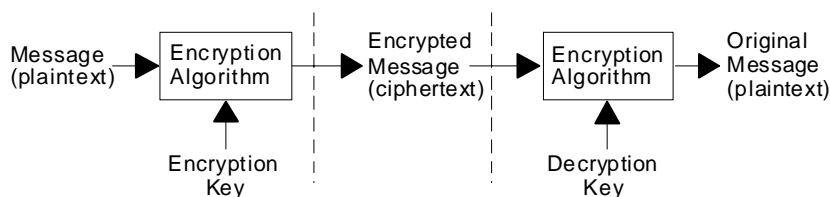


Figure 2.1: The encryption/decryption process

It should be emphasized here that the security of encrypted data is not due to keeping the encryption algorithm secret. Cryptographers have researched the standard encryption algorithms over the years since they were proposed, trying to break them through various types of attacks. The best algorithms are considered unbreakable because of this academic research, not because of secrecy. Of course, one of the good encryption algorithms could be broken tomorrow because of some new research. No algorithm can be said to be absolutely unbreakable. Instead, it is accepted that there is a high probability the algorithm is

unbreakable. The more research done on any algorithm, the higher is that probability. The best encryption algorithms derive their security entirely through the secrecy of the keys used. Messages will be safe if the keys are kept secret.

Symmetric and asymmetric algorithms

There are two classes of encryption algorithms: *symmetric* and *asymmetric*. The differences between the two lie in the use of the encryption and decryption keys. With symmetric algorithms, the same key is used for encryption and decryption. Since there is only one key, and that key must obviously be kept secret, symmetric algorithms are sometimes also known as private-key algorithms. Since both the person doing the encrypting and the person doing the decrypting must know the key, and since the encryption is worthless once that key becomes public, the keys for symmetric algorithms are frequently changed. For the highest security, they are calculated from random number generators. Key lengths vary: DES, for example, uses 56-bit keys (usually quoted as 64 bits instead, because the specification requires 8 check bits to verify that the key is valid).

Symmetric algorithms are the most popular encryption algorithms, mainly because they tend to be fast (essentially all symmetric algorithms shuffle and manipulate the bits in a message and the bits in the key through several similar cycles) and hence are very efficient at encrypting large amounts of data. LockBox provides support for the most well known symmetric algorithms: DES, triple-DES, Blowfish and Rijndael (the Advanced Encryption Standard, AES).

Asymmetric algorithms, on the other hand, use one key for encryption and another for decryption. The two keys are related in a mathematical sense. Yet, even if one of the keys is known, it is very hard to calculate the other. Asymmetric algorithms are popular because one of the keys can be published. This published key is known as the *public key*. Public keys allow anyone to use the key to encrypt a message and send it to a recipient. The only person who can decrypt the message is the recipient because the recipient is the only one who knows the other key, which is known as the *private key*. Hence, anybody can send an encrypted message, even people unknown to the recipient. Compare this to the symmetric case: for someone to send an encrypted message, the sender and recipient must exchange information about the secret key. The transmission of this secret key could be intercepted, removing any security.

LockBox provides support for the most well known asymmetric algorithm, RSA (named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman). RSA is a slow algorithm—at least compared with the standard symmetric algorithms—and so is generally used to encrypt small messages.

This limitation seems like a drawback, but in reality it doesn't much matter. In a situation where a recipient wants to allow anyone to send large amounts of data in a secure manner, using a symmetric algorithm may not be the best solution. Yes, it has a fast implementation,

but it has no public key. Neither is an asymmetric algorithm the best solution since it has a public key, but is very slow. Instead, a hybrid can be used: the sender generates a random key for a symmetric algorithm, (Blowfish for example), encrypts it with the recipient's public RSA key, and sends it. The sender then encrypts the large volume of data using Blowfish, along with this randomly generated key, and sends the ciphertext to the recipient. The recipient can decrypt the encrypted Blowfish key using the private RSA key and then easily decrypt the large ciphertext that was received. The key used for the Blowfish encryption step is then discarded and never reused.

The keys for the RSA algorithm have two parts. The first part is called the *modulus*. A modulus is a 512-bit number (64 bytes) and is the product of two 256-bit primes. The modulus is the same for both the public and the private key. The second part of an RSA key is called the *exponent*. This is a number of variable length, with the exponent of the public key usually being the smaller of the two. The two exponents, the public and the private, are related in a mathematical way, but the derivation of the private exponent from the public, and the common modulus, go beyond the scope of this manual. Essentially, with RSA encryption, the plaintext is viewed as a binary number, is raised to the power of the public exponent, and the remainder after dividing by the modulus is the ciphertext. To decrypt, the ciphertext is raised to the power of the private exponent, and the remainder after dividing by the modulus is the plaintext again. The mathematics behind RSA ensures that this all works the way it is supposed to, but again, the explanation goes beyond the scope of this manual.

Key lengths and attacks

Although this chapter does not discuss in detail any possible attacks on the encryption algorithms in LockBox, there is one that bears mentioning. This is the *brute-force attack*.

A brute-force attack is nothing more than trying every single key value until the one that decrypts the ciphertext is found. Obviously, this attack requires very fast machines (or, indeed, a large number of slower machines) in order to perform the attack in a reasonable amount of time.

To understand the scope of a brute-force attack, imagine trying to break an encrypted message encrypted with Rijndael with a key 128 bits long (16 bytes). Rijndael creates approximately 2^{128} possible keys. Even with a machine or a network of machines that can test keys at the rate of 1000 billion a second (roughly 2^{40}), it would take 2^{88} seconds to test them all, or, on average, 2^{87} seconds to find the key. Since there are approximately 2^{25} seconds in a year, the search would take 2^{62} years, a number with 19 digits. So even with this hypothetically fast machine, it would, to all intents and purposes, be impractical to break ciphertext encrypted using 128-bit Rijndael (or any other cipher with that key length).

DES is a popular algorithm to use, but be warned. It uses a 56-bit key, which is not very long. Using the putative machine or network of machines, the key could be discovered, on average, within about 9 hours. Indeed, in 1997 a network of machines on the Internet brute-force attacked ciphertext encrypted with DES in few months.

The argument just advanced was for symmetric algorithms. For asymmetric algorithms the best attack is not a brute-force attack. For RSA, as an example, the private exponent could be 256 bits long. Trying 2^{256} possible exponents is infeasible. Instead, the attack concentrates on the modulus of the public key, trying to factorize it (in an equivalent brute-force fashion) into its two 256-bit prime factors. Although there exist various mathematical tricks to factorize a large number, current knowledge is limited to factorizing a 300-bit product in a reasonable amount of time. Also, because the two primes in RSA are the same length, some of these tricks are of lesser use.

Stream and block ciphers

All of the popular encryption algorithms, whether symmetric or asymmetric, are *block ciphers*. Block ciphers are so called because they operate on the message a block at a time. Taking DES as an example, the message is divided up into 64-bit blocks (8 bytes) and then the DES algorithm and private key is applied to each 64-bit block.

Of course, seven out of eight messages, on average, will have a smaller final block than is required. On dividing up a message into 8-byte blocks, the final block will be anywhere from one byte to eight bytes long. If it is eight bytes, encrypting it is not a problem. If it is less than eight bytes it is necessary to pad it up to eight bytes somehow. The final block can be padded with zero bytes; bytes containing all set bits, random bytes, or anything else. Some block ciphers specify how the final block is to be padded. LockBox removes this issue: it performs the correct padding automatically.

Generally, with block ciphers, the ciphertext is longer than the plaintext. Sometimes it's only a few bytes longer, as with DES, and sometimes much longer. RSA increases the size of data by about 20% during encryption. With block ciphers, you cannot allocate buffers of equal size for the plaintext and the ciphertext.

Each block within a block cipher can be encrypted in one of two ways: each block can be encrypted independently or each block can be encrypted using the previous encrypted block in some way, using a feedback mechanism.

The first method is known as the *Electronic Code Book* (ECB) mode. With ECB mode, a given plaintext block, no matter where it appears, is always encrypted to the same ciphertext block. Indeed, it is theoretically possible to list all possible plaintext blocks and their corresponding ciphertext blocks somehow. Encryption would then become a process of finding a plaintext block in this "code book", which would give the corresponding

ciphertext block. This mode has one advantage: if a block accidentally gets garbled during transmission, only the single block would get corrupted during decryption—the blocks on either side would still be decrypted properly.

ECB mode also has several disadvantages. If the plaintext has lots of repetition, it's likely that the ciphertext would also have repeating blocks and this may be a point of attack for the cracker, especially if the repetition is known. Also, because each block is independently encrypted from any other, the attacker could try and substitute blocks in a possible attack.

The second method is the *Cipher Block Chaining* (CBC) mode. This mode adds a *feedback* mechanism where the prior encrypted block is fed back into the encryption of the current block, usually with an XOR operation. In other words, each block is used to modify the encryption of the next block. Each ciphertext block is then dependent not only on encrypting the corresponding plaintext block, but also on every previous plaintext block. This ensures that even if the plaintext contains many identical blocks, each similar plaintext block will encrypt to a different ciphertext block.

CBC mode works by encrypting the first plaintext block and then storing the resulting ciphertext in a *feedback register* (a fancy name for a buffer). For the next plaintext block, first XOR the block with the feedback register, and then encrypt it with the encryption algorithm. Again, output the resulting ciphertext block, and also store it in the feedback register ready for the next plaintext block. In this way, each ciphertext block will depend on every previous block.

Decryption is just as straightforward. Store the first ciphertext block in the feedback register. Since the first block is a simple encrypted block, just decrypt it and output it. For the next block, decrypt it and then XOR it with the feedback register. Store the encrypted block in the feedback register again. Continue this until the entire message is decrypted.

Because CBC mode links every block but the first to every previous block, if the ciphertext is corrupted during storage or transmission, the plaintext cannot be recovered from the point of the corruption onwards.

CBC mode forces identical plaintext blocks to encrypt to different ciphertext blocks only when some previous plaintext block is different. Two identical messages still encrypt to the same ciphertext. Even worse perhaps, two messages that begin with the same data would produce the same ciphertext up to the first difference in the plaintext. (An example of this would be a series of memos that always start with the same recipient name and the same sender name in exactly the same format.) This possibility is prevented by creating a special block containing random bytes, inserting it prior to the first block of the actual message, and then encrypting this expanded plaintext (random block plus original message). This block of random data is called the *initialization vector*. The initialization vector has no meaning; it's just there to make each message unique. When the block containing the

initialization vector is decrypted, it is just used to fill the feedback register and is otherwise ignored. A timestamp often makes a good initialization vector, but any random data can in fact be used. When an initialization vector is required, LockBox creates it automatically.

A *stream cipher* works in an entirely different way than a block cipher by encrypting the plaintext one bit at a time. A stream cipher generates a random stream of bits and XORs each bit of the plaintext to generate a bit of ciphertext. Decryption proceeds in the same manner and requires that the same stream of random bits are generated during decryption as was generated during encryption. A big advantage of this technique is that the ciphertext is the same size as the plaintext. Although stream ciphers are very fast—the code can be optimized to work a byte at a time, for example—that speed comes at the expense of security. There are several well-known attacks on stream ciphers and so, although LockBox provides some encapsulations of stream ciphers, the standard block ciphers offer much higher levels of security.

Hashing and Digital Signing

Although nearly everyone is familiar with cryptography in terms of encryption, there is another part of cryptography that is equally as important: cryptographic hashing.

A *cryptographic hash* is an algorithm that takes an entire message and, through a process of shuffling, manipulating, and processing the bytes using logical operations, generates a small *message digest* of the data.

Hashing algorithms are commonly used with hash tables and string dictionaries. The standard hashes for these applications usually generate a 32-bit integer value used to access the string in the hash table. The remainder after dividing the hash value by the table size is used as an index. One example of such a hash is a Cyclic Redundancy Check (CRC). The point of the hash is two-fold: to calculate an integer value from some data that is generally not integer-like, and to try and ensure that similar looking data has different hashes. The more efficiently a hash algorithm produces random looking values, the less likely it is that collisions will occur between items in the hash table.

Cryptographic hashes generate much larger hash values than dictionary hashes. LockBox provides support for two standard hashes: MD5 and SHA-1. MD5 produces 16 byte hashes (128 bits), whereas SHA-1 generates 20-byte hashes (160 bits). Larger hashes make it more difficult to devise a message that has the same hash value as another. Hence, to ensure that an important document isn't altered to gain some advantage, or to cause a dispute later on, a hash of the message is appended to the message itself before distributing it. It is very hard with cryptographic hashes to alter the original message in such a way that its digest does not change.

Of course, merely appending a hash to the document is not that secure—it is easy to alter the document, calculate a new digest and append it—so it is necessary to somehow stamp the hash to indicate that it is the original and has not been altered. This is the domain of the digital signature.

A *digital signature* is a combination of a cryptographic hash and an asymmetric encryption algorithm. To stamp a document in such a way that it is obvious that it is the unaltered original, calculate a message digest of the document (either with MD5 or SHA-1). This hash contains the “essence” of the document in a concise form. Encrypting the message digest with a private RSA key creates a digital signature. To verify the signature, someone else would calculate the same message digest of the document then decrypt the digital signature using the public RSA key. If the two values are the same, the document is the same as the one originally signed. If they are different then the document has been changed.

The premise behind digital signatures is that the signature was created using a secure key—a private RSA key—and yet it can be verified by anybody else by using a public key. It is very hard to fake a digital signature; the difficulty is exactly the same as trying to break the RSA algorithm. LockBox provides support for two main digital signature algorithms, DSA and RSASSA.

Using LockBox

Cryptography is such a broad and detailed science, it may be difficult to perceive how LockBox can help implement cryptographic security in specific applications with specific data.

The answer is that LockBox provides complete implementations of the standard algorithms. For example, LockBox contains routines and classes with which to encrypt a block of data using the Blowfish algorithm. LockBox helps encrypt a complete buffer into an output buffer, encrypt one stream into another, or encrypt one file to produce a second. This breadth of usage extends to DES, triple-DES, Rijndael, and RSA, as well as Blowfish.

For ease of use, the reference sections of this manual have been divided up into separate sections, one for each algorithm. In this way, all of the information about how to use one particular algorithm appears in the same vicinity, within the same few pages; it is not necessary to jump from one part of the reference section to another.

The individual algorithms are introduced in Chapter 3 “Low Level Routines” on page 33. For convenience, Table 2.1 shows the page numbers where each algorithm is discussed.

Table 2.1: *Where to find descriptions of algorithms*

Algorithm	Page
Blowfish	32
DES	41
Triple-DES	97
Rijndael (AES)	78
RSA	86
MD5	73
SHA-1	93

Notes on encryption support

Prior to using the low-level encryption routines for a particular cipher, it is necessary to initialize a data block that LockBox can use for the encryption step. This data block is called the *context*. The context for a given algorithm is initialized by calling an `InitEncryptXxx` routine (for example, `InitEncryptBF` sets up the context for encrypting with Blowfish), and then passed to the routine that does the actual encrypting or decrypting.

The context for an encryption algorithm varies from cipher to cipher. However, the initial creation of the context usually involves encoding the key in some way.

This leads to the question: how is a key generated? The passwords used to sign onto systems, PCs and web sites are keys. In general, these passwords are small strings, say about 10 characters in length. Some symmetric algorithms require a key length of 128 bits (16 bytes). It is possible to use 16 letter words but in general these are not very secure. Think of it this way: the power of a 128-bit key comes from the fact that any key is as good as any other. A brute-force attack would have to try them all. If the variations are limited to bits that are ASCII characters, the number of bits that are different in the key is significantly reduced. Instead of 2^{128} possible keys, only 26^{16} possible keys would exist if the characters are limited to lower-case letters. This would be the same as a 75-bit key. Of course, the limit would be even smaller because the key would be using actual words, or combinations of them, which are much fewer in number.

However, not many people can memorize a 128-bit key for a symmetric algorithm. The same goes for RSA: the private part of a private key, the exponent, is just as long and, even worse, it is calculated based on the values of two 256-bit prime numbers.

For the symmetric encryption algorithms, LockBox generates keys using two techniques. The first one is designed for those situations that require a key that will be reused often and that must be memorized in some fashion by someone. The second method is for those cases where the key can be stored electronically somewhere (dangerous!) or for one-off applications where a key can be securely exchanged using the RSA public-key algorithm and then used just once before it is destroyed completely.

The first method uses the MD5 secure hash on a *passphrase*. A passphrase is a string, much like a password but longer. It is usually a complete sentence instead of a single word. The longer the passphrase the better, but in general it should be at least 32 characters long. A normal sentence is relatively easy to remember, and the longer it is, the harder it is to crack. The MD5 hash is exactly 128 bits long, which is long enough for a key for most of the symmetric algorithms. For an algorithm that requires less length, like the DES algorithm, any group of 64 bits within the hash can be used: the first 64 bits, or the last 64, or any concatenated set of 64 bits. The hash is also randomized—it's a feature of cryptographic hashes—and so doesn't have easily exploitable weaknesses. Indeed, the weakness with this scheme is the passphrase and only the passphrase. This method is implemented by the `GenerateMD5Key` routine, described on page 59.

The second method is even simpler: the key is generated as a set of random bytes. For a 128-bit key, 16 random bytes are needed. With this method, the weakness is in the random number generator and in the seed used to start the sequence off. LockBox contains a stronger random number generator than the standard one provided with your compiler (it is more cryptographically secure) and uses it to generate the random keys. It uses the system clock as a random seed. This method is implemented by the `GenerateRandomKey` routine described on page 59.

For the RSA public-key algorithm, the keys are generated through a rigorous mathematical process. LockBox provides a single routine, `CreateRSAKeys`, which will do all of the work. It generates two random primes of 256 bits each, and from those will calculate the modulus for the RSA keys and the public and the private exponents.

There are encryption routines for each cipher to encrypt streams, files, and arbitrary buffers of data. One commonly requested option is to encrypt strings, and LockBox supports this requirement by not only encrypting a string but also encoding it with the Base64 algorithm. This ensures that LockBox-encrypted strings only contain standard printable characters, and avoids the problems that nulls and non-printable characters may cause. Of course these encrypted strings can be decrypted with LockBox.

The following steps describe how to use LockBox's low-level routines for a symmetric algorithm. This recipe uses Blowfish as an example, but applies to all symmetric algorithms.

1. Generate the key using `GenerateMD5Key` or `GenerateRandomKey`.
2. To encrypt a buffer of data, proceed to step 3. To encrypt a stream or a file, proceed to step 6.
3. Initialize the context for the encryption engine, using `InitEncryptBF`.
4. Split the buffer into blocks, and encrypt them in order using `EncryptBF` or `EncryptBFCBC`. (Note that different ciphers have different sized blocks, so take this into consideration). The blocks are encrypted in place, so copy them into a temporary block prior to calling the routine.
5. Once all the blocks are encrypted, the encryption process is over. There is no clean up to do.
6. To encrypt a stream, call `BFEncryptStream` or `BFEncryptStreamCBC`, passing the key generated in step 1. To encrypt a file, use `BFEncryptFile` or `BFEncryptFileCBC`.

For the RSA algorithm, LockBox's only asymmetric algorithm, proceed as follows:

1. Generate the keys using `GenerateRSAKeys`.
2. To encrypt a buffer of data, proceed to step 3. To encrypt a stream or a file, proceed to step 6.
3. Split the buffer into blocks, and encrypt them in order using `EncryptRSA`. The blocks are encrypted in place, so copy them into a temporary block prior to calling the routine.
4. Once you've encrypted all the blocks, the encryption process is over. There is no clean up to do, apart from freeing the RSA key objects.
5. To encrypt a stream, call `RSASyncStream`, passing a key generated in step 1. To encrypt a file, use `RSASyncFile`.

Rather than use the full names of the various algorithms in the various low-level routines, LockBox uses a simple acronym convention. Table 2.2 shows this naming convention.

Table 2.2: *LockBox naming conventions*

Cipher name	Acronym
Blowfish (ECB mode)	BF
Blowfish (CBC mode)	BFCBC
DES (ECB mode)	DES
DES (CBC mode)	DESCBC
Triple-DES (ECB mode)	TripleDES
Triple-DES (CBC mode)	TripleDESCBC
Rijndael (AES) (ECB mode))	RDL
Rijndael (AES) (CBC mode)	RDLCBC
RSA	RSA

As you can see from Table 2.2, the bare acronym is used for ECB mode, whereas the acronym decorated with “CBC” denotes CBC mode. (Note that RSA does not support either ECB or CBC mode. It uses a different methodology entirely that injects some random noise into each encrypted block.)

Notes on hashing support

The low-level hashing support for MD5 and SHA-1 operates in a similar way to the encryption support. To use a message digest algorithm directly, first initialize the context of the hash using the `InitMD5` or the `InitSHA1`. This does not require the use of a key; instead the initialization routine merely prepares the context structure for the hash that is about to begin. Then update the context with blocks of data with the `UpdateMD5` or `UpdateSHA1`. Once all of the data is processed, the final step is to finalize the hash using the `FinalizeMD5` or `FinalizeSHA1`, which returns the message digest.

LockBox also provides routines to hash a string with the MD5 algorithm. This makes it possible to take a passphrase and generate a message digest from it for use as an encryption key. Please see `GenerateMD5Key` on page 59 for more information.

Classes

In addition to the algorithms and the low-level support, LockBox also provides easy-to-use encryption classes and components. All of the initialization and taking care of context is hidden inside the class implementation. Using the class hierarchy, it is easy to create encryption objects, pass in the key and then use the objects. LockBox's encryption classes provide routines to encrypt blocks, files, streams, and strings.

Other Algorithms

Apart from the standard, academically tested algorithms, LockBox also contains a set of LockBox specific ciphers and hashes. In general, these are variations of well-known techniques. Since these algorithms have not been rigorously tested and analyzed, it is unknown how secure they are and use of them is not recommended. They are incorporated in LockBox for backwards compatibility only. Applications created with earlier versions of LockBox that use any of these algorithms should be converted to use one of the standard algorithms.

It is not that the algorithms are insecure per se, it is just that they have not been subject to the same intense critical and academic analysis that the accepted cryptographic algorithms have had. They may be secure, but, equally well, they may have hidden flaws that would only be revealed through cryptanalysis. The LockBox-specific algorithms should therefore be viewed with caution.

The algorithms concerned are the LockBox Block Cipher (LBC), the LockBox Stream Cipher (LSC), the LockBox Quick Cipher (LQC), the LockBox Message Digest (LMD), and the Mix128 message digest.

In a different category are the random number ciphers, the 32-bit and the 64-bit Random Number Generator algorithms. To be sure, these algorithms provide a rapid method for encrypting data with the ciphertext being the same size as the plaintext, but, like all random number generator ciphers, they suffer from certain standard attacks. If major security is an issue, ignore these algorithms and concentrate instead on the standard algorithms.

In the reference sections, these untested algorithms will be marked with the ⚠ caution symbol.

Chapter 3: Low-Level Cryptographic Routines

At its heart, Lockbox consists of a set of units that encapsulate the cryptographic algorithms as low-level routines. These units concerned are listed in Table 3.1.

Table 3.1: *Low-level routine units*

Unit Name	Description
LbCipher	The setup, use, and finalization of the various cryptographic algorithms.
LbRandom	A cryptographically secure random number generator class.
LbBigInt	The big integer arithmetic routines for RSA.
LbAsym	The low-level classes and routines for RSA.
LbProc	The routines for encrypting files and streams.
LbString	The routines for encrypting strings.

This chapter will describe these low-level routines, whereas the following two chapters will describe the classes built upon these routines. Note that this manual does not document the big integer routines in LbBigInt. These routines are solely designed for the RSA key generation and encryption and are not intended for general use.

To make it easier to understand the low-level routines, this chapter is divided up into multiple sections, one per algorithm. That way the information can be easily found about a particular cryptographic algorithm in one place and not have to be searched for in the entire manual.

For each algorithm, there is a brief discussion pointing out the salient points of the algorithm, a how-to on using LockBox to implement the algorithm for the important algorithms, and finally a reference section detailing the individual routines, the unit in which unit they can be found, and a small example of their use.

Blowfish Cipher

Bruce Schneier designed Blowfish in 1993. It is a symmetric block cipher with key lengths from 32 bits to 448 bits (LockBox's implementation limits the key length to 128 bits, however). The block length is 64 bits, just like DES. The algorithm is patent, royalty and license free.

Since its introduction, Blowfish has been extensively analyzed, with papers being published by many cryptographers. It is acknowledged as a strong cipher, especially with the recommended number of rounds. There have been attacks and weaknesses published on using Blowfish with fewer rounds, but note that LockBox implements the recommended number and so doesn't suffer from these problems.

For more information please see <http://www.counterpane.com/blowfish.html>.

In the next example the ciphertext is going to be larger than the plaintext so the routine should pass along the size of the original plaintext buried somehow in the ciphertext. This example uses a simple technique of inserting a new first block that consists of the size together with a set of random bytes. The final block of the buffer, if it is smaller than 8 bytes, will be padded with random bytes from the previous encryption step. The following example shows how to encrypt and decrypt a stream of data with the Blowfish algorithm, from basic principles:

```
const
    Passphrase = 'Three can keep a secret, if two are dead.';

procedure TestEncryptBF(aInStream, aOutStream : TStream);
var
    Key          : TKey128;
    Context      : TBFCContext;
    BFBlock      : TBFBBlock;
    BytesRead    : longint;
    Seed         : integer;
begin
    {reset the streams}
    aInStream.Position := 0;
    aOutStream.Position := 0;
    {create a key}
    GenerateMD5Key(Key, Passphrase);
    {initialize the Blowfish context}
    InitEncryptBF(Key, Context);
    {set up the first block to contain the size of
     the plaintext, together with random noise}
    BFBlock[1] := aInStream.Size;
    Seed := $12345678;
```

```

BFBlock[0] := Ran03(Seed);
{encrypt and write this block}
EncryptBF(Context, BFBlock, True);
aOutputStream.Write(BFBlock, sizeof(BFBlock));
{read the first block from the stream}
BytesRead := aInStream.Read(BFBlock, sizeof(BFBlock));
{while there is still data to encrypt, do so}
while (BytesRead <> 0) do begin
    EncryptBF(Context, BFBlock, True);
    aOutputStream.Write(BFBlock, sizeof(BFBlock));
    BytesRead := aInStream.Read(BFBlock, sizeof(BFBlock));
end;
end;

procedure TestDecryptBF(aInStream, aOutputStream : TStream);
var
    Key      : TKey128;
    Context  : TBFCContext;
    BFBlock  : TBFBBlock;
    BytesToDecrypt : longint;
    BytesToWrite  : longint;
begin
    {reset the streams}
    aInStream.Position := 0;
    aOutputStream.Position := 0;
    {create a key}
    GenerateMD5Key(Key, Passphrase);
    {initialize the Blowfish context}
    InitEncryptBF(Key, Context);
    {read the first block and decrypt to get the
     size of the plaintext}
    aInStream.ReadBuffer(BFBlock, sizeof(BFBlock));
    EncryptBF(Context, BFBlock, False);
    BytesToDecrypt := BFBlock[1];
    {while there is still data to decrypt, do so}
    while (BytesToDecrypt <> 0) do begin
        aInStream.ReadBuffer(BFBlock, sizeof(BFBlock));
        EncryptBF(Context, BFBlock, False);
        if (BytesToDecrypt > sizeof(BFBlock)) then
            BytesToWrite := sizeof(BFBlock)
        else
            BytesToWrite := BytesToDecrypt;
        dec(BytesToDecrypt, BytesToWrite);
        aOutputStream.WriteBuffer(BFBlock, BytesToWrite);
    end;
end;
end;

```

The following example shows how to encrypt a stream into another, using the routine provided in LockBox:

```
var
    Key : TKey128;
begin
    GenerateMD5Key(Key, Passphrase);
    BFEncryptStream(InStream, OutStream, Key, True);
end;
```

Procedures / Functions

BFEncryptFile	BFEncryptStreamCBC	EncryptBF
BFEncryptFileCBC	BFEncryptString	EncryptBFCBC
BFEncryptStream	BFEncryptStringCBC	InitEncryptBF

Reference Section

BFEncryptFile

procedure, LbProc

```
procedure BFEncryptFile(const InFile, OutFile : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a file using the Blowfish cipher in ECB mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`

BFEncryptFileCBC

procedure, LbProc

```
procedure BFEncryptFileCBC(const InFile, OutFile : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a file using the Blowfish cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`


```
procedure BFEncryptStream(InStream, OutStream : TStream;  
    Key : TKey128; Encrypt : Boolean);
```

```
TKey128 = array [0..15] of Byte;
```

3 ↪ Encrypts or decrypts a stream using the Blowfish cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`BFEncryptStream` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`

BFEncryptStreamCBC**procedure, LbProc**

```
procedure BFEncryptStreamCBC(  
    InStream, OutStream : TStream; Key : TKey128; Encrypt : Boolean);
```

```
TKey128 = array [0..15] of Byte;
```

↪ Encrypts or decrypts a stream using the Blowfish cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`BFEncryptStreamCBC` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`

```
procedure BFEncryptString(const InString : string;  
    var OutString : string; const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a string using the Blowfish cipher in ECB mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`

BFEncryptStringCBC**procedure, LbProc**

```
procedure BFEncryptFileCBC(const InString, OutString : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a string using the Blowfish cipher in CBC mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateRandomKey`, `LbCipher.GenerateMD5Key`

```

procedure EncryptBF(const Context : TBFBContext;
  var Block : TBFBBlock, Encrypt : Boolean);

TBFBContext = packed record
  PBox : array[0..(bf_N+1)] of LongInt;
  SBox : array[0..3, 0..255] of LongInt;
end;

TBFBBlock = array[0..1] of LongInt;

```

↪ Encrypts or decrypts a block of data using the Blowfish cipher in ECB mode.

Context must be initialized by a call to `InitEncryptBF` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted. If `Encrypt` is `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

See also: `InitEncryptBF`

EncryptBFCBC

procedure, LbCipher

```

procedure EncryptBFCBC(const Context : TBFBContext;
  const Prev : TBFBBlock; var Block : TBFBBlock, Encrypt : Boolean);

TBFBContext = packed record
  PBox : array[0..(bf_N+1)] of LongInt;
  SBox : array[0..3, 0..255] of LongInt;
end;

TBFBBlock = array[0..1] of LongInt;

```

↪ Encrypts or decrypts a block of data using the Blowfish cipher in CBC mode.

Context must be initialized by a call to `InitEncryptBF` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted. If `Encrypt` is `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

The `Prev` parameter specifies the previously encrypted block. The way this routine works is to take the current value of `Block`, XOR it with `Prev`, and then encrypt it. It is then required to pass the new value of `Block` as the `Prev` parameter for the next call to `EncryptBFCBC`. This code snippet (which should be read in conjunction with the example code in the introduction to Blowfish) shows how to encrypt using CBC mode:

```

{set up the first block to contain the size of
 the plaintext, together with random noise}
BFBBlock[1] := aInStream.Size;
Seed := $12345678;
BFBBlock[0] := Ran03(Seed);

```

```

{encrypt and write this block}
FillChar(PrevBFBlock, sizeof(PrevBFBlock), 0);
EncryptBF CBC(Context, PrevBFBlock, BFBlock, True);
aOutputStream.Write(BFBlock, sizeof(BFBlock));
PrevBFBlock := BFBlock;
{read the first block from the stream}
BytesRead := aInStream.Read(BFBlock, sizeof(BFBlock));
{while there is still data to encrypt, do so}
while (BytesRead <> 0) do begin
    EncryptBF CBC(Context, PrevBFBlock, BFBlock, True);
    aOutputStream.Write(BFBlock, sizeof(BFBlock));
    PrevBFBlock := BFBlock;
    BytesRead := aInStream.Read(BFBlock, sizeof(BFBlock));
end;

```

This code example is a little more involved since we need two spare Blowfish blocks, one to temporarily hold the just-about-to-be-decrypted block, and the other the previous encrypted block. Again read this code in conjunction with the example code in the introduction. This code snippet shows how to decrypt using CBC mode:

```

{read the first block and decrypt to get the
 size of the plaintext}
aInStream.ReadBuffer(BFBlock, sizeof(BFBlock));
FillChar(PrevBFBlock, sizeof(PrevBFBlock), 0);
TempBFBlock := BFBlock;
EncryptBF CBC(Context, PrevBFBlock, BFBlock, False);
PrevBFBlock := TempBFBlock;
BytesToDecrypt := BFBlock[1];
{while there is still data to decrypt, do so}
while (BytesToDecrypt <> 0) do begin
    aInStream.ReadBuffer(BFBlock, sizeof(BFBlock));
    TempBFBlock := BFBlock;
    EncryptBF CBC(Context, PrevBFBlock, BFBlock, False);
    PrevBFBlock := TempBFBlock;
    if (BytesToDecrypt > sizeof(BFBlock)) then
        BytesToWrite := sizeof(BFBlock)
    else
        BytesToWrite := BytesToDecrypt;
    dec(BytesToDecrypt, BytesToWrite);
    aOutputStream.WriteBuffer(BFBlock, BytesToWrite);
end;

```

See also: `InitEncryptBF`

```
procedure InitEncryptBF(Key : TKey128; var Context : TBFCContext);  
TKey128 = array [0..15] of Byte;  
  
TBFCContext = packed record  
    PBox : array[0..(bf_N+1)] of LongInt;  
    SBox : array[0..3, 0..255] of LongInt;  
end;
```

↪ Initializes a context for use with the Blowfish cipher.

InitEncryptBF initializes Context for use with either EncryptBF or EncryptBFCBC. Key is the key used to encrypt and decrypt each block of data. This key can be created with GenerateRandomKey or GenerateMD5Key.

For encrypting a set of data, either use EncryptBF exclusively or EncryptBFCBC exclusively. Do not mix the two block encryption routines for a single encryption round.

See also: EncryptBF, EncryptBFCBC, LbCipher.GenerateRandomKey,
LbCipher.GenerateMD5Key

DES Cipher

The Data Encryption Standard (DES) algorithm was the first encryption standard. It was adopted as a federal standard in 1976. IBM proposed the algorithm to the National Bureau of Standards (the precursor of the National Institute of Standards and Technology) based upon an earlier algorithm called LUCIFER. The National Security Agency refined some internal aspects of the algorithm during a review process. The algorithm is free of royalties and licenses.

DES is a symmetric block cipher. The key length is usually quoted as 64 bits, but in reality only 56 bits of these 64 participate in the encryption algorithm. The other eight bits are parity or check bits and are stripped prior to the encryption or decryption process. The block size is 64 bits (8 bytes).

During its acceptance phase, DES was extensively analyzed and is now accepted to be a strong algorithm, except for one area: the key length. A 56-bit key is within the reach of brute-force attacks using a network of high-powered machines. DES should therefore only be used for temporary data; data that would be out-of-date before someone brute-force cracked an encrypted message.

To counteract the key length problem, the triple-DES algorithm is now widely used. See page 97 for details on the triple-DES algorithm.

In the next example the ciphertext is going to be larger than the plaintext so the routine should pass along the size of the original plaintext buried somehow in the ciphertext. This example uses a simple technique of inserting a new first block that consists of the size together with a set of random bytes. The final block of the buffer, if it is smaller than 8 bytes, will be padded with random bytes from the previous encryption step. The following example shows how to encrypt and decrypt a stream of data with the DES algorithm, from basic principles:

```
const
    Passphrase = 'Three can keep a secret, if two are dead.';

procedure TestEncryptDES(aInStream, aOutStream : TStream);
type
    TMyDESBlock = array [0..1] of LongInt;
var
    Key          : TKey128;
    DESKey       : TKey64;
    Context      : TDESContext;
    DESBlock     : TDESBlock;
    BytesRead    : longint;
    Seed         : integer;
begin
```

```

{reset the streams}
aInStream.Position := 0;
aOutStream.Position := 0;
{create a key}
GenerateMD5Key(Key, Passphrase);
Move(Key, DESKey, sizeof(DESKey));
{initialize the DES context}
InitEncryptDES(DESKey, Context, True);
{set up the first block to contain the size of
 the plaintext, together with random noise}
TMyDESBlock(DESBlock)[1] := aInStream.Size;
Seed := $12345678;
TMyDESBlock(DESBlock)[0] := Ran03(Seed);
{encrypt and write this block}
EncryptDES(Context, DESBlock);
aOutStream.Write(DESBlock, sizeof(DESBlock));
{read the first block from the stream}
BytesRead := aInStream.Read(DESBlock, sizeof(DESBlock));
{while there is still data to encrypt, do so}
while (BytesRead <> 0) do begin
    EncryptDES(Context, DESBlock);
    aOutStream.Write(DESBlock, sizeof(DESBlock));
    BytesRead := aInStream.Read(DESBlock, sizeof(DESBlock));
end;
end;

procedure TestDecryptDES(aInStream, aOutStream : TStream);
type
    TMyDESBlock = array [0..1] of LongInt;
var
    Key          : TKey128;
    DESKey       : TKey64;
    Context      : TDESContext;
    DESBlock     : TDESBlock;
    BytesToDecrypt : longint;
    BytesToWrite  : longint;
begin
    {reset the streams}
    aInStream.Position := 0;
    aOutStream.Position := 0;
    {create a key}
    GenerateMD5Key(Key, Passphrase);
    Move(Key, DESKey, sizeof(DESKey));
    {initialize the Blowfish context}
    InitEncryptDES(DESKey, Context, False);
    {read the first block and decrypt to get

```

```

    the size of the plaintext}
aInStream.ReadBuffer(DESBLOCK, sizeof(DESBLOCK));
EncryptDES(Context, DESBLOCK);
BytesToDecrypt := TMyDesBlock(DESBLOCK)[1];
{while there is still data to decrypt, do so}
while (BytesToDecrypt <> 0) do begin
    aInStream.ReadBuffer(DESBLOCK, sizeof(DESBLOCK));
    EncryptDES(Context, DESBLOCK);
    if (BytesToDecrypt > sizeof(DESBLOCK)) then
        BytesToWrite := sizeof(DESBLOCK)
    else
        BytesToWrite := BytesToDecrypt;
    dec(BytesToDecrypt, BytesToWrite);
    aOutStream.WriteBuffer(DESBLOCK, BytesToWrite);
end;
end;

```

Procedures / Functions

DESEncryptFile	DESEncryptString	InitEncryptDES
DESEncryptFileCBC	DESEncryptStringCBC	ShrinkDESKey
DESEncryptStream	EncryptDES	
DESEncryptStreamCBC	EncryptDESCBC	

Reference Section

DESEncryptFile

procedure, LbProc

```
procedure DESEncryptFile(const InFile, OutFile : string;  
    Key : TKey64; Encrypt : Boolean);  
  
TKey64 = array [0..7] of Byte;
```

↪ Encrypts or decrypts a file using the DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`. Since `GenerateMD5Key` creates a 128-bit key, it does not matter which 64 bits are taken out of this generated key for the DES key (the easiest 64 to take are probably the first 64 bits).

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

DESEncryptFileCBC

procedure, LbProc

```
procedure DESEncryptFileCBC(const InFile, OutFile : string;  
    Key : TKey64; Encrypt : Boolean);  
  
TKey64 = array [0..7] of Byte;
```

↪ Encrypts or decrypts a file using the DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`. Since `GenerateMD5Key` creates a 128-bit key, it does not matter which 64 bits are taken out of this generated key for the DES key (the easiest 64 to take are probably the first 64 bits).

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

```
procedure DESEncryptStream(InStream, OutStream : TStream;  
    Key : TKey64; Encrypt : Boolean);
```

```
TKey64 = array [0..7] of Byte;
```

↳ Encrypts or decrypts a stream using the DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`. Since `GenerateMD5Key` creates a 128-bit key, it does not matter which 64 bits are taken out of this generated key for the DES key (the easiest 64 to take are probably the first 64 bits).

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`DESEncryptStream` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

DESEncryptStreamCBC**procedure, LbProc**

```
procedure DESEncryptStreamCBC(InStream, OutStream : TStream;  
    Key : TKey64; Encrypt : Boolean);
```

```
TKey64 = array [0..7] of Byte;
```

↳ Encrypts or decrypts a stream using the DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`. Since `GenerateMD5Key` creates a 128-bit key, it does not matter which 64 bits are taken out of this generated key for the DES key (the easiest 64 to take are probably the first 64 bits).

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`DESEncryptStreamCBC` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

```
procedure DESEncryptString(const InString : string;  
    var OutString : string; const Key : TKey64; Encrypt : Boolean);  
  
TKey64 = array [0..7] of Byte;
```

3 ↪ Encrypts or decrypts a string using the DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

```
procedure DESEncryptFileCBC(const InString, OutString : string;  
    Key : TKey64; Encrypt : Boolean);  
  
TKey64 = array [0..7] of Byte;
```

↪ Encrypts or decrypts a string using the DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

EncryptDES**procedure, LbCipher**

```

procedure EncryptDES(const Context : TDESContext;
  var Block : TDESBlock);

TDESContext = packed record
  TransformedKey : array [0..31] of LongInt;
  Encrypt : Boolean;
end;

TDESBlock = array[0..7] of Byte;

```

↪ Encrypts or decrypts a block of data using the DES cipher in ECB mode.

Context must be initialized by a call to InitEncryptDES prior to calling this routine. If Encrypt is True, Block is encrypted; if False, Block is decrypted. Note that the Block variable is directly modified by this routine.

Note that it is the Context variable that defines whether the block is encrypted or decrypted.

See also: InitEncryptDES

EncryptDESCBC**procedure, LbCipher**

```

procedure EncryptDESCBC(const Context : TDESContext;
  const Prev : TDESBlock; var Block : TDESBlock);

TDESContext = packed record
  TransformedKey : array [0..31] of LongInt;
  Encrypt : Boolean;
end;

TDESBlock = array[0..7] of Byte;

```

↪ EncryptDESCBC encrypts or decrypts a block of data using the Data Encryption Standard and CBC mode.

Context must be initialized by a call to InitEncryptDES prior to calling this routine. If Encrypt is True, Block is encrypted. If Encrypt is False, Block is decrypted. Note that the Block variable is directly modified by this routine.

The Prev parameter specifies the previously encrypted block. The way this routine works is to take the current value of Block, XOR it with Prev, and then encrypt it. It is then required to pass the new value of Block as the Prev parameter for the next call to EncryptDESCBC. The technique is the same as shown by the example for EncryptBFCBC.

Note that it is the Context variable that defines whether the block is encrypted or decrypted.

See also: EncryptBFCBC, InitEncryptDES

```

procedure InitEncryptDES(const Key : TKey64;
  var Context : TDESContext; Encrypt : Boolean);

TKey64 = array [0..7] of Byte;

TDESContext = packed record
  TransformedKey : array [0..31] of LongInt;
  Encrypt : Boolean;
end;

```

↪ Initializes a context for use with the DES cipher.

InitEncryptDES initializes Context for use with either EncryptDES or EncryptDESCBC. Key is the key used to encrypt and decrypt each block of data. This key can be created with GenerateRandomKey or GenerateMD5Key. Since GenerateMD5Key creates a 128-bit key, it does not matter which 64 bits are taken out of this generated key for the DES key (the easiest 64 to take are probably the first 64 bits).

For encrypting a set of data, either use EncryptDES exclusively or EncryptDESCBC exclusively. Do not mix the two block encryption routines for a single encryption round.

See also: EncryptDES, EncryptDESCBC, LbCipher.GenerateMD5Key,
LbCipher.GenerateRandomKey, ShrinkDESKey

ShrinkDESKey

function, LbCipher

```

procedure ShrinkDESKey(var Key : TKey64);

TKey64 = array [0..7] of Byte;

```

↪ Reduces the effective number of bits in the key.

Key can then be passed to InitEncryptDES or InitEncryptTripleDES to prepare for encryption or decryption using the weaker key.

☛ **Caution:** Do not use. This routine is provided for backwards compatibility only. The DES key length is short enough already, making brute-force attacks possible; reducing the effective number of bits increases the probability that such an attack will succeed in less time. Also, having a weaker key does not speed up the DES implementation.

See also: InitEncryptDES, InitEncryptTripleDES

ELF Hash

The ELF (Executable and Linking Format) hash is a simple hash algorithm, invented to digest strings into 32-bit integers for the linking of object files. This hash is used in string dictionaries and hash tables. It is provided as part of LockBox for completeness' sake; because the hash is so small (only 32-bits) it is not secure.

Procedures / Functions

HashELF

StringHashELF

Reference Section

HashELF

function, LbCipher

```
procedure HashELF(var Digest : LongInt;  
  const Buf; BufSize : LongInt);
```

3

↳ Generates a hash of a buffer using the ELF hashing algorithm.

HashELF hashes the contents of Buf using the ELF hash algorithm. BufSize is the number of bytes in Buf. The hash is returned in Digest.

StringHashELF

function, LbCipher


```
procedure StringHashELF(var Digest : LongInt; const Str : string);
```

↳ Generates a hash of a string using the ELF hashing algorithm.

StringHashELF hashes the contents of the string Str using the ELF hash algorithm. The hash is returned in Digest.

LockBox Block Cipher

The LockBox Block Cipher is a symmetric block cipher. The key size is 128 bits. The block size is 16 bytes (128 bits). It is not secure.

 **Caution:** This cipher is provided for backward compatibility only. Please do not use this cipher in new applications; use one of the well-known ciphers instead. LockBox Block Cipher has not been cryptanalyzed.

3

Procedures / Functions

EncryptLBC	LBCEncryptFile	LBCEncryptStreamCBC
EncryptLBCCBC	LBCEncryptFileCBC	
InitEncryptLBC	LBCEncryptStream	

Reference Section

EncryptLBC

procedure, LbCipher

```
procedure EncryptLBC(const Context : TLBCContext;
  var Block : TLBCBlock);

TLBCContext = packed record
  Encrypt : Boolean;
  Rounds : LongInt;
  case Byte of
    0: (SubKeys64 : array [0..15] of TKey64);
    1: (SubKeysInts : array [0..3, 0..7] of LongInt);
  end;

TKey64 = array [0..7] of Byte;

TLBCBlock = array[0..3] of LongInt;
```

↳ Encrypts or decrypts a block of data using the LockBox Block Cipher in ECB mode.

Context must be initialized by a call to `InitEncryptTripleDES` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted. If `Encrypt` is `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

See also: `InitEncryptLBC`

```
procedure EncryptLBCCBC(const Context : TLBCContext;  
    const Prev : TLBCBlock; var Block : TLBCBlock);  
  
TLBCContext = packed record  
    Encrypt : Boolean;  
    Rounds : LongInt;  
    case Byte of  
        0: (SubKeys64 : array [0..15] of TKey64);  
        1: (SubKeysInts : array [0..3, 0..7] of LongInt);  
    end;  
  
TKey64 = array [0..7] of Byte;  
TLBCBlock = array[0..3] of LongInt;
```

↳ Encrypts or decrypts a block of data using the LockBox Block Cipher in CBC mode.

Context must be initialized by a call to InitEncryptLBC prior to calling this routine. Whether EncryptLBCCBC encrypts or decrypts Block is determined by the value passed as the Encrypt parameter to InitEncryptLBC.

See also: InitEncryptLBC

InitEncryptLBC**procedure, LbCipher**

```

procedure InitEncryptLBC(const Key : TKey128;
  var Context : TLBCCContext; Rounds : LongInt; Encrypt : Boolean);
TKey128 = array [0..15] of Byte;
TLBCCContext = packed record
  Encrypt : Boolean;
  Rounds : LongInt;
  case Byte of
    0: (SubKeys64 : array [0..15] of TKey64);
    1: (SubKeysInts : array [0..3, 0..7] of LongInt);
end;
TKey64 = array [0..7] of Byte;

```

↳ Initializes a context for use with the LockBox Block Cipher.

InitEncryptLBC initializes Context for use with EncryptLBC. Key is the key used to encrypt and decrypt each block of data. If Encrypt is True, calls to EncryptLBC will encrypt the data block. If Encrypt is False, calls to EncryptLBC will decrypt the data block.

Rounds should be between 4 and 16: the higher the value, the better the security. If Rounds is less than 4 it will be forced silently to 4; if greater than 16 it will be forced to 16.

See also: EncryptLBC

LBCEncryptFile**procedure, LbProc**

```

procedure LBCEncryptFile(const InFile, OutFile : string;
  Key : TKey128; Rounds : LongInt; Encrypt : Boolean);
TKey128 = array [0..15] of Byte;

```

↳ Encrypts or decrypts a file using the LockBox Block Cipher in ECB mode.

Key is the key used to encrypt or decrypt the file. This key can be created with GenerateRandomKey or GenerateMD5Key.

If Encrypt is True, the contents of InFile are encrypted and written to OutFile. If Encrypt is False, the contents of InFile are decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

Rounds should be between 4 and 16: the higher the value, the better the security. If Rounds is less than 4 it will be forced silently to 4; if greater than 16 it will be forced to 16.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

```
procedure LBCEncryptFileCBC(const InFile, OutFile : string;  
    Key : TKey128; Rounds : LongInt; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a file using the LockBox Block Cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

Rounds should be between 4 and 16: the higher the value, the better the security. If Rounds is less than 4 it will be forced silently to 4; if greater than 16 it will be forced to 16.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

LBCEncryptStream**procedure, LbProc**

```
procedure LBCEncryptStream(InStream, OutStream : TStream;  
    Key : TKey128; Rounds : LongInt; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a stream using the LockBox Block Cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`LBCEncryptStream` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

Rounds should be between 4 and 16: the higher the value, the better the security. If Rounds is less than 4 it will be forced silently to 4; if greater than 16 it will be forced to 16.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

```
procedure LBCEncryptStreamCBC(InStream, OutStream : TStream;  
    Key : TKey128; Rounds : LongInt; Encrypt : Boolean);  
  
    TKey128 = array [0..15] of Byte;
```

3

↳ Encrypts or decrypts a stream using the LockBox Block Cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InStream` are encrypted and written to `OutStream`. If `Encrypt` is `False`, the contents of `InStream` are decrypted and written to `OutStream`.

`LBCEncryptStreamCBC` does not reset the position of `InStream` or `OutStream` prior to processing. The routine will, however, process all of the remaining bytes in `InStream`.

`Rounds` should be between 4 and 16: the higher the value, the better the security. If `Rounds` is less than 4 it will be forced silently to 4; if greater than 16 it will be forced to 16.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

LockBox Key Generation Routines

For several of the block ciphers, a key must be provided. LockBox provides a couple of key generation routines for the purpose of creating a key.

The first key generation routine, `GenerateRandomKey`, is the simplest to understand. It creates a key containing random bytes. Use this routine to generate keys that are only used once before discarding. The one-off key will need to be transmitted to the recipient of the ciphertext. A public-key method is recommended for this. Please see the discussion of using a public-key method in “Symmetric and asymmetric algorithms” on page 19.

For cases where it would be useful to have a key that can be reused, and yet not be difficult to remember 16 random-looking bytes, the MD5 hash of a Passphrase can be used. A passphrase is a long phrase or sentence (we recommend at least 32 letters long, including spaces and punctuation) that can be easily memorize. Use lines from Shakespeare, lyrics from a favorite song, sentences from famous documents, and the like; whatever is easy to remember. The `GenerateMD5Key` routine will take the passphrase and generate a 128-bit key from it.

Procedures / Functions

`GenerateLMDKey`

`GenerateMD5Key`

`GenerateRandomKey`

Reference Section

GenerateLMDKey

procedure, LbCipher

```
procedure GenerateLMDKey (var Key; KeySize : Integer;  
    const Str : string);
```

↳ Generates a key using the LockBox Message Digest algorithm.

GenerateLMDKey generates a Key of KeySize bytes by applying the LockBox Message Digest algorithm to the text contained in Str.

☛ **Caution:** This routine uses the LockBox Message Digest algorithm, which is not secure.

GenerateMD5Key

procedure, LbCipher

```
procedure GenerateMD5Key (var Key : TKey128; const Str : string);  
TKey128 = array [0..15] of Byte;
```

↳ Generates a 128-bit key using the MD5 message digest algorithm.

GenerateMD5Key generates a 128-bit (16 byte) Key, by applying the MD5 secure hashing algorithm to the text contained in Str.

GenerateRandomKey

procedure, LbCipher

```
procedure GenerateRandomKey (var Key; KeySize : Integer);
```

↳ Generates a key using a random number generator.

GenerateRandomKey generates a Key of KeySize bytes, by using a random number generator.

The random number generator used is the standard one provided by the compiler. The seed for this generator is initialized from the system clock prior to generating the key.

LockBox Message Digest

The LockBox Message Digest is a hashing algorithm. It does not generate a fixed-size hash, instead the hash size can be specified to be anywhere from one byte to 256 bytes long.

The LMD algorithm requires an initialization phase as well as a finalization phase. LockBox provides a single HashLMD routine, however, if full control is required over the hashing process (maybe the data will be hashed in separate pieces at separate times, and they are not to be concatenated all in one buffer), LockBox provides separate routines for the initialization, hashing, and finalization phases. First, call InitLMD to prepare the context structure used by the other two routines. Second, call UpdateLMD as many times as necessary to hash each of the pieces of the data. The final step is to convert the context structure into the hash by calling FinalizeLMD.

⚠ **Caution:** The LockBox Message Digest algorithm has not been cryptanalyzed. It is provided for backwards compatibility only. Please use MD5 or SHA-1 instead; they have been more rigorously analyzed.

Procedures / Functions

FinalizeLMD

InitLMD

UpdateLMD

HashLMD

StringHashLMD

Reference Section

FinalizeLMD

function, LbCipher

```
procedure FinalizeLMD(var Context : TLMDContext; var Digest;
    DigestSize : LongInt);
    TLMDContext = array[0..279] of Byte;
```

↳ Completes the generation of a LockBox Message Digest by returning the hash.

FinalizeLMD takes the LMD context structure obtained from a call to InitLMD followed by one or more to UpdateLMD and calculates the final message digest from it. The hash is returned in Digest. DigestSize is the size of the Digest buffer in bytes. Its value must be between 1 and 256, and will be forced into that range if required.

See also: HashLMD, InitLMD, UpdateLMD

HashLMD

procedure, LbCipher

```
procedure HashLMD (var Digest; DigestSize : LongInt;
    const Buf; BufSize : LongInt);
```

↳ Generates a secure hash of a buffer using the LockBox Message Digest algorithm.

HashLMD performs a hash on the contents of Buf. BufSize is the number of bytes in Buf. The hash is returned in Digest. DigestSize is the size of the Digest buffer in bytes. Its value must be between 1 and 256, and will be forced into that range if required.

HashLMD hashes the buffer by calling InitLMD, UpdateLMD, and FinalizeLMD. If more control is required over the process, these three routines can be called directly. For example, if all of the data is not available at one time, multiple calls will need to be made to UpdateLMD as the data becomes available. However, for the majority of requirements, using HashLMD is the easiest course.

See also: FinalizeLMD, InitLMD, UpdateLMD

InitLMD

procedure, LbCipher

```
procedure InitLMD(var Context : TLMDContext);
    TLMDContext = array[0..279] of Byte;
```

↳ Initializes the context structure for the LockBox Message Digest algorithm.

The normal sequence of events for generating an LMD hash is to call InitLMD to prepare the context buffer, call UpdateLMD one or more times, and then call FinalizeLMD to retrieve the message digest.

HashLMD performs this sequence of events automatically. If no special processing is required during the generation of the message digest, use HashLMD instead of the sequence of calls to InitLMD, UpdateLMD, and FinalizeLMD.

See also: HashLMD, FinalizeLMD, UpdateLMD

StringHashLMD

procedure, LbCipher

```
procedure StringHashLMD (var Digest; DigestSize : LongInt;
    const Str : string);
```

↳ Generates a hash (or message digest) using the LockBox Message Digest.

StringHashLMD uses the HashLMD routine to perform a hash on the contents of Str. The hash is returned in Digest. DigestSize is the size of the Digest buffer in bytes. Its value must be between 1 and 256, and will be forced into that range if required.

See also: HashLMD

UpdateLMD

procedure, LbCipher

```
procedure UpdateLMD(var Context : TLMDContext; const Buf;
    BufSize : LongInt);
```

```
TLMDContext = array[0..279] of Byte;
```

↳ Updates a LockBox Message Digest context with a hashed form of the data.

UpdateLMD is one step in the generation of a hash using the LMD algorithm. It updates Context with a digested form of the data in Buf. Call UpdateLMD once if all of the data is ready, or multiple times if all of the data is not available. BufSize is the number of bytes to be processed from Buf.

See also: FinalizeLMD, HashLMD, InitLMD

LockBox Quick Cipher

The LockBox Quick Cipher is a symmetric block cipher. The key size is 128 bits. The block size is 8 bytes (64 bits). It is not secure.

3

Unlike other block ciphers, the LockBox Quick Cipher does not require any context to be maintained across encryption calls. The context is generated from the key at run-time.

- ⚠ **Caution:** This cipher is provided for backward compatibility only. Please do not use this cipher in new applications; use one of the well-known ciphers instead. LockBox Quick Cipher has not been cryptanalyzed.

Procedures / Functions

EncryptLQC

LQCEncryptFile

LQCEncryptStream

EncryptLQCCBC

LQCEncryptFileCBC

LQCEncryptStreamCBC

Reference Section

EncryptLQC

procedure, LbCipher

```
procedure EncryptLQC(const Key : TKey128;  
    var Block : TLQCBLOCK; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;  
TLQCBLOCK = array[0..1] of Integer;
```

↳ Encrypts or decrypts a block of data using the LockBox Quick Cipher in ECB mode.

Key contains the key value used to encrypt or decrypt the block of data. If Encrypt is True, Block is encrypted. If Encrypt is False, Block is decrypted. Note that the Block variable is directly modified by this routine.

The LockBox Quick Cipher does not require an initialization step, and therefore no context structure.

EncryptLQCCBC

procedure, LbCipher

```
procedure EncryptLQCCBC(  
    const Key : TKey128; const Prev : TLQCBLOCK;  
    var Block : TLQCBLOCK; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;  
TLQCBLOCK = array[0..1] of Integer;
```

↳ Encrypts or decrypts a block of data using the LockBox Quick Cipher in CBC mode.

Key contains the key value used to encrypt or decrypt the block of data. If Encrypt is True, Block is encrypted. If Encrypt is False, Block is decrypted. Note that the Block variable is directly modified by this routine.

The LockBox Quick Cipher does not require an initialization step, and therefore no context structure.

```
procedure LQCEncryptFile(const InFile, OutFile : string;  
    const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

3

↳ Encrypts or decrypts a file using the LockBox Quick Cipher in ECB mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

LQCEncryptFileCBC**procedure, LbProc**

```
procedure LQCEncryptFileCBC(const InFile, OutFile : string;  
    const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a file using the LockBox Quick Cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

```
procedure LQCEncryptStream(InStream, OutStream : TStream;  
    const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a stream using the LockBox Quick Cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with GenerateRandomKey or GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

LQCEncryptStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

LQCEncryptStreamCBC**procedure, LbProc**

```
procedure LQCEncryptStreamCBC(InStream, OutStream : TStream;  
    const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↳ Encrypts or decrypts a stream using the LockBox Quick Cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with GenerateRandomKey or GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

LQCEncryptStreamCBC does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

LockBox Random Number Ciphers

LockBox provides two random number ciphers, the first based on a 32-bit random number generator and the second on a 64-bit generator. Both are stream ciphers, and hence the ciphertext is the same size as the original plaintext from which it was produced.

A random number cipher works by generating a reproducible series of random numbers from some seed. The cipher encrypts plaintext by XORing it with the random numbers produced by the generator. The key used by the cipher is merely the initial seed value for the random number generator.

Since the algorithm is based on the XOR operation, it is not necessary to specify whether encrypting or decrypting is occurring with this cipher.

☛ **Caution:** This cipher is provided for backward compatibility only. Please do not use this cipher in new applications; use one of the well-known ciphers instead. The random number generators used by LockBox are linear congruential generators and there are well-known attacks against ciphers built on these types of generators.

Procedures / Functions

EncryptRNG32

InitEncryptRNG32

RNG32EncryptFile

EncryptRNG64

InitEncryptRNG64

RNG64EncryptFile

Reference Section

EncryptRNG32

procedure, LbCipher

```
procedure EncryptRNG32(  
  var Context : TRNG32Context; var Buf; BufSize : LongInt);  
  
  TRNG32Context = array [0..3] of Byte;
```

↳ Encrypts or decrypts a buffer of data using the 32-bit Random Number cipher.

EncryptRNG32 encrypts or decrypts BufSize bytes of data contained in Buf. If Buf contains plaintext (unencrypted data), it is encrypted. If Buf contains ciphertext (encrypted data), it is decrypted. Context must be initialized by a call to InitEncryptRNG32 prior to calling this routine.

The reason that it does not need to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

See also: InitEncryptRNG32

EncryptRNG64

procedure, LbCipher

```
procedure EncryptRNG64(  
  var Context: TRNG64Context; var Buf; BufSize : LongInt);  
  
  TRNG64Context = array [0..7] of Byte;
```

↳ Encrypts or decrypts a buffer of data using the 64-bit Random Number cipher.

EncryptRNG64 encrypts or decrypts BufSize bytes of data contained in Buf. If Buf contains plaintext (unencrypted data), it is encrypted. If Buf contains ciphertext (encrypted data), it is decrypted. Context must be initialized by a call to InitEncryptRNG64 prior to calling this routine.

The reason that it does not need to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

See also: InitEncryptRNG64

InitEncryptRNG32**procedure, LbCipher**

```

procedure InitEncryptRNG32(Key : LongInt;
    var Context : TRNG32Context);

TRNG32Context = array [0..3] of Byte;

```

↳ Initializes a context for use with the 32-bit Random Number cipher.

InitEncryptRNG32 initializes Context for use with EncryptRNG32. Key is used as the seed to prime the 32-bit random number generator.

See also: EncryptRNG32

InitEncryptRNG64**procedure, LbCipher**

```

procedure InitEncryptRNG64(KeyHi, KeyLo : LongInt;
    var Context : TRNG64Context);

TRNG64Context = array [0..7] of Byte;

```

↳ Initializes a context for use with the 64-bit Random Number cipher.

InitEncryptRNG64 initializes Context for use with EncryptRNG64. KeyHi and KeyLo are used as the seed to prime the 64-bit random number generator.

See also: EncryptRNG64

RNG32EncryptFile**procedure, LbProc**

```

procedure RNG32EncryptFile(
    const InFile, OutFile : string; Key : LongInt);

```

↳ Encrypts or decrypts a file using the 32-bit Random Number cipher.

Key is used as the seed to prime the 32-bit random number generator.

If InFile contains plaintext (unencrypted data), it is encrypted and written to OutFile. If InFile contains ciphertext (encrypted data), it is decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

The reason that it does not have to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

```
procedure RNG64EncryptFile(const InFile, OutFile : string;  
    KeyHi, KeyLo : LongInt);
```

↳ Encrypts or decrypts a file using the 64-bit Random Number cipher.

KeyHi and KeyLo are used as the seed to prime the 64-bit random number generator.

If InFile contains plaintext (unencrypted data), it is encrypted and written to OutFile. If InFile contains ciphertext (encrypted data), it is decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

The reason that it does not have to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

LockBox Stream Cipher

The LockBox Stream Cipher is a symmetric stream cipher. Essentially it works performing XOR operations with the data and context. Some shuffling also occurs. It is fast, but very insecure.

⚠ **Caution:** This cipher is provided for backward compatibility only. Please do not use this cipher in new applications. It is easily breakable.

Procedures / Functions

EncryptLSC

InitEncryptLSC

LSCEncryptFile

Reference Section

EncryptLSC

procedure, LbCipher

```
procedure EncryptLSC(  
    var Context : TLSCContext; var Buf; BufSize : LongInt);  
  
TLSCContext = packed record  
    Index : LongInt;  
    Accumulator : LongInt;  
    SBox : array [0..255] of Byte;  
end;
```

↳ Encrypts or decrypts a buffer of data using the LockBox Stream Cipher.

EncryptLSC encrypts or decrypts the BufSize bytes of data contained in Buf. If Buf contains plaintext (unencrypted data), it is encrypted. If Buf contains ciphertext (encrypted data), it is decrypted. Context must be initialized by a call to InitEncryptLSC prior to calling this routine.

The reason that it does not have to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

See also: InitEncryptLSC

InitEncryptLSC

procedure, LbCipher

```
procedure InitEncryptLSC(  
    const Key; KeySize : Integer; var Context : TLSCContext);  
  
TLSCContext = packed record  
    Index : LongInt;  
    Accumulator : LongInt;  
    SBox : array [0..255] of Byte;  
end;
```

↳ Initializes a context for use with the LockBox Stream Cipher.

InitEncryptLSC initializes Context for use with EncryptLSC. Key can be any size, but no more than the first 256 bytes are used by EncryptLSC to encrypt and decrypt the data block. KeySize is the size (in bytes) of Key.

See also: EncryptLSC

```
procedure LSEncryptFile(  
    const InFile, OutFile : string; const Key; KeySize : Integer);
```

↳ Encrypts or decrypts a file using the LockBox Stream Cipher.

3

Key is the key used to encrypt or decrypt the file. Key can be any size, but no more than the first 256 bytes are used to encrypt and decrypt the data block. KeySize is the size (in bytes) of Key.

If InFile contains plaintext (unencrypted data), it is encrypted and written to OutFile. If InFile contains ciphertext (encrypted data), it is decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

The reason that it does not have to be specified whether to encrypt or decrypt with this routine is that the algorithm essentially works via the XOR operation. It is not secure.

MD5 Hash

MD5 is a cryptographically secure message digest algorithm. Ron Rivest (of RSA fame) designed it in 1992 to be a stronger version of the MD4 algorithm, which had several attacks devised against it.

MD5 generates a 128-bit hash (16 bytes) from its input. The input data is divided up into 512-bit blocks (the algorithm goes into some detail about how to pad out the last block, the one that may be less than 512 bits in size) and then each block is divided up into 32-bit integers ready for the mixing stage. Since all operations act on 32-bit integers, MD5 is very efficient on 32-bit CPUs.

The MD5 algorithm requires an initialization phase as well as a finalization phase. Although LockBox provides separate routines for the initialization, hashing and finalization phases, it is recommended that the single HashMD5 routine is used. However, should full control be required over the hashing process (maybe the data will be hashed in separate pieces at separate times, and not concatenated all in one buffer), the LockBox routines are designed to work as follows. First, call InitMD5 to prepare the context structure used by the other two routines. Second, call UpdateMD5 as many times as necessary to hash each of the pieces of the data. The final step is to convert the context structure into the hash by calling FinalizeMD5. The following example shows how to hash a simple buffer:

```
var
    Context : TMD5Context;
    Digest   : TMD5Digest;
begin
    InitMD5(Context);
    UpdateMD5(Context, Buffer, sizeof(Buffer));
    FinalizeMD5(Context, Digest);
    ..Digest now contains the MD5 hash of buffer..
```

The MD5 algorithm is one of the most prevalent secure hashing algorithms in wide use, the other being SHA-1.

Procedures / Functions

FinalizeMD5

HashMD5

InitMD5

StringHashMD5

UpdateMD5

Reference Section

FinalizeMD5

function, LbCipher

```
procedure FinalizeMD5(  
    var Context : TMD5Context; var Digest : TMD5Digest);  
  
TMD5Digest = array[0..15] of Byte;  
  
TMD5Context = array[0..87] of Byte;
```

↳ Completes the generation of an MD5 hash by returning the digest.

FinalizeMD5 takes the MD5 context structure obtained from a call to InitMD5 followed by one or more to UpdateMD5 and calculates the final message digest from it. The hash is returned in Digest.

See also: HashMD5, InitMD5, UpdateMD5

HashMD5

function, LbCipher

```
procedure HashMD5(  
    var Digest : TMD5Digest; const Buf; BufSize : LongInt);  
  
TMD5Digest = array[0..15] of Byte;
```

↳ Generates a secure hash of a buffer using the MD5 hashing algorithm.

HashMD5 performs a hash on the contents of Buf. BufSize is the number of bytes in Buf. The hash is returned in Digest.

HashMD5 hashes the buffer by calling InitMD5, UpdateMD5, and FinalizeMD5. If more control over the process is needed, call these three routines directly. For example, if all of the data is not available at one time it will be necessary to make multiple calls to UpdateMD5 as the data becomes available. However, for the majority of requirements, using HashMD5 is the easiest course.

See also: FinalizeMD5, InitMD5, UpdateMD5

InitMD5**procedure, LbCipher**

```
procedure InitMD5(var Context : TMD5Context);
```

```
TMD5Context = array[0..87] of Byte;
```

↳ Initializes the context structure for the MD5 message digest algorithm.

The normal sequence of events for generating an MD5 hash is to call InitMD5 to prepare the context buffer, call UpdateMD5 one or more times, and then call FinalizeMD5 to retrieve the message digest.

HashMD5 performs this sequence of events automatically. If no special processing is required during the generation of the message digest, use HashMD5 instead of the sequence of calls to InitMD5, UpdateMD5, and FinalizeMD5.

See also: HashMD5, FinalizeMD5, UpdateMD5

StringHashMD5**function, LbCipher**

```
procedure StringHashMD5(
  var Digest : TMD5Digest; const Str : string);
```

↳ Generates a hash (or message digest) of an input string using the MD5 message digest algorithm.

StringHashMD5 uses the HashMD5 routine to perform a hash on the contents of Str. The hash value is returned in Digest.

See also: HashMD5

UpdateMD5**procedure, LbCipher**

```
procedure UpdateMD5(
  var Context : TMD5Context; const Buf; BufSize : LongInt);
```

↳ Updates an MD5 context with a hashed form of the data.

UpdateMD5 is one step in the generation of a hash using the MD5 algorithm. It updates Context with a digested form of the data in Buf. Call UpdateMD5 once if all of the data is ready, or multiple times if all of the data is not available at once. BufSize is the number of bytes to be processed from Buf.

See also: FinalizeMD5, HashMD5, InitMD5

Mix128 Hash

The Mix128 hash is a simple hash algorithm, designed to digest strings into 32-bit integers. Because the hash is so small (only 32-bits) it is not secure.

Procedures / Functions

HashMix128

StringHashMix128

Reference Section

HashMix128

function, LbCipher

```
procedure HashMix128(  
    var Digest : LongInt; const Buf; BufSize : LongInt);
```

↳ Generates a hash of a buffer using the Mix128 hashing algorithm.

HashMix128 hashes the contents of Buf using the Mix128 hash algorithm. BufSize is the number of bytes in Buf. The hash is returned in Digest.

StringHashMix128

function, LbCipher

```
procedure StringHashMix128(  
    var Digest : LongInt; const Str : string);
```

↳ Generates the hash of a string using the Mix128 hashing algorithm.

StringHashMix128 hashes the contents of the string Str using the Mix128 hash algorithm. The hash is returned in Digest.

Rijndael Cipher

The Rijndael encryption algorithm was the finalist in the Advanced Encryption Standard (AES) selection process run by the National Institute of Standards and Technology (NIST). AES is the replacement for the Data Encryption Standard (DES).

Two Belgian cryptographers, Joan Daemen and Vincent Rijmen, devised Rijndael for the AES competition in 1998. It is a symmetric block cipher with variable key lengths and block sizes. In their paper, the authors promoted key lengths of 128, 192, and 256 bits, with block sizes of 128, 192, and 256 bits. LockBox supports 128, 192, and 256-bit keys and 128-bit blocks. Rijndael (pronounced “Rhine Dahl”) is interesting from a cryptographic viewpoint as, unlike many of the ciphers invented since DES, is it not a Feistel cipher.

For more information please see <http://csrc.nist.gov/encryption/aes/>.

In the following example, since the ciphertext is larger than the plaintext, the routine should pass along the size of the original buffer somehow. This example uses a simple technique of inserting a new first block that consists of the size together with a set of random bytes. The final block of the buffer, if it is smaller than 8 bytes, is padded with random bytes. The following example shows how to encrypt and decrypt a buffer of data with the Rijndael algorithm:

```
procedure TestEncryptRijndael(aInStream, aOutStream : TStream);
type
  TMyRDLBlock = array [0..3] of LongInt;
var
  Key      : TKey128;
  Context  : TRDLContext;
  RDLBlock : TRDLBlock;
  BytesRead : longint;
  Seed      : integer;
begin
  {reset the streams}
  aInStream.Position := 0;
  aOutStream.Position := 0;
  {create a key}
  GenerateMD5Key(Key, Passphrase);
  {initialize the Rijndael context}
  InitEncryptRDL(Key, sizeof(Key), Context, True);
  {set up the first block to contain the size of
   the plaintext, together with random noise}
  TMyRDLBlock(RDLBlock)[1] := aInStream.Size;
  Seed := $12345678;
```

```

TMyRDLBlock(RDLBlock)[0] := Ran03(Seed);
TMyRDLBlock(RDLBlock)[2] := Ran03(Seed);
TMyRDLBlock(RDLBlock)[3] := Ran03(Seed);
{encrypt and write this block}
EncryptRDL(Context, RDLBlock);
aOutputStream.Write(RDLBlock, sizeof(RDLBlock));
{read the first block from the stream}
BytesRead := aInStream.Read(RDLBlock, sizeof(RDLBlock));
{while there is still data to encrypt, do so}
while (BytesRead <> 0) do begin
    EncryptRDL(Context, RDLBlock);
    aOutputStream.Write(RDLBlock, sizeof(RDLBlock));
    BytesRead := aInStream.Read(RDLBlock, sizeof(RDLBlock));
end;
end;

procedure TestDecryptRijndael(aInStream, aOutputStream : TStream);
type
    TMyRDLBlock = array [0..3] of LongInt;
var
    Key          : TKey128;
    Context      : TRDLContext;
    RDLBlock     : TRDLBlock;
    BytesToDecrypt : longint;
    BytesToWrite  : longint;
begin
    {reset the streams}
    aInStream.Position := 0;
    aOutputStream.Position := 0;
    {create a key}
    GenerateMD5Key(Key, Passphrase);
    {initialize the Blowfish context}
    InitEncryptRDL(Key, sizeof(Key), Context, False);
    {read the first block and decrypt to get the
     size of the plaintext}
    aInStream.ReadBuffer(RDLBlock, sizeof(RDLBlock));
    EncryptRDL(Context, RDLBlock);
    BytesToDecrypt := TMyRDLBlock(RDLBlock)[1];
    {while there is still data to decrypt, do so}
    while (BytesToDecrypt <> 0) do begin
        aInStream.ReadBuffer(RDLBlock, sizeof(RDLBlock));
        EncryptRDL(Context, RDLBlock);
        if (BytesToDecrypt > sizeof(RDLBlock)) then
            BytesToWrite := sizeof(RDLBlock)
        else

```

```
        BytesToWrite := BytesToDecrypt;  
        dec(BytesToDecrypt, BytesToWrite);  
        aOutStream.WriteBuffer(RDLBlock, BytesToWrite);  
    end;  
end;
```

3

Procedures / Functions

EncryptRDL

EncryptRDLCBC

InitEncryptRDL

RDLEncryptFile

RDLEncryptFileCBC

RDLEncryptStream

RDLEncryptStreamCBC

RDLEncryptString

RDLEncryptStringCBC

Reference Section

EncryptRDL

procedure, LbCipher

```
procedure EncryptRDL(
  const Context : TRDLContext; var Block : TRDLBlock);

TRDLContext = packed record
  Encrypt : Boolean;
  Dummy : array[0..2] of Byte;
  Rounds : DWord;
  case Byte of
    0 : (W : array[0..56] of TRDLVector);
    1 : (Rk : array[0..14] of TRDLBlock);
  end;

TRDLBlock = array[0..15] of Byte;

TKey192 = array[0..23] of Byte;
TKey256 = array[0..31] of Byte;

TRDLVector = record
  case Byte of
    0 : (dw : DWord);
    1 : ( bt : array[0..3] of Byte);
  end;
```

↪ Encrypts or decrypts a block of data using the Rijndael cipher in ECB mode.

Context must be initialized by a call to InitEncryptRDL prior to calling this routine. If Encrypt is True, Block is encrypted; if False, Block is decrypted. Note that the Block variable is directly modified by this routine.

See also: InitEncryptRDL

```

procedure EncryptRDLCBC(const Context : TRDLContext;
  const Prev : TRDLBlock; var Block : TRDLBlock);

TRDLContext = packed record
  Encrypt : Boolean;
  Dummy : array[0..2] of Byte;
  Rounds : DWord;
  case Byte of
    0 : (W : array[0..56] of TRDLVector);
    1 : (Rk : array[0..14] of TRDLBlock);
  end;

TRDLBlock = array[0..15] of Byte;

TKey192 = array[0..23] of Byte;
TKey256 = array[0..31] of Byte;

TRDLVector = record
  case Byte of
    0 : (dw : DWord);
    1 : ( bt : array[0..3] of Byte);
  end;

```

↪ Encrypts or decrypts a block of data using the Rijndael cipher in CBC mode.

Context must be initialized by a call to `InitEncryptRDL` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted; if `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

The `Prev` parameter specifies the previously encrypted block. The way this routine works is to take the current value of `Block`, XOR it with `Prev`, and then encrypt it. It is then required to pass the new value of `Block` as the `Prev` parameter for the next call to `EncryptRDLCBC`. The technique is the same as shown by the example for `EncryptBF CBC`.

See also: `EncryptBF CBC`, `InitEncryptRDL`

```

procedure InitEncryptRDL(const Key; KeySize : Longint;
  var Context : TRDLContext; Encrypt : Boolean);

TRDLContext = packed record
  Encrypt : Boolean;
  Dummy : array[0..2] of Byte;
  Rounds : DWord;
  case Byte of
    0 : (W : array[0..56] of TRDLVector);
    1 : (Rk : array[0..14] of TRDLBlock);
  end;

TRDLBlock = array[0..15] of Byte;

TKey192 = array[0..23] of Byte;
TKey256 = array[0..31] of Byte;

TRDLVector = record
  case Byte of
    0 : (dw : DWord);
    1 : ( bt : array[0..3] of Byte);
  end;

```

↪ Initializes a context for use with the Rijndael cipher.

InitEncryptRDL initializes Context for use with either EncryptRDL or EncryptRDLCBC. Key is the key used to encrypt and decrypt each block of data. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

For encrypting a set of data, either use EncryptRDL exclusively or EncryptRDLCBC exclusively. Do not mix the two block encryption routines for a single encryption round.

See also: EncryptRDL, EncryptRDLCBC, GenerateRandomKey, GenerateMD5Key


```
procedure RDLEncryptFile(const InFile, OutFile : string;  
    const Key; KeySize : Longint; Encrypt : Boolean);
```

↳ Encrypts or decrypts a file using the Rijndael cipher in ECB mode.

3

Key is the key used to encrypt or decrypt the file. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InFile are encrypted and written to OutFile. If Encrypt is False, the contents of InFile are decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

RDLEncryptFileCBC**procedure, LbProc**

```
procedure RDLEncryptFileCBC(const InFile, OutFile : string;  
    const Key; KeySize : Longint; Encrypt : Boolean);
```

↳ Encrypts or decrypts a file using the Rijndael cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InFile are encrypted and written to OutFile. If Encrypt is False, the contents of InFile are decrypted and written to OutFile. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

```
procedure RDLEncryptStream(InStream, OutStream : TStream;  
    const Key; KeySize : Longint; Encrypt : Boolean);
```

↪ Encrypts or decrypts a stream using the Rijndael cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

RDLEncryptStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

RDLEncryptStreamCBC

procedure, LbProc

```
procedure RDLEncryptStreamCBC(InStream, OutStream : TStream;  
    const Key; KeySize : Longint; Encrypt : Boolean);
```

↪ Encrypts or decrypts a stream using the Rijndael cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

RDLEncryptStreamCBC does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

```
procedure RDLEncryptString(const InString : string;
  var OutString : string; const Key : KeySize : Longint;
  Encrypt : Boolean);
```

```
TKey128 = array [0..15] of Byte;
```

3 ↪ Encrypts or decrypts a string using the Rijndael cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InString are encrypted, converted to Base64 encoding and written to OutString. If Encrypt is False, the contents of InString are converted from Base64 encoding, decrypted and written to OutString.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

RDLEncryptStringCBC**procedure, LbProc**

```
procedure RDLEncryptFileCBC(const InString, OutString : string;
  const Key : KeySize : Longint; Encrypt : Boolean);
```

```
TKey128 = array [0..15] of Byte;
```

↪ Encrypts or decrypts a string using the Rijndael cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. KeySize is the length of the key and should have a value of 16 (for 128-bits), 24 for (192-bits), or 32 (for 256 bits). This key can be created with GenerateRandomKey or, if it is 128-bits in length, GenerateMD5Key.

If Encrypt is True, the contents of InString are encrypted, converted to Base64 encoding and written to OutString. If Encrypt is False, the contents of InString are converted from Base64 encoding, decrypted and written to OutString.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

RSA Cipher

The RSA public key cipher was invented by Ron Rivest, Adi Shamir, and Leonard Adleman. It is an asymmetric block cipher using two keys, a public key that can be published, and a private key.

Although RSA was originally patented, the patent ran out on September 20, 2000. It is now patent, royalty and license free.

The RSA algorithm consists of two processes. The first one is an *encoding method*, which splits the plaintext into small blocks and then pads those small blocks into 64-byte blocks. There are several such encoding methods, each of which uses a different padding scheme. The second process is the RSA encryption itself, which is fixed.

LockBox supports the algorithm known as RSAES-PKCS1-v1_5 (RSA Encryption Scheme, Public-Key CryptoSystem #1, version 1.5).

This algorithm specifies the following encoding method. It divides the plaintext into 53-bit blocks and pads them out to 64 bits before passing them on for encryption. The padding provides extra security and also enables the algorithm to pass along the size of the plaintext to the person doing the decryption (in fact, the encoding method hides an end of file “marker” rather than encrypting the file size). The padding also provides the possibility of using smaller plaintext blocks, if needed, although the padding process always grows a block to 64 bits before encryption.

Because of the minimum of 11 bits of padding per 53 bits of plaintext, the RSA encoded plaintext block is at least 20% larger in size than the plaintext block. The result of the encryption step is also 64 bytes; therefore, overall, RSA ciphertext is 20% larger than the original plaintext.

Without going into any great mathematical details, the encryption part of the RSA algorithm works by raising very large numbers to a large exponent and then taking the modulus of the result with respect to another large number. This process is relatively slow. Symmetric algorithms are almost always faster, so it makes sense to minimize the amount of encryption that occurs with RSA.

For an example of a technique that minimizes the amount of RSA encryption for a large amount of plaintext, suppose Alice wanted to send a lot of data to Bob securely. She would generate a random key for Rijndael, say, encrypt it with Bob's public key and send it to Bob. She would then encrypt all of the data with Rijndael and this randomly generated key and send the ciphertext to Bob. Bob would then recover the Rijndael key by decrypting Alice's first message with his private key, and then decrypt Alice's second, much larger, message with the recovered key and Rijndael.

The modulus n in an RSA key is the product of two large 256-bit primes, usually known as p and q . The modulus is therefore a 512-bit number; it is this value that is quoted as the RSA key size. The public and private exponents, e and d , are mathematically related to each other and to p and q :

$$de = 1 \bmod m, \text{ where } m = (p-1)(q-1)$$

If the encoded plaintext block were described as a large number x less than n (the encoding method makes sure that this is so), then the following equation is a mathematical representation of encryption using RSA to generate the ciphertext y (which is forced to be less than n by the modulus operation):

$$y = x^e \bmod n$$

And the following is RSA decryption expressed as a mathematical equation:

$$x = y^d \bmod n$$

Breaking RSA is equivalent to being able to factorize n to retrieve the primes p and q . Once these two are known the private key can be derived from the public key using the first equation above. Fortunately it is extremely difficult to factorize a large number into its two equal-sized primes in a reasonable amount of time, at least with our current mathematical knowledge.

The RSA support in LockBox not only provides encryption and decryption, but also a class for the RSA keys (this class is described in Chapter 7). LockBox also provides a routine that generates new RSA key pairs; this routine takes care of all the housekeeping and arithmetic needed to generate large prime numbers, the underpinning of RSA's keys.

The RSA algorithm was designed to encode the size of the original block as part of the ciphertext so it is not necessary to make alterations for that. Remember that encryption takes place with one of the RSA keys (usually the public key) and decryption with the other (the private key), although digital signatures do it the other way round. The following example shows how to encrypt and decrypt a stream of data with RSA:

```
procedure ExampleEncryptRSA(aKey : TLbAsymmetricKey;
                           aInStream, aOutStream : TStream);
var
  PlainBuf   : TRSAPlainBlock;
  CipherBuf  : TRSACipherBlock;
  BytesRead  : longint;
begin
  {reset the streams}
  aInStream.Position := 0;
  aOutStream.Position := 0;
  {read the first block from the stream}
  BytesRead := aInStream.Read(PlainBuf, sizeof(PlainBuf));
```

```

    {while there is still data to encrypt, do so}
    while (BytesRead <> 0) do begin
        EncryptRSA(aKey, PlainBuf, BytesRead, CipherBuf);
        aOutStream.Write(CipherBuf, sizeof(CipherBuf));
        BytesRead := aInStream.Read(PlainBuf, sizeof(PlainBuf));
    end;
end;

procedure ExampleDecryptRSA(aKey : TLbAsymmetricKey;
    aInStream, aOutStream : TStream);
var
    PlainBuf      : TRSAPlainBlock;
    CipherBuf     : TRSACipherBlock;
    PlainLen      : integer;
    BytesToWrite  : longint;
begin
    {reset the streams}
    aInStream.Position := 0;
    aOutStream.Position := 0;
    {read the first block from the stream}
    BytesRead := aInStream.ReadBuffer(CipherBuf,
sizeof(CipherBuf));
    {while there is still data to decrypt, do so}
    while (BytesRead = sizeof(CipherBuf)) do begin
        BytesToWrite := DecryptRSA(aKey, CipherBuf, PlainBuf);
        aOutStream.WriteBuffer(PlainBuf, BytesToWrite);
    end;
end;
end;

```

Procedures / Functions

DecryptRSA	GenerateRSAKeys	RSAEncryptStream
EncryptRSA	RSAEncryptFile	RSAEncryptString

Reference Section

DecryptRSA

function, LbAsym

```
function DecryptRSA(PrivateKey : LbAsymmetricKey;
  const CipherBlock : TRSACipherBlock;
  var PlainBlock : TRSAPlainBlock) : Integer;

TRSAPlainBlock = array[0..52] of Byte;
TRSACipherBlock = array[0..63] of Byte;
```

↳ Decrypts a ciphertext block with the RSA algorithm.

PrivateKey is the private key, the one related to the public key used for encryption.

The RSA cipher block, CipherBlock, always contains 64 encrypted bytes (to be strict, the ciphertext block size is equal to the size of the modulus, in LockBox's case 512 bits). The decrypted block, PlainBlock, is at most 53 bytes in size. The return value is the number of plaintext bytes decrypted from CipherBlock.

See also: EncryptRSA

EncryptRSA

procedure, LbAsym

```
procedure EncryptRSA(PublicKey : TLbAsymmetricKey;
  const PlainBlock : TRSAPlainBlock; PlainBlockLen : Integer;
  var CipherBlock : TRSACipherBlock);

TRSAPlainBlock = array[0..52] of Byte;
TRSACipherBlock = array[0..63] of Byte;
```

↳ Encrypts a plaintext block with the RSA algorithm.

PublicKey is the recipient's public key, and will be used to encrypt the block. The only key that will be able to decrypt this block will be the recipient's private key.

The plaintext block, PlainBlock, has at most 53 bytes of data to be encrypted. PlainBlockLen is the length of the plaintext data. EncryptRSA will encode the plaintext block, padding it to 64 bytes, and then encrypt that encoded block. The result is placed in CipherBlock. CipherBlock will always contain 64 bytes of ciphertext.

See also: DecryptRSA

GenerateRSAKeys**procedure, LbAsym**

```
procedure GenerateRSAKeys(
  var PrivateKey, PublicKey : TLbAsymmetricKey);
```

↳ Creates a new public and private key pair.

GenerateRSAKeys will calculate two new keys for RSA, a public key and a private key.

The routine generates, and verifies the primality of, two 256-bit prime numbers from a cryptographically secure random number generator. From these two keys it calculates the modulus for the keys, and then generates the exponents.

RSAShieldFile**procedure, LbAsym**

```
procedure RSAShieldFile(const InFile, OutFile : string;
  Key : TLbAsymmetricKey; Encrypt : Boolean);
```

↳ Encrypts a file with the RSA algorithm.

If Encrypt is True, the contents of InFile are encrypted and written to OutFile. Key is assumed to be a public key. If Encrypt is False, the contents of InFile are decrypted and written to OutFile. Key is assumed to be the private key corresponding to the public key used for encryption. In either case, if OutFile exists when this routine is called, it will be overwritten without warning.

RSAShieldStream**procedure, LbAsym**

```
procedure RSAShieldStream(InStream, OutStream : TStream;
  Key : TLbAsymmetricKey; Encrypt : Boolean);
```

↳ Encrypts a stream with the RSA algorithm.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. Key is assumed to be a public key. If Encrypt is False, the contents of InStream are decrypted and written to OutStream. Key is assumed to be the private key corresponding to the public key used for encryption.

RSAShieldStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.


```
procedure RSAEncryptString(const InString : string;  
    var OutString : string; Key : TLbAsymmetricKey;  
    Encrypt : Boolean);
```

3

↳ Encrypts a string with the RSA algorithm.

If Encrypt is True, the contents of InString are encrypted, converted to Base64 encoding and written to OutString. Key is assumed to be a public key. If Encrypt is False, the contents of InString are converted from Base64 encoding, decrypted and written to OutString. Key is assumed to be the private key corresponding to the public key used for encryption.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

Secure Hashing Algorithm (SHA-1)

The National Institute of Standards and Technology (NIST) and the National Security Agency (NSA) invented SHA-1 for use with the Digital Signature Algorithm (DSA). It is a cryptographically secure message digest algorithm.

SHA-1 generates a 160-bit hash (20 bytes) from its input. The input data is divided up into 512-bit blocks (the algorithm goes into some detail about how to pad out the last block, the one that may be less than 512 bits in size) and then each block is divided up into 32-bit integers ready for the mixing stage. Since all operations act on 32-bit integers, SHA-1 is very efficient on 32-bit CPUs.

The SHA-1 algorithm requires an initialization phase as well as a finalization phase. Although LockBox provides separate routines for the initialization, hashing and finalization phases, it is recommended that the single HashSHA1 routine is used. However, should full control be required over the hashing process (maybe the data is to be hashed in separate pieces at separate times, and not to be concatenated all in one buffer), the LockBox routines are designed to work as follows. First, call InitSHA1 to prepare the context structure used by the other two routines. Second, call UpdateSHA1 as many times as necessary to hash each of the pieces of the data. The final step is to convert the context structure into the hash by calling FinalizeSHA1. The following example shows how to hash a simple buffer:

```
var
    Context : TSHA1Context;
    Digest   : TSHA1Digest;
begin
    InitSHA1(Context);
    UpdateSHA1(Context, Buffer, sizeof(Buffer));
    FinalizeSHA1(Context, Digest);
    ..Digest now contains the SHA-1 hash of buffer..
```

The SHA-1 algorithm is viewed as being very secure, even better than the MD5 algorithm.

Procedures / Functions

FinalizeSHA1
HashSHA1

InitSHA1
StringHashSHA1

UpdateSHA1

Reference Section

FinalizeSHA1

function, LbCipher

```
procedure FinalizeSHA1(  
  var Context: TSHA1Context; var Digest : TSHA1Digest);  
  
TSHA1Context = record  
  sdHi : DWord;  
  sdLo : DWord;  
  sdIndex : DWord;  
  sdHash : array [0..4] of DWord;  
  sdBuf : array [0..63] of byte;  
end;  
  
TSHA1Digest = array[ 0..19 ] of byte;
```

↳ Completes the generation of a SHA-1 hash by returning the digest.

FinalizeSHA1 takes the SHA-1 context structure obtained from a call to InitSHA1 followed by one or more to UpdateSHA1 and calculates the final message digest from it. The hash is returned in Digest.

See also: HashSHA1, InitSHA1, UpdateSHA1

HashSHA1

function, LbCipher

```
procedure HashSHA1(  
  var Digest : TSHA1Digest; const Buf; BufSize : Longint);  
  
TSHA1Digest = array[ 0..19 ] of byte;
```

↳ Generates a secure hash of a buffer using the SHA-1 hashing algorithm.

HashSHA1 performs a hash on the contents of Buf. BufSize is the number of bytes in Buf. The hash is returned in Digest.

HashSHA1 hashes the buffer by calling InitSHA1, UpdateSHA1, and FinalizeSHA1. If more control over the process is needed, call these three routines directly. For example, if all of the data is not available at one time multiple calls will need to be made to UpdateSHA1 as the data becomes available. However, for the majority of requirements, using HashSHA1 is the easiest course.

See also: FinalizeSHA1, InitSHA1, UpdateSHA1

InitSHA1**procedure, LbCipher**

```

procedure InitSHA1(var Context: TSHA1Context);

TSHA1Context = record
    sdHi : DWord;
    sdLo : DWord;
    sdIndex : DWord;
    sdHash : array [0..4] of DWord;
    sdBuf : array [0..63] of byte;
end;

```

↪ Initializes the context structure for the SHA-1message digest algorithm.

The normal sequence of events for generating a SHA-1 hash is to call InitSHA1 to prepare the context buffer, call UpdateSHA1 one or more times, and then call FinalizeSHA1 to retrieve the message digest.

HashSHA1 performs this sequence of events automatically. If no special processing is required during the generation of the message digest, use HashSHA1 instead of the sequence of calls to InitSHA1, UpdateSHA1, and FinalizeSHA1.

See also: HashSHA1, FinalizeSHA1, UpdateSHA1

StringHashSHA1**function, LbCipher**

```

procedure StringHashSHA1(
    var Digest : TSHA1Digest; const Str : string);

TSHA1Digest = array[ 0..19 ] of byte;

```

↪ Generates a hash (or message digest) of the input string using the SHA-1message digest algorithm.

StringHashSHA1 uses the HashSHA1 routine to perform a hash on the contents of Str. The hash value is returned in Digest.

See also: HashSHA1

```
procedure UpdateSHA1(  
  var Context : TSHA1Context;  const Buf; BufSize: Longint);  
  
TSHA1Context = record  
  sdHi : DWord;  
  sdLo : DWord;  
  sdIndex : DWord;  
  sdHash : array [0..4] of DWord;  
  sdBuf : array [0..63] of byte;  
end;
```

↪ Updates a SHA-1 context with a hashed form of the data.

UpdateSHA1 is one step in the generation of a hash using the SHA-1 algorithm. It updates Context with a digested form of the data in Buf. Call UpdateSHA1 once if all of the data is ready, or multiple times if all of the data is not available at once. BufSize is the number of bytes to be processed from Buf.

See also: FinalizeSHA1, HashSHA1, InitSHA1

Triple-DES Cipher

Since DES's 56-bit key is perceived as being too short to counter brute-force attacks, several modifications to the DES standard have been proposed in order to strengthen it. The most successful of these is the triple-DES algorithm.

The algorithm is simple. Instead of encrypting a block with a single DES pass, the block is encrypted three times using DES with different keys. There are several variations on this theme, some of the most popular being:

- Encrypt it three times with three different keys. Here the effective key length is 168 bits.
- Encrypt it in the first pass with one key, decrypt it in the second pass with a second key, and finally encrypt it again with the first key. Since two keys are used, the effective key length is 112 bits. To decrypt the block, decrypt with the first key, encrypt with the second, and decrypt again with the first.
- Use the same process as the second option, but use three keys.

Obviously, triple-DES is three times slower than DES since three complete passes are made for each block. However, the cipher is much more secure than simple DES. LockBox implements the second option above: it uses two keys (specified as a single 128-bit value) and follows the encrypt-decrypt-encrypt process. Blocks are still 64 bits long.

In the following example, since the ciphertext is going to be larger than the plaintext, the routine should pass along the size of the original plaintext buried somehow in the ciphertext. This example uses a simple technique of inserting a new first block that consists of the size together with a set of random bytes. The final block of the buffer, if it is smaller than 8 bytes, will be padded with random bytes from the previous encryption step. The following example shows how to encrypt and decrypt a stream of data with the triple-DES algorithm, from basic principles:

```
procedure TestEncryptTripleDES(aInStream, aOutStream : TStream);
type
  TMyDESBlock = array [0..1] of LongInt;
var
  Key          : TKey128;
  Context      : TTripleDESContext;
  DESBlock     : TDESBlock;
  BytesRead    : longint;
  Seed         : integer;
```

```

begin
  {reset the streams}
  aInStream.Position := 0;
  aOutStream.Position := 0;
  {create a key}
  GenerateMD5Key(Key, Passphrase);
  {initialize the DES context}
  InitEncryptTripleDES(Key, Context, True);
  {set up the first block to contain the size of
   the plaintext, together with random noise}
  TMyDESBlock(DESBlock)[1] := aInStream.Size;
  Seed := $12345678;
  TMyDESBlock(DESBlock)[0] := Ran03(Seed);
  {encrypt and write this block}
  EncryptTripleDES(Context, DESBlock);
  aOutStream.Write(DESBlock, sizeof(DESBlock));
  {read the first block from the stream}
  BytesRead := aInStream.Read(DESBlock, sizeof(DESBlock));
  {while there is still data to encrypt, do so}
  while (BytesRead <> 0) do begin
    EncryptTripleDES(Context, DESBlock);
    aOutStream.Write(DESBlock, sizeof(DESBlock));
    BytesRead := aInStream.Read(DESBlock, sizeof(DESBlock));
  end;
end;

procedure TestDecryptTripleDES(aInStream, aOutStream : TStream);
type
  TMyDESBlock = array [0..1] of LongInt;
var
  Key      : TKey128;
  Context  : TTripleDESContext;
  DESBlock : TDESBlock;
  BytesToDecrypt : longint;
  BytesToWrite   : longint;
begin
  {reset the streams}
  aInStream.Position := 0;
  aOutStream.Position := 0;
  {create a key}
  GenerateMD5Key(Key, Passphrase);
  {initialize the Blowfish context}
  InitEncryptTripleDES(Key, Context, False);
  {read the first block and decrypt to get
   the size of the plaintext}
  aInStream.ReadBuffer(DESBlock, sizeof(DESBlock));

```

```

EncryptTripleDES(Context, DESBlock);
BytesToDecrypt := TMyDesBlock(DESBlock)[1];
{while there is still data to decrypt, do so}
while (BytesToDecrypt <> 0) do begin
    aInStream.ReadBuffer(DESBlock, sizeof(DESBlock));
    EncryptTripleDES(Context, DESBlock);
    if (BytesToDecrypt > sizeof(DESBlock)) then
        BytesToWrite := sizeof(DESBlock)
    else
        BytesToWrite := BytesToDecrypt;
    dec(BytesToDecrypt, BytesToWrite);
    aOutStream.WriteBuffer(DESBlock, BytesToWrite);
end;
end;

```

Procedures / Functions

EncryptTripleDES	TripleDESEncryptFile	TripleDESEncryptStreamCBC
EncryptTripleDESCBC	TripleDESEncryptFileCBC	TripleDESEncryptString
InitEncryptTripleDES	TripleDESEncryptStream	TripleDESEncryptStringCBC

Reference Section

EncryptTripleDES

procedure, LbCipher

```
procedure EncryptTripleDES(  
  const Context : TTripleDESContext; var Block : TDESBlock);  
  
TTripleDESContext = array [0..1] of TDESContext;  
  
TDESContext = packed record  
  TransformedKey : array [0..31] of LongInt;  
  Encrypt : Boolean;  
end;  
  
TDESBlock = array[0..7] of Byte;
```

↳ Encrypts or decrypts a block of data using the triple-DES cipher in ECB mode.

Context must be initialized by a call to `InitEncryptTripleDES` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted. If `Encrypt` is `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

Note that it is the `Context` variable that defines whether the block is encrypted or decrypted.

See also: `InitEncryptTripleDES`

EncryptTripleDESCBC

procedure, LbCipher

```
procedure EncryptTripleDESCBC(const Context : TTripleDESContext;  
  const Prev : TDESBlock; var Block : TDESBlock);  
  
TTripleDESContext = array [0..1] of TDESContext;  
  
TDESContext = packed record  
  TransformedKey : array [0..31] of LongInt;  
  Encrypt : Boolean;  
end;  
  
TDESBlock = array[0..7] of Byte;
```

↳ Encrypts or decrypts a block of data using the triple-DES cipher in CBC mode.

Context must be initialized by a call to `InitEncryptTripleDES` prior to calling this routine. If `Encrypt` is `True`, `Block` is encrypted. If `Encrypt` is `False`, `Block` is decrypted. Note that the `Block` variable is directly modified by this routine.

The Prev parameter specifies the previously encrypted block. The way this routine works is to take the current value of Block, XOR it with Prev, and then encrypt it. It is then required to pass the new value of Block as the Prev parameter for the next call to EncryptTripleDESCBC. The technique is the same as shown by the example for EncryptBFBCBC.

Note that it is the Context variable that defines whether the block is encrypted or decrypted.

See also: LbCipher.EncryptBFBCBC, InitEncryptTripleDES

InitEncryptTripleDES

procedure, LbCipher

```

procedure InitEncryptTripleDES(const Key : TKey128;
    var Context : TTripleDESContext; Encrypt : Boolean);

TKey128 = array [0..15] of Byte;
TTripleDESContext = array [0..1] of TDESContext;
TDESContext = packed record
    TransformedKey : array [0..31] of LongInt;
    Encrypt : Boolean;
end;
```

↳ Initializes a context for use with the triple-DES cipher.

InitEncryptTripleDES initializes Context for use with either EncryptTripleDES or EncryptTripleDESCBC. Key is the key used to encrypt and decrypt each block of data. This key can be created with GenerateRandomKey or GenerateMD5Key.

For encrypting a set of data, either use EncryptTripleDES exclusively or EncryptTripleDESCBC exclusively. Do not mix the two block encryption routines for a single encryption round.

See also: EncryptTripleDES, EncryptTripleDESCBC, GenerateRandomKey, GenerateMD5Key

```
procedure TripleDESEncryptFile(const InFile, OutFile : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

3 ↪ Encrypts or decrypts a file using the triple-DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

TripleDESEncryptFileCBC**procedure, LbProc**

```
procedure TripleDESEncryptFileCBC(const InFile, OutFile : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

↪ Encrypts or decrypts a file using the triple-DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the file. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InFile` are encrypted and written to `OutFile`. If `Encrypt` is `False`, the contents of `InFile` are decrypted and written to `OutFile`. In either case, if `OutFile` exists when this routine is called, it will be overwritten without warning.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

TripleDESEncryptStream**procedure, LbProc**

```

procedure TripleDESEncryptStream(InStream, OutStream : TStream;
  Key : TKey128; Encrypt : Boolean);
  TKey128 = array [0..15] of Byte;

```

↳ Encrypts or decrypts a stream using the triple-DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with GenerateRandomKey or GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

TripleDESEncryptStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

TripleDESEncryptStreamCBC**procedure, LbProc**

```

procedure TripleDESEncryptStreamCBC(InStream, OutStream : TStream;
  Key : TKey128; Encrypt : Boolean);
  TKey128 = array [0..15] of Byte;

```

↳ Encrypts or decrypts a stream using the triple-DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the stream. This key can be created with GenerateRandomKey or GenerateMD5Key.

If Encrypt is True, the contents of InStream are encrypted and written to OutStream. If Encrypt is False, the contents of InStream are decrypted and written to OutStream.

TripleDESEncryptStreamCBC does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: LbCipher.GenerateMD5Key, LbCipher.GenerateRandomKey

```
procedure TripleDESEncryptString(const InString : string;  
    var OutString : string; const Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..7] of Byte;
```

3 ➤ Encrypts or decrypts a string using the triple-DES cipher in ECB mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

TripleDESEncryptStringCBC**procedure, LbString**

```
procedure TripleDESEncryptFileCBC(  
    const InString, OutString : string;  
    Key : TKey128; Encrypt : Boolean);  
  
TKey128 = array [0..15] of Byte;
```

➤ Encrypts or decrypts a string using the triple-DES cipher in CBC mode.

Key is the key used to encrypt or decrypt the string. This key can be created with `GenerateRandomKey` or `GenerateMD5Key`.

If `Encrypt` is `True`, the contents of `InString` are encrypted, converted to Base64 encoding and written to `OutString`. If `Encrypt` is `False`, the contents of `InString` are converted from Base64 encoding, decrypted and written to `OutString`.

Note that the ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: `LbCipher.GenerateMD5Key`, `LbCipher.GenerateRandomKey`

Chapter 4: Encryption Components

Although LockBox provides all of the routines necessary to incorporate cryptographic algorithms into user applications, it's obvious that not all users need the full level of control afforded by these low-level routines. Indeed, in the vast majority of cases, developers would just like to encrypt buffers, streams, files, or strings with a particular cipher and anything to make that easier would be highly beneficial. They are not interested in the nuances of encryption; they have a particular need and the simpler it is to implement that need, the better.

LockBox contains an encryption component hierarchy to fulfill this need. Figure 4.1 shows this class hierarchy. At its base is an encryption engine class, TLbCipher. This class has virtual methods to encrypt and decrypt an arbitrary buffer of data into another, a file into another, a stream into another, or a string into another.

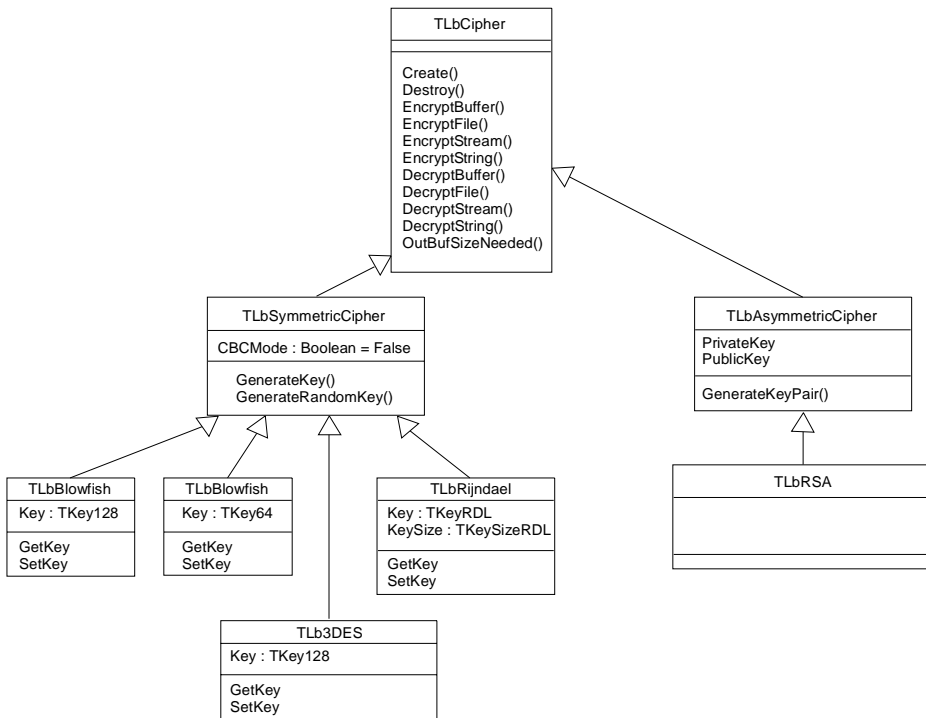


Figure 4.1: The LockBox encryption component hierarchy

There are two simple descendant classes that act as roots for the two types of ciphers, symmetric and asymmetric. These classes have some extra functionality dealing with keys; in the symmetric case, a single private key, in the asymmetric case, the public and the private keys.

Finally, there are the descendant components that perform the actual encryption and decryption using specific ciphers. You'll find a component for Blowfish, DES, triple-DES, and Rijndael off the symmetric branch of the hierarchy, and RSA off the asymmetric branch.

The following code shows how to use the Blowfish component to encrypt a stream into another stream:

```
const
    Passphrase = 'Three can keep a secret, if two are dead.';

var
    Blowfish      : TLbBlowfish;
    PlainStream   : TFileStream;
    CipherStream  : TFileStream;
begin
    {open the streams}
    PlainStream := TFileStream.Create('plain.txt', fmOpenRead);
    CipherStream := TFileStream.Create('cipher.blf', fmCreate);

    {create the Blowfish cipher}
    Blowfish := TLbBlowfish.Create(nil);

    {set the Blowfish key from our passphrase}
    Blowfish.GenerateKey(Passphrase);

    {encrypt the plaintext stream to the ciphertext one}
    Blowfish.EncryptStream(PlainStream, CipherStream);

    {clean up}
    Blowfish.Free;
    CipherStream.Free;
    PlainStream.Free;
end;
```

TLbBaseComponent Class

All LockBox components descend from TComponent through TLbBaseComponent. This class implements the Version property.

Hierarchy

TComponent (VCL)

 TLbBaseComponent (LbClass)

Properties

 Version

Reference Section

Version

property

property Version : string

↳ Shows the current version of LockBox.

Version is provided in order that the version of LockBox can be easily identified should technical support be needed. In the Object Inspector, display the LockBox about box by double-clicking this property or selecting the dialog button to the right of the property value.

4

TLbCipher Class

The TLbCipher class is the base class for the encryption class and component hierarchy. It descends from TLbBaseComponent in order to inherit the Version property.

TLbCipher manages the basic functionality of a cipher engine: encrypting and decrypting buffers of data, files, streams and strings. These options are implemented as virtual methods and are overridden in the descendants that implement the different ciphers.

With the methods that encrypt and decrypt a buffer of data, block ciphers pose a thorny problem. When encrypting or decrypting files, streams, and strings, LockBox can easily grow the output object if needed, but with the buffer methods, the output buffer needs to be sized before calling the method. On encryption, the cipher component can easily calculate the output buffer size (the algorithm concerned defines how the encrypted data grows in size). The OutBufSizeNeeded virtual method performs this calculation.

However, in the opposite direction, the only way with the majority of ciphers to calculate the plaintext size from the ciphertext size is to perform the decryption. This is too inefficient. Since the plaintext size is never greater than the ciphertext size, a better methodology would be to provide an output buffer equal in size to the input buffer, call the decryption method, and have that method return the number of decrypted bytes in the output buffer. This is what the DecryptBuffer method does.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
- TLbCipher (LbClass)

Properties

- ❶ Version

Methods

DecryptBuffer	DecryptString	EncryptStream
DecryptFile	EncryptBuffer	EncryptString
DecryptStream	EncryptFile	OutBufSizeNeeded

Reference Section

DecryptBuffer

virtual abstract method

```
function DecryptBuffer(const InBuf; InBufLength : Longint;  
    var OutBuf) : Longint; virtual; abstract;
```

↳ Decrypts a buffer of ciphertext.

DecryptBuffer decrypts the ciphertext in the InBuf buffer and puts the resulting plaintext in OutBuf. InBufLength is the number of bytes of ciphertext in InBuf. The function returns the number of bytes of plaintext.

Prior to decrypting the input buffer, the only thing known about the plaintext size is that it will be less than or equal to the ciphertext size. Hence, prior to calling this method, the output buffer must be created to be the same size as the input buffer.

See also: EncryptBuffer

DecryptFile

virtual abstract method

```
procedure DecryptFile(  
    const InFile, OutFile : string); virtual; abstract;
```

↳ Decrypts a file containing ciphertext.

The contents of InFile are decrypted and written to OutFile. If OutFile exists when this routine is called, it will be overwritten without warning.

See also: EncryptFile

DecryptStream

virtual abstract method

```
procedure DecryptStream(  
    InStream, OutStream : TStream); virtual; abstract;
```

↳ Decrypts a stream containing ciphertext.

The contents of InStream are decrypted and written to OutStream.

DecryptStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: EncryptStream

DecryptString

virtual abstract method

```
function DecryptString(  
    const InString : string); virtual; abstract;
```

↪ Decrypts a string containing ciphertext.

The contents of InString are converted from Base64 encoding, decrypted and returned.

See also: EncryptString

EncryptBuffer

virtual abstract method

```
function EncryptBuffer(const InBuf; InBufLength : Longint;  
    var OutBuf) : Longint; virtual; abstract;
```

↪ Encrypts a buffer of plaintext.

EncryptBuffer encrypts the ciphertext in the Inbuf buffer and puts the resulting ciphertext in OutBuf. Ther value of InBufLength is the number of bytes of plaintext in InBuf.

Each cipher has different space requirements for its ciphertext. The virtual OutBufSizeNeeded method will return the size of the output buffer needed for a particular input buffer size. EncryptBuffer should only be called after calling OutBufSizeNeeded and allocating an output buffer of the correct size. The following example code shows the steps required:

```
var  
    OutBufSize    : integer;  
    CipherText    : pointer;  
    PlainText     : pointer;  
    PlaintextLen  : integer;  
begin  
    ..get the plaintext and its length..  
  
    OutBufSize := MyCipher.OutBufSizeNeeded(PlaintextLen);  
    GetMem(CipherText, OutBufSize);  
    MyCipher.EncryptBuffer(PlainText^, CipherText^, PlaintextLen);  
  
    ..process the ciphertext..  
  
    FreeMem(CipherText);
```

See also: DecryptBuffer, OutBufSizeNeeded

EncryptFile**virtual abstract method**

```
procedure EncryptFile(
  const InFile, OutFile : string); virtual; abstract;
```

↳ Encrypts a file containing plaintext.

The contents of InFile are encrypted and written to OutFile. If OutFile exists when this routine is called, it will be overwritten without warning.

See also: DecryptFile

EncryptStream**virtual abstract method**

```
procedure EncryptStream(
  InStream, OutStream : TStream); virtual; abstract;
```

↳ Encrypts a stream containing plaintext.

The contents of InStream are encrypted and written to OutStream.

EncryptStream does not reset the position of InStream or OutStream prior to processing. The routine will, however, process all of the remaining bytes in InStream.

See also: DecryptStream

EncryptString**virtual abstract method**

```
function EncryptString(
  const InString : string); virtual; abstract;
```

↳ Encrypts a string containing plaintext.

The contents of InString are encrypted, converted from Base64 encoding, and returned.

Note that the created ciphertext string will always be longer than the plaintext string. Because the ciphertext string is encoded using Base64, it will contain only printable ASCII characters.

See also: DecryptString

```
function OutBufSizeNeeded(  
    InBufSize : Longint) : Longint; virtual; abstract;
```

↪ Calculates the size of the output buffer needed for encrypting some plaintext.

Each cipher has different space requirements for the ciphertext created from some plaintext. The virtual `OutBufSizeNeeded` method will return the size of the output buffer needed for a particular input buffer containing plaintext. `InBufSize` is the number of bytes to be encrypted. `EncryptBuffer` should only be called after calling `OutBufSizeNeeded` and allocating an output buffer of the correct size.

See also: `EncryptBuffer`

TLbSymmetricCipher Class

The TLbSymmetricCipher class is the ancestor class for the symmetric encryption components. It descends form the TLbCipher base class.

The class provides functionality for key management for the descendants. There are two virtual abstract methods that can generate keys for the ciphers, one from a passphrase, GenerateKey, and the other from a random number generator, GenerateRandomKey.

This ancestor class also has a public property, CipherMode, which defines whether ECB or CBC modes should be used for encryption and decryption.

Hierarchy

TComponent (VCL)	
❶ TLbBaseComponent (LbClass)	107
❷ TLbCipher (LbClass).....	109
TLbSymmetricCipher (LbClass)	

Properties

CipherMode	❶ Version
------------	-----------

Methods

❷ DecryptBuffer	❷ EncryptBuffer	GenerateKey
❷ DecryptFile	❷ EncryptFile	GenerateRandomKey
❷ DecryptStream	❷ EncryptStream	❷ OutBufSizeNeeded
❷ DecryptString	❷ EncryptString	

Reference Section

CipherMode

property

```
property CipherMode : TLbCipherMode
TLbCipherMode = (cmECB, cmCBC);
```

↳ Defines the encryption mode for the block ciphers.

If this property is cmECB, encryption and decryption will be done in ECB (Electronic Code Book) mode. If the property is cmCBC, they will be done in CBC (Cipher Block Chaining Mode).

GenerateKey

virtual abstract method

```
procedure GenerateKey(const PassPhrase : string);
virtual; abstract;
```

↳ Generates the key from a passphrase.

For better security the passphrase should be at least 32 characters long. The method sets the internal key from the MD5 hash of the passphrase. A key should always be set prior to performing any encryption or decryption. The following example shows how to get the key for a Blowfish cipher object from a passphrase:

```
var
  MyCipher : TLbBlowFish;
begin
  MyCipher := TLbBlowFish.Create(nil);
  MyCipher.GenerateKey(
    'Three can keep a secret, if two are dead.');
```

```
MyCipher.EncryptStream(InStream, OutStream);
```

See also: GenerateRandomKey


```
procedure GenerateRandomKey; virtual; abstract;
```

↳ Generates the key from a random number generator.

GenerateRandomKey sets the internal key from a random number generator. A key should always be set prior to performing any encryption or decryption. The following example shows how to get the key for a Blowfish cipher object from a random source:

```
var
    MyCipher : TLbBlowfish;
begin
    MyCipher := TLbBlowfish.Create(nil);
    MyCipher.GenerateRandomKey;
    MyCipher.EncryptStream(InStream, OutStream);
```

See also: [GenerateKey](#)

TLbBlowfish Component

The TLbBlowfish component provides encryption and decryption facilities using the Blowfish cipher. It descends from TLbSymmetricCipher.

The methods implemented by TLbBlowfish are all overridden from virtual methods defined by its ancestors, except for the key access methods, GetKey and SetKey.

For more information about the Blowfish cipher, see page 32.

Hierarchy

TComponent (VCL)

❶ TLbBaseComponent (LbClass)	107
❷ TLbCipher (LbClass)	109
❸ TLbSymmetricCipher (LbClass)	114
TLbBlowfish (LbClass)	

Properties

❸ CipherMode	❶ Version
--------------	-----------

Methods

❷ DecryptBuffer	❷ EncryptFile	GetKey
❷ DecryptFile	❷ EncryptStream	❷ OutBufSizeNeeded
❷ DecryptStream	❷ EncryptString	SetKey
❷ DecryptString	❸ GenerateKey	
❷ EncryptBuffer	❸ GenerateRandomKey	

Reference Section

GetKey

method

```
procedure GetKey(var Key : TKey128);  
TKey128 = array [0..15] of Byte;
```

↪ Returns the key used for encryption and decryption.

When a TLbBlowfish object is created, the key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call SetKey, GenerateKey, or GenerateRandomKey to do this.

See also: SetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

SetKey

method

```
procedure SetKey(const Key : TKey128);  
TKey128 = array [0..15] of Byte;
```

↪ Sets the key used for encryption and decryption.

When a TLbBlowfish object is created, Key is set to all binary zeros. A key should always be set prior to performing any encryption or decryption. Call this routine, GenerateKey, or GenerateRandomKey to do this.

See also: GetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

TLbDES Component

The TLbDES component provides encryption and decryption facilities using the DES (Data Encryption Standard) cipher. It descends from TLbSymmetricCipher.

The methods implemented by TLbDES are all overridden from virtual methods defined by its ancestors, except for the key access methods, GetKey and SetKey.

For more information about the DES cipher, see page 41.

Hierarchy

TComponent (VCL)

❶ TLbBaseComponent (LbClass)	107
❷ TLbCipher (LbClass)	109
❸ TLbSymmetricCipher (LbClass)	114
TLbDES (LbClass)	

Properties

CipherMode	❶ Version
------------	-----------

Methods

❷ DecryptBuffer	❷ EncryptFile	GetKey
❷ DecryptFile	❷ EncryptStream	❷ OutBufSizeNeeded
❷ DecryptStream	❷ EncryptString	SetKey
❷ DecryptString	❸ GenerateKey	
❷ EncryptBuffer	❸ GenerateRandomKey	

Reference Section

GetKey

method

```
procedure GetKey(var Key : TKey64);  
TKey64 = array [0..7] of Byte;
```

↪ Returns the key used for encryption and decryption.

When a TLbDES object is created, the key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call SetKey, GenerateKey, or GenerateRandomKey to do this.

See also: SetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

SetKey

method

```
procedure SetKey(const Key : TKey64);  
TKey64 = array [0..7] of Byte;
```

↪ Sets the key used for encryption and decryption.

When a TLbDES object is created, Key is set to all binary zeros. A key should always be set prior to performing any encryption or decryption. Call this routine, GenerateKey, or GenerateRandomKey to do this.

See also: GetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

TLb3DES Component

The TLb3DES component provides encryption and decryption facilities using the triple-DES (Data Encryption Standard) cipher. It descends from TLbSymmetricCipher.

The methods implemented by TLb3DES are all overridden from virtual methods defined by its ancestors, except for the key access methods, GetKey and SetKey.

For more information about the DES cipher, see page 41. For more information about the triple-DES cipher, see page 97.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
 - ❷ TLbCipher (LbClass) 109
 - ❸ TLbSymmetricCipher (LbClass) 114
 - TLb3DES (LbClass)

Properties

- ❸ CipherMode
- ❶ Version

Methods

- | | | |
|-----------------|---------------------|--------------------|
| ❷ DecryptBuffer | ❷ EncryptFile | GetKey |
| ❷ DecryptFile | ❷ EncryptStream | ❷ OutBufSizeNeeded |
| ❷ DecryptStream | ❷ EncryptString | SetKey |
| ❷ DecryptString | ❸ GenerateKey | |
| ❷ EncryptBuffer | ❸ GenerateRandomKey | |

Reference Section

GetKey

method

```
procedure GetKey(var Key : TKey128);  
TKey128 = array [0..15] of Byte;
```

↪ Returns the key used for encryption and decryption.

When a TLb3DES object is created, the key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call SetKey, GenerateKey, or GenerateRandomKey to do this.

See also: SetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

SetKey

method

```
procedure SetKey(const Key : TKey128);  
TKey128 = array [0..15] of Byte;
```

↪ Sets the key used for encryption and decryption.

When a TLb3DES object is created, Key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call this routine, GenerateKey, or GenerateRandomKey to do this.

See also: GetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

TLbRijndael Component

The TLbRijndael component provides encryption and decryption facilities using the Rijndael (or AES, Advanced Encryption Standard) cipher. It descends from TLbSymmetricCipher.

The methods implemented by TLbRijndael are all overridden from virtual methods defined by its ancestors, except for the key access routines, GetKey and SetKey.

For more information about the Rijndael cipher, see page 78.

Hierarchy

TComponent (VCL)

❶ TLbBaseComponent (LbClass)	107
❷ TLbCipher (LbClass)	109
❸ TLbSymmetricCipher (LbClass)	114
TLbRijndael (LbClass)	

Properties

❸ CipherMode	KeySize	❶ Version
--------------	---------	-----------

Methods

❷ DecryptBuffer	❷ EncryptFile	GetKey
❷ DecryptFile	❷ EncryptStream	❷ OutBufSizeNeeded
❷ DecryptStream	❷ EncryptString	SetKey
❷ DecryptString	❸ GenerateKey	
❷ EncryptBuffer	❸ GenerateRandomKey	

Reference Section

GetKey

method

```
procedure GetKey(var Key);
```

- ↳ Returns the key used for encryption and decryption.

The KeySize property defines how big the Key parameter should be.

When a TLbRijndael object is created, the key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call SetKey, GenerateKey, or GenerateRandomKey to do this.

See also: SetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

KeySize

property

```
property KeySize : TLbKeySizeRDL
```

```
TLbKeySizeRDL = (ks128, ks192, ks256);
```

- ↳ Defines the key size for the Rijndael key.

Default: ks128

The LockBox implementation of Rijndael enables the use of keys that are 128, 192 or 256 bits in length. Reading this property returns the length of the key in use.

Writing to the property not only sets the value, but also clears the current key. On changing the KeySize property, the key must be set again before encrypting or decrypting. This is done by calling SetKey, GenerateKey, or GenerateRandomKey.

See also: SetKey, TLbSymmetricCipher.GenerateKey,
TLbSymmetricCipher.GenerateRandomKey

```
procedure SetKey(const Key);
```

↪ Sets the key used for encryption and decryption.

The `KeySize` property defines how big the `Key` parameter should be.

When a `TLbRijndael` object is created, the key is set to all binary zeros. The key should always be set prior to performing any encryption or decryption. Call this routine, `GenerateKey`, or `GenerateRandomKey` to do this.

See also: `GetKey`, `TLbSymmetricCipher.GenerateKey`,
`TLbSymmetricCipher.GenerateRandomKey`

TLbAsymmetricCipher Class

The TLbAsymmetricCipher class is the ancestor class for the asymmetric encryption components. It descends form the TLbCipher base class. Do not create any instances of this class.

The class provides the two keys, public and private, for the descendant asymmetric components. It also provides a convenient method to generate a new asymmetric key pair.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
- ❷ TLbCipher (LbClass) 109
- TLbAsymmetricCipher (LbClass)

Properties

PrivateKey	PublicKey	❶ Version
------------	-----------	-----------

Methods

❷ DecryptBuffer	❷ EncryptBuffer	GenerateKeyPair
❷ DecryptFile	❷ EncryptFile	❷ OutBufSizeNeeded
❷ DecryptStream	❷ EncryptStream	
❷ DecryptString	❷ EncryptString	

Reference Section

GenerateKeyPair

method

```
procedure GenerateKeyPair;
```

↪ Generates a new public/private key pair.

When an instance of a descendant class is created the two keys, `PrivateKey` and `PublicKey`, are internally set to zeros. The keys should be always set to meaningful values prior to performing any encryption or decryption. They can be set them by calling this routine, in which case the keys are generated from a random source, or by calling their `Assign` or `LoadFromStream` methods.

For an example of the use of this method, see `TLbRSA`.

See also: `GenerateRSAKeys` routine (`LbCipher`), `TLbAsymmetricKey.Assign`,
`TLbAsymmetricKey.LoadFromStream`

PrivateKey

read-only property

```
property PrivateKey : TLbAsymmetricKey
```

↪ Returns the private key for the asymmetric cipher.

`PrivateKey` is only used for decryption.

The asymmetric cipher class manages the private key internally. Although `PrivateKey` is read-only, its value can be changed either by calling `GenerateKeyPair`, or by calling `PrivateKey`'s `LoadFromStream` or `Assign` methods. Since a new instance of a descendant class has both its private and public keys set to zeros, `PrivateKey`'s value must be set through one of these routines before performing any encryption or decryption.

See the code with the `TLbRSA` class for an example of how to use this property.

See also: `GenerateKeyPair`, `PublicKey`, `TLbAsymmetricKey.Assign`,
`TLbAsymmetricKey.LoadFromStream`

property PublicKey : TLbAsymmetricKey

↳ Returns the public key for the asymmetric cipher.

PublicKey is only used for encryption.

The asymmetric cipher class manages the public key internally. Although PrivateKey is read-only, its value can be changed either by calling GenerateKeyPair, or by calling PrivateKey's LoadFromStream or Assign methods. Since a new instance of a descendant class has both its private and public keys set to zeros, PrivateKey's value must be set through one of these routines before performing any encryption or decryption.

See the code with the TLbRSA class for an example of how to use this property.

See also: GenerateKeyPair, PublicKey, TLbAsymmetricKey.Assign,
TLbAsymmetricKey.LoadFromStream

TLbRSA Class

The TLbRSA class provides encryption and decryption facilities using the RSA cipher. It descends from TLbAsymmetricCipher.

The class provides no new methods or properties. It overrides the virtual methods from its ancestors in order to encrypt and decrypt data.

Using the TLbRSA class involves a little more work than the symmetric cipher components, mainly to do with managing the key pair. In the following example, a file stream is encrypted with the eventual recipient's public key. The recipient will have published the key in ASN.1 format (the standard form for publishing RSA keys, page 155 has some more details). This example assumes that the ASN.1 formatted key has been read into a TMemoryStream:

```
var
    ASN1Key      : TMemoryStream;
    RSACipher    : TLbRSA;
    PlainStream  : TFileStream;
    CipherStream : TFileStream;
begin
    ..read the public key in ASN1Key..

    {open the two file streams}
    PlainStream := TFileStream.Create('plain.txt', fmOpenRead or
fmShareDenyWrite);
    CipherStream : TFileStream('cipher.rsa', fmCreate);

    {create the RSA cipher object}
    RSACipher := TLbRSA.Create(nil);

    {set the public key}
    ASN1Key.Position := 0;
    RSACipher.PublicKey.LoadFromStream(ASN1Key);

    {encrypt the plaintext stream}
    RSACipher.EncryptStream(PlainStream, CipherStream);

    ..clean up..
```

After this code has executed, the cipher.rsa file can be sent to the recipient. Since he is the only one that knows the corresponding private key, he is the only one that can decrypt it. The following example shows how. (It is assumed here, by the way, that the recipient has saved the private key on his or her disk, encrypted with Blowfish in a file. Private keys should never be written to a persistent medium in plaintext.)

```
var
    Blowfish      : TLbBlowfish;
    ASN1Key       : TMemoryStream;
    RSACipher     : TLbRSA;
    PlainStream   : TFileStream;
    CipherStream  : TFileStream;
begin
    {create the RSA cipher object}
    RSACipher := TLbRSA.Create(nil);

    {initialize the Blowfish cipher}
    Blowfish := TLbBlowfish.Create(nil);
    Blowfish.GenerateKey(MyPassphrase);
    Blowfish.CipherMode := cmCBC;

    {decrypt the ASN.1 formatted key}
    CipherStream := TFileStream('private.key', fmOpenRead or
fmShareDenyWrite);
    ASN1Key := TMemoryStream.Create;
    Blowfish.DecryptStream(CipherStream, ASN1Key);

    {get the private key from the ASN.1 format}
    ASN1Key.Position := 0;
    RSACipher.PrivateKey.LoadFromStream(ASN1Key);

    {we're done with the Blowfish part now}
    CipherStream.Free;
    ASN1Key.Free;
    Blowfish.Free;

    {open the two file streams}
    CipherStream := TFileStream.Create('cipher.rsa', fmOpenRead or
fmShareDenyWrite);
    PlainStream := TFileStream('plain.txt', fmCreate);

    {decrypt the ciphertext stream}
    RSACipher.DecryptStream(CipherStream, PlainStream);

    ..clean up..
```

The code first of all creates the RSA cipher object. The next step is to read the private.key file from disk, and decrypt it using Blowfish and the supplied passphrase. This will result in the ASN1Key memory stream containing the ASN.1 formatted private key. Then the value is

loaded into the PrivateKey property of the RSA cipher object. At that point the reading, decrypting and decoding of the private key is done. Now the process is easy: decrypt the cipher.rsa file into the plain.txt file using the RSA cipher object.

Hierarchy

TComponent (VCL)

❶ TLbBaseComponent (LbClass)	107
❷ TLbCipher (LbClass)	109
❸ TLbAsymmetricCipher (LbAsym)	126
TLbRSA (LbAsym)	

Properties

❸ PrivateKey	❸ PublicKey	❶ Version
--------------	-------------	-----------

Methods

❷ DecryptBuffer	❷ DecryptString	❷ EncryptStream
❷ DecryptFile	❷ EncryptBuffer	❷ EncryptString
❷ DecryptStream	❷ EncryptFile	

Chapter 5: Hash Components

Although LockBox provides all of the routines necessary to incorporate cryptographic hashing algorithms into your applications, not everyone needs the full level of control afforded by these low-level routines. In the vast majority of cases, developers would just like to calculate the message digests of buffers, streams, files, or strings using a particular algorithm and anything to make that easier would be highly beneficial. They are not interested in the nuances of hashing; they have a particular need and the simpler it is to implement that need, the better.

LockBox contains a hash component hierarchy to fulfill this need. Figure 5.1 shows this class hierarchy. At its base is a hashing engine class, TLbHash. This class has virtual methods to hash an arbitrary buffer of data, a file, a stream, or a string.

5

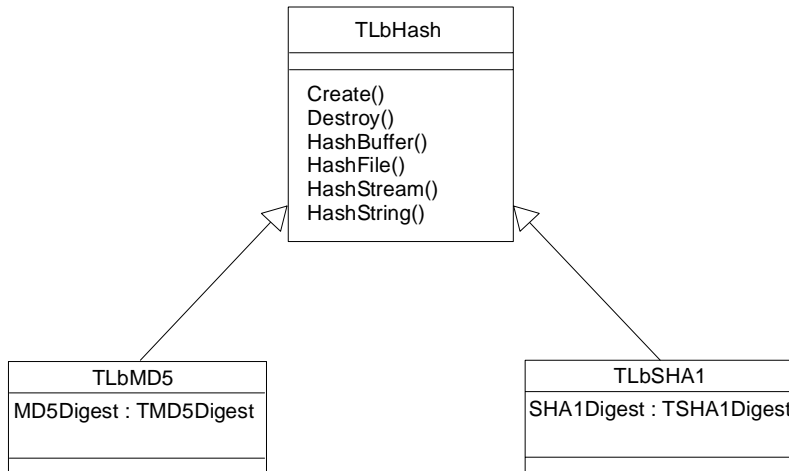


Figure 5.1: The LockBox hash component hierarchy

There are two simple descendant classes for each of the supported hash algorithms, one for MD5 and the other for SHA-1.

The following code shows how to use the TLbSHA1 component, a descendant of TLbHash, to hash the contents of a stream.

```
var
    SHA1 : TLbSHA1;
    SHA1Digest : TSHA1Digest;
begin
    SHA1 := TLbSHA1.Create(nil);
    SHA1.HashStream(MyStream);
    SHA1.GetDigest(SHA1Digest);
    ..use the digest..
    SHA1.Free;
end;
```

TLbHash Class

The TLbHash class is the base class for the hash component hierarchy. It descends from TLbBaseComponent in order to inherit the Version property.

TLbHash manages the basic functionality of a hashing engine: calculating the message digest of buffers of data, files, streams and strings. These options are implemented as virtual methods and will be overridden in the descendants that implement the different hash algorithms. The descendant classes contain the methods needed to access the calculated message digest.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
 - TLbHash (LbClass)

Properties

- ❶ Version

Methods

HashBuffer	HashStream
HashFile	HashString

Reference Section

HashBuffer

virtual method

```
procedure HashBuffer(  
    const Buf; BufLength : Longint); virtual; abstract;
```

↳ Hashes a buffer of data.

HashBuffer calculates the message digest of the data in the Buf buffer. BufLength is the number of bytes of data in Buf.

The message digest is not available in this ancestor class. The descendant classes define methods to read the hash.

HashFile

virtual method

```
procedure HashFile(const AFileName : string);  
    virtual; abstract;
```

↳ Hashes a file.

HashFile calculates the message digest of the data in the AFileName file.

The message digest is not available in this ancestor class. The descendant classes define methods to read the hash.

HashStream

virtual method

```
procedure HashStream(AStream : TStream); virtual; abstract;
```

↳ Hashes a stream.

HashStream calculates the message digest of the data in the AStream stream. HashStream does not reset the position of Stream prior to processing, but will process all of the remaining bytes in Stream.

The message digest is not available in this ancestor class. The descendant classes define methods to read the hash.

```
procedure HashString(const AStr : string); virtual; abstract;
```

↳ Hashes a string.

HashStream calculates the message digest of the characters in the AStr string.

The message digest is not available in this ancestor class. The descendant classes define methods to read the hash.

TLbMD5 Component

The TLbMD5 component provides secure hashing facilities using the MD5 algorithm. It descends from TLbHash.

The majority of methods implemented by TLbMD5 are all overridden from virtual methods defined by its ancestor. It has a method, GetDigest, to return the message digest calculated from one of the HashXxx methods.

For more information about the MD5 algorithm, see page 73.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
- ❷ TLbHash (LbClass)..... 135

TLbMD5 (LbClass)

Properties

- ❶ Version

Methods

- | | | |
|--------------|--------------|--------------|
| GetDigest | ❷ HashFile | ❷ HashString |
| ❷ HashBuffer | ❷ HashStream | |

Reference Section

GetDigest

method

```
procedure GetDigest(var Digest : TMD5Digest);  
TMD5Digest = array [0..15] of Byte;
```

↳ Retrieves the message digest.

Once you have called one of the HashXxx methods (defined in the ancestor class, TLbHash), you obtain the actual MD5 hash by calling GetDigest. Every call to a HashXxx method will reset the digest prior to hashing the data.

TLbSHA1 Component

The TLbSHA1 component provides secure hashing facilities using the SHA-1 algorithm. It descends from TLbHash.

The majority of methods implemented by TLbSHA1 are overridden from virtual methods defined by its ancestor. It has a method, GetDigest, to return the message digest calculated from one of the HashXxx methods.

For more information about the SHA-1 algorithm, see page 93.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
 - ❷ TLbHash (LbClass) 135
 - TLbSHA1 (LbClass)

Properties

- ❶ Version

Methods

- | | | |
|--------------|--------------|--------------|
| GetDigest | ❷ HashFile | ❷ HashString |
| ❷ HashBuffer | ❷ HashStream | |

Reference Section

GetDigest

method

```
procedure GetDigest(var Digest : TSHA1Digest);  
TSHA1Digest = array [0..19] of Byte;
```

↳ Retrieves the message digest.

Once you have called one of the HashXxx methods (defined in the ancestor class, TLbHash), you obtain the actual SHA-1 hash by calling GetDigest. Every call to a HashXxx method will reset the digest prior to hashing the data.

Chapter 6: Digital Signature Components

A digital signature for a document is a hash of the data in the document, encrypted using an asymmetric algorithm with a private key. To verify the signature, the hash is recalculated from the same document and the hash obtained from decrypting the signature with the public key should match. Although the term “document” seems to imply a file, in actuality a document can be any block, stream or file of data.

A digital signature performs two functions: it authenticates the integrity of the signed document (has the document changed since being signed?) and it verifies the identity of the signer (since the signature was decrypted with Bob’s public key, Bob must have signed it with his private one).

To be formal, LockBox implements *signature schemes with appendix*. A signature scheme with appendix is a signature algorithm with two operations. The first calculates the signature of a document using the signer’s private key; the second verifies the signature using the signer’s public key. For the verification operation the document is required as a separate entity (that is, the appendix).

In general, there is no need to physically link a signature to the entity to which it refers. There only has to be the understanding between the parties concerned that a particular document is signed. If a signature cannot be found for a document, that omission is taken to be equivalent to an invalid signature: that is, either the document has changed or the sender didn’t sign the document. You could certainly ensure that the signature file and the document file don’t get separated by archiving them both in the same Zip file, for example, but this falls outside the realm of digital signatures.

Although LockBox provides all of the low-level routines necessary to incorporate signature schemes into your applications, a class hierarchy implementing digital signatures will make it much easier to do so, and that is the subject of this chapter.

Figure 6.1 shows the LockBox digital signature component hierarchy. At its base is a digital signing engine class, TLbSignature. This class has virtual methods to digitally sign an arbitrary buffer of data, a file, a stream, or a string.

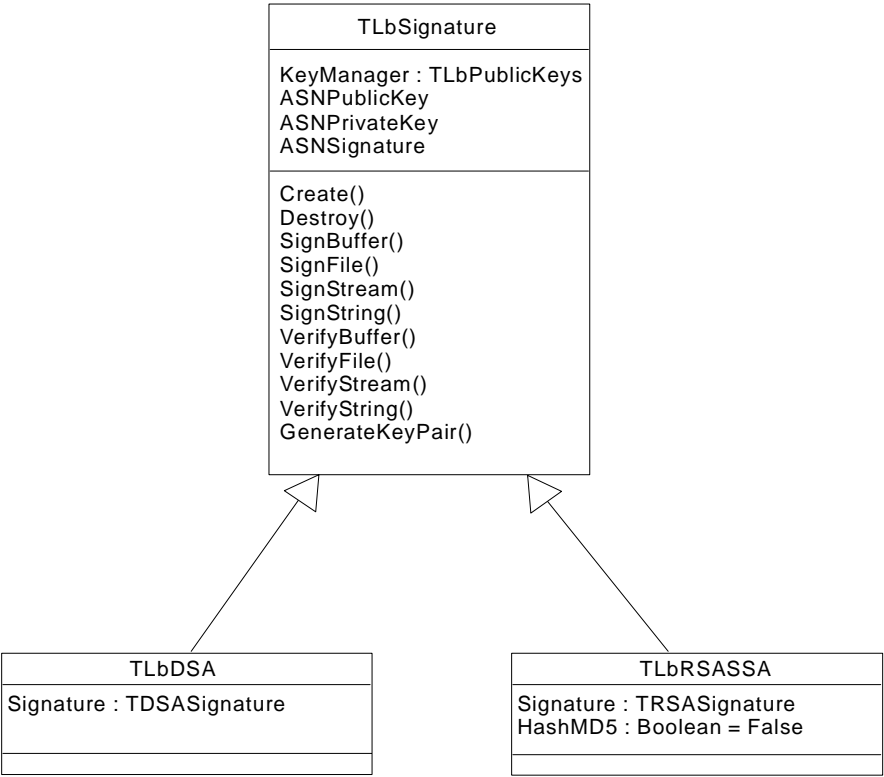


Figure 6.1: The LockBox digital signature component hierarchy

There are then two simple descendant classes for each of the supported digital signatures, one for DSA (Digital Signature Algorithm) and the other for RSASSA (RSA Signature Scheme with Appendix).

Note that, for simplicity's sake, no exception handling is shown in the example that follows. For a discussion and details on how to store and retrieve a private key, please see the example in chapter 7 on page 156. The following code shows how to use the TLbDSA component, a descendant of TLbSignature, to sign the contents of a file:

```
var
    DSA      : TLbDSA;
    DSASig   : TLbDSASig;
    SigFile  : TFileStream;
    ASN1Key  : TMemoryStream;
begin
    {create the signer}
    DSA := TLbDSA.Create(nil);

    {read the private key value}
    ..read the private key into the ASN1Key stream..
    DSA.PrivateKey.LoadFromStream(ASN1Key);
    ASN1Key.Free;

    {sign the document file}
    DSA.SignFile('plain.doc', fmOpenRead, DSASig);

    {clean up}
    DSA.Free;

    {save the signature in another file}
    SigFile := TFileStream.Create('plain.sig', fmCreate);
    SigFile.WriteBuffer(DSASig, sizeof(DSASig));
    SigFile.Free;
```

The recipient of the document and the signature now has to verify the signature. The following example shows how. Again, for simplicity's sake, no exception handling is shown:

```
var
    DSA      : TLbDSA;
    DSASig   : TLbDSASig;
    SigFile  : TFileStream;
    ASN1Key  : TMemoryStream;
begin
    {read the signature from its file}
    SigFile := TFileStream.Create('plain.sig', fmOpenRead);
    SigFile.ReadBuffer(DSASig, sizeof(DSASig));
    SigFile.Free;

    {create the verifier}
    DSA := TLbDSA.Create(nil);

    {read the public key value}
    ..read the public key into the ASN1Key stream..
    DSA.PublicKey.LoadFromStream(ASN1Key);
    ASN1Key.Free;

    {verify the document file}
    if not DSA.VerifyFile('plain.doc', fmOpenRead, DSASig) then
        raise Exception.Create('Signature is invalid');

    {clean up}
    DSA.Free;
```

TLbSignature Class

The TLbSignature class is the base class for the hash component hierarchy. It descends from TLbBaseComponent in order to inherit the Version property.

TLbSignature manages the basic functionality of a signing engine: calculating the digital signature of buffers of data, files, streams and strings. These options are implemented as virtual methods and are overridden in the descendants that implement the different signing algorithms. Since the size of the signature itself varies according to the signing algorithm, at this level the signatures are declared as untyped buffers.

There are two properties declared in this ancestor class: the public and private key. These are instances of the TLbAsymmetricKey class, which is described in Chapter 7 on page 156.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
- TLbSignature (LbClass)

Properties

PrivateKey

PublicKey

❶ Version

Methods

GenerateKeyPair

SignStream

VerifyFile

SignBuffer

SignString

VerifyStream

SignFile

VerifyBuffer

VerifyString

Reference Section

GenerateKeyPair

virtual abstract method

```
procedure GenerateKeyPair;
```

↳ Calculates a new private and public key pair.

Each descendant overrides this method to generate a new key pair, one for the `PrivateKey` and one for the `PublicKey` properties.

PrivateKey

read-only property

```
property PrivateKey : TLbAsymmetricKey
```

↳ Returns the private key for the digital signature.

`PrivateKey` is only used for creating the signature.

The digital signature class manages the private key internally. Although `PrivateKey` is read-only, its value can be changed either by calling `GenerateKeyPair`, or by calling `PrivateKey`'s `LoadFromStream` or `Assign` methods. Since a new instance of a descendant class has both its private and public keys set to zeros, `PrivateKey`'s value must be set through one of these routines before performing any signing or verification.

See the code on page 145 for an example of how to use this property.

See also: `GenerateKeyPair`, `PublicKey`, `TLbAsymmetricKey.Assign`,
`TLbAsymmetricKey.LoadFromStream`

PublicKey

read-only property

```
property PublicKey : TLbAsymmetricKey
```

↳ Returns the public key for the digital signature.

`PublicKey` is only used for verifying the signature.

The digital signature class manages the public key internally. Although `PublicKey` is read-only, its value can be changed either by calling `GenerateKeyPair`, or by calling `PrivateKey`'s `LoadFromStream` or `Assign` methods. Since a new instance of a descendant class has both its private and public keys set to zeros, `PrivateKey`'s value must be set through one of these routines before performing any signing or verification.

See the code on page 146 for an example of how to use this property.

See also: `GenerateKeyPair`, `PublicKey`, `TLbAsymmetricKey.Assign`,
`TLbAsymmetricKey.LoadFromStream`

SignBuffer

virtual abstract method

```
procedure SignBuffer(  
    const Buf; BufLen : Longint; var Sig); virtual; abstract;
```

↪ Signs a buffer of data.

SignBuffer calculates the digital signature of the data in the Buf buffer. BufLen is the number of bytes of data in Buf to sign. Signing uses the PrivateKey property; hence its value must be set before calling this routine.

The digital signature is returned in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the type of their signature.

See also: VerifyBuffer

SignFile

virtual abstract method

```
procedure SignFile(  
    const AFileName : string; var Sig); virtual; abstract;
```

↪ Signs a file.

SignFile calculates the digital signature of the data in the AFileName file. Signing uses the PrivateKey property; hence its value must be set before calling this routine.

The digital signature is returned in the Sig buffer. The type (and hence size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the type of their signature.

See also: VerifyFile

SignStream**virtual abstract method**

```
procedure SignStream(AStream : TStream; var Sig); virtual;
abstract;
```

↳ Signs a stream.

SignStream calculates the digital signature of the data in the AStream stream. SignStream does not reset the position of AStream prior to processing, but will, however, process all of the remaining bytes in AStream. Signing uses the PrivateKey property; hence its value must be set before calling this routine.

The digital signature is returned in the Sig buffer. The type (and hence size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: VerifyStream

SignString**virtual abstract method**

```
procedure SignString(const AStr : string; var Sig); virtual;
abstract;
```

↳ Signs a string.

SignStream calculates the digital signature of the characters in the AStr string. SignString uses the PrivateKey property; hence, its value must be set before calling this routine.

The digital signature is returned in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: VerifyString

```
procedure VerifyBuffer(  
    const Buf; BufLen : Longint; const Sig); virtual; abstract;
```

↪ Verifies the digital signature of a buffer of data.

VerifyBuffer checks that the digital signature matches the data in the Buf buffer. BufLen is the number of bytes of data in Buf. Verification uses the PublicKey property; hence, its value must be set before calling this routine.

The digital signature is passed to the routine in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: SignBuffer

```
procedure VerifyFile(  
    const AFileName : string; const Sig); virtual; abstract;
```

↪ Verifies the digital signature of a file.

VerifyFile checks that the digital signature matches the data in the AFileName file. Verification uses the PublicKey property; hence, its value must be set before calling this routine.

The digital signature is passed to the routine in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: SignFile

```
procedure VerifyStream(  
  AStream : TStream; const Sig); virtual; abstract;
```

↳ Verifies the digital signature of a stream.

VerifyStream checks that the digital signature matches the data in the AStream stream. VerifyStream does not reset the position of AStream prior to processing, but will, however, process all of the remaining bytes in AStream. Verification uses the PublicKey property; hence, its value must be set before calling this routine.

The digital signature is passed to the routine in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: SignStream

VerifyString

virtual abstract method

```
procedure VerifyString(  
  const AStr : string; const Sig); virtual; abstract;
```

↳ Verifies the digital signature of a string.

VerifyString checks that the digital signature matches the characters in the AStr string. Verification uses the PublicKey property; hence, its value must be set before calling this routine.

The digital signature is passed to the routine in the Sig buffer. The type (and size) of this buffer depends on the actual signature algorithm being employed. See the description of the descendant classes, TLbDSA and TLbRSASSA, for information about the types of their signatures.

See also: SignString

TLbDSA Component

The TLbDSA component provides digital signing facilities using the DSA (Digital Signature Algorithm) algorithm. It descends from TLbSignature.

The signature for DSA is 40 bytes long and is defined by the following type:

```
TLbDSASig = packed record
  R : TSHA1Digest;
  S : TSHA1Digest;
end;
TSHA1Digest = array [0..19] of Byte;
```

The methods implemented by TLbDSA are all overridden from virtual methods defined by its ancestor.

For more information about the DSA algorithm, go to www.abanet.org/scitech/ec/isc/dsg-tutorial.html.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass) 107
- ❷ TLbSignature (LbClass) 147

TLbDSA (LbClass)

Properties

- ❷ PrivateKey
- ❷ PublicKey
- ❶ Version

Methods

- ❷ GenerateKeyPair
- ❷ SignStream
- ❷ VerifyFile
- ❷ SignBuffer
- ❷ SignString
- ❷ VerifyStream
- ❷ SignFile
- ❷ VerifyBuffer
- ❷ VerifyString

TLbRSASSA Component

The TLbRSASSA component provides digital signing facilities using the RSASSA (RSA Signature Scheme with Appendix) algorithm. It descends from TLbSignature.

The signature for RSASSA is 64 bytes long and is defined by the following type:

```
LbRSAKeySize = 64;  
TLbRSASig = array[0..LbRSAKeySize-1] of Byte;
```

The methods implemented by TLbRSASSA are all overridden from virtual methods defined by its ancestor.

For more information about the RSASSA algorithm, go to www.abanet.org/scitech/ec/isc/dsg-tutorial.html.

Hierarchy

TComponent (VCL)

- ❶ TLbBaseComponent (LbClass)107
 - ❷ TLbSignature (LbClass)147
 - TLbRSASSA (LbClass)

Properties

- | | | |
|--------------|-------------|-----------|
| ❷ PrivateKey | ❷ PublicKey | ❶ Version |
|--------------|-------------|-----------|

Methods

- | | | |
|-------------------|----------------|----------------|
| ❷ GenerateKeyPair | ❷ SignStream | ❷ VerifyFile |
| ❷ SignBuffer | ❷ SignString | ❷ VerifyStream |
| ❷ SignFile | ❷ VerifyBuffer | ❷ VerifyString |

Chapter 7: Asymmetric Key Class

Both the RSA encryption algorithm and the digital signature algorithms require two keys: a private key and a public key. The two keys form a pair; they are connected in a mathematical relationship. Public keys are usually created using the ASN.1 format (Abstract Syntax Notation One). For details on the ASN.1 format, please see CCIT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*. 1988. The *Request For Comments 2459* (RFC-2459) also has an appendix that gives a brief overview of the ASN.1 format.

To make the management of these keys easier, LockBox provides an asymmetric key class, `TLbAsymmetricKey`. This class provides properties to get at the modulus and exponent as big integer objects for calculation purposes, and it provides routines to write the key out and to read in a key in ASN.1 format.

TLbAsymmetricKey Class

The TLbAsymmetricKey class is the class that encapsulates an asymmetric key, either the private key or the public key. It descends from TLbBaseComponent in order to inherit the Version property.

Both the RSA cipher and the digital signature components require two asymmetric keys, one the public key and the other the private key. Although the TLbAsymmetricKey class does not prevent the keys from getting mixed up, it shouldn't happen. There is a strong mathematical relationship between the public and private key and they should be kept together.

The TLbSymmetricKey class enables the reading or writing of keys in ASN.1 format. Note that the format is merely an encapsulation of the data, no compression or encryption is applied, for example. Although it is certainly possible to store private keys on disk in files using the StoreToStream method, this is strongly advised against because of the security risks. Instead, encrypt private keys using one of LockBox's encryption algorithms. (A public key should of course be stored as plaintext; otherwise it is of little use.) The following example sketches this out using the Blowfish cipher:

```
var
    TempStream : TMemoryStream;
    FileStream : TFileStream;
    Blowfish    : TLbBlowfish;
    PrivateKey  : TLbAsymmetricKey;
begin
    ..create the objects..

    {get the private key in ASN.1 format}
    PrivateKey.StoreToStream(TempStream);

    {initialize the Blowfish cipher}
    Blowfish.GenerateKey(MyPassphrase);
    Blowfish.CipherMode := cmCBC;

    {encrypt the ASN.1 formatted key}
    TempStream.Position := 0;
    Blowfish.EncryptStream(TempStream, FileStream);

    ..free the objects..
```

The MyPassphrase string variable would presumably be obtained at run-time from the user.

Retrieving the key is a simple matter of reversing the process:

```
var
  TempStream : TMemoryStream;
  FileStream : TFileStream;
  Blowfish   : TLbBlowfish;
  PrivateKey : TLbAsymmetricKey;
begin
  ..create the objects..

  {initialize the Blowfish cipher}
  Blowfish.GenerateKey(MyPassphrase);
  Blowfish.CipherMode := cmCBC;

  {decrypt the ASN.1 formatted key}
  Blowfish.DecryptStream(FileStream, TempStream);

  {get the private key in ASN.1 format}
  TempStream.Position := 0;
  PrivateKey.LoadFromStream(TempStream);

  ..free the objects except for the key..
```

Hierarchy

TComponent (VCL)

❶ TLbBaseComponent (LbClass)	107
TLbAsymmetricKey (LbClass)	

Properties

Exponent	Modulus	❶ Version
----------	---------	-----------

Methods

Assign	LoadFromStream	StoreToStream
--------	----------------	---------------

Reference Section

Assign

method

```
procedure Assign(aKey : TLbAsymmetricKey);
```

↳ Duplicates another asymmetric key.

Assign copies the modulus and exponent of aKey to this instance, making them equal. aKey is not changed by this routine.

Exponent

property

```
property Exponent : TLbBigInt
```

```
TLbBigInt = class
```

↳ Returns the exponent for the asymmetric key.

This property is used internally for the encryption or signing calculations.

See also: Modulus

Modulus

property

```
property Modulus : TLbBigInt
```

```
TLbBigInt = class
```

↳ Returns the modulus for the asymmetric key.

This property is used internally for the encryption or signing calculations.

See also: Exponent

LoadFromStream

method

```
procedure LoadFromStream(aStream : TStream);
```

↳ Retrieves a key value from a stream.

Asymmetric keys are stored and transmitted in ASN.1 format. This method reads a key from aStream in ASN.1 format.

The LoadFromStream does not reset the stream prior to reading from it. On return the stream will be positioned after the key data.

See also: StoreToStream

```
procedure StoreToStream(aStream : TStream);
```

↪ Saves a key value to a stream.

Asymmetric keys are stored and transmitted in ASN.1 format. This method writes the key value to aStream in ASN.1 format.

The StoreToStream does not reset the stream prior to writing to it. On return the stream will be positioned after the just-written key data.

● **Caution:** Although this method will write out a private key to a stream, you should never write a private key unencrypted on a persistent medium (e.g., disk). Please see page 156 for a way of storing a private key by encrypting it.

See also: StoreToStream

Chapter 8: Stream Classes

The LbClass unit implements several stream classes descended from the VCL TMemoryStream and TFileStream classes. These classes provide a simple way to use the LockBox's stream ciphers on streams. Writing to a LockBox stream class instance encrypts the data. Reading from such a stream decrypts the data. Because these classes only encapsulate stream ciphers, the ciphertext will be the same size as the original plaintext from which it was encrypted

- **Caution:** These stream classes are provided for backwards compatibility only. LockBox now contains simpler to use encryption components that are not limited to file streams or memory streams and that instead work on any stream descendant class. Also these classes encapsulate stream ciphers only, and all of LockBox's stream ciphers are not secure.

The following example reads data from one stream, encrypts the data, and writes the data to another stream:

```
var
  Key : TKey128; {a 128 bit key}
  PlainText : TFileStream;
  CipherText : TLbSCFileStream;
begin
  {create an encryption key}
  GenerateLMDKey(Key, SizeOf(Key),
    'Three can keep a secret, if two are dead.');
```

8

```
  PlainText := nil;
  CipherText := nil;
  try
    {create the plaintext stream}
    PlainText := TFileStream.Create('TEST1.TXT', fmOpenRead or
fmShareDenyWrite);
    {create the ciphertext stream}
    CipherText := TLbSCFileStream.Create('TEST2.LSC', fmCreate,
Key, SizeOf(Key));
    {copy and encrypt the plaintext stream to the ciphertext
stream}
    CipherText.CopyFrom(PlainText, PlainText.Size);
  finally
    CipherText.Free;
    PlainText.Free;
  end;
end;
```

With the file stream class you get to specify what mode is used to open the input file. Additionally, if you use the Read and Write methods of the stream class instead of the CopyFrom method to process the files, you could pre-process the data bytes prior to encrypting them as shown in the following example:

```
BytesRead := PlainText.Read(Buffer, sizeof(Buffer));
while (BytesRead <> 0) do begin
    ..process the data in Buffer..
    CipherText.Write(Buffer, BytesRead);
    BytesRead := PlainText.Read(Buffer, sizeof(Buffer));
end;
```

TLbSCStream Class

The TLbSCStream class encapsulates the capabilities of the LockBox Stream Cipher (LSC) making it easy to encrypt and decrypt data stored in memory (e.g., in buffers or records) without directly using the low-level encryption routines.

Use a TLbSCStream class just as you would a normal stream class. When writing to the stream, the data is encrypted. When reading from the stream, the data is decrypted.

Hierarchy

TMemoryStream (VCL)
 TLbSCStream (LbClass)

Methods

ChangeKey	Read	Write
Create	Reinitialize	

Reference Section

ChangeKey

method

```
procedure ChangeKey(const Key; KeySize : Integer); dynamic;
```

- ↳ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to `Create` is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used by the cipher.

See also: `Create`, `Reinitialize`

Create

constructor

```
constructor Create(const Key; KeySize : Integer);
```

- ↳ Creates the stream and initializes the context used to process the stream data.

Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used.

See also: `ChangeKey`

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

- ↳ Reads the specified number of bytes from the stream.

Count bytes of data are read from the stream, decrypted, and put in Buffer. If there are less than Count bytes in the stream, only the number of bytes in the stream are decrypted and placed in Buffer. The stream position is adjusted by Count bytes. The function result is the actual number of bytes decrypted and placed in Buffer.

See also: `Write`

Reinitialize

method

```
procedure Reinitialize(const Key; KeySize : Integer); dynamic;
```

↪ Changes the key and resets the stream position.

Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↪ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

TLbSCFileStream Class

The TLbSCFileStream class encapsulates the capabilities of the LockBox Stream Cipher (LSC) making it easy to encrypt and decrypt files without directly using the low-level encryption routines.

You use a TLbSCFileStream class just as you would a normal stream class. When writing to the file stream, the data is encrypted and written to the file. When reading from the file stream, the data is decrypted and stored in a user-supplied buffer.

Hierarchy

TFileStream (VCL)

 TLbSCFileStream (LbClass)

Methods

ChangeKey

Read

Write

Create

Reinitialize

Reference Section

ChangeKey

method

```
procedure ChangeKey(const Key; KeySize : Integer); dynamic;
```

- ↳ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to Create is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used by the cipher.

See also: Create, Reinitialize

Create

constructor

```
constructor Create(const FileName : string;  
  Mode : Word; const Key; KeySize : Integer);
```

- ↳ Creates the stream and initializes the context used to process the stream data.

FileName and Mode are the same as those described for the VCL TFileStream class. Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used.

See also: ChangeKey

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

- ↳ Reads the specified number of bytes from the stream.

Count bytes of data are read from the stream, decrypted, and put in Buffer. If there are less than Count bytes in the stream, only the number of bytes in the stream are decrypted and placed in Buffer. The stream position is adjusted by Count bytes. The function result is the actual number of bytes decrypted and placed in Buffer.

See also: Write

Reinitialize

method

```
procedure Reinitialize(const Key; KeySize : Integer); dynamic;
```

↳ Changes the key and resets the stream position.

Key is the key used to encrypt and decrypt the stream data. KeySize is the size (in bytes) of Key. Key can be any size, but only the first 256 bytes are used. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↳ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

TLbRNG32Stream Class

TLbRNG32Stream encapsulates the capabilities of the 32-bit key Random Number Generator stream cipher (RNG32) making it easy to encrypt and decrypt data stored in memory (e.g., in buffers or records) without directly using the low-level encryption routines.

You use a TLbRNG32Stream class just as you would a normal stream class. When writing to the stream, the data is encrypted. When reading from the stream, the data is decrypted.

Hierarchy

TMemoryStream (VCL)

TLbRNG32Stream (LbClass)

Methods

ChangeKey

Read

Write

Create

Reinitialize

Reference Section

ChangeKey

method

```
procedure ChangeKey(const Key : LongInt); dynamic;
```

- ↳ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to `Create` is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid.

See also: `Create`, `Reinitialize`

Create

constructor

```
constructor Create(const Key : LongInt);
```

- ↳ Creates the stream and initializes the context used to process the stream data.

Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid.

See also: `ChangeKey`

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

- ↳ Reads the specified number of bytes from the stream.

Count bytes of data are read from the stream, decrypted, and put in `Buffer`. If there are less than Count bytes in the stream, only the number of bytes in the stream are decrypted and placed in `Buffer`. The stream position is adjusted by Count bytes. The function result is the actual number of bytes decrypted and placed in `Buffer`.

See also: `Write`

Reinitialize

method

```
procedure Reinitialize(const Key : LongInt); dynamic;
```

↪ Changes the key and resets the stream position.

Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↪ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

TLbRNG32FileStream Class

TLbRNG32FileStream encapsulates the capabilities of the 32-bit key Random Number Generator stream cipher (RNG32), making it easy to encrypt and decrypt files without directly using the low-level encryption routines.

You use a TLbRNG32FileStream class just as you would a normal stream class. When writing to the file stream, the data is encrypted and written to the file. When reading from the file stream, the data is decrypted and stored in a user-supplied buffer.

Hierarchy

TFileStream (VCL)

 TLbRNG32FileStream (LbClass)

Methods

ChangeKey

Read

Write

Create

Reinitialize

Reference Section

ChangeKey

method

```
procedure ChangeKey(const Key : LongInt); dynamic;
```

- ↳ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to Create is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid.

See also: Create, Reinitialize

Create

constructor

```
constructor Create(  
    const FileName : string; Mode : Word; const Key : LongInt);
```

- ↳ Creates the stream and initializes the context used to process the stream data.

FileName and Mode are the same as those described for the Delphi TFileStream. Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid.

See also: ChangeKey

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

- ↳ Reads the specified number of bytes from the stream.

Count bytes of data are read from the stream, decrypted, and put in Buffer. If there are less than Count bytes in the stream, only the number of bytes in the stream are decrypted and placed in Buffer. The stream position is adjusted by Count bytes. The function result is the actual number of bytes decrypted and placed in Buffer.

See also: Write

Reinitialize

method

```
procedure Reinitialize(const Key : LongInt); dynamic;
```

↳ Changes the key and resets the stream position.

Key is the key used to encrypt and decrypt the stream data. All possible values (negative and positive) for key are valid. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↳ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

TLbRNG64Stream Class

TLbRNG64Stream encapsulates the capabilities of the 64-bit key Random Number Generator stream cipher (RNG64), making it easy to encrypt and decrypt data stored in memory (e.g., in buffers or records) without directly using the low-level encryption routines.

You use a TLbRNG64Stream class just as you would a normal stream class. When writing to the stream, the data is encrypted. When reading from the stream, the data is decrypted.

Hierarchy

TMemoryStream (VCL)

TLbRNG64Stream (LbClass)

Methods

ChangeKey

Read

Write

Create

Reinitialize

Reference Section

ChangeKey

method

```
procedure ChangeKey(const KeyHi, KeyLo : LongInt); dynamic;
```

↪ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to `Create` is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

`KeyHi` and `KeyLo` are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for `KeyHi` and `KeyLo` are valid.

See also: `Create`, `Reinitialize`

Create

constructor

```
constructor Create(const KeyHi, KeyLo : LongInt);
```

↪ Creates the stream and initializes the context used to process the stream data.

`KeyHi` and `KeyLo` are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for `KeyHi` and `KeyLo` are valid.

See also: `ChangeKey`

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

↪ Reads the specified number of bytes from the stream.

`Count` bytes of data are read from the stream, decrypted, and put in `Buffer`. If there are less than `Count` bytes in the stream, only the number of bytes in the stream are decrypted and placed in `Buffer`. The stream position is adjusted by `Count` bytes. The function result is the actual number of bytes decrypted and placed in `Buffer`.

See also: `Write`

Reinitialize

method

```
procedure Reinitialize(const KeyHi, KeyLo : LongInt); dynamic;
```

↪ Changes the key and resets the stream position.

KeyHi and KeyLo are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for KeyHi and KeyLo are valid. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↪ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

TLbRNG64FileStream Class

TLbRNG64FileStream encapsulates the capabilities of the 64-bit key Random Number Generator stream cipher (RNG64), making it easy to encrypt and decrypt files without directly using the low-level encryption routines.

You use a TLbRNG64FileStream class just as you would a normal stream class. When writing to the file stream, the data is encrypted and written to the file. When reading from the file stream, the data is decrypted and stored in a user-supplied buffer.

Hierarchy

TFileStream (VCL)

 TLbRNG64FileStream (LbClass)

Methods

ChangeKey

Read

Write

Create

Reinitialize

Reference Section

ChangeKey

method

```
procedure ChangeKey(const KeyHi, KeyLo : LongInt); dynamic;
```

- ↳ Replaces the existing key and initializes the context used to process the stream.

This method can be called to change the encryption key while processing the stream. Normally, the key specified in the call to Create is used to process all of the stream data, but this method is provided so that you can change the key at any point. It is recommended that the key not be changed part way through processing the stream.

KeyHi and KeyLo are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for KeyHi and KeyLo are valid.

See also: Create, Reinitialize

Create

constructor

```
constructor Create(const FileName : string;  
  Mode : Word; const KeyHi, KeyLo : LongInt);
```

- ↳ Creates the stream and initializes the context used to process the stream data.

FileName and Mode are the same as those described for the Delphi TFileStream. KeyHi and KeyLo are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for KeyHi and KeyLo are valid.

See also: ChangeKey

Read

method

```
function Read(var Buffer; Count : LongInt) : LongInt; override;
```

- ↳ Reads the specified number of bytes from the stream.

Count bytes of data are read from the stream, decrypted, and put in Buffer. If there are less than Count bytes in the stream, only the number of bytes in the stream are decrypted and placed in Buffer. The stream position is adjusted by Count bytes. The function result is the actual number of bytes decrypted and placed in Buffer.

See also: Write

Reinitialize

method

```
procedure Reinitialize(const KeyHi, KeyLo : LongInt); dynamic;
```

↳ Changes the key and resets the stream position.

KeyHi and KeyLo are combined to create the 64-bit key used to encrypt and decrypt the stream data. All possible values (negative and positive) for KeyHi and KeyLo are valid. After calling this method, the stream is repositioned to the beginning.

Reinitialize performs the same function as ChangeKey, but also repositions the stream.

See also: ChangeKey

Write

method

```
function Write(const Buffer; Count : LongInt) : LongInt; override;
```

↳ Writes the specified number of bytes to the stream.

Count bytes of data are read from Buffer, encrypted, and written to the stream. The stream position is adjusted by Count bytes. The function result is the actual number of bytes encrypted and written to the stream.

See also: Read

Identifier Index

A

Assign 158

B

BFEncryptFile 35, 37
BFEncryptFileCBC 35, 37
BFEncryptStream 36
BFEncryptStreamCBC 36
Blowfish Cipher 32

C

ChangeKey 164, 167, 170, 173, 176, 179
CipherMode 115
Create 164, 167, 170, 173, 176, 179

D

DecryptBuffer 110
DecryptFile 110
DecryptRSA 90
DecryptStream 110
DecryptString 111
DES Cipher 41
DESEncryptFile 44
DESEncryptFileCBC 44
DESEncryptStream 45
DESEncryptStreamCBC 45
DESEncryptString 46, 104
DESEncryptStringCBC 46, 104

E

ELF Hash 49

EncryptBF 38
EncryptBFCBC 38
EncryptBuffer 111
EncryptDES 47
EncryptDESCBC 47
EncryptFile 112
EncryptLBC 52
EncryptLBCCBC 53
EncryptLQC 63
EncryptLQCCBC 63
EncryptLSC 71
EncryptRDL 81
EncryptRDLCBC 82
EncryptRNG32 67
EncryptRNG64 67
EncryptRSA 90
EncryptStream 112
EncryptString 112
EncryptTripleDES 100
EncryptTripleDESCBC 100
Exponent 158

F

FinalizeLMD 60
FinalizeMD5 74
FinalizeSHA1 94

G

GenerateKey 115
GenerateKeyPair 127
GenerateLMDKey 58
GenerateMD5Key 58
GenerateRandomKey 58, 116
GenerateRSAKeys 91
GetDigest 139, 141

GetKey 118, 120, 122, 124

H

HashBuffer 136
 HashElf 50
 HashFile 136
 HashLMD 60
 HashMD5 74
 HashMix128 77
 HashSHA1 94
 HashStream 136
 HashString 137

I

InitEncryptBF 40
 InitEncryptDES 48
 InitEncryptLBC 54
 InitEncryptLSC 71
 InitEncryptRDL 83
 InitEncryptRNG32 68
 InitEncryptRNG64 68
 InitEncryptTripleDES 101
 InitLMD 60
 InitMD5 75
 InitSHA1 95

K

KeySize 124

L

LBCEncryptFile 54, 64
 LBCEncryptFileCBC 55, 64
 LBCEncryptStream 55
 LBCEncryptStreamCBC 56
 LoadFromStream 158
 LockBox Block Cipher 51

LockBox Key Generation 57
 LockBox Message Digest 59
 LockBox Random Number Ciphers 66
 LQCEncryptStream 65
 LQCEncryptStreamCBC 65
 LSCEncryptFile 72

M

Mix128 Hash 76
 Modulus 158

N

NewKeyPair 148

O

OutBufSizeNeeded 113

P

PrivateKey 127, 148
 PublicKey 128, 148

R

RDLEncryptFile 84, 86
 RDLEncryptFileCBC 84
 RDLEncryptStream 85
 RDLEncryptStreamCBC 85
 Read 164, 167, 170, 173, 176, 179
 Reinitialize 165, 168, 171, 174, 177, 180
 RNG32EncryptFile 68
 RNG64EncryptFile 69
 RSA Cipher 87
 RSAEncryptFile 91
 RSAEncryptStream 91
 RSAEncryptString 92

S

Secure Hashing Algorithm (SHA-1) 93
 SetKey 118, 120, 122, 125
 ShrinkDESKey 48
 SignBuffer 149
 SignFile 149
 SignStream 150
 SignString 150
 StoreToStream 159
 StringHashElf 50
 StringHashLMD 61
 StringHashMD5 75
 StringHashMix128 77
 StringHashSHA1 95

T

TLb3DES 121
 TLbAsymmetricCipher 126
 TLbBaseComponent 107
 TLbBlowfish 117
 TLbCipher 109
 TLbDES 119
 TLbDSA 153
 TLbHash 135
 TLbMD5 138
 TLbRijndael 123
 TLbRNG32FileStream class 172
 TLbRNG32Stream class 169
 TLbRNG64FileStream 178
 TLbRNG64FileStream class 178
 TLbRNG64Stream class 175

TLbRSA 129
 TLbRSASSA 154
 TLbSCFileStream 166
 TLbSCFileStream class 166
 TLbSCStream 163
 TLbSCStream class 163
 TLbSHA1 140
 TLbSignature 147
 TLbSymmetricCipher 114
 Triple-DES Cipher 97
 TripleDESEncryptFile 102
 TripleDESEncryptFileCBC 102
 TripleDESEncryptStream 103
 TripleDESEncryptStreamCBC 103

U

UpdateLMD 61
 UpdateMD5 75
 UpdateSHA1 96

V

VerifyBuffer 151
 VerifyFile 151
 VerifyStream 152
 VerifyString 152
 Version 108

W

Write 165, 168, 171, 174, 177, 180

Subject Index

A

- Advanced Encryption Standard 2
- AES 2
- asymmetric cipher
 - defined 17
 - returning private key 127
 - returning public key 128
- asymmetric key
 - defined 17
 - duplicating 158
 - returning exponent 158
 - returning modulus 158

B

- bits, reducing number in key 48
- block cipher
 - defined 19
 - defining encryption mode 115
 - list of 27
- Blowfish cipher 2, 35, 36, 37, 38, 46, 81, 82, 84, 85, 86, 104, 108, 110, 111, 112, 113, 115, 116, 118, 120, 122, 124, 136, 137, 139, 141, 148, 149, 150, 151, 152, 158, 159
- buffer
 - calculating digital signature of data 149
 - calculating size 113
 - decrypting 110
 - encrypting 111
 - generating a secure hash 60, 94
 - generating a secure hash using MD5
 - hashing algorithm 74
 - generating hash 77
 - hashing 136
 - signing 149
 - verifying digital signature 151

C

- calculating digital signature 150
- changing key 165
- character, calculating digital signature 150
- cipher
 - choosing 27
 - defined 19
 - defining encryption mode 115
- Cipher Block Chaining 20
- ciphertext
 - decrypting block 90
 - decrypting buffer 110
 - decrypting file 110
 - decrypting stream 110
 - decrypting string 111
 - defined 19
- context
 - defined 24
 - initializing 167, 170, 173, 176, 179
 - initializing structure 75
 - initializing structure for SHA-1 95
 - updating SHA-1 with hashed form of data 96
 - updating with hashed form of data 61, 75
- Cormen, Thomas H. 14
- creating stream 164, 167, 170, 173, 176, 179
- cryptography 1

D

- data
 - calculating digital signature of data
 - in AFileName file 149
 - in AStream stream 150
 - in buffer 149
- Data Encryption Standard 2
- Data Encryption Standard cipher 44, 45, 47

- Davies, D. W. 14
 - decrypting blocks of data using
 - Blowfish cipher in CBC mode 38
 - Blowfish cipher in ECB mode 38
 - DES and CBC mode 47
 - DES cipher in ECB mode 47
 - LockBox Block Cipher in CBC mode 53
 - LockBox Block Cipher in ECB mode 52
 - LockBox Quick Cipher in CBC mode 63
 - LockBox Quick Cipher in ECB mode 63
 - Rijndael cipher in CBC mode 82
 - Rijndael cipher in ECB mode 81
 - RSA algorithm 90
 - triple-DES cipher in CBC mode 100
 - triple-DES cipher in ECB mode 100
 - decrypting buffers of data using
 - 32-bit Random Number cipher 67
 - 64-bit Random Number cipher 67
 - LockBox Stream Cipher 71
 - decrypting files using
 - 32-bit Random Number cipher 68
 - 64-bit Random Number cipher 69
 - Blowfish cipher in CBC mode 35
 - Blowfish cipher in ECB mode 35
 - DES cipher in CBC mode 44
 - DES cipher in ECB mode 44
 - LockBox Block Cipher in CBC mode 55
 - LockBox Block Cipher in ECB mode 54
 - LockBox Quick Cipher in CBC mode 64
 - LockBox Quick Cipher in ECB mode 64
 - LockBox Stream Cipher 72
 - Rijndael cipher is CBC mode 84
 - Rijndael cipher is ECB mode 84
 - triple-DES cipher in CBC mode 102
 - triple-DES cipher in ECB mode 102
 - decrypting streams using
 - Blowfish cipher in CBC mode 36
 - Blowfish cipher in ECB mode 36
 - DES cipher in CBC mode 45
 - DES cipher in ECB mode 45
 - LockBox Block Cipher in CBC mode 56
 - LockBox Block Cipher in ECB mode 55
 - LockBox Quick Cipher in CBC mode 65
 - LockBox Quick Cipher in ECB mode 65
 - Rijndael cipher is CBC mode 85
 - Rijndael cipher is ECB mode 85
 - triple-DES cipher in CBC mode 103
 - triple-DES cipher in ECB mode 103
 - decrypting strings using
 - Blowfish cipher in CBC mode 37
 - Blowfish cipher in ECB mode 37
 - DES cipher in CBC mode 46
 - DES cipher in ECB mode 46
 - Rijndael cipher is CBC mode 86
 - Rijndael cipher is ECB mode 86
 - triple-DES cipher in CBC mode 104
 - triple-DES cipher in ECB mode 104
 - decryption
 - defined 16
 - private key 127
 - returning key 118, 120, 122, 124
 - setting the key 118, 120, 122, 125
 - defining encryption mode 115
 - defining size of Rijndael key 124
 - Denning, Dorothy E. 14
 - DES cipher 2, 48
 - digest, returning 74, 94
 - digital signature 22, 143
 - calculating 149, 150
 - creating 148
 - returning private key 148
 - returning public key 148
 - verifying 148
 - verifying buffer of data 151
 - verifying of file 151
 - verifying of stream 152
 - verifying of string 152
 - Digital Signature Algorithms 2
 - DSA 2
- ## E
- ELF hash

- defined 49
- generating a hash of a buffer 50
- generating a hash of a string 50
- encrypting
 - initializing a context 40
 - initializing a context for DES cipher 48
 - initializing a context with LockBox Block Cipher 54
- encrypting blocks of data using
 - Blowfish cipher in CBC mode 38
 - Blowfish cipher in ECB mode 38
 - DES and CBC mode 47
 - DES cipher in ECB mode 47
 - LockBox Block Cipher in CBC mode 53
 - LockBox Block Cipher in ECB mode 52
 - LockBox Quick Cipher in CBC mode 63
 - LockBox Quick Cipher in ECB mode 63
 - Rijndael cipher in CBC mode 82
 - Rijndael cipher in ECB mode 81
 - RSA algorithm 90
 - triple-DES cipher in CBC mode 100
 - triple-DES cipher in ECB mode 100
- encrypting buffers of data using
 - 32-bit Random Number cipher 67
 - 64-bit Random Number cipher 67
 - LockBox Stream Cipher 71
- encrypting files using
 - 32-bit Random Number cipher 68
 - 64-bit Random Number cipher 69
 - Blowfish cipher in CBC mode 35
 - Blowfish cipher in ECB mode 35
 - DES cipher in CBC mode 44
 - DES cipher in ECB mode 44
 - LockBox Block Cipher in CBC mode 55
 - LockBox Block Cipher in ECB mode 54
 - LockBox Quick Cipher in CBC mode 64
 - LockBox Quick Cipher in ECB mode 64
 - LockBox Stream Cipher 72
 - Rijndael cipher in CBC mode 84
 - Rijndael cipher in ECB mode 84
 - RSA algorithm 91
 - triple-DES cipher in CBC mode 102
 - triple-DES cipher in ECB mode 102
- encrypting streams using
 - Blowfish cipher in CBC mode 36
 - Blowfish cipher in ECB mode 36
 - DES cipher in CBC mode 45
 - DES cipher in ECB mode 45
 - LockBox Block Cipher in CBC mode 56
 - LockBox Block Cipher in ECB mode 55
 - LockBox Quick Cipher in CBC mode 65
 - LockBox Quick Cipher in ECB mode 65
 - Rijndael cipher in CBC mode 85
 - Rijndael cipher in ECB mode 85
 - RSA algorithm 91
 - triple-DES cipher in CBC mode 103
 - triple-DES cipher in ECB mode 103
- encrypting strings using
 - Blowfish cipher in CBC mode 37
 - Blowfish cipher in ECB mode 37
 - DES cipher in CBC mode 46
 - DES cipher in ECB mode 46
 - Rijndael cipher in CBC mode 86
 - Rijndael cipher in ECB mode 86
 - RSA algorithm 92
 - triple-DES cipher in CBC mode 104
 - triple-DES cipher in ECB mode 104
- encryption
 - defined 16
 - public key 128
 - returning key 118, 120, 122, 124
 - setting the key 118, 120, 122, 125
- encryption mode, defining 115
- exponent, returning for asymmetric key 158

F

- feedback 20
- file
 - decrypting 110
 - encrypting 112
 - hashing 136
 - signing 149

verifying digital signature 151

G

generating key from
 passphrase 115
 random number generator 116
 generating new private key pair 127
 generating new public key pair 127

H

hardware requirements 4
 hash
 buffer 136
 file 136
 generating 50, 61
 generating buffer 94
 generating secure 60, 74
 generating string using Mix128 77
 generating using MD5 75
 generating using Mix128 77
 generating using SHA-1 95
 returning 60
 stream 136
 string 137
 hashing
 file 136
 stream 136
 string 137
 help 8

I

inheritance 11
 initialization vector 20
 initializing a context for DES cipher 48
 initializing context 167, 170, 173, 176, 179
 initializing context for use with
 32-bit Random Number cipher 68

64-bit Random Number cipher 68
 LockBox Stream Cipher 71
 Rijndael cipher 83
 triple-DES cipher 101
 initializing context structure 60
 initializing context used to process a stream
 164

K

key
 changing 165, 168, 171, 174, 177, 180
 defining size 124
 duplicating asymmetric 158
 generating 58, 115
 generating 128-bit using MD5 message
 digest 58
 generating from random number
 generator 116
 generating using LockBox Message Digest
 58
 generating using random number
 generator 58
 initializing context 164
 processing stream 164
 reducing number of bits 48
 replacing 164, 167, 170, 173, 176, 179
 resetting stream position 165
 returning 118, 120, 122, 124
 returning private 148
 returning public 148
 returning value from a stream 158
 saving value to a stream 159
 setting 118, 120, 122, 125
 key pair
 calculating private 148
 calculating public 148
 creating private 91
 creating public 91
 generating 127

L

- LbClass unit 161
- Leiserson, Charles E. 14
- LMD key 58
- LockBox Block Cipher 52, 53, 54, 55, 56, 64, 65, 68, 69, 72
- LockBox functionality 2
- LockBox Message Digest
 - completing the generation 60
 - generating a hash 61
 - generating a message digest 61
 - generating a secure hash of a buffer 60
 - generating key 58
 - initializing the context structure 60
 - returning the hash 60
 - updating with hashed form of data 61
- LockBox Quick Cipher 63
- LockBox Stream Cipher 71
- low-level routines 31

M

- MD5
 - defined 22
 - generating key 58
 - generating secure hash of a buffer 74
 - initializing context structure 75
 - updating with hashed form of data 75
- MD5 hash, completing the generation 74
- MD5 key 58
- message digest
 - generating 61
 - generating using MD5 75
 - generating using SHA-1 95
 - retrieving 139, 141
- method 118
- Mix128
 - generating hash of a buffer 77
 - generating hash of a string 77
- modulus, returning for asymmetric key 158

N

- naming conventions 8

O

- output buffer, calculating size 113

P

- passphrase
 - defined 25
 - generating key from 115
- plaintext
 - calculating size of buffer 113
 - defined 16
 - encrypting block 90
 - encrypting buffer 111
 - encrypting file 112
 - encrypting stream 112
 - encrypting string 112
- Price, W. L. 14
- private key
 - defined 17
 - returning 127
- private key pair, generating 127
- processing stream 167, 170, 173, 176, 179
- public key
 - defined 17
 - returning 128
- public key pair, generating 127

R

- random number generator, generating key 58, 116
- RDLEncryptStringCBC 86
- reading bytes from stream 164, 167, 170, 173, 176, 179
- replacing key 164, 167, 170, 173, 176, 179
- requirements, hardware and software 4

- resetting stream position 165, 168, 171, 174, 177, 180
- retrieving message digest 139, 141
- returning key for decryption 120, 122
- returning key for encryption 120, 122
- returning key used for decryption 118
- returning key used for encryption 118
- returning the key for decryption 124
- returning the key for encryption 124
- Rijndael decryption algorithm 17
- Rijndael key, defining size 124
- Rivest, Ronald L. 14
- RSA algorithm
 - decrypting ciphertext block 90
 - defined 17
 - encrypting files 91
 - encrypting plaintext block 90
 - encrypting streams 91
 - encrypting strings 92

S

- Schneier, Bruce 14
- self-delegation 13
- sequence diagram 13
- setting the key for decryption 118, 120, 122, 125
- setting the key for encryption 118, 120, 122, 125
- SHA-1
 - completing the generation of 94
 - defined 22
 - generating a hash 95
 - generating a message digest 95
 - generating secure hash of a buffer 94
 - initializing context structure 95
 - updating context with hashed form of data 96
- signature scheme 143
- Software requirements 4
- software requirements 4

- Stinson, D. R. 14
- stream
 - calculating digital signature 150
 - creating 164, 167, 170, 173, 176, 179
 - decrypting 110
 - encrypting 91, 112
 - hashing 136
 - initializing context 164, 167
 - processing 167, 170, 173, 176, 179
 - processing data 164, 167, 170, 173, 176, 179
 - reading bytes from 164, 167, 170, 173, 176, 179
 - resetting 168, 171, 174, 177, 180
 - returning a key value 158
 - saving a key value 159
 - signing 150
 - verifying digital signature 152
 - writing bytes to 165

string

- calculating digital signature 150
- decrypting 111
- encrypting 92, 112
- generating hash 77
- hashing 137
- signing 150
- verifying digital signature 152

T

- Triple Data Encryption Standard cipher 100, 102, 103
- Triple Defense Encryption Standard cipher 103
- triple-DES 17

U

- UML 10
- Unified Modeling Language 10

V

version, showing current 108
visibility characters 11

W

writing bytes to stream 165, 168, 171, 174,
177, 180



The TurboPower family of tools—
Winners of 6 *Delphi Informant* Readers' Choice Awards for 2000!

For over fifteen years you've depended on TurboPower to provide the best tools and libraries for your development tasks. Now try FlashFiler 2 and Internet Professional—two of TurboPower's best selling products—risk free. Both are fully compatible with Delphi and C++Builder, and are backed with our expert support and 60-day money back guarantee.

DEVELOP, DEBUG, OPTIMIZE
FROM START TO FINISH, TURBOPOWER
HELPS YOU **BUILD YOUR BEST**

FLASHFILER 2™

FlashFiler 2 is the high performance database engine that's ideal for your next project. It's amazingly easy to use, easy to license, and easy to deploy no matter how big your development project is. And if you're looking for a database with advanced features that will grow with you, there's no better choice than new FlashFiler 2.

INTERNET PROFESSIONAL™

Now you don't need to be a guru to create an Internet powerhouse. Internet Professional's rich mix of components support all popular protocols—they'll make your life easier and your apps more professional. Components like our elegant VCL-only HTML viewer, our telnet terminal with VT100 emulation, or an FTP directory viewer that navigates sites with the click of a mouse.



Try the full range of
TurboPower products.
Download free Trial-Run Editions
from our Web site.

www.turbopower.com

LockBox 2 requires Microsoft Windows 9X, Me, NT or 2000 or Linux operating systems,
and Borland Delphi 3 and above, C++Builder 3 and above, or Borland Kylix

TurboPower Software Company
©2000, TurboPower Software Co.



TURBOPOWER®
Software Company