cryptlib

Security Toolkit

Version 3.0 *beta 2*

May 2000

INTRODUCTION 1

cryptlib Overview 1

cryptlib features 2

Programming Interface 2
Standards Compliance 2
Y2K Compliance 3
Encrypted Object Management 3
S/MIME 3
Certificate Management 4
Key Database Interface 4
Smart Card Support 5
Security Features 5
Performance 6
Cryptographic Random Number Management 6
Configuration Options 7

Document conventions 7

Recommended Reading 7

INSTALLATION 8

Installing cryptlib for Windows 3.x 8

Installing cryptlib for Windows'95/98 and Windows NT 8

Installing from Source Code 8

BeOS 8
DOS 8
DOS32 8
Macintosh 8
MVS 8
OS2 9
VM/CMS 9
Windows 3.x 9
Windows'95/98 and Windows NT 10
Unix 10
Other systems 11
Key Database Setup 12
Certificate Installation 12
Cut-down cryptlib Versions 13
Support for Vendor-specific Algorithms 13

CRYPTLIB BASICS 14

Programming interfaces 14

Container object interface 15
Mid-level interface 15
Action object interface 15

Objects and Interfaces 15

Objects and Attributes 16

Interfacing with cryptlib 16

Initialisation 16

Working with Object Attributes 17

Attribute Types 18

Object Security 19

Interaction with External Events 21

ENVELOPING CONCEPTS 22

Creating/Destroying Envelopes 22

The Data Enveloping Process 23

Data Size Considerations 24

Basic Data Enveloping 25
Compressed Data Enveloping 26
Password-based Encryption Enveloping 27
De-enveloping Mixed Data 27
Enveloping Large Data Quantities 28
Alternative Processing Techniques 29
Enveloping with Many Enveloping Attributes 30
ADVANCED ENVELOPING 32
Public-Key Encrypted Enveloping 32
Digitally Signed Enveloping 35
Enveloping with Multiple Attributes 36
Envelope Attribute Cursor Management 36
Processing Multiple Deenveloping Attributes 37
Nested Envelopes 39
KEY DATABASES 40
Creating/Destroying Keyset Objects 40
File Keysets 41
HTTP Keysets 42
LDAP Keysets 42
Relational Database Keysets 43
Smart Card Keysets 45
Extended Keyset Initialisation 46
LDAP Keysets 46
Relational Database Keysets 46
Smart Card Keysets 47
Accessing a Keyset 49
Reading a Key from a Keyset 50
Obtaining a Key for a User 50
General Keyset Queries 52
Handling Multiple Certificates with the Same Name 54
Writing a Key to a Keyset 54
Deleting a Key 55
ENCRYPTION AND DECRYPTION 56
Creating/Destroying Encryption Contexts 56
Generating a Key into an Encryption Context 57
Public/Private Key Generation 58
Deriving a Key into an Encryption Context 59
Loading Keys into Encryption Contexts 60
Loading Initialisation Vectors 61
Working with Public/Private Keys 61
Loading Multibyte Integers 62
Querying Encryption Contexts 63
ENCRYPTING/DECRYPTING DATA 64
Using Encryption Contexts to Encrypt/Decrypt Data 64
EXCHANGING KEYS 66
Exporting a Key 66
Exporting using Conventional Encryption 67
Importing a Key 68
Importing using Conventional Encryption 69
Querying an Exported Key Object 69
Extended Key Export/Import 70

Key Agreement **70**

SIGNING DATA 72

Querying a Signature Object **73**

Extended Signature Creation/Checking **74**

CERTIFICATE MANAGEMENT 76

Overview of Certificates **76**
Certificates and Standards Compliance 76

The Certification Process **77**

Creating/Destroying Certificate Objects **78**

Working with Certificate Attributes **79**

Certificate Structures **79**
Attribute Certificate Structure 79
Certificate Structure 81
Certification Request Structure 83
CRL Structure 83

Basic Certificate Management **84**

Certificate Identification Information **86**
DN Structure for Business Use 87
DN Structure for Private Use 88
Other DN Structures 88
Working with Distinguished Names 88

Extended Certificate Identification Information **88**
Working with GeneralNames 90
Certificate Fingerprints 90

Importing/Exporting Certificates **90**

Signing/Verifying Certificates **93**

Certificate Trust Management **95**
Working with Trust Settings 95

Certificate Errors **96**

CERTIFICATE EXTENSIONS 97

Extension Structure **97**

Working with Extension Attributes **98**
Extension Cursor Management 98
Composite Extension Attributes 100

X.509 Extensions **101**
Alternative Names 101
Basic Constraints 101
Certificate Policies, Policy Mappings, and Policy Constraints 102
CRL Distribution Points and Authority Information Access 103
Directory Attributes 103
Key Usage, Extended Key Usage, and Netscape cert-type 103
Name Constraints 106
Private Key Usage Period 107
Subject and Authority Key Identifiers 107

CRL Extensions **108**
CRL Reasons, CRL Numbers, Delta CRL Indicators 108
Hold Instruction Code 109
Invalidity Date 109
Issuing Distribution Point and Certificate Issuer 109

Digital Signature Legislation Extensions **110**
Certificate Generation Date 110
Other Restrictions 110
Reliance Limit 111

Signature Delegation 111
SET Extensions 111
SET Card Required and Merchant Data 111
SET Certificate Type, Hashed Root Key, and Tunneling 112
Vendor-specific Extensions 112
Netscape Certificate Extensions 112
Thawte Certificate Extensions 113
MAINTAINING KEYS AND CERTIFICATES 114
Updating a Private Key with Certificate Information 114
Changing a Private Key Password 115
The Certification Process 115
Certificate Chains 117
Working with Certificate Chains 117
Signing Certificate Chains 118
Checking Certificate Chains 119
Exporting Certificate Chains 120
Certificate Revocation Lists 120
Working with CRL's 121
Creating CRL's 121
Advanced CRL Creation 122
Checking Certificates against CRL's 122
Automated CRL Checking 123
FURTHER CERTIFICATE OBJECTS 124
Certificate-like Object Structure 124
CMS Attributes 124
CMS Attributes 124
Content Type 124
Countersignature 125
MAC Value 125
Message Digest 126
Signing Time 126
Extended CMS Attributes 126
AuthentiCode Attributes 126
Content Hints 127
Mail List Expansion History 128
Receipt Request 128
Security Label, Equivalent Label 128
S/MIME Capabilities 129
Signing Certificate 130
S/MIME 131
S/MIME Enveloping 131
Encrypted Enveloping 132
Digitally Signed Enveloping 132
Detached Signatures 133
Extra Signature Information 135
From Envelopes to S/MIME 136
S/MIME Content Types 136
Data 136
Signed Data 136
Detached Signature 137
Encrypted Data 137
Nested Content 137
Implementing S/MIME using cryptlib 138
Example: c-client/IMAP4 138

Example: Eudora 138
Example: MAPI 138
Example: Windows'95/98 and NT Shell 139

ENCRYPTION DEVICES AND MODULES 140

Creating/Destroying Device Objects 140

Activating and Controlling Cryptographic Devices 141
Initialise Device 141
User Authentication 141
Zeroise Device 142

Extended Device Control Functions 142
Setting/Changing User Authentication Values 142

Working with Device Objects 142
Key Storage in Crypto Devices 143
Considerations when Working with Devices 143

PKCS #11 Devices 144
Installing New PKCS #11 Modules 144
PKCS #11 Functions used by cryptlib 145

MISCELLANEOUS TOPICS 146

Querying cryptlib's Capabilities 146

Working with Configuration Options 146
Querying/Setting Configuration Options 149
Saving Configuration Options 150

Obtaining Information About Cryptlib 151

Random Numbers 151
Gathering Random Information 151
Obtaining Random Numbers 152
Random Information Gathering Techniques 152
Hardware Random Number Generation 156

Working with Newer Versions of cryptlib 156

ERROR HANDLING 158

Extended Error Reporting 160

ALGORITHMS AND MODES 163

Blowfish 163

CAST-128 163

DES 163

Triple DES 163

Diffie-Hellman 163

DSA 163

ElGamal 164

HMAC-MD5 164

HMAC-SHA1 164

HMAC-RIPEMD-160 164

IDEA 164

MD2 165

MD4 165

MD5 165

MDC2 165

RC2 165

RC4 166

RC5 166

RIPEMD-160 166
RSA 166
SAFER 166
SAFER-SK 166
SHA 166
Skipjack 166

DATA TYPES AND CONSTANTS 167

CRYPT_ALGO 167
CRYPT_ATTRIBUTE_TYPE 168
CRYPT_CERTFORMAT_TYPE 168
CRYPT_CERTTYPE_TYPE 168
CRYPT_DEVICE_TYPE 169
CRYPT_FORMAT_TYPE 169
CRYPT_KEYID_TYPE 169
CRYPT_KEYOPT 170
CRYPT_KEYSET_TYPE 170
CRYPT_MODE 170
CRYPT_OBJECT_TYPE 171
Data Size Constants 171
Miscellaneous Constants 172

DATA STRUCTURES 173

CRYPT_OBJECT_INFO Structure 173
CRYPT_PKCINFO Structures 173
CRYPT_QUERY_INFO Structure 174

FUNCTION REFERENCE 175

cryptAddCertExtension 175
cryptAddPrivateKey 175
cryptAddPublicKey 175
cryptAddRandom 176
cryptAsyncCancel 176
cryptAsyncQuery 176
cryptCheckCert 176
cryptCheckSignature 177
cryptCheckSignatureEx 177
cryptCreateCert 178
cryptCreateContext 178
cryptCreateEnvelope 178
cryptCreateSignature 179
cryptCreateSignatureEx 179
cryptDecrypt 180
cryptDeleteAttribute 180
cryptDeleteCertExtension 180
cryptDeleteKey 181
cryptDestroyCert 181
cryptDestroyContext 181
cryptDestroyEnvelope 182
cryptDestroyObject 182

cryptDeviceClose **182**
cryptDeviceControlEx **182**
cryptDeviceCreateContext **183**
cryptDeviceOpen **183**
cryptEncrypt **183**
cryptEnd **184**
cryptExportCert **184**
cryptExportKey **184**
cryptExportKeyEx **185**
cryptGenerateKey **186**
cryptGenerateKeyAsync **186**
cryptGenerateKeyAsyncEx **186**
cryptGenerateKeyEx **187**
cryptGetAttribute **187**
cryptGetAttributeString **188**
cryptGetCertExtension **188**
cryptGetPrivateKey **189**
cryptGetPublicKey **189**
cryptImportCert **190**
cryptImportKey **190**
cryptImportKeyEx **190**
cryptInit **191**
cryptInitEx **191**
cryptKeysetClose **191**
cryptKeysetOpen **191**
cryptKeysetOpenEx **192**
cryptPopData **192**
cryptPushData **193**
cryptQueryCapability **193**
cryptQueryDeviceCapability **194**
cryptQueryObject **194**
cryptSetAttribute **194**
cryptSetAttributeString **195**
cryptSignCert **195**

STANDARDS CONFORMANCE 196

Blowfish 196
CAST-128 196
DES 196
Triple DES 197
Diffie-Hellman 197
DSA 197
Elgamal 197
HMAC-MD5 198
HMAC-SHA1 198
IDEA 198
MD2 198
MD4 198
MD5 199
MDC-2 199
RC2 199
RC4 199

RC5 199
RIPEMD-160 200
RSA 200
SHA/SHA1 200
Safer/Safer-SK 200
Skipjack 201
Certificates 201
Data Structures 201
S/MIME 201
Y2K Compliance 202
General 202

ACKNOWLEDGEMENTS 203

Introduction

The information age has seen the development of electronic pathways which carry vast amounts of valuable commercial, scientific, and educational information between financial institutions, companies, individuals, and government organisations. Unfortunately the unprecedented levels of access provided by systems like the Internet also expose this data to breaches of confidentiality, disruption of service, and outright theft. As a result, there is an enormous (and still growing) demand for the means to secure these online transactions. One report by the Computer Systems Policy Project (a consortium of virtually every large US computer company, including Apple, AT&T, Compaq, Digital, IBM, Silicon Graphics, Sun, and Unisys) estimates that the potential revenue arising from these security requirements in the US alone could be as much as US$30-60 billion by the year 2000, and the potential exposure to global users from a lack of this security is projected to reach between US$320 and 640 billion by the year 2000.

Unfortunately the security systems required to protect data are generally extremely difficult to design and implement, and even when available tend to require considerable understanding of the underlying principles in order to be used. This has lead to a proliferation of "snake oil" products which offer only illusionary security, or to organisations holding back from deploying online information systems because the means to secure them aren't readily available, or (in the case of US products) because they employ weak, easily broken security which is unacceptable to users.

The cryptlib security toolkit is one answer to this problem. A complete description of the capabilities provided by cryptlib is given below.

cryptlib Overview

cryptlib is a powerful security toolkit which allows even inexperienced crypto programmers to easily add encryption and authentication security services to their software. The high-level interface provides anyone with the ability to add strong encryption and authentication capabilities to an application in as little as half an hour, without needing to know any of the low-level details which make the encryption or authentication work. Because of this, cryptlib dramatically reduces the cost involved in adding security to new or existing applications.

cryptlib provides a transparent and consistent interface to a number of widely-used security services and algorithms which are accessed through a straightforward, standardised interface with parameters such as the algorithm and key size being selectable by the user. Included as core components are implementations of the most popular encryption and authentication algorithms, Blowfish, CAST, DES, triple DES, IDEA, RC2, RC4, RC5, Safer, Safer-SK, and Skipjack conventional encryption, MD2, MD4, MD5, MDC-2, RIPEMD-160 and SHA hash algorithms, HMAC-MD5, HMAC-SHA, and HMAC-RIPEMD-160 MAC algorithms, and Diffie-Hellman, DSA, ElGamal, and RSA public-key encryption, with elliptic-curve encryption currently under development. The algorithm parameters are summarised below:

Algorithm	Key size	Block size
Blowfish	448	64
CAST-128	128	64
DES	56	64
Triple DES	112 / 168	64
IDEA	128	64
RC2	1024	64
RC4	2048	8
RC5	832	64
Safer	128	64
Safer-SK	128	64
Skipjack	80	64
MD2	—	128

Algorithm	Key size	Block size
MD4	—	128
MD5	—	128
MDC-2	—	128
RIPEMD-160	—	160
SHA	—	160
HMAC-MD5	128	128
HMAC-SHA	160	160
HMAC-RIPEMD-160	160	160
Diffie-Hellman	4096	—
DSA	4096[1]	—
ElGamal	4096	—
RSA	4096	—

Unlike similar products sourced from the US, cryptlib contains no deliberately weakened encryption or backdoors, and allows worldwide use of keys of up to 4096 bits. In contrast products originating from the US contain either extremely weak encryption with keys a mere 40 bits in length (sometimes referred to as "8-cent keys" in reference to the cost of breaking one key), or, if they use longer keys, are required to contain backdoors which allow easy access by the US government (and, by extension, US business interests) to all data "protected" by the encryption. This makes US products unsuited for protecting sensitive, confidential data, and gives cryptlib an automatic advantage over all US products.

cryptlib features

cryptlib provides a standardised interface to a number of popular encryption algorithms, as well as providing a high-level interface which hides most of the implementation details and provides an operating-system-independent encoding method which makes it easy to transfer secured data from one operating environment to another. Although use of the high-level interface is recommended, experienced programmers can directly access the lower-level encryption routines for implementing custom encryption protocols or methods not directly provided by cryptlib.

Programming Interface

The application programming interface (API) serves as an interface to a range of plug-in encryption modules which allow encryption algorithms to be added in a fairly transparent manner, so that adding a new algorithm or replacing an existing software implementation with custom encryption hardware can be done without any trouble. The standardised API allows any of the algorithms and modes supported by cryptlib to be used with a minimum of coding effort. In addition the easy-to-use high-level routines allow for the exchange of encrypted session keys and data and the creation and checking of digital signatures with a minimum of programming overhead.

cryptlib has been written to be as foolproof as possible. On initialization it performs extensive self-testing against test data from encryption standards documents, and the API's check each parameter and function call for errors before any actions are performed, with error reporting down to the level of individual parameters. In addition logical errors such as, for example, a key exchange function being called in the wrong sequence, are checked for and identified.

Standards Compliance

All algorithms, security methods, and data encoding systems in cryptlib either comply with one or more national and international banking and security standards, or are implemented and tested to conform to a reference implementation of a particular

[1] The DSA standard only defines key sizes from 512 to 1024 bits, cryptlib supports longer keys but there is no extra security to be gained from using these keys.

algorithm or security system. Compliance with national and international security standards is automatically provided when cryptlib is integrated into an application. These standards include ANSI X3.92, ANSI X3.106, ANSI X9.9, ANSI X9.17, ANSI X9.30-1, ANSI X9.30-2, ANSI X9.31-1, ANSI X9.42, ANSI X9.52, FIPS PUB 46-2, FIPS PUB 46-3, FIPS PUB 74, FIPS PUB 81, FIPS PUB 113, FIPS PUB 180, FIPS PUB 180-1, FIPS PUB 186, ISO/IEC 8372, ISO/IEC 8731 ISO/IEC 8732, ISO/IEC 8824/ITU-T X.680, ISO/IEC 8825/ITU-T X.690, ISO/IEC 9797, ISO/IEC 10116, ISO/IEC 10118, PKCS #1, PKCS #3, PKCS #7, PKCS #9, PKCS #10, RFC 1319, RFC 1320, RFC 1321, RFC 1750, RFC 2040, RFC 2104, RFC 2144, RFC 2268, RFC 2312, RFC 2313, RFC 2314, RFC 2315, RFC 2459, RFC 2528, RFC 2585, RFC 2630, RFC 2631, RFC 2632, and RFC 2634. Because of the use of internationally recognised and standardised security algorithms, cryptlib users will avoid the problems caused by homegrown, proprietary algorithms and security techniques which often fail to provide any protection against attackers, resulting in embarrassing bad publicity and expensive software recalls.

Y2K Compliance

cryptlib handles all date information using the ANSI/ISO C time format which does not suffer from Y2K problems. Although earlier versions of the X.509 certificate format do have Y2K problems, cryptlib transparently converts the date encoded in certificates to and from the ANSI/ISO format, so cryptlib users will never see this. cryptlib's own time/date format is not affected by any Y2K problems, and cryptlib itself conforms to the requirements in the British Standards Institutions DISC PD2000-1:1998 Y2K compliance standard.

Encrypted Object Management

cryptlibs powerful object management interface provides the ability to add encryption and authentication capabilities to an application without needing to know all the low-level details which make the encryption or authentication work. The automatic object-management routines take care of encoding issues and cross-platform portability problems, so that a single function call is enough to export a public-key encrypted session key with all the associated information and parameters needed to recreate the session key on the other side of a communications channel, or to generate a digital signature on a piece of data. This provides a considerable advantage over other encryption toolkits which often require hundreds of lines of code and the manipulation of complex encryption data structures to perform the same task.

S/MIME

cryptlib employs the IETF-standardised Cryptographic Message Syntax (CMS, formerly called PKCS #7) format as its native data format. CMS is the underlying format used in the S/MIME secure mail standard, as well as a number of other standards covering secure EDI and related systems like HL7 messaging. As an example of its use in secure EDI, cryptlib provides security services for the Sypmhonia EDI messaging toolkit which is used to communicate medical lab reports, patient data, drug prescription information, and similar information requiring a high level of security.

The S/MIME implementation uses cryptlibs enveloping interface which allows simple, rapid integration of strong encryption and authentication capabilities into existing email agents and messaging software. The resulting signed data format provides message integrity and origin authentication services, the enveloped data format provides confidentiality. The complexity of the S/MIME format means that the few other toolkits which are available require a high level of programmer knowledge of S/MIME processing issues. In contrast cryptlib's enveloping interface makes the process as simple as pushing raw data into an envelope and popping the processed data back out, a total of three function calls, plus one more call to add the appropriate encryption or signature key.

Certificate Management

cryptlib implements full X.509 certificate support, including all X.509 version 3 extensions as well as extensions defined in the IETF PKIX certificate profile. cryptlib also supports additional certificate types and extensions including SET certificates, Microsoft AuthentiCode and Netscape and Microsoft server-gated crypto certificates, S/MIME and SSL client and server certificates, and various vendor-specific extensions such as Netscape certificate types and the Thawte secure extranet.

In addition to certificate handling, cryptlib allows the generation of PKCS #10 certification requests with CMMF extensions suitable for submission to certification authorities (CA's) in order to obtain a certificate. Since cryptlib is itself capable of processing certification requests into certificates, it is also possible to use cryptlib to provide full CA services. cryptlib also supports the creating and handling of the certificate chains required for S/MIME, SSL, and other applications, and the creation of certificate revocation lists (CRL's) with the capability to check certificates against existing or new CRL's either automatically or under programmer control.

cryptlib can import and export certification requests, certificates, and CRL's in straight binary format, as PKCS #7 certificate chains, and as Netscape certificate sequences, with or without base64 armouring. This covers the majority of certificate and certificate transport formats used by a wide variety of software such as web browsers and servers.

The certificate types which are supported include:

- Basic X.509 version 1 certificates
- Extended X.509 version 3 certificates
- SSL server and client certificates
- S/MIME email certificates
- SET certificiates
- AuthentiCode code signing certificates
- IPSEC server, client, end-user, and tunneling certificates
- Server-gated crypto certificates
- Timestamping certificates

In addition cryptlib supports all X.509v3, IETF, S/MIME, and SET certificate extensions and a many vendor-specific extensions including ones covering public and private key usage, certificate policies, path and name constraints, policy constraints and mappings, and alternative names and other identifiers. This comprehensive coverage makes cryptlib a single solution for almost all certificate processing requirements.

To handle certificate trust and revocation issues, cryptlib includes a certificate trust manager which can be used to automatically manage CA trust settings, for example a CA can be designated as a trusted issuer which will allow cryptlib to automatically evaluate trust along certificate chains. Similarly, cryptlib can automatically check certificates against CRL's published by CA's, removing from the user the need to perform complex manual checking.

Key Database Interface

cryptlib utilizes commercial-strength RDBMS's to store keys in the internationally standardised X.509 format. The key database integrates seamlessly into existing databases and can be managed using existing tools. For example a key database stored on an MS SQL Server might be managed using Visual Basic or MS Access; a key database stored on an Oracle server might be managed through SQL*Plus. cryptlib currently supports mSQL, MySQL, Oracle, and Postgres databases under Unix, and most databases which can be accessed through Windows ODBC drivers.

This includes MS Access, dBase, Oracle, Paradox, MS SQL Server, and many more. Extending the interface to support new database types requires approximately 200 lines of code to tie the cryptlib routines into a particular database backend.

In addition to key databases, cryptlib supports the storage and retrieval of certificates in LDAP directories. This interface provides full LDAPv3 support, with optional SSL protection of the connection to the directory. cryptlib also supports HTTP access for keys accessible via the web, as well as external flat-file key collections such as PGP key rings. The key collections may be freely mixed (so for example a private key could be stored in a disk file, a PGP keyring or on a smart card with the corresponding X.509 public key certificate being stored in an Oracle or SQL Server database, an LDAP directory, or on the web).

Private keys may be stored on disk encrypted with an algorithm such as triple DES (selectable by the user), with the password processed using several hundred iterations of a hashing algorithm such as SHA-1 (also selectable by the user). Where the operating system supports it, cryptlib will apply system security features such as ACL's under Windows NT and file permissions under Unix to the private key file to further restrict access.

Smart Card Support

cryptlib allows private keys to be stored on a variety of smart cards accessed through a selection of smart card readers — use of cryptlib won't tie you to a single card or reader vendor. As an extra precaution, cryptlib encrypts all data written to the smart card so that even if the card is hacked, the data remains secure. Support for new smart card types and/or readers can be added on request.

Crypto Devices

In addition to its built-in capabilities, cryptlib can make use of the crypto capabilities of a variety of external crypto devices such as:

- Hardware crypto accelerators
- Fortezza cards
- PKCS #11 devices
- Crypto smart cards

cryptlib provides full crypto device management capabilities, allowing you to initialise and program a crypto device, generate or load keys into it, add certificates for the generated/loaded keys, update and change PINs, and perform other management functions. For Fortezza cards, cryptlib provides full certificate authority workstation (CAW) capabilities.

In addition, the crypto device interface provides a convenient general-purpose plug-in capability for adding new functionality which will be automatically used by cryptlib in its higher-level routines which handle key management, digital signatures, and message encryption.

Security Features

cryptlib is built around a security kernel with Orange Book B3-level security features to implement its security mechanisms. This kernel provides the interface between the outside world and the architecture's objects (intra-object security) and between the objects themselves (interobject security). The security kernel is the basis of the entire cryptlib architecture — all objects are accessed and controlled through it, and all object attributes are manipulated through it. The kernel is implemented as an interface layer which sits on top of the objects, monitoring all accesses and handling all protection functions.

Each cryptlib object is contained entirely within the security perimeter, so that data and control information can only flow in and out in a very tightly-controlled manner,

and that objects are isolated from each other within the perimeter by the security kernel. For example once keying information has been sent to an object, it can't be retrieved by the user except under tightly-controlled conditions (the only real case where this can occur is when an object's access control list (ACL) permits a short-term session key to be exported in encrypted form, or a private key to be stored in encrypted form to a permanent storage medium such as a smart card or disk). In general keying information isn't even visible to the user, since it is generated inside the object itself and never leaves the security perimeter. This design is ideally matched to hardware implementations which perform strict red/black separation, since sensitive information can never leave the hardware.

Associated with each object is a mandatory ACL which determines who can access a particular object and under which conditions the access is allowed. At a very coarse level, each object has a mandatory access control setting which determines whether it is externally visible or not (that is, whether it has a handle which is valid outside the security perimeter). Only externally visible objects can be (directly) manipulated by the user, with ACL enforcement being handled by the architectures security kernel.

Operating at a much finer level of control than the object ACL is the discretionary access control (DACL) mechanism through which only certain capabilities in an object may be enabled. For example once an encryption context is established, it can be restricted to only allow basic data encryption and decryption, but not encrypted session key export.

If the operating system supports it, all sensitive information used will be page-locked to ensure it is never swapped to disk from where it could be recovered using a disk editor. All memory corresponding to security-related data is managed by cryptlib and will be automatically sanitized and freed when cryptlib shuts down even if the calling program forgets to release the memory itself.

Where the operating system supports it, cryptlib will apply operating system security features to any objects it creates or manages. For example under Windows NT cryptlib private key files will be created with an access control list (ACL) which allows only the key owner access to the file; under Unix the file permissions will be set to achieve the same result.

Performance

cryptlib is re-entrant and completely thread-safe, allowing it to be used with multithreaded applications under Windows 95/98 and Windows NT, OS/2, and Unix systems which support threads. Because it is thread-safe, lengthy cryptlib operations can be run in the background if required while other processing is performed in the foreground. In addition cryptlib itself is multithreaded so that computationally intensive internal operations take place in the background without impacting the performance of the calling application.

Most of the core algorithms used in cryptlib have been implemented in assembly language in order to provide the maximum possible performance. These routines provide an unprecedented level of performance, in some cases running faster than expensive, specialised encryption hardware designed to perform the same task. This means cryptlib can be used for high-bandwidth applications such as video/audio encryption and online network and disk encryption without the need to resort to expensive, hard-to-get encryption hardware.

Cryptographic Random Number Management

cryptlib contains an internal secure random data management system which provides the cryptographically strong random data used to generate session keys and public/private keys, in public-key encryption operations, and in various other areas which require secure random data. The random data pool is updated with unpredictable process-specific information as well as system-wide data such as current disk I/O and paging statistics, network, SMB, LAN manager, and NFS traffic, packet filter statistics, multiprocessor statistics, process information, users, VM

statistics, process statistics, open files, inodes, terminals, vector processors, streams, and loaded code, objects in the global heap, loaded modules, running threads, process, and tasks, and an equally large number of system performance-related statistics covering virtually every aspect of the operation of the system.

The exact data collected depends on the hardware and operating system, but generally includes quite detailed operating statistics and information. In addition if a `/dev/random`-style randomness driver (which continually accumulates random data from the system) is available, cryptlib will use this is a source of randomness. Finally, cryptlib supports a number of cryptographically strong hardware random number generators such as the Protego SG100 and various serial-port-based generators which can be used to supplement or replace the internal generator. This level of secure random number management ensures that security problems such as those present in Netscape's web browser (which allowed encryption keys to be predicted without breaking the encryption because the random data gathered wasn't at all random) can't occur with cryptlib.

Configuration Options

cryptlib works with a configuration database which can be used to tune its operation for different environments using the Windows registry or Unix `rc` files. This allows a system administrator to set a consistent security policy (for example mandating the use of 1024-bit public keys on a company-wide basis instead of the insecure 512-bit keys used in most US-sourced products). These configuration options are then automatically applied by cryptlib to operations such as key generation and data encryption and signing, although they can be overridden on a per-application or per-user basis if required.

Document conventions

This manual uses the following document conventions:

Example	Description
capi.h	This font is used for filenames.
cryptCreateContext	Bold type indicates cryptlib function names.
value	Words or portions of words in italics indicate placeholders for information which you need to supply.
`if( i == 0 )`	This font is used for sample code and operating system commands.

Recommended Reading

One of the best books to help you understand how to use cryptlib is *Network Security* by Charlie Kaufman, Radia Perlman, and Mike Speciner, which covers general security principles, encryption techniques, and a number of potential cryptlib applications such as X.400/X.500 security, PEM/S/MIME/PGP, Kerberos, and various other security, authentication, and encryption techniques. The book also contains a wealth of practical advice for anyone considering implementing a cryptographic security system.

A tutorial in 8 parts totalling over 500 slides covering all aspects of encryption and general network security, including encryption and security basics, algorithms, key management and certificates, CA's, certificate profiles and policies, PEM, PGP, S/MIME, SSL, ssh, SET, smart cards, and a wide variety of related topics, is available through `http://www.cs.auckland.ac.nz/~pgut001/`.

In addition to this, there are a number of excellent books available which will help you in understanding the operating principles behind cryptlib. The foremost of these are *Applied Cryptography* by Bruce Schneier and the *Handbook of Applied*

Cryptography by Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Applied Cryptography* provides an easy-to-read overview while the *Handbook of Applied Cryptography* provides extremely comprehensive, in-depth coverage of the field.

For general coverage of computer security issues, *Security in Computing* by Charles Pfleeger provides a good overview of security, access control, and secure operating systems and databases, and also goes into a number of other areas such as ethical issues which aren't covered in most books on computer security.

Installation

This chapter describes how to install cryptlib for a variety of operating systems.

Installing cryptlib for Windows

[Windows install procedure]

Installing from Source Code

If the precompiled version of cryptlib isn't available or if you have source code access, you can also install cryptlib from the source code, although this is somewhat more work than using the precompiled version. Instructions on installing cryptlib from source code are given below. Note that the versions which run on mainframe systems aren't kept as current as the more mainstream versions, and some will require adaptation to match the configuration of a particular system. See the entry for the particular mainframe operating system below for more information.

BeOS

The BeOS version of cryptlib can be built using a procedure which is mostly identical to that given further down for Unix. The BeOS version uses the Unix makefile, to change it for use with BeOS uncomment the marked lines at the start of the file.

DOS

The 16-bit DOS version of cryptlib can be built from the same files as the 16-bit Windows version, so no separate makefile is provided. The resulting library is about 500K in size, and any attempt to use any high-level routines which require random data will fail with a CRYPT_ERROR_RANDOM error code unless a `/dev/random`-style driver is available because there isn't any way to reliably obtain random data under DOS. Using cryptlib under 16-bit DOS is possible, but not recommended.

DOS32

The 32-bit DOS version of cryptlib can be built using the supplied makefile, which requires the djgpp compiler. The DOS32 version of cryptlib uses the same 32-bit assembly language code used by the Win32 and 80x86 Unix versions, so it runs significantly faster than the 16-bit DOS version. Like the 16-bit DOS version, any attempt to use the high-level key export routines will fail with a CRYPT_ERROR_-RANDOM error code unless a `/dev/random`-style driver is available because there isn't any way to reliably obtain random data under DOS.

Macintosh

The Macintosh version of cryptlib can be built using the Metroworks Codewarrior and the project file Mac_MWCW_Project.sit, which will build cryptlib as a shared library. After you've built cryptlib, you should run the self-test program to make sure everything is working OK.

MVS

The MVS version of cryptlib can be built using the standard C/370 compiler and accompanying tools. The supplied JCL file MVSBUILD JOB can be used as a basis for building cryptlib under MVS. Since MVS sites typically have significantly different system configurations, this file and possibly portions of the source code will require some tuning in order to adjust it to suit the build process normally used at your site. After you've built cryptlib, you should run the self-test program to make sure everything is working OK.

OS2

The OS/2 version of cryptlib can be built using the command-line version of the IBM compiler. The supplied makefile will build the DLL version of cryptlib, and can also build the cryptlib self-test program, which is a console application. You should run the self-test program after you've built cryptlib to make sure everything is working OK.

If you are working with the IBM OS/2 compiler you should set enumerated types to always be 32-bit values because the compiler by default uses variable-length types depending on the enum range (so one enum could be an 8-bit type and another 32). cryptlib is immune to this "feature", and function calls from your code to cryptlib should also be unaffected because of type promotion to 32-bit integers, but the variable-range enums may cause problems in your code if you try to work with them under the assumption that they have a fixed type.

VM/CMS

The VM/CMS version of cryptlib can be built using the standard C/370 compiler and accompanying tools. The supplied EXEC2 file VMBUILD EXEC will build cryptlib as a TXTLIB and then build the self-test program as an executable MODULE file. Since VM sites typically have different system configurations, this file and possibly portions of the source code may require tuning in order to adjust it to suit the build process normally used at your site. You should run the self-test program after you've built cryptlib to make sure everything is working OK.

Windows 3.x

The 16-bit cryptlib DLL can be built using the cl16.mak makefile, which is for version 1.5x of the Visual C++ compiler. To use the files you should create a network share pointing at the cryptlib root directory and then connect to the share and work from that. This makes the makefile paths independent of the directory you put the files in. For example you might want to share the directory as CRYPT, so you could connect to it as \\machine\CRYPT and access the project file as \\machine\CRYPT\CL16.MAK.

The mixed C/assembly language encryption and hashing code will give a number of warnings, the remaining code should compile without warnings. Once the DLL has been built, test.mak will build the cryptlib self-test program, which is a console application. You should run this after you've built cryptlib to make sure everything is working OK.

If you will be accessing certificates stored on web pages, you need to install the required HTTP client software on your system. The cryplib package includes the TCP4U HTTP DLL and will automatically detect and use this if present. If you don't enable the use of the HTTP interface, the self-test code will issue a warning that no HTTP interface is present and continue without testing this interface

If you will be using special encryption hardware or an external encryption device such as a PCMCIA card or smart card, you need to install the required device drivers on your system, and if you're using a generic PKCS #11 device you need to configure the appropriate driver for it as described in "Encryption Devices and Modules" on page 140. cryptlib will automatically detect and use any devices which it recognises and which have drivers present. If you don't enable the use of a crypto device, the self-test code will issue a warning that no devices are present and continue without testing the crypto device interface.

The use of the 16-bit DLL on a Windows'95/98 system is not recommended, as the randomness-polling required by some of the high-level routines performs very poorly in the emulated 16-bit environment. Under Windows NT 4, the 16-bit DLL may cause a memory fault if a randomness poll is run because the polling process used relies on the presence of certain Windows VxD components which don't exist under NT.

Windows'95/98 and Windows NT

The 32-bit cryptlib DLL can be built using the crypt32.dsp and crypt32.dsw make/project files, which are for version 6 of the Visual C++ compiler. To use the files you should create a network share pointing at the cryptlib root directory and then connect to the share and work from that. This makes the makefile paths independent of the directory you put the files in. For example you might want to share the directory as CRYPT, so you could connect to it as \\machine\CRYPT and access the project file as \\machine\CRYPT\CRYPT32.DSW.

You may also need to rescan dependencies since Visual C++ handles makes in a somewhat broken manner. Once the DLL has been built, test32.dsp will build the cryptlib self-test program, which is a console application. You should run this after you've built cryptlib to make sure everything is working OK.

If you will be using an LDAP directory, you need to install the required LDAP client DLL on your system. The cryptlib package includes the Netscape LDAPv3 client DLL and will automatically detect and use this if present. If you don't enable the use of an LDAP directory interface, the self-test code will issue a warning that no LDAP directory is present and continue without testing the LDAP interface.

If you will be accessing certificates stored on web pages, you need to install the required HTTP client software on your system. The cryplib package includes the TCP4U HTTP DLL and will automatically detect and use this if present. If you don't enable the use of the HTTP interface, the self-test code will issue a warning that no HTTP interface is present and continue without testing this interface

If you will be using special encryption hardware or an external encryption device such as a PCMCIA card or smart card, you need to install the required device drivers on your system, and if you're using a generic PKCS #11 device you need to configure the appropriate driver for it as described in "Encryption Devices and Modules" on page 140. cryptlib will automatically detect and use any devices which it recognises and which have drivers present. If you don't enable the use of a crypto device, the self-test code will issue a warning that no devices are present and continue without testing the crypto device interface.

If you're using Borland C++ rather than Visual C++, you'll need to set up the .def and .lib files for use with the Borland compiler. To do this, run the following commands in the cryptlib directory:

```
impdef cl32 cl32
implib cl32 cl32.def
```

The first one will produce a Borland-specific .def file from the DLL, the second one will produce a Borland-specific .lib file from the DLL and .def file.

Unix

To unzip the code under Unix use the `-a` option to ensure that the text files are converted to the Unix format. The makefile by default will build the statically-linked library when you invoke it with `make`. To build the shared library, use `make shared`. Once cryptlib has been built, use `make testlib` to build the cryptlib self-test program. For example to build the shared library and self-test program, you would use `make shared; make testlib`. testlib will run fairly extensive self-tests of cryptlib and you should run this after you've built it to make sure everything is working OK. Depending on your system setup and privileges you may need to either copy the shared library to `/usr/lib` or set the `LD_LIBRARY_PATH` environment variable to make sure the shared library is used.

If you will be using a key database, you need to enable the use of the appropriate interface module for the database backend. To do this, you need to define one or more of `DBX_MSQL`, `DBX_MYSQL`, `DBX_ORACLE`, or `DBX_POSTGRES` (depending on the database or databases you're using) in the makefile before you build cryptlib. You can do this by adding the appropriate defines (for example `-DDBX_MSQL`) to the

CFLAGS or SCFLAGS setting at the start of the makefile, depending on whether you're building the static or shared library. In addition you also need to link the database library or libraries (for example libmsql.a) into your executable. For the cryptlib self-test code you can define the database libraries using the TESTLIBS setting at the start of the makefile. If you don't enable the use of a database interface, the self-test code will issue a warning that no key database is present and continue without testing the database interface.

If you will be using an LDAP directory, you need to install the required LDAP client library on your system, define DBX_LDAP before you build cryptlib in the manner described above for the database defines, and link the LDAP client library into your executable. cryptlib uses the Netscape LDAPv3 client, and will automatically detect the presence of the SSL or non-SSL version of the client depending on which one you link in. If you don't enable the use of an LDAP directory interface, the self-test code will issue a warning that no LDAP directory is present and continue without testing the LDAP interface.

If you will be accessing certificates stored on web pages, you need to install the required HTTP library on your system, define DBX_HTTP before you build cryptlib in the manner described above for the LDAP define, and link the HTTP client library into your executable. cryptlib uses the TCP4U client which can be obtained from a number of locations on the Internet. If you don't enable the use of a the HTTP interface, the self-test code will issue a warning that no HTTP interface is present and continue without testing this interface.

If you will be using special encryption hardware or an external encryption device such as a PCMCIA card or smart card, you need to install the required device drivers on your system and enable their use when you build cryptlib by linking in the required interface libraries. If you don't enable the use of a crypto device, the self-test code will issue a warning that no devices are present and continue without testing the crypto device interface.

For any common Unix system, cryptlib will build without any problems, but in some rare cases you may need to edit misc/rndunix.c and possibly keymgmt/stream.c if you're running an unusual Unix variant which puts include files in strange places or has broken Posix support. If you get compile errors from misc/rndunix.c or keymgmt/stream.c you may need to change the header files included near the start of the file.

Other systems

cryptlib should be fairly portable to other systems, the only two parts which need attention is the memory locking in cryptkrn.c (cryptlib will work without this, but won't be as secure as a version with memory locking because sensitive data may be paged out to disk) and the randomness-gathering in misc/rnd*os_name*.c (cryptlib won't work without this, the code will generate a compiler error). The idea behind the randomness-gathering code is to perform a comprehensive poll of every possible entropy source in the system in a separate thread or background task ("slowPoll"), as well as providing a less useful but much faster poll of quick-response sources ("fastPoll").

To find out what to compile, look at the Unix makefile which contains all the necessary source files (the *group_name*_OBJS dependencies) and compiler options. Link all these into a library (as the makefile does) and then compile and link test*xxx*.c modules in the test subdirectory with the library to create the self-test program. There is additional assembly-language code included which will lead to noticeable speedups on some systems, you should modify your build options as appropriate to use these if possible.

Depending on your compiler you may get a few warnings about some of the encryption and hashing code (one or two) and the bignum code (one or two). This code mostly relates to the use of C as a high-level assembler and changing things

around to remove the warnings on one system could cause the code to break on another system.

Key Database Setup

If you want to work with a public key database, you need to configure a database for cryptlib to use. Under Windows, go to the Control Panel and click on the ODBC/ODBC32 item. Click on "Add" and select the ODBC data source (that is, the database type) you want to use. If it's on the local machine, this will probably be an Access database, if it's a centralised database on a network this will probably be SQL Server. Once you've selected the data source type, you need to give it a name for cryptlib to use. "Public Keys" is a good choice (the self-test code expects to find a source called "testkeys" for use during the self-test procedure). In addition you may need to set up other parameters like the server the database is located on and other access information. Once the data source is set up, you can access it as a CRYPT_KEYSET_ODBC keyset using the name you've assigned to it.

Under Unix, the database type you use will require a specific database interface to be enabled in cryptlib. To enable the use of one of the cryptlib interfaces, you need to define the appropriate `DBX_`*`name`* setting in misc/dbms.h and link in the appropriate database library as described previously.

Certificate Installation

The PKCS #15 keyfile format currently exists only as a draft standard, until the final version is published it won't be possible to store certificate trust information because the format for this is still subject to revision. Because of this, the –t option described below doesn't have any effect. It's likely that when PKCS #15 is finalised the trusted certs will be stored in a fixed config file, removing the need to store them in the certificate database.

Before you use cryptlib, you should run the certificate installation program `certinst` from the cryptlib directory. This will install default certification authority (CA) certificates into a key database and update cryptlibs trust information to make the certificates trusted by cryptlib. Without these standard certificates installed and marked as trusted by cryptlib, it becomes difficult to automatically verify signatures in certificate chains signed by widely-recognised CA's such as Verisign and Thawte.

To install the default certificates, run `certinst` with the option `-it` to install the certificates and mark them as trusted and `-n` to specify the name of the key database you want to use to store the certificates in. You can also make the certificates trusted without installing them by only using the `-i` option. For example to install the default certificates into the key database "Public Keys" and mark them as trusted, you would use:

```
certinst -it -n"Public Keys"
```

Note that fact that the database name is quoted since it contains a space. To mark the certificates as trusted without installing them, you would use:

```
certinst -t
```

To access the database you may also need to specify a user name and password (the details depend on how the database has been configured). You can specify the username with `-u`*`name`* and the password with `-p`*`password`*. For example to install the certificates as before into a key database which only allows write access by an administrator using the password "password", you would use:

```
certinst -it -n"Public Keys" -uadministrator -ppassword
```

`certinst` has a number of other options, run it without any arguments for a help screen.

Cut-down cryptlib Versions

In some cases you may want to create a cut-down version of cryptlib which omits certain algorithms because of size constraints or patent problems. You can do this by building cryptlib with one or more NO_*algorithmname* preprocessor defines set to exclude the use of that algorithm. The defines for algorithms which can be excluded are NO_CAST, NO_ELGAMAL, NO_HMAC_MD5, NO_HMAC_RIPEMD160, NO_IDEA, NO_MD4, NO_MDC2, NO_RC2, NO_RC4, NO_RC5, NO_SAFER, and NO_SKIPJACK. This will remove any references to that algorithm from the code and make it impossible to use. Most linkers will discard the low-level algorithm code (since the remaining cryptlib code doesn't reference it any more), but if you're using a more primitive linker you may need to explicitly remove references to the appropriate files from the link phase. The files are the algorithm-specific lib_*name* file and the matching files from the crypt or hash subdirectories, for example for IDEA the files to remove are lib_idea.c and crypt/idea.c.

In addition to removing individual algorithms, you can also remove certain types of functionality from cryptlib in order to reduce code size: NO_COMPRESSION will remove support for compressed-data enveloping and NO_PGP will remove support for handling of PGP keyrings and data.

Support for Vendor-specific Algorithms

cryptlib supports the use of vendor-specific algorithm types with the predefined values CRYPT_ALGO_VENDOR1, CRYPT_ALGO_VENDOR2, and CRYPT_ALGO_VENDOR3. For each of the algorithms you use, you need to add the appropriate cryptlib capability definitions as used in cryptcap.c to a file called vendalgo.c, which will be automatically compiled into cryptlib. Finally, rebuild cryptlib with the preprocessor define USE_VENDOR_ALGOS set, which will include the new algorithm types in cryptlibs capabilities.

For example if you wanted to add support for the Foo256 cipher to cryptlib you would create the file vendalgo.c containing the capability definitions and then rebuild cryptlib with USE_VENDOR_ALGOS defined. The Foo256 algorithm would then become available as algorithm type CRYPT_ALGO_VENDOR1.

cryptlib Basics

cryptlib works with two classes of objects, container objects and action objects. A container object is an object which contains one or more items such as data, keys or certificates. An action object is an object which is used to perform an action such as encrypting or signing data. The container types used in cryptlib are envelopes (for data), sessions (for communications sessions), keysets (for keys), and certificates (for attributes such as key usage restrictions and signature information). Container objects can have items such as data or public/private keys placed in them and retrieved from them. In addition to containing data or keys, container objects can also contain other objects which affect the behaviour of the container object. For example pushing an encryption object into an envelope container object will result in all data which is pushed into the envelope being encrypted or decrypted using the encryption object.

The action object types used in cryptlib are encryption contexts (for encryption/ hashing/digital signatures). Action objects are used to act on data, for example to encrypt or decrypt a piece of data or to digitally sign or check the signature on a piece of data.

The usual mechanism for processing data is to use the envelope container object. The process of pushing data into an envelope and popping the processed data back out is known as enveloping the data. The reverse process is known as de-enveloping the data. The first section of this manual covers the basics of enveloping data, which introduces the enveloping mechanism and covers various aspects of the enveloping process such as processing data streams of unknown length and handling errors. Once you have the code to perform basic enveloping in place, you can add extra functionality such as password-based data encryption to the processing.

Once the basic concepts behind enveloping have been explained, more advanced techniques such as public-key based enveloping and digital signature enveloping are covered. The use of public keys for enveloping requires the use of key management functions, and the next section covers storing and retrieving keys from keyset objects.

So far all the objects which have been covered are container objects. The next section covers the creation of action objects which you can either push into a container object or apply directly to data, including the various ways of loading or generating keys into them. The next three sections explain how to apply the action objects to data and cover the process of encryption, key exchange, and signature generation and verification.

The public portions of public/private key pairs are typically managed using X.509 certificates and certificate revocation lists. The next three sections cover certificates, certificate extensions and attributes, and the management of certificates including certificate creation, certificate chains, and certificate revocation list (CRL) creation and checking. This covers the full key life cycle from creation through certification to revocation and/or destruction.

Once certificate management is in place, it’s possible to use cryptlib envelopes to process S/MIME messages. The next two sections cover the creation and handling of S/MIME messages using the same enveloping mechanisms which were covered earlier.

The next section covers the use of encryption devices such as smart cards and Fortezza cards, and explains how to use them to perform many of the tasks covered in previous sections. Finally, the last section covers miscellaneous topics such as random number management and the cryptlib configuration database.

Programming interfaces

cryptlib provides three levels of interface, of which the highest-level one is the easiest to use and therefore the recommended one. At this level cryptlib works with

envelope and session container objects, an abstract object into which you can insert and remove data which is processed as required while it is in the object (this is explained in more detail below). Using envelopes requires no knowledge of encryption or digital signature techniques. At an intermediate level, cryptlib works with encryption action objects, and requires some knowledge of encryption techniques. In addition you will need to handle some of the management of the encryption objects yourself. At the very lowest level cryptlib works directly with the encryption action objects and requires you to know about algorithm details (which can be queried from cryptlib) and key and data management methods.

Before you begin you should decide which interface you want to use, as each one has its own distinct advantages and disadvantages. The three interfaces are:

Container object interface

This interface requires no knowledge of encryption and digital signature techniques, and is easiest for use with languages like Visual Basic which don't interface to C data structures very well. The container object interface provides services to create and destroy envelopes and secure sessions, to add security attributes such as encryption information and signature keys to a container object, and to move data into and out of a container.

Mid-level interface

This interface requires some knowledge of encryption and digital signature techniques. Because it handles encoding of things like session keys and digital signatures but not of the data itself, it is better suited for applications which require high-speed data encryption, or encryption of many small data packets (such as an encrypted terminal session). The container object interface is built on top of this interface. The mid-level interface provides services such as routines to export and import encrypted keys and to create and check digital signatures.

Action object interface

This interface requires quite a bit of knowledge of encryption and digital signature techniques. It provides a direct interface to the raw encryption capabilities of cryptlib. The only real reason for using the low-level routines is if you need them as building blocks for your own custom encryption protocol. The mid-level interface is built on top of this interface. The low-level interface serves as an interface to a range of plug-in encryption modules which allow encryption algorithms to be added in a fairly transparent manner, with a standardised interface allowing any of the algorithms and modes supported by cryptlib to be used with a minimum of coding effort. As such the main function of the action object interface is to provide a standard, portable, easy-to-use interface between the underlying encryption routines and the user software.

Objects and Interfaces

The cryptlib object types are key certificate objects of type CRYPT_CERTIFICATE which usually contain a key certificate for an individual or organisation but can also contain other information such as certificate chains or digital signature attributes, encryption context objects of type CRYPT_CONTEXT which contain encryption or digital signature key information, envelope container objects of type CRYPT_ENVELOPE which provide an abstract container for performing encryption and security-related operations on an item of data, key collection container objects of type CRYPT_KEYSET which contain collections of public or private keys, and secure session objects of type CRYPT_SESSION which manage a secure session with a server or client. Finally, there are device objects of type CRYPT_DEVICE which provide a mechanism for working with crypto devices such as crypto hardware accelerators and PCMCIA and smart cards.

These objects are referred to via arbitrary integer values, or handles, which have no meaning outside of cryptlib. All data pertaining to an object is managed internally by cryptlib, with no outside access to security-related information being possible. There is also a generic object handle of type CRYPT_HANDLE which is used in cases where the exact type of an object is not important. For example most cryptlib functions which require keys can work with either encryption contexts or key certificate objects, so the objects they use have a generic CRYPT_HANDLE which is equivalent to either a CRYPT_CONTEXT or a CRYPT_CERTIFICATE.

Objects and Attributes

Each cryptlib object has a number of attributes of type CRYPT_ATTRIBUTE_TYPE which you can read, write, and in some cases delete. For example an encryption context would have a key attribute, a certificate would have issuer name and validity attributes, and an envelope would have attributes such as passwords or signature information, depending on the type of the envelope. Most cryptlib objects are controlled by manipulating these attributes.

The attribute classes are CRYPT_ATTRIBUTE_*name* for generic attributes, CRYPT_CERTINFO_*name* for certificate attributes, CRYPT_CTXINFO_*name* for encryption context attributes, CRYPT_DEVINFO_*name* for device attributes, CRYPT_ENVINFO_*name* for envelope attributes, CRYPT_KEYSETINFO_*name* for keyset attributes, CRYPT_OPTION_*name* for global configuration options, CRYPT_PROPERTY_*name* for object properties, and CRYPT_SESSINFO_*name* for session attributes. Some of the attributes apply only to a particular object type but others may apply across multiple objects (for example a certificate contains a public key, so the key size attribute, which is normally associated with a context, would also apply to a certificate). To determine the key size for the key in a certificate, you would read its key size attribute as if it were an encryption context.

Attribute data is either a single numeric value or variable-length data consisting of a (data, length) pair. Numeric attribute values are used for objects, boolean values,and integers. Variable-length data attribute values are used for text strings, binary data blobs, and representations of time (which uses the ANSI/ISO standard seconds-since-1970 format).

Interfacing with cryptlib

All necessary constants, types, structures, and function prototypes are defined in the header file cryptlib.h (for C and C++), cryptlib.bas (for Visual Basic), or cryptlib.pas (for Delphi). You need to include one of these files in each module which makes use of cryptlib. Although the examples given in this manual are for C/C++, they apply equally for the other languages.

Initialisation

Before you can use any of the cryptlib functions, you need to call the **cryptInit** function to initialise cryptlib. You also need to call its companion function **cryptEnd** at the end of your program. **cryptInit** initializes cryptlib for use, and **cryptEnd** performs various cleanup functions including automatic garbage collection of any objects you may have forgotten to destroy. You don't have to worry about inadvertently calling **cryptInit** multiple times (for example if you're calling it from multiple threads), it will handle the initialisation correctly.

If you call **cryptEnd** and there are still objects in existence, it will return CRYPT_-ERROR_INCOMPLETE to inform you that there were leftover objects present. cryptlib can tell this because it keeps track of each object so it can erase any sensitive data which may be present in the object (**cryptEnd** will return a CRYPT_ERROR_-INCOMPLETE error to warn you, but will nevertheless clean up and free each object for you).

To make the use of **cryptEnd** in a C or C++ program easier, you may want to use the C `atexit()` function or add a call to **cryptEnd** to a C++ destructor in order to have **cryptEnd** called automatically when your program exits.

Any use of cryptlib will then be as follows:

```
#include "cryptlib.h"

cryptInit();

/* Calls to cryptlib routines */

cryptEnd();
```

If you're going to be doing something which needs encryption keys, you should also perform a randomness poll fairly early on to give cryptlib enough random data to create keys:

```
cryptAddRandom( NULL, CRYPT_RANDOM_SLOWPOLL );
```

The randomness poll executes asynchronously, so it won't stall the rest of your code while it's running.

Working with Object Attributes

All object attributes are read, written, and deleted using a common set of functions: **cryptGetAttribute**/**cryptGetAttributeString** to get the value of an attribute, **cryptSetAttribute**/**cryptSetAttributeString** to set the value of an attribute, and **cryptDeleteAttribute** to delete an attribute. Attribute deletion is only valid for a small subset of attributes for which it makes sense, for example you can delete the validity date attribute from a certificate before the certificate is signed but not after it's signed, and you can never delete the algorithm type attribute from an encryption context.

cryptGetAttribute and **cryptSetAttribute** take as argument an integer value or a pointer to a location to receive an integer value:

```
int keySize;

cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PUBLICKEY, cryptKey );
cryptGetAttribute( cryptContext, CRYPT_CTXINFO_KEYSIZE, &keySize );
```

cryptGetAttributeString and **cryptSetAttributeString** take as argument a pointer to the data value to get or set and a length value or pointer to a location to receive the length value:

```
char emailAddress[ 128 ]
int emailAddressLength;

cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   "1234", 4 );
cryptGetAttributeString( cryptCertificate, CRYPT_CERTINFO_RFC822NAME,
   emailAddress, &emailAddressLength );
```

This leads to a small problem: How do you know how big to make the buffer? The answer is to use **cryptGetAttributeString** to tell you. If you pass in a null pointer for the data value, the function will set the length value to the size of the data, but not do anything else. You can then use code like:

```
char *emailAddress;
int emailAddressLength;

cryptGetAttributeString( cryptCertificate, CRYPT_CERTINFO_RFC822NAME,
   NULL, &emailAddressLength );
emailAddress = malloc( emailAddressLength );
cryptGetAttributeString( cryptCertificate, CRYPT_CERTINFO_RFC822NAME,
   emailAddress, &emailAddressLength );
```

to obtain the data value. In most cases this two-step process isn't necessary, the standards cryptlib conforms to generally place limits on the size of most attributes so that cryptlib will never return more data than the fixed limit. For example most strings in certificates are limited to a maximum length set by the CRYPT_MAX_-

TEXTSIZE constant. More information on these sizes is given with the descriptions of the attributes.

Finally, **cryptDeleteAttribute** lets you delete an attribute in the cases where that's possible:

```
cryptDeleteAttribute( cryptCertificate, CRYPT_CERTINFO_VALIDFROM );
```

All access to objects and object attributes is enforced by cryptlib's security kernel. If you try to access or manipulate an attribute in a manner which isn't allowed (for example by trying to read a write-only attribute, trying to assign a string value to a numeric attribute, trying to delete an attribute which can't be deleted, trying to set a certificate-specific attribute for an envelope, or some similar action) cryptlib will return an error code to tell you that this type of access is invalid. If there's a problem with the object you're trying to manipulate, cryptlib will return CRYPT_ERROR_-PARAM1 to tell you that the object handle parameter passed to the function is invalid. If there's a problem with the attribute type (typically because it's invalid for this object type) cryptlib will return CRYPT_ERROR_PARAM2. If there's a problem with the attribute value, cryptlib will return CRYPT_ERROR_PARAM3, and if there's a problem with the length (for thje functions which take a length parameter) cryptlib will return CRYPT_ERROR_PARAM4. If you try to perform an attribute access which is disallowed (reading an attribute which can't be read, writing to or deleting a read-only attribute, or something similar) cryptlib will return CRYPT_ERROR_PERMISSION. If you try to delete an attribute which isn't set, cryptlib will return CRYPT_ERROR_NOTFOUND.

Attribute Types

Attribute values can be boolean or numeric values, text strings, or binary data:

Type	Description
Binary	A binary data string which can contain almost anything.
Boolean	Flags which can be set to 'true' (any nonzero value) or 'false' (a zero value) and which control whether a certain option or operation is enabled or not (note that cryptlib uses the value 1 to represent 'true', some languages may represent this by the value -1). For example the CRYPT_CERTINFO_CA attribute in a certificate controls whether a certificate is marked as being a CA certificate or not.
Numeric	A numeric constant such as an integer value, a bitflag, or a handle to a cryptlib object. For example the CRYPT_-CERTINFO_SUBJECTPUBLICKEYINFO attribute specifies the public key to be added to a certificate, and the CRYPT_CERTINFO_CRLREASON attribute specifies a bitflag which indicates why a CRL was issued.
String	A text string which contains information such as a name, message, email address, or URL. For example CRYPT_-CTXINFO_LABEL contains a human-readable label used to identify private keys. The most frequently used text string components are those which make up a certificates distinguished name, which identifies the certificate owner. All of these components are limited to a maximum of 64 characters by the X.500 standard which covers certificates and their components, and cryptlib provides the CRYPT_MAX_TEXTSIZE constant for use with these components. Since this limit is specified in characters rather than bytes, Unicode strings in certificates can be several times as long as this value when their length is expressed in bytes, depending on which data type the system uses to represent Unicode characters. cryptlib also uses the

Type	Description
	CRYPT_MAX_TEXTSIZE limit for most other strings such as key labels.

Since most text strings have a fixed maximum length, you can use code like:

```
char commonName[ CRYPT_MAX_TEXTSIZE + 1 ];
int commonNameLength;

/* Retrieve the component and null-terminate it */
cryptGetAttributeString( cryptCert, CRYPT_CERTINFO_COMMONNAME,
   commonName, &commonNameLength );
commonName[ commonNameLength ] = '\0';
```

to read the value, in this case the common name of a certificate owner. This assumes that the common name is expressed in a single-byte character set. For Unicode strings, you need to multiply the size of the buffer by the size of a Unicode character on your system.

Note the addition of the terminating null character, since the text strings returned aren't null-terminated.

Object Security

Each cryptlib object has its own security settings which affect the way you can use the object. You can set these attributes, identified by CRYPT_PROPERTY_*name*, after you create an object to provide enhanced control over how it is used. For example on a system which supports threads you can bind an object to an individual thread within a process so that only the thread which owns the object can see it. For any other thread in the process, the object handle is invalid.

You can get and set an objects properties using **cryptGetAttribute** and **cryptSetAttribute**, passing as arguments the object whose property attribute you want to change, the type of property attribute to change, and the attribute value or a pointer to a location to receive the attributes' value. The object property attributes which you can get or set are:

Property/Description	Type
CRYPT_PROPERTY_DECRYPTONLY CRYPT_PROPERTY_ENCRYPTONLY Whether an encryption action object can be used only to encrypt or decrypt data. This attribute is useful when you want to restrict the way an encryption action object can be used, for example before you change the ownership of an encryption object to allow it to be used by other threads you could restrict it to be usable only for encryption or decryption purposes. These attributes are write-once attributes, once you've set them they can't be reset.	Boolean
CRYPT_PROPERTY_FORWARDCOUNT The number of times an object can be forwarded (that is, the number of times the ownership of the object can be changed). Each time the objects ownership is changed, the forwarding count decreases by one; once it reaches zero, the object can't be forwarded any further. For example if you set this attributes' value to 1 then you can forward the object to another thread, but that thread can't forward it further. After you set this attribute (and any other security-related attributes), you should set the CRYPT_PROPERTY_LOCKED attribute to ensure that it can't be changed later.	Numeric
CRYPT_PROPERTY_HIGHSECURITY This is a composite value which sets all general security-related attributes to their highest security setting. Setting this value will make an object owned, non-exportable (if appropriate), non-forwardable, and locked. Since this is a composite value representing a number of separate attributes, its value can't	Boolean

Property/Description	Type

be read or unset after being set.

CRYPT_PROPERTY_LOCKED Boolean

Locks the security-related object attributes so that they can no longer be changed. You should set this attribute once you've set other security-related attributes such as CRYPT_PROPERTY_FORWARDCOUNT.

This attribute is a write-once attribute, once you've set it can't be reset.

CRYPT_PROPERTY_NONEXPORTABLE Boolean

Whether a key in an encryption action object can be exported from the object in encrypted form. Normally only session keys can be exported, and only in encrypted form, however in some cases private keys are also exported in encrypted form when they can are saved to a keyset. By setting this attribute you can make them non-exportable in any form (some keys, such as those held in crypto devices, are nonexportable by default).

This attribute is a write-once attribute, once you've set it can't be reset.

CRYPT_PROPERTY_OWNER Numeric

The identity of the thread which owns the object. The thread's identity is specified using a value which depends on the operating system, but is usually a thread handle or thread ID. For example under Windows 95/98 and NT, the thread ID is the value returned by the `GetCurrentThreadID` function, which returns a systemwide unique handle for the current thread.

You can also pass in a value of CRYPT_UNUSED, which unbinds the object from the thread and makes it accessible to all threads in the process.

CRYPT_PROPERTY_USAGECOUNT Numeric

The number of times an action object can be used before it deletes itself and becomes unusable. Every time an action object is used (for example when a signature encryption object is used to create a signature), its usage count is decremented; once the usage count reaches zero, the object can't be used to perform any further actions (although you can still perform non-action operations such as reading its attributes).

This attribute is useful when you want to restrict the number of times an object can be used by other code. For example, before you change the ownership of a signature object to allow it to be used by another thread, you would set the usage count to 1 to ensure that it can't be used to sign arbitrary numbers of messages or transactions. This eliminates a troubling security problem with objects such as smart cards where, once a user has authenticated themselves to the card, the software can ask the card to sign arbitrary numbers of (unauthorised) transactions alongside the authorised ones.

This attribute is a write-once attribute, once you've set it can't be reset.

For example to create a triple DES encryption context in one thread and transfer ownership of the context to another thread you would use:

```
CRYPT_CONTEXT cryptContext;

/* Create a context and claim it for exclusive use */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptSetAttribute( cryptContext, CRYPT_PROPERTY_OWNER, threadID );

/* Generate a key into the context */
cryptGenerateKey( cryptContext );

/* Transfer ownership to another thread */
cryptSetAttribute( cryptContext, CRYPT_PROPERTY_OWNER, otherThreadID
   );
```

The other thread now has exclusive ownership of the context containing the loaded key. If you wanted to prevent the other thread from transferring the context further, you would also have to set the CRYPT_PROPERTY_FORWARDCOUNT property to 1 (to allow you to transfer it) and then set the CRYPT_PROPERTY_LOCKED attribute (to prevent the other thread from changing the attributes you've set).

Note that in the above code the object is claimed as soon as it's created (and before any sensitive data is loaded into it) to ensure that another thread isn't given a chance to use it when it contains sensitive data. The use of this type of object binding is recommended when working with sensitive information under Windows 95/98 and Windows NT, since the Win32 API provides several security holes whereby any process in the system may interfere with resources owned by any other process in the system. The checking for object ownership which is performed typically adds a few microseconds to each call, so in extremely time-critical applications you may want to avoid binding an object to a thread. On the other hand for valuable resources such as private keys, you should always consider binding them to a thread, since the small overhead becomes insignificant compared to the cost of the public-key operation.

Although the example shown above is for encryption contexts, the same applies to other types of objects such as keysets and envelopes (although in that case the information they contain isn't as sensitive as it is for encryption contexts). For container objects which can themselves contain objects (for example keysets), if the container is bound to a thread then any objects which are retrieved from it are also bound to the thread. For example if you're reading a private key from a keyset, you should bind the keyset to the current thread after you open it (but before you read any keys) so that any keys read from it will also automatically be bound to the current thread. In addition if a key which is used to generate another key (for example the key which imports a session key) is bound, then the resulting generated key will also be bound.

On non-multithreaded systems, CRYPT_PROPERTY_OWNER and CRYPT_-PROPERTY_FORWARDCOUNT have no effect, so you can include them in your code for any type of system.

Interaction with External Events

Internally, cryptlib consists of a number of security-related objects, some of which can be controlled by the user through handles to the objects. These objects may also be acted on by external forces such as information coming from encryption and system hardware, which will result in a message related to the external action being sent to any affected cryptlib objects. An example of such an event is the withdrawal of a smart card from a card reader, which would result in a card removal message being sent to all cryptlib objects which were created using information stored on the card. This can affect quite a number of objects.

Typically, the affected cryptlib objects will destroy any sensitive information held in memory and disable themselves from further use. If you try to use any of the objects, cryptlib will return CRYPT_ERROR_SIGNALLED to indicate that an external event has caused a change in the state of the object.

After an object has entered the signalled state, the only remaining operation you can perform with the object is to destroy it using the appropriate function.

Enveloping Concepts

Encryption envelopes are the easiest way to use cryptlib. An envelope is a container object whose behaviour is modified by the data and resources which you push into it. To use an envelope, you push into it other container and action objects and resources such as passwords which control the actions performed by the envelope, and then push in data and pop out data which is processed according to the resources you've pushed in. cryptlib takes care of the rest. For example to encrypt the message "This is a secret" with the password "Secret password" you would do the following:

```
create the envelope;
add the password attribute "Secret password" to the envelope;
push data "This is a secret" into the envelope;
pop encrypted data from the envelope;
destroy the envelope;
```

That's all that's necessary. Since you've added a password attribute, cryptlib knows that you want to encrypt the data in the envelope with the password, so it encrypts the data and returns it to you. This process is referred to as enveloping the data.

The opposite, de-enveloping process consists of:

```
create the envelope;
push encrypted data into the envelope;
add the password attribute "Secret password" to the envelope;
pop decrypted data from the envelope;
destroy the envelope;
```

cryptlib knows the type of encrypted data that it's working with (it can inform you that you need to push in a password if you don't know that in advance), decrypts it with the provided password, and returns the result to you.

This example illustrates a feature of the de-enveloping process which may at first seem slightly unusual: You have to push in some encrypted data before you can add the password attribute needed to decrypt it. This is because cryptlib will automatically determine what to do with the data you give it, so if you added a password before you pushed in the encrypted data cryptlib wouldn't know what to do with the password.

Signing data is almost identical, except that you add a signature key attribute instead of a password. You can also add a number of other encryption attributes depending on the type of funtionality you want. Since all of these require further knowledge of cryptlib's capabilities, only basic data, compressed-data, and password-based enveloping will be covered in this section.

Due to constraints in the underlying data formats which cryptlib supports, it is not currently possible to perform more than one of compression, encryption, or signing using a single envelope (the resulting data stream can't be encoded using most of the common data formats supported by cryptlib). If you want to perform more than one of these operations, you need to use multiple envelopes, one for each of the processing steps you want to perform. If you try and add an encryption attribute to an envelope which is set up for signing, or a signing attribute to an envelope which is set up for encryption, cryptlib will return a parameter error to indicate that the attribute type is invalid for this envelope since it is already being used for a different purpose.

Creating/Destroying Envelopes

Envelopes are accessed through envelope objects which work in the same general manner as the other container objects used by cryptlib. Before you can envelope or de-envelope data you need to create the appropriate type of envelope for the job. If you want to envelope some data, you would create the envelope with **cryptCreateEnvelope**, specifying the format for the enveloped data (for now you should use CRYPT_FORMAT_CRYPTLIB, the default format):

```
CRYPT_ENVELOPE cryptEnvelope;
```

```
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Perform enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

If you want to de-envelope the result of the previous enveloping process, you would create the envelope with a format CRYPT_FORMAT_AUTO, which tells cryptlib to automatically detect and use the appropriate format to process the data:

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Perform de-enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

By default the envelope object which is created will have a 16K data buffer on DOS and 16-bit Windows systems, and a 32K buffer elsewhere. The size of the internal buffer affects the amount of extra processing which cryptlib needs to perform; a large buffer will reduce the amount of copying to and from the buffer, but will consume more memory (the ideal situation to aim for is one in which the data fits completely within the buffer, which means that it can be processed in a single operation). Since the process of encrypting and/or signing the data can increase its overall size, you should make the buffer 1-2K larger than the total data size if you want to process the data in one go. The minimum buffer size is 4K, and on 16-bit systems the maximum buffer size is 32K-1.

If want to use a buffer which is smaller or larger than the default size, you can specify its size using the CRYPT_ATTRIBUTE_BUFFERSIZE attribute after the envelope has been created. For example if you knew you were going to be processing a single 80K message on a 32-bit system (you can't process more than 32K-1 bytes at once on a 16-bit system) you would use:

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );
cryptSetAttribute( cryptEnvelope, CRYPT_ATTRIBUTE_BUFFERSIZE, 90000L
   );
/* Perform enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

(the extra 10K provides a generous safety margin for message expansion due to the enveloping process). When you specify the size of the buffer, you should try and make it as large as possible, unless you're pretty certain you'll only be seeing messages up to a certain size. Remember, the larger the buffer, the less processing overhead is involved in handling data. However, if you make the buffer excessively large it increases the probability that the data in it will be swapped out to disk, so it's a good idea not to go overboard on buffer size. You don't have to process the entire message at once, cryptlib provides the ability to envelope or de-envelope data in multiple sections to allow processing of arbitrary amounts of data even on systems with only small amounts of memory available.

Note that the CRYPT_ENVELOPE is passed to the envelope creation functions by reference, as they modify it when they create the envelope. In all other routines in cryptlib, CRYPT_ENVELOPE is passed by value.

The Data Enveloping Process

Although this section only covers basic data and password-based enveloping, the concepts it covers apply to all the other types of enveloping as well, so you should familiarise yourself with this section even if you're only planning to use the more advanced types of enveloping such as digitally signd data enveloping. The general model for enveloping data is:

```
add any attributes such as passwords or keys
push in data
```

```
pop out processed data
```

To de-envelope data:

```
push in data
(cryptlib will inform you what resource(s) it needs to process the
   data)
add the required attribute such as a password or key
pop out processed data
```

The enveloping/de-enveloping functions perform a lot of work in the background. For example when you add a password attribute to an envelope and follow it with some data, the function hashes the variable-length password down to create a fixed-length key for the appropriate encryption algorithm, generates a temporary session key to use to encrypt the data you'll be pushing into the envelope, uses the fixed-length key to encrypt the session key, encrypts the data (taking into account the fact that most encryption modes can't encrypt individual bytes but require data to be present in fixed-length blocks), and then cleans up by erasing any keys and other sensitive information still in memory. This is why it's recommended that you use the envelope interface rather than trying to do the same thing yourself.

The **cryptPushData** and **cryptPopData** functions are used to push data into and pop data out of an envelope. For example to push the message "Hello world" into an envelope, you would use:

```
cryptPushData( envelope, "Hello world", 11, &bytesCopied );
```

The function will return an indication of how many bytes were copied into the envelope in `bytesCopied`. Usually this is the same as the number of bytes you pushed in, but if the envelope is almost full or you're trying to push in a very large amount of data, only some of the data may be copied in. This is useful when you want to process a large quantity of data in multiple sections, which is explained further on.

Popping data works similarly to pushing data:

```
cryptPopData( envelope, buffer, bufferSize, &bytesCopied );
```

In this case you supply a buffer to copy the data to, and an indication of how many bytes you want to accept, and the function will return the number of bytes actually copied in `bytesCopied`. This could be anything from zero up to the full buffer size, depending on how much data is present in the envelope.

Once you've pushed the entire quantity of data which you want to process into an envelope, you need perform a final push with a length of zero bytes to tell the envelope object to wrap up the data processing. If you try to push in any more data after this point, cryptlib will return a CRYPT_ERROR_COMPLETE error to indicate that processing of the data in the envelope has been completed and no more data can be added. Since the enveloped data contains all the information necessary to de-envelope it, it isn't necessary to perform the final zero-byte push during de-enveloping.

The **cryptSetAttribute** and **cryptSetAttributeString** functions are used to add attributes to an envelope, with the attribute being identified by a CRYPT_ENVINFO_*type* value. For example to add the password attribute "password" to an envelope, you would use:

```
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   "password", 8 );
```

The various types of attributes which you can add are explained in more detail further on.

Data Size Considerations

When you add data to an envelope, cryptlib processes and encodes it in a manner which allows arbitrary amounts of data to be added. If cryptlib knows in advance how much data will be pushed into the envelope, it can use a more efficient encoding method since it doesn't have to take into account an indefinitely long data stream.

You can notify cryptlib of the overall data size by setting the CRYPT_ENVINFO_-DATASIZE attribute:

```
cryptSetAttribute( envelope, CRYPT_RESOURCE_DATASIZE, dataSize );
```

This tells cryptlib how much data will be added, and allows it to use the more efficient encoding format. If you push in more data than this before you wrap up the enveloping with a zero-byte push, cryptlib will return CRYPT_ERROR_-OVERFLOW; if you push in less, it will return CRYPT_ERROR_UNDERFLOW.

The amount of data popped out of an envelope never matches the amount pushed in, because the enveloping process adds encryption headers, digital signature information, and assorted other paraphernalia which is required to process a message. In many cases the overhead involved in wrapping up a block of data in an envelope can be noticeable, so you should always push and pop as much data at once into and out of an envelope as you can. For example if you have a 100-byte message and push it in as 10 lots of 10 bytes, this is much slower than pushing a single lot of 100 bytes. This behaviour is identical to the behaviour in applications like disk or network I/O, where writing a single big file to disk is a lot more efficient than writing 10 smaller files, and writing a single big network data packet is more efficient than writing 10 smaller data packets.

Push and popping unnecessarily small blocks of data when the total data size is unknown can also affect the overall enveloped data size. If you haven't told cryptlib how much data you plan to process with CRYPT_ENVINFO_DATASIZE then each time you pop a block of data from an envelope, cryptlib has to wrap up the current block and add header information to it to allow it to be de-enveloped later on. Because this encoding overhead consumes extra space, you should again try to push and pop a single large data block rather than many small ones (to prevent worst-case behaviour, cryptlib will coalesce adjacent small blocks into a mimum block size of 10 bytes, so it won't return an individual block containing less than 10 bytes unless it's the last block in the envelope). This is again like disk data storage or network I/O, where many small files or data packets lead to greater fragmentation and wasted storage space or network overhead than a single large file or packet.

Basic Data Enveloping

In the simplest case the entire message you want to process will fit into the envelopes internal buffer. The simplest type of enveloping does nothing to the data at all, but just wraps it and unwraps it:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

/* Create the envelope */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Add the data size information and data followed by a zero-length
   block to wrap up the processing, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

/* Destroy the envelope */
cryptDestroyEnvelope( cryptEnvelope );
```

To de-envelope the resulting data you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

/* Create the envelope */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataSize,
   &bytesCopied );
```

```
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

/* Destroy the envelope */
cryptDestroyEnvelope( cryptEnvelope );
```

This type of enveloping isn't terribly useful, but it does demonstrate how the enveloping process works.

Compressed Data Enveloping

A variation of basic data enveloping is compressed data enveloping which compresses or decompresses data during the enveloping process. Compressing data before signing or encryption improves the overall enveloping throughput (compressing data and encrypting the compressed data is faster than just encrypting the larger, uncompressed data), increases security by removing known patterns in the data, and saves storage space and network bandwidth.

To tell cryptlib to compress data which you add to an envelope, you should set the CRYPT_ENVINFO_COMPRESSION attribute before you add the data. This attribute doesn't take a value, so you should set it to CRYPT_UNUSED. The code to compress a message is then:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Tell cryptlib to compress the data */
cryptSetAttribute ( cryptEnvelope, CRYPT_ENVINFO_COMPRESSION,
   CRYPT_UNUSED );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Deenveloping compressed data works exactly like decompressing normal data, cryptlib will transparently decompress the data for you and return the decompressed result when you call **cryptPopData**.

The compression/decompression process can cause a large change in data size between what you push and what you pop back out, so you typically end up pushing much more than you pop or popping much more than you push. In particular, you may end up pushing multiple lots of data before you can pop any compressed data out, or pushing a single lot of compressed data and having to pop multiple lots of uncompressed data. This applies particularly to the final stages of enveloping when you push a zero-byte block to flush out any remaining data, which signals the compressor to wrap up processing and move any remaining data into the envelope. This means that the flush can return CRYPT_ERROR_OVERFLOW to indicate that there is more data to be flushed, requiring multiple iterations of flushing and copying out data:

```
/* ... */

/* Flush out any remaining data */
do
   {
   cryptPushData( cryptEnvelope, NULL, 0, NULL );
   cryptPopData( cryptEnvelope, outBuffer, BUFFER_SIZE, &bytesCopied
      );
   }
while( bytesCopied );
```

To handle this in a more general manner, you should use the processing techniques described in “Enveloping Large Data Quantities” on page 28.

Password-based Encryption Enveloping

To do something useful (security-wise) to the data, you need to add a container or action object or other type of attribute to tell the envelope to secure the data in some way. For example if you wanted to encrypt a message with a password you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Add the password */
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

To de-envelope the resulting data you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and the password required to deenvelope
   it, and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PASSWORD, password,
   passwordLength );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

cryptDestroyEnvelope( cryptEnvelope );
```

If you add the wrong password, cryptlib will return a CRYPT_ERROR_-WRONGKEY error. You can use this to request a new password from the user and try again. For example to give the user the traditional three attempts at getting the password right you would replace the code to add the password with:

```
for( i = 0; i < 3; i++ )
   {
   password = ...;
   if( cryptSetAttributeString( envelope, CRYPT_RESOURCE_PASSWORD,
      password, passwordLength ) == CRYPT_OK )
      break;
   }
```

De-enveloping Mixed Data

Sometimes you won’t know exactly what type of processing has been applied to the data you’re trying to de-envelope, so you can let cryptlib tell you what to do. When cryptlib needs some sort of resource (such as a password or an encryption key) to process the data which you’ve pushed into an envelope, it will return a CRYPT_-ENVELOPE_RESOURCE error if you try and push in any more data or pop out the processed data. This error code is returned as soon as cryptlib knows enough about the data you’re pushing into the envelope to be able to process it properly. Typically, as soon as you start pushing in encrypted, signed, or otherwise processed data,

cryptPushData will return CRYPT_ENVELOPE_RESOURCE to tell you that it needs some sort of resource in order to continue.

If you knew that the data you were processing was either plain, unencrypted or compressed data or password-encrypted data created using the code shown earlier, you could de-envelope it with:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, status

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and pop out the recovered message */
status = cryptPushData( cryptEnvelope, envelopedData,
   envelopedDataLength, &bytesCopied );
if( status == CRYPT_ENVELOPE_RESOURCE )
   cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
      password, passwordLength );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

cryptDestroyEnvelope( cryptEnvelope );
```

If the data is enveloped without any processing or is compressed data, cryptlib will de-envelope it without requiring any extra input. If the data is enveloped using password-based encryption, cryptlib will return CRYPT_ENVELOPE_RESOURCE to indicate that it needs a password before it can continue.

This illustrates the manner in which the enveloped data contains enough information to allow cryptlib to process it automatically. If the data had been enveloped using some other form of processing (for example public-key encryption or digital signatures), cryptlib would ask you for the private decryption key or the signature check key at this time (it's actually slightly more complex than this, the details are explained in "Advanced Enveloping" on page 32).

Enveloping Large Data Quantities

Sometimes, a message may be too big to process in one go or may not be available in its entirety, an example being data which is being sent or received over a network interface where only the currently transmitted or received portion is available. Although it's much easier to process a message in one go, it's also possible to envelope and de-envelope it a piece at a time (bearing in mind the earlier comment that the enveloping is most efficient when you push and pop data a single large block at a time rather than in many small blocks). With unknown amounts of data to be processed it generally isn't possible to use CRYPT_ENVINFO_DATASIZE, so in the sample code below this is omitted.

There are several strategies for processing data in multiple parts. The simplest one simply pushes and pops a fixed amount of data each time:

```
loop
   push data
   pop data
```

Since there's a little overhead added by the enveloping process, you should always push in slightly less data than the envelope buffer size. Alternatively, you can use the CRYPT_ATTRIBUTE_BUFFERSIZE to specify an envelope buffer which is slightly larger than the data block size you want to use. The following code uses the first technique to password-encrypt a file in blocks of BUFFER_SIZE - 4K bytes:

```
CRYPT_ENVELOPE cryptEnvelope;
void *buffer;
int bufferCount;

/* Create the envelope with a buffer of size BUFFER_SIZE and add the
   password attribute */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );
cryptSetAttribute( cryptEnvelope, CRYPT_ATTRIBUTE_BUFFERSIZE,
   BUFFER_SIZE );
```

```
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );

/* Allocate a buffer for file I/O */
buffer = malloc( BUFFER_SIZE );

/* Process the entire file */
while( !endOfFile( inputFile ) )
   {
   int bytesCopied;

   /* Read a BUFFER_SIZE - 4K block from the input file, envelope it,
      and write the result to the output file */
   bufferCount = readFile( inputFile, buffer, BUFFER_SIZE - 4096 );
   cryptPushData( cryptEnvelope, buffer, bufferCount, &bytesCopied );
   cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
   writeFile( outputFile, buffer, bytesCopied );
   }

/* Flush the last lot of data out of the envelope */
cryptPushData( cryptEnvelope, NULL, 0 );
cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
if( bytesCopied )
   writeFile( outputFile, buffer, bytesCopied );
free( buffer );

cryptDestroyEnvelope( cryptEnvelope );
```

The code allocates a BUFFER_SIZE byte I/O buffer, reads up to BUFFER_SIZE - 4K from the input file, and pushes it into the envelope. It then tells cryptlib to pop up to BUFFER_SIZE bytes of enveloped data back out into the buffer, takes whatever is popped out, and writes it to the output file. When it has processed the entire file, it pushes in the usual zero-length data block to flush any remaining data out of the buffer.

Note that the upper limit on BUFFER_SIZE depends on the system you're running the code on. If you need to run it on a 16-bit system, BUFFER_SIZE is limited to 32K–1 bytes because of the length limit imposed by 16-bit integers, and the default envelope buffer size is 16K bytes unless you specify a larger default size using the CRYPT_ATTRIBUTE_BUFFERSIZE attribute.

Going to a lot of effort to exactly match a certain data size such as a power of two when pushing and popping data isn't really worthwhile, since the overhead added by the envelope encoding will always change the final encoded data length.

When you're performing compressed data enveloping or de-enveloping, the processing usually results in a large change in data size, in which case you may need to use the technique described below which can handle arbitrarily-sized input and output quantities.

Alternative Processing Techniques

A slightly more complex technique is to always stuff the envelope as full as possible before trying to pop anything out of it:

```
loop
   do
      push data
   while push status != CRYPT_ERROR_OVERFLOW
   pop data
```

This results in the most efficient use of the envelopes internal buffer, but is probably overkill for the amount of code complexity required:

```
CRYPT_ENVELOPE cryptEnvelope;
void *inBuffer, *outBuffer;
int bytesCopiedIn, bytesCopiedOut, bufferCount;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
```

```
/* Allocate input and output buffers */
inBuffer = malloc( BUFFER_SIZE );
outBuffer = malloc( BUFFER_SIZE );

/* Process the entire file */
while( !endOfFile( inputFile ) )
   {
   int offset = 0;

   /* Read a buffer full of data from the file and push and pop it
      to/from the envelope */
   bufferCount = readFile( inputFile, inBuffer, BUFFER_SIZE );
   while( bufferCount )
      {
      /* Push as much as we can into the envelope */
      cryptPushData( cryptEnvelope, inBuffer + offset, bufferCount,
         &bytesCopiedIn );
      offset += bytesCopiedIn;
      bufferCount -= bytesCopiedIn;

      /* If we couldn't push everything in, the envelope is full, so
         we empty a buffers worth out */
      if( bufferCount )
         {
         cryptPopData( cryptEnvelope, outBuffer, BUFFER_SIZE,
            &bytesCopiedOut );
         writeFile( outputFile, outBuffer, bytesCopiedOut );
         }
      }
   }

/* Flush out any remaining data */
do
   {
   cryptPushData( cryptEnvelope, NULL, 0, NULL );
   cryptPopData( cryptEnvelope, outBuffer, BUFFER_SIZE,
      &bytesCopiedOut );
   if( bytesCopiedOut)
      writeFile( outputFile, outBuffer bytesCopiedOut );
   }
while( bytesCopiedOut );
free( inBuffer );
free( outBuffer );

cryptDestroyEnvelope( cryptEnvelope );
```

Running the code to fill/empty the envelope in a loop is useful when a transformation such as data compression, which dramatically changes the length of the enveloped/de-enveloped data, is being applied. In this case it's not possible to tell how much data can still be pushed into or popped out of the envelope because the length is transformed by the compression operation. It's also generally good practice to not write code which makes assumptions about the amount of internal buffer space available in the envelope, the above code will make optimal use of the envelope buffer no matter what its size.

Enveloping with Many Enveloping Attributes

There may be a special-case condition when you begin the enveloping which occurs if you have added a large number of password, encryption, or keying attributes to the envelope so that the header prepended to the enveloped data is particularly large. For example if you encrypt a message with different keys or passwords for several dozen recipients, the header information for all the keys could become large enough that it occupies a noticeable portion of the envelopes buffer. In this case you can push in a small amount of data to flush out the header information, and then push and pop data as usual:

```
add many password/encryption/keying attributes;
push a small amount of data;
pop data;
loop
   push data;
   pop data;
```

If you use this strategy then you can trim the difference between the envelope buffer size and the amount of data you push in at once down to about 1K; the 4K difference shown earlier took into account the fact that a little extra data would be generated the first time data was pushed due to the overhead of adding the envelope header:

```
CRYPT_ENVELOPE cryptEnvelope;
void *buffer;
int bufferCount;

/* Create the envelope and add many passwords */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password1, password1Length );
   /* ... */
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password100, password100Length );

buffer = malloc( BUFFER_SIZE );

/* Read up to 100 bytes from the input file, push it into the envelope
   to flush out the header data, and write all the data in the
   envelope to the output file */
bufferCount = readFile( inputFile, buffer, 100 );
cryptPushData( cryptEnvelope, buffer, bufferCount, &bytesCopied );
cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
writeFile( outputFile, buffer, bytesCopied );

/* Process the entire file */
while( !endOfFile( inputFile ) )
   {
   int bytesCopied;

   /* Read a BUFFER_SIZE block from the input file, envelope it, and
      write the result to the output file */
   bufferCount = readFile( inputFile, buffer, BUFFER_SIZE );
   cryptPushData( cryptEnvelope, buffer, bufferCount, &bytesCopied );
   cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
   writeFile( outputFile, buffer, bytesCopied );
   }

/* Flush the last lot of data out of the envelope */
cryptPushData( cryptEnvelope, NULL, 0 );
cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
if( bytesCopied )
   writeFile( outputFile, buffer, bytesCopied );
free( buffer );

cryptDestroyEnvelope( cryptEnvelope );
```

In the most extreme case (hundreds or thousands of resources added to an envelope), the header could fill the entire envelope buffer, and you would need to pop the initial data in multiple sections before you could process any more data using the usual push/pop loop. If you plan to use this many resources, it's better to specify the use of a larger envelope buffer using CRYPT_ATTRIBUTE_BUFFERSIZE in order to eliminate the need for such special-case processing for the header.

Deenveloping data which has been enveloped with multiple keying resources also has special requirements and is covered in the next section.

Advanced Enveloping

The previous chapter covered basic enveloping concepts and simple password-based enveloping. Extending beyond these basic forms of enveloping, you can also envelope data using public-key encryption or digitally sign the contents of the envelope. These types of enveloping require the use of public and private keys which are explained in various other chapters which cover key generation, key databases, and certificates.

cryptlib automatically manages objects such as public and private keys and keysets, so you can destroy them as soon as you've pushed them into the envelope. Although the object will appear to have been destroyed, the envelope maintains its own reference to it which it can continue to use for encryption or signing. This means that instead of the obvious:

```
create the key object;
create the envelope;
add the key object to the envelope;
push data into the envelope;
pop encrypted data from the envelope;
destroy the envelope;
destroy the key object;
```

it's also quite safe to use something like:

```
create the envelope;
create the key object;
add the key object to the envelope;
destroy the key object;
push data into the envelope;
pop encrypted data from the envelope;
destroy the envelope;
```

Keeping an object active for the shortest possible time makes it much easier to track, it's a lot easier to let cryptlib manage these things for you by handing them off to the envelope.

Public-Key Encrypted Enveloping

Public-key based enveloping works just like password-based enveloping except that instead of adding a password attribute you add a public key or certificate (when encrypting) or a private decryption key (when decrypting). For example if you wanted to encrypt data using a public key contained in `pubKeyContext`, you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Add the public key */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PUBLICKEY,
   pubKeyContext );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

You can also use a certificate in place of the public key, the envelope will handle both in the same way. The certificate is typically obtained by reading it from a keyset, either directly using **cryptGetPublicKey** as described in "Reading a Key from a Keyset" on page 52, or by setting the CRYPT_ENVINFO_RECIPIENT attribute as

described in “S/MIME Enveloping” on page 134. Using the CRYPT_ENVINFO_-RECIPIENT attribute is the preferred option since it lets cryptlib handle a number of the complications which arise from reading keys for you.

De-enveloping is slightly more complex since, unlike password-based enveloping, there are different keys used for enveloping and de-enveloping. In the simplest case if you know in advance which private decryption key is required to decrypt the data, you can add it to the envelope in the same way as with password-based enveloping:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and the private decryption key required
   to deenvelope it, and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PRIVATEKEY,
   privKeyContext );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

cryptDestroyEnvelope( cryptEnvelope );
```

Although this leads to very simple code, it’s somewhat awkward since you may not know in advance which private key is required to decrypt a message. To make the private key handling process easier, cryptlib provides the ability to automatically fetch decryption keys from a private key keyset for you, so that instead of adding a private key, you add a private key keyset object and cryptlib takes care of obtaining the key for you. Alternatively, you can use a crypto device such as a smart card or Fortezza card to perform the decryption.

Using a private key from a keyset is slightly more complex than pushing in the private key directly since the private key stored in the keyset is usually encrypted or PIN-protected and will require a password or PIN supplied by the user to access it. This means that you have to supply a password to the envelope before the private key can be used to decrypt the data in it. This works as follows:

```
create the envelope;
add the decryption keyset;
push encrypted data into the envelope;
if( required resource = private key )
   add password to decrypt the private key;
pop decrypted data from the envelope;
destroy the envelope;
```

When you add the password, cryptlib will use it to try to recover the private key stored in the keyset you added previously. If the password is incorrect, cryptlib will return CRYPT_ERROR_WRONGKEY, otherwise it will recover the private key and then use that to decrypt the data. The full code to decrypt public-key enveloped data is therefore:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_ATTRIBUTE_TYPE requiredAttribute;
int bytesCopied, status;

/* Create the envelope and add the private key keyset and data */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_DECRYPT,
   privKeyKeyset );
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );

/* Find out what we need to continue and, if it's a private key, add
   the password to recover it from the keyset */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_CURRENT_COMPONENT, &
   requiredAttribute );
if( requiredAttribute != CRYPT_ENVINFO_PRIVATEKEY )
   /* Error */
```

```
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Pop the data and clean up */
cryptPopData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptDestroyEnvelope( cryptEnvelope );
```

In the unusual case where the private key isn't protected by a password or PIN, there's no need to add the password since cryptlib will use the private key as soon as you access the attribute information by reading it using **cryptGetAttribute**.

In order to ask the user for a password, it can be useful to know the name or label attached to the private key so you can display it as part of the password request message. You can obtain the label for the required private key by reading the envelope's CRYPT_ENVINFO_PRIVATEKEY_LABEL attribute:

```
char label[ CRYPT_MAX_TEXTSIZE + 1 ];
int labelLength;

cryptGetAttributeString( cryptEnvelope,
   CRYPT_ENVINFO_PRIVATEKEY_LABEL, label, &labelLength );
label[ labelLength ] = '\0';
```

You can then use the key label when you ask the user for the password for the key.

Using a crypto device to perform the decryption is somewhat simpler since the PIN will already have been entered after **cryptDeviceOpen** was called, so there's no need to supply it as CRYPT_ENVINFO_PASSWORD. To use a crypto device, you add the device in place of the private key keyset:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_ATTRIBUTE_TYPE requiredAttribute;
int bytesCopied, status;

/* Create the envelope and add the crypto device and data */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_DECRYPT,
   cryptDevice );
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );

/* Find out what we need to continue.  Since we've told the envelope
   to use a crypto device, it'll perform the decryption as soon as we
   ask it to using the device, so we shouldn't have to supply anything
   else */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_CURRENT_COMPONENT, &
   requiredAttribute );
if( requiredAttribute != CRYPT_ATTRIBUTE_NONE )
   /* Error */
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Pop the data and clean up */
cryptPopData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptDestroyEnvelope( cryptEnvelope );
```

Note how **cryptGetAttribute** now reports that there's nothing further required (since the envelope has used the private key in the crypto device to performed the decryption), and you can continue with the de-enveloping process.

Code which can handle the use of either a private key keyset or a crypto device for the decryption is a straightforward extension of the above:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_ATTRIBUTE_TYPE requiredAttribute;
int bytesCopied, status;

/* Create the envelope and add the keyset or crypto device and data */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_DECRYPT,
   cryptKeysetOrDevice );
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
```

```
/* Find out what we need to continue.  If what we added was a crypto
   device, the decryption will occur once we query the envelope.  If
   what we added was a keyset, we need to supply a password for the
   decryption to happen */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_CURRENT_COMPONENT, &
   requiredAttribute );
if( requiredAttribute != CRYPT_ATTRIBUTE_NONE )
   {
   char label[ CRYPT_MAX_TEXTSIZE + 1 ];
   int labelLength;

   if( requiredAttribute != CRYPT_ENVINFO_PASSWORD )
      /* Error */

   /* Get the label for the private key and obtain the required
      password from the user */
   cryptGetAttributeString( cryptEnvelope,
      CRYPT_ENVINFO_PRIVATEKEY_LABEL, label, &labelLength );
   label[ labelLength ] = '\0';
   getPassword( label, password, &passwordLength );

   /* Add the password required to decrypt the private key */
   cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
      password, passwordLength );
   }
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Pop the data and clean up */
cryptPopData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptDestroyEnvelope( cryptEnvelope );
```

Digitally Signed Enveloping

Digitally signed enveloping works much like the other enveloping types except that instead of adding an encryption or decryption attribute you supply a private signature key (when enveloping) or a public key or certificate (when de-enveloping). For example if you wanted to sign data using a private signature key contained in `sigKeyContext`, you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Add the signing key */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

The signature key could be a native cryptlib key, but it could also be a key from a crypto device such as a smart card or Fortezza card. They both work in the same way for signing data.

As with public-key based enveloping, verifying the signed data requires a different key for this part of the operation, in this case a public key or key certificate. In the simplest case if you know in advance which public key is required to verify the signature, you can add it to the envelope in the same way as with the other envelope types:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
```

```
/* Add the enveloped data and the signature check key required to
   verify the signature, and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigCheckKeyContext );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

cryptDestroyEnvelope( cryptEnvelope );
```

Although this leads to very simple code, it's somewhat awkward since you may not know in advance which public key or key certificate is required to verify the signature on the message. To make the signature verification process easier, cryptlib provides the ability to automatically fetch signature verification keys from a public-key keyset for you, so that instead of supplying a public key or key certificate, you add a public-key keyset object before you start de-enveloping and cryptlib will take care of obtaining the key for you. This option works as follows:

```
create the envelope;
add the signature check keyset;
push signed data into the envelope;
pop plain data from the envelope;
if( required resource = signature check key )
   read signature verification result;
```

The full code to verify signed data is therefore:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_ENVINFO_TYPE requiredResource;
int bytesCopied, signatureResult, status;

/* Create the envelope and add the signature check keyset */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_SIGCHECK,
   sigCheckKeyset );

/* Push in the signed data and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

/* Determine the result of the signature check */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_RESULT,
   &signatureResult );
```

The signature result will typically be CRYPT_OK (the signature verified), CRYPT_ERROR_SIGNATURE (the signature did not verify), or CRYPT_ERROR_NOTFOUND (the key needed to check the signature wasn't found in the keyset).

Enveloping with Multiple Attributes

Sometimes enveloped data can have multiple sets of attributes applied to it, for example encrypted data might be encrypted with two different passwords to allow it to be decrypted by two different people:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CRYPTLIB );

/* Add two different passwords to the envelope */
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password1, password1Length );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password2, password2Length );
```

```
/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

In this case either of the two passwords can be used to decrypt the data. This can be extended indefinitely, so that 5, 10, 50, or 100 passwords could be used (of course with 100 different passwords able to decrypt the data, it's questionable whether it's worth the effort of encrypting it at all, however this sort of multiuser encryption could be useful for public-key encrypting messages sent to collections of people such as mailing lists). The same applies for public-key enveloping, in fact the various encryption types can be mixed if required so that (for example) a private decryption key or a password could be used to decrypt data.

When deenveloping data which has been enveloped with a choice of multiple attributes, cryptlib builds a list of the attributes required to decrypt the data and allows you to query the required attribute information and choose the one you want to work with.

Envelope Attribute Cursor Management

The attributes required for deenveloping are managed through the use of an envelope attribute cursor which cryptlib maintains for each envelope object. You can set or move the cursor either to an absolute position or relative to the current position.

You move the cursor by setting an envelope attribute which tells cryptlib where to move the envelope attribute cursor. This attribute, identified by CRYPT_-ENVINFO_CURRENT_COMPONENT takes as value a cursor movement code which moves the cursor either to an absolute position (the first or last required attribute) or relative to its current position. The movement codes are:

Code	Description
CRYPT_CURSOR_FIRST	Move the cursor to the first attribute.
CRYPT_CURSOR_LAST	Move the cursor to the last attribute.
CRYPT_CURSOR_NEXT	Move the cursor to the next attribute.
CRYPT_CURSOR_PREV	Move the cursor to the previous attribute.

For example to move the cursor to the first required attribute you would use:

```
cryptSetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   CRYPT_CURSOR_FIRST );
```

To advance the cursor to the next required attribute you would use:

```
cryptSetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   CRYPT_CURSOR_NEXT );
```

To obtain the type of required attribute at the current cursor position you would use:

```
CRYPT_ATTRIBUTE_TYPE requiredAttribute;

cryptSetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   &requiredAttribute );
```

The attribute cursor provides a convenient mechanism for stepping through every required attribute which is present in an envelope to obtain information about it. To iterate through each required decryption attribute when de-enveloping encrypted data you would use:

```
if( cryptSetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
```

```
    CRYPT_ATTRIBUTE_TYPE requiredAttribute;

    /* Get the type of the required attribute at the cursor position
       */
    cryptGetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
       &requiredAttribute );

    /* Handle the attribute if possible */
    /* ... */
    }
 while( cryptSetAttribute( envelope,
    CRYPT_ENVINFO_CURRENT_COMPONENT, CRYPT_CURSOR_NEXT ) == CRYPT_OK
    );
```

As soon as one of the attributes required to continue is added to the envelope, cryptlib will delete the required-attribute list and continue, so the attempt to move the cursor to the next entry in the list will fail and the program will drop out of the loop.

Iterating through each required signature attribute when de-enveloping signed data is similar, but instead of trying to provide the necessary decryption information you would provide the necessary signature check information (if requested) and display the resulting signature information. Unlike encryption deenveloping attributes, cryptlib won't delete the signature information once it has been processed, so you can re-read the information multiple times.

Processing Multiple Deenveloping Attributes

The previous section explained how to step through the list of required attributes to find one which can be processed to allow the enveloped data to be decrypted. All that's left to do is to plug in the appropriate handler routines to manage each attribute requirement which could be encountered. For example to try a password against all of the possible passwords which might decrypt the message which was enveloped above, you would use:

```
int status

/* Get the decryption password from the user */
password = ...;

if( cryptSetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      CRYPT_ATTRIBUTE_TYPE requiredAttribute;

      /* Get the type of the required attribute at the cursor position
         */
      cryptGetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
         &requiredAttribute );

      /* Make sure we really do require a password resource */
      if( requiredAttribute != CRYPT_ENVINFO_PASSWORD )
         /* Error */

      /* Try the password.  If everything is OK, we'll drop out of the
         loop */
      status = cryptSetAttributeString( envelope,
         CRYPT_ENVINFO_PASSWORD, password, passwordLength );
      }
   while( status == CRYPT_WRONGKEY && \
          cryptSetAttribute( envelope,
             CRYPT_ENVINFO_CURRENT_COMPONENT, CRYPT_CURSOR_NEXT ) ==
             CRYPT_OK );
```

This steps through each required attribute in turn and tries the supplied password to see if it matches. As soon as the password matches, the data can be decrypted, and we drop out of the loop and continue the de-enveloping process.

To extend this a bit further, let's assume that the data could be enveloped using a password or a public key (requiring a private decryption key to decrypt it, either one from a keyset or a crypto device such as a smart card or Fortezza card). The code inside the loop above then becomes:

```
CRYPT_ATTRIBUTE_TYPE requiredAttribute;

/* Get the type of the required resource at the cursor position */
cryptGetAttribute( envelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   &requiredAttribute );

/* If the decryption is being handled via a crypto device, we don't
   need to take any further action, the data has already been
   decrypted */
if( requiredAttribute != CRYPT_ATTRIBUTE_NONE )
   {
   /* Make sure we really do require a password attribute */
   if( requiredAttribute != CRYPT_ENVINFO_PASSWORD && \
       requiredAttribute != CRYPT_ENVINFO_PRIVATEKEY )
      /* Error */

   /* Try the password.  If everything is OK, we'll drop out of the
      loop */
   status = cryptSetAttributeString( envelope, CRYPT_ENVINFO_PASSWORD,
      password, passwordLength );
   }
```

If what's required is a CRYPT_ENVINFO_PASSWORD, cryptlib will apply it directly to decrypt the data. If what's required is a CRYPT_ENVINFO_-PRIVATEKEY, cryptlib will either use the crypto device to decrypt the data if it's available, or otherwise use the password to try to recover the private key from the keyset and then use that to decrypt the data.

Nested Envelopes

Sometimes it may be necessary to apply multiple levels of processing to data, for example you may want to both sign and encrypt data. cryptlib allows enveloped data to be arbitrarily nested, with each nested content type being either further enveloped data or (finally) the raw data payload. For example to sign and encrypt data you would do the following:

```
create the envelope;
add the signature key;
push in the raw data;
pop out the signed data;
destroy the envelope;

create the envelope;
add the encryption key;
push in the previously signed data;
pop out the signed, encrypted data;
destroy the envelope;
```

This nesting process can be extended arbitrarily with any of the cryptlib content types.

Since cryptlib's enveloping isn't sensitive to the content type (that is, you can push in any type of data and it'll be enveloped in the same way), you need to notify cryptlib of the actual content type being enveloped if you're using nested envelopes. You can set the content type being enveloped using the CRYPT_ENVINFO_CONTENTTYPE attribute, giving as value the appropriate CRYPT_CERTINFO_CONTENT_*type*. For example to specify that the data being enveloped is signed data, you would use:

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_CONTENTTYPE,
   CRYPT_CERTINFO_CONTENT_SIGNEDDATA );
```

The default content type is plain data, so if you don't explicitly set a content type cryptlib will assume it's just raw data.

Using the nested enveloping example shown above, the full enveloping procedure would be:

```
create the envelope;
add the signature key;
(cryptlib sets the content type to the default 'plain data')
push in the raw data;
```

```
pop out the signed data;
destroy the envelope;

create the envelope;
set the content type to 'signed data';
add the encryption key;
push in the previously signed data;
pop out the signed, encrypted data;
destroy the envelope;
```

This will mark the innermost content as plain data (the default), the next level as signed data, and the outermost level as encrypted data.

Unwrapping nested enveloped data is the opposite of the enveloping process. For each level of enveloped data, you can obtain its type (once you've pushed enough of it into the envelope to allow cryptlib to decode it) by reading the CRYPT_ENVINFO_CONTENTTYPE attribute:

```
CRYPT_ATTRIBUTE_TYPE contentType;

cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_CONTENTTYPE,
   &contentType );
```

Processing nested enveloped data therefore involves unwrapping successive layers of data until you finally reach the raw data content type.

Key Databases

The most direct way to load a public or private key into an encryption object is by setting the CRYPT_CTXINFO_KEY attribute as described in "Loading Keys into Encryption Contexts" on page 60. However this method is rather clumsy, requires detailed knowledge of the key format and parameters, and isn't available in some environments like Delphi and Visual Basic. A much easier way to work with public and private keys is to store them in a keyset, an abstract container which can hold one or more keys. In practice a keyset might be a cryptlib key file, a PGP keyring, a relational database, an LDAP directory (using a standard or SSL-protected link), a URL accessed via HTTP, or a smart card containing a key. cryptlib accesses all of these keyset types using a uniform interface which hides all of the background details of the underlying keyset implementations.

Creating/Destroying Keyset Objects

Keysets are accessed as keyset objects which work in the same general manner as the other container objects used by cryptlib. You create a keyset object with **cryptKeysetOpen**, specifying the type of keyset you want to attach it to, the location of the keyset, and any special options you want to apply for the keyset. This opens a connection to the keyset. Once you've finished with the keyset, you use **cryptKeysetClose** to sever the connection and destroy the keyset object:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, keysetType, keysetLocation,
   keysetOptions );

/* Load/store keys */

cryptKeysetClose( cryptKeyset );
```

The available keyset types are:

Keyset Type	Description
CRYPT_KEYSET_FILE	A flat-file keyset, either a cryptlib key file or a PGP keyring.
CRYPT_KEYSET_HTTP	URL specifying the location of a certificate or CRL.
CRYPT_KEYSET_LDAP	LDAP directory using a standard or SSL-protected link.
CRYPT_KEYSET_SMARTCARD	Smart card key carrier.
CRYPT_KEYSET_MSQL	MSQL RDBMS.
CRYPT_KEYSET_MYSQL	MySQL RDBMS.
CRYPT_KEYSET_ODBC	Generic ODBC interface.
CRYPT_KEYSET_ORACLE	Oracle RDBMS.
CRYPT_KEYSET_POSTGRES	Postgres RDBMS.

These keyset types are covered in more detail below.

The keyset options are:

Keyset Option	Description
CRYPT_KEYOPT_CREATE	Create a new keyset. This option is only valid for writeable keyset types, which includes keysets implemented as relational databases, cryptlib key files, and some smart cards.
CRYPT_KEYOPT_NONE	No special access options (this option

Keyset Option	Description
	implies read/write access).
CRYPT_KEYOPT_READONLY	Read-only keyset access. This option is automatically enabled by cryptlib for keyset types which have read-only restrictions enforced by the nature of the keyset, the operating system, or user access rights. Unless you specifically require write access to the keyset, you should use this option since it allows cryptlib to optimise its buffering and access strategies for the keyset.

These options are also covered in more detail below.

The `keysetLocation` varies depending on the keyset type and is explained in more detail below. Note that the CRYPT_KEYSET is passed to **cryptKeysetOpen** by reference, as the function modifies it when it creates the keyset object. In all other routines, CRYPT_KEYSET is passed by value.

More details on opening connections to each type of keyset are given below.

File Keysets

For cryptlib key files and PGP keyrings, the keyset location is the path to the disk file. For example to open a connection to a PGP public keyring located in /usr/pub/, you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE,
   "/usr/pub/pubring.pgp", CRYPT_KEYOPT_READONLY );
```

cryptlib will automatically determine the file type and access it in the appropriate manner. As another example, to open a connection to a cryptlib key located in the KEYS share on the server FILESERVER, you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE,
   "\\FILESERVER\KEYS\KEY.P15", CRYPT_KEYOPT_READONLY );
```

For PGP keysets, cryptlib will automatically set the access mode to read-only even if you don't specify the CRYPT_KEYOPT_READONLY option, since writes to this keyset type aren't supported. If you try to write a key to this keyset type, cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you don't have permission to write to the file. The only file keyset type which can be written to is a cryptlib private key file. This keyset contains a one or more (usually encrypted) private keys and certificates. To create a new cryptlib keyset you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE, "Private key
   file.p15", CRYPT_KEYOPT_CREATE );
```

If a cryptlib keyset of the given name already exists and you open it with CRYPT_-KEYOPT_CREATE, cryptlib will erase it before creating a new one in its place. The erasure process involves overwriting the original keyset with random data and committing the write to disk to ensure that the data really is overwritten, truncating its length to 0 bytes, resetting the file timestamp and attributes, and deleting the file to ensure that no trace of the previous key remains. The new keyset is then created in its place.

For security reasons, cryptlib won't overwrite an existing file if it isn't a normal file (for example if it's a hard or symbolic link, if it's a device name, or if it has other unusual properties such as having a stream `fattach()`'d to it).

Where the operating system supports it, cryptlib will set the security options on the keyset so that only the person who created it (and, in some cases, the system administrator) can access it. For example under Unix the file access bits are set to allow only the file owner to access the file, and under Windows NT the files access control list is set so that only the user who owns the file can access or change it. Since not even the system administrator can access the keyset under Windows NT, the user may need to manually enable access for others to allow the file to be backed up or copied.

When you open a keyset which contains private keys, you should bind the it to the current thread for added security to ensure that no other threads can access the file or the keys read from it:

```
CRYPT_KEYSET cryptKeyset;

/* Open a keyset and claim it for exclusive use */
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE, "Private key file",
   CRYPT_KEYOPT_READONLY );
cryptSetAttribute( cryptKeyset, CRYPT_PROPERTY_OWNER, threadID );
```

You can find out more about binding objects to threads in "Object Security" on page 16.

HTTP Keysets

For keys accessed via a web page URL, there's no need to specify a keyset name since the certificate or CRL to be fetched is implicitly specified by the URL. Because of this, HTTP keysets don't have names:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_HTTP, NULL,
   CRYPT_KEYOPT_READONLY );
```

For each certificate or CRL that you want to fetch you give the location (URL) as the key name:

```
cryptGetPublicKey( cryptKeyset, &cryptCert, CRYPT_KEYID_NAME, url );
```

Once you've created an HTTP keyset object, you can read multiple keys through it by specifying a new URL for each key read.

cryptlib provides various HTTP-related configuration options which allow you to specify the use of an HTTP proxy used to access the Internet, and control the timeout when accessing a URL. These options are:

Configuration Option	Description
CRYPT_OPTION_KEYS_- HTTP_PROXY	The HTTP proxy to use for Internet access, defaulting to none.
CRYPT_OPTION_KEYS_- HTTP_TIMEOUT	The timeout when reading a certificate or CRL via HTTP, defaulting to 60 seconds.

The CRL's provided by some CA's can become quite large, so you may need to play with timeouts in order to allow the entire CRL to be downloaded if the link is slow or congested.

LDAP Keysets

For keys stored in an LDAP directory, the keyset location is the name of the LDAP server, with an optional port if access is via a nonstandard port. For example if the LDAP server was called `directory.ldapserver.com`, you would access the keyset with:

```
CRYPT_KEYSET cryptKeyset;
```

```
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_LDAP,
   "directory.ldapserver.com", CRYPT_KEYOPT_READONLY );
```

If the server is configured to allow access on a nonstandard port, you can append the port to the server name in the usual manner for URL's. For example if the server mentioned above listened on port 8389 instead of the usual 389 you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_LDAP,
   "directory.ldapserver.com:8389", CRYPT_KEYOPT_READONLY );
```

You can also optionally include the `ldap://` or `ldaps://` protocol specifiers in the URL, these are ignored by cryptlib.

The storage of certificates in LDAP directories is currently somewhat haphazard and vendor-dependent, and you may need to adjust cryptlib's LDAP configuration options to work with a particular vendors idea of how certificates and CRL's should be stored on a server. In order to make it easier to adapt cryptlib to work with different vendors ways of storing information in a directory, cryptlib provides various LDAP-related configuration options which allow you to specify the X.500 objects and attributes used for certificate storage. These options are:

Configuration Option	Description
CRYPT_OPTION_KEYS_- LDAP_CERTNAME CRYPT_OPTION_KEYS_- LDAP_CACERTNAME	The X.500 attribute which certificates are stored as. For some reason certificates belonging to certification authorities (CA's) are stored under their own attribute type, so if a search for a certificate fails cryptlib will try again using the CA certificate attribute (there's no easy way to tell in advance how a certificate will be stored, so it's necessary to do it this way). The default settings for these options are `userCertificate;binary` and `cACertificate;binary`. Note the use of the `binary` qualifier, this is required for a number of directories which would otherwise try and encode the returned information as text rather than returning the raw certificate.
CRYPT_OPTION_KEYS_- LDAP_CRLNAME	The X.500 attribute which certificate revocation lists (CRL's) are stored as, defaulting to `certificateRevocationList;binary`.
CRYPT_OPTION_KEYS_- LDAP_EMAILNAME	The X.500 attribute which email addresses are stored as, defaulting to `emailAddress`. Since X.500 never defined an email address attribute, various groups defined their own ones, `emailAddress` is the most common one.
CRYPT_OPTION_KEYS_- LDAP_OBJECTCLASS	The X.500 object class, defaulting to `inetOrgPerson`.

The default settings used by cryptlib have been chosen to have the best chance of working with the most widely-deployed LDAP servers currently in use.

Relational Database Keysets

For keys stored in a relational database, the keyset location is the access path to the database. The nature of the access path depends on the database type, and ranges

from an alias or label which identifies the database (for example an ODBC data source) through to a complex combination of the name or address of the server which contains the database, the name of the database on the server, and the user name and password required to access the database. In some cases you may need to use **cryptKeysetOpenEx** to access keysets which require the more complex types of access parameters.

The exact keyset type also depends on the operating system with which cryptlib is being used. Under Windows 3.x, Windows'95/98, and Windows NT, all database keyset types are accessed as ODBC data sources with the keyset type CRYPT_KEYSET_ODBC. Under Unix, which doesn't provide a general vendor-independant database access system, database keyset types are accessed in a manner which specifies the database type being used, for example an Oracle database keyset would be accessed using a keyset type of CRYPT_KEYSET_ORACLE. With some systems such as DOS, which don't support easy external database access, cryptlib can't be used with a database keyset and is restricted to the simpler keyset types such as cryptlib private key files and PGP keyrings.

The simplest type of keyset to access is a local database which requires no extra parameters such as a user name or password. An example of this is an ODBC data source on the local machine. Let's assume that the keyset is stored in an MS Access database which is accessed through the "PublicKeys" data source. This keyset is accessed with:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_ODBC, "PublicKeys",
   CRYPT_KEYOPT_READONLY );
```

The CRYPT_KEYSET_ODBC keyset type is used to access any keyset which is configurable as an ODBC data source (which in practice means virtually any kind of database, although some of the more primitive legacy formats will have trouble storing the long records required to hold public keys).

Some databases allow a collection of parameters to be specified by combining them into an access path with special delimiters. For example Oracle databases allow an access path to take the form `user@server:name`, so you could access a keyset stored in the Oracle database "services" located on the server "dbhost" with the user name "system" using:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_ORACLE,
   "system@dbhost:services", CRYPT_KEYOPT_READONLY );
```

An alternative way to do this is to use **cryptKeysetOpenEx**, which allows the individual parameters to be specified separately.

In the examples shown above, the keyset was opened with the CRYPT_KEYOPT_-READONLY option. The use of this option is recommended when you will use the keyset to retrieve a key but not store one (which is usually the case) since it allows cryptlib to optimise its transaction management with the database backend. This can lead to significant performance improvements due to the different data buffering and locking strategies which can be employed if it is known that the database won't be updated. If you try to write a key to a keyset which has been opened in read-only mode, cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you don't have permission to write to the database.

To create a new key database, you can use the CRYPT_KEYOPT_CREATE flag. If a keyset of the given name already exists, cryptlib will return CRYPT_ERROR_-DUPLICATE, otherwise it will create a new key database ready to have keys added to it.

Smart Card Keysets

For cryptlib private key keysets stored on smart cards, the keyset location is the name of the smart card driver interface. For example to open a connection to a private key stored on a basic memory card in a Gemplus reader, you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_SMARTCARD, "Gemplus",
   CRYPT_KEYOPT_READONLY );
```

The interface name isn't case sensitive, so you could specify it as "Gemplus", "GEMPLUS", or "gemplus".

To create a new keyset containing a private key on the card, you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_SMARTCARD, "Gemplus",
   CRYPT_KEYOPT_CREATE );
```

If a keyset already exists on the card, cryptlib will erase it before creating a new keyset in its place.

cryptlib supports the following smart card interfaces:

Interface	Details
ASE	Aladdin readers accessed via the Aladding Smartcard Environment (ASE). The interface defaults to using the default Aladdin reader type which is set by the ASE software.
Auto	This is a generic reader interface which works with any card reader which is accessed directly through the serial port (most readers require special drivers and can't be accessed directly). The interface defaults to using a reader connected to the COM2 serial port under Windows or `/dev/ttyb` or `/dev/ttyS1` under Unix (depending on the Unix variant), with an ISO 7816 memory card.
Gemplus	Gemplus card readers accessed via the Gemplus drivers. The interface defaults to using a GCR 410 reader connected to the COM2 serial port, with automatic card type detection.
Towitoko	Towitoko card readers accessed via the Towitoko drivers. The interface defaults to using the default Towitoko reader which is set by the driver software.

Since many reader types will require further parameters to identify the exact reader and card type, and possibly reader communications parameters, you may need to use **cryptKeysetOpenEx** to open a connection to these keyset types.

cryptlib is structured to allow easy support for many types of smart cards to be added to the keyset interface, due to the lack of standard card types and driver interfaces (the interfaces which are supported by manufacturers aren't standardised, and the interfaces which are standardised aren't supported by manufacturers), most new cards and readers require custom code to be added to cryptlib to interface them with the keyset routines. If you require specific support for a particular card type or card reader, please contact the cryptlib developers. For crypto cards, the preferred interface is the encryption device interface covered in "Encryption Devices and Modules" on page 140.

When you open a smart card keyset, you should bind the keyset to the current thread for added security to ensure that no other threads can access the card or the keys read from it:

```
CRYPT_KEYSET cryptKeyset;
```

```
/* Open a keyset and claim it for exclusive use */
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_SMARTCARD, "Gemplus",
   CRYPT_KEYOPT_READONLY );
cryptSetAttribute( cryptKeyset, CRYPT_PROPERTY_OWNER, threadID );
```

You can find out more about binding objects to threads in "Object Security" on page 16.

Note that private keys typically range in size from 500-2K bytes, which means that the smaller memory cards don't have enough capacity to store the entire key. If the write to the card fails, cryptlib will erase the card before returning an error code to ensure that no traces of the partially-written key remain on the card.

Extended Keyset Initialisation

The **cryptKeysetOpen** function has a companion function **cryptKeysetOpenEx** which may be used to perform an extended open on a keyset. This is needed for some types of database and LDAP keysets which require database and server names and possibly a user name and password, and by some smart card types which require extended information about card types and card reader variants. If a particular parameter isn't needed, you can set it to null and cryptlib will ignore it.

LDAP Keysets

For keys stored in an LDAP directory, the extra parameters which can be supplied using **cryptKeysetOpenEx** are the the user name, the user password, and information needed for an SSL connection to the LDAP server. Using the previous example of an LDAP directory located at `directory.ldapserver.com`, the **cryptKeysetOpenEx** version of the call would be:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_LDAP,
   "directory.ldapserver.com", "username", "password", NULL,
   CRYPT_KEYOPT_READONLY );
```

which specifies the user name and password for connecting to the server rather than using the basic anonymous connection name is used with **cryptKeysetOpen**. This provides a standard, unsecured connection to the LDAP server.

If the server requires the use of an SSL connection for security, you would supply the name of the SSL information as the last parameter in place of the null pointer. For example for Netscape LDAP client access you need to specify the location of the client certificate database:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_LDAP,
   "directory.ldapserver.com", "username", "password",
   "/users/mozilla/.netscape/cert5.db", CRYPT_KEYOPT_READONLY );
```

If the server allows anonymous access over an SSL connection, you would omit the user name and password and only provide the SSL information:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_LDAP,
   "directory.ldapserver.com", NULL, NULL,
   "/users/mozilla/.netscape/cert5.db", CRYPT_KEYOPT_READONLY );
```

This gives more control over access to the keyset than that provided by the simpler **cryptKeysetOpen** form.

Relational Database Keysets

For keys stored in a relational database, the extra parameters which can be supplied using **cryptKeysetOpenEx** are the server name, the user name, and the user password. Using the previous example of a keyset stored in an Oracle database, the **cryptKeysetOpenEx** version of the call would be:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_ORACLE, "dbhost",
   "services", "system", NULL, CRYPT_KEYOPT_READONLY );
```

which specifies the database name, server, and user name as separate parameters instead of using the unified `"system@dbhost:services"` name which was used with **cryptKeysetOpen**. If the database requires a password to access it alongside the parameters given above, you would supply the password as the last parameter in place of the null pointer.

If the keyset were stored in the Postgres "keys" database on the local machine, you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_POSTGRES, "localhost",
   "keys", NULL, NULL, CRYPT_KEYOPT_READONLY );
```

cryptKeysetOpenEx also allows extended control over access to keyset types which would normally be accessed using **cryptKeysetOpen**. For example if the keyset were stored in an SQL Server database accessed through the ODBC data source "ServerKeys" with the user name "Key User" and the password "password", you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_ODBC, NULL,
   "ServerKeys", "Key Users", "password", CRYPT_KEYOPT_READONLY );
```

This gives more control over access to the keyset than that provided by the simpler **cryptKeysetOpen** form.

Smart Card Keysets

For keys stored on a smart card, the extra parameters which can be supplied using **cryptKeysetOpenEx** are the reader type, card type, and reader communications parameters. The communications parameters are specified in a character string which contains the serial port the reader is connected to, the baud rate, number of data bits, parity type, and number of stop bits. Except for the Unix serial port device names, the parameters aren't case-sensitive. If you don't specify a parameter by passing in a null pointer, the default setting for that parameter will be used.

If the reader is accessed via the serial port, the communications parameters will be as follows:

Parameter	Settings
Serial port	The name of the serial port to which the reader is connected, which is COM1-4 under Windows or the serial port device name under Unix (for example `/dev/ttyS1`).
Baud rate	The baud rate at which the reader is accessed, typically 9600bps. Some readers will support access at 19,200bps, 38,400bps, or even higher.
Data bits	The number of data bits, usually 8.
Parity	The parity type, 'N' for no parity, 'E' for even parity, 'O' for odd parity.
Stop bits	The number of stop bits, usually 1.

These settings are combined into a single string by seperating them with commas. There are two ways to specify the parameters, the short form which only specifies the serial port (for example "COM3"), and the long form which specifies the entire set of parameters (for example "COM3,9600,8,N,1").

The "ASE" reader interface supports the following parameters:

Parameter	Settings
Reader type	The name of the reader as set up using the ASE software. If this parameter isn't supplied, the reader will be accessed using the port specified in the comms parameters.
Card type	This interface supports the card types "Auto" (which tries to determine which card is in the reader by scanning it), "T0" (for ISO 7816 T=0 cards), "T14" (for ISO 7816 T=14 cards), "Memory Auto" (which tries to determine which type of non-ISO 7816 memory card is in the reader by scanning it), "2-Wire" (for Siemens 2-wire protocol cards), and "I2C" (for I2C memory cards).
Comms parameters	This interface accesses readers using the short form of the standard serial parameters, which specifies only the port on which the reader is connected. The long form isn't used since the Aladdin readers use fixed serial port settings. If this parameter isn't supplied, the reader will be access using the name specified in the reader type parameters.

The "Auto" reader interface supports the following parameters:

Parameter	Settings
Reader type	This interface works with any serial-port based reader, so you should set this parameter to null.
Card type	This interface works with any ISO 7816 type card, so you should set this parameter to null.
Comms parameters	This interface accesses readers using either the short or long form of the standard serial parameters.

The "Gemplus" reader interface supports the following parameters:

Parameter	Settings
Reader type	This interface supports all Gemplus reader types, including "GCR200", "GCR400FD A", "GCR400FD B", "GCR400", "GCR500", "GCI400DC", "GCR610", "GCR680", "GCR420", "GPR", "GPR400", "GCM AUTO", "GCM CONN", "IFD140", "IFD140 200", "IFD140 400", and "IFD220" (assuming that the Gemplus driver you're using also supports the given reader type). The default setting for this parameter is "GCR 410".
Card type	This interface supports the card types "Auto" (which tries to determine which type of card is present in the reader by scanning it), "I2C" (for I2C memory cards), "ISO" and "COS" (for standard ISO 7816 cards), and "FastISO" (for double-clock-frequency ISO 7816 cards). If you definitely know the card type in advance, you should specify the exact type since this will make the initial access faster by avoiding the card scanning which is performed by the "Auto" setting. The default setting for this parameter is "Auto".
Comms parameters	This interface accesses readers using the short form of the standard serial parameters, which specifies only the port on which the reader is connected. The long form isn't used since the Gemplus drivers use fixed serial port settings. The default setting for this parameter is "COM2".

The "Towitoko" reader interface supports the following parameters:

Parameter	Settings
Reader type	This interface supports all Towitoko reader types, including "CHIPDRIVE intern", "CHIPDRIVE extern", "CHIPDRIVE extern II", "CHIPDRIVE twin", and "KartenZwerg". The default setting for this parameter is "CHIPDRIVE extern".
Card type	This interface automatically determines the correct card type using the Towitoko drivers.
Comms parameters	This interface automatically determines the correct port and comms parameters using the Towitoko drivers.

Using the earlier example of a keyset stored on a card accessed via a Gemplus reader, the **cryptKeysetOpenEx** version of the call to read a GP4K card (an I2C memory card) using a GCR500 reader connected to the default serial port would be:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpenEx( &cryptKeyset, CRYPT_KEYSET_SMARTCARD, "Gemplus",
   "GCR 500", "I2C", NULL, CRYPT_KEYOPT_READONLY );
```

Accessing a Keyset

Once you've established a connection to a keyset, you can read and write keys to it. The type of access you can perform depends on the keyset type:

Type	Access Allowed
cryptlib	Read/write access to public/private keys and certificates stored in a file, with the private key portion encrypted. This is the cryptlib native keyset format for private keys.
Crypto device	Read access to public/private keys and and read/write access to certificates stored in the device. Devices aren't general-purpose keysets but can act like them for keys contained within them.
Database	Read/write access to X.509 public-keys stored in a relational database. This is the cryptlib native keyset format for public keys and provides a fast, scalable key storage mechanism. The exact database format used depends on the platform, but would typically include any ODBC database under Windows, and Oracle, Postgres, and mSQL databases under Unix.
HTTP	Read access to X.509 certificates and CRL's accessed via URL's.
LDAP	Read/write access to X.509 certificates and CRL's stored in an LDAP directory.
PGP	Read-only access to PGP keyrings. This capability is provided for compatibility reasons, actual key storage should be performed using the cryptlib native keysets which are more flexible than a PGP keyring (for example a number of cryptlib key types can't be stored as PGP-style keys).
Smart card	Read/write access to a single private key stored (usually) in encrypted form on a smart card.

The recommended method for public key storage is to use a relational database keyset, which usually outperforms the other keyset types by a large margin, is highly scalable, and is well suited for use in cases where data is already administered through existing database servers.

Reading a Key from a Keyset

Once you've set up a connection to a keyset, you can read one or more keys from it. Some keysets such as HTTP URL's and smart cards can contain only one key, whereas cryptlib key files, PGP keyrings, relational databases, and LDAP keysets may contain multiple keys.

You can also use a crypto device such as a smart card or Fortezza card as a keyset. Reading a key from a device creates an encryption context which is handled via the crypto device, so that although it looks just like any other encryption context it uses the device to perform any encryption or signing.

The two functions which are used to read keys are **cryptGetPublicKey** and **cryptGetPrivateKey**, which get a public and private key respectively. The key to be read is identified through a key identifier, either the name or the email address of the keys' owner, specified as CRYPT_KEYID_NAME and CRYPT_KEYID_EMAIL, or the label assigned to the key when it's generated or when it's written to the keyset, also specified as CRYPT_KEYID_NAME.

cryptGetPublicKey returns a generic CRYPT_HANDLE which can be either a CRYPT_CONTEXT or a CRYPT_CERTIFICATE depending on the keyset type. Most public-key keysets will return a key certificate, but some keysets (like PGP keyrings) don't store the full certificate information and will return only an encryption context rather than a key certificate. You don't have to worry about the difference between the two, they are interchangeable in most cryptlib functions.

Obtaining a Key for a User

The rules used to match the key ID to a key depend on the keyset type, and are as follows:

Type	User ID Handling
cryptlib Smart card	The key ID is a label attached to the key when it's generated or loaded into the keyset or card, and is specified using CRYPT_KEYID_NAME.
	The key ID is matched as a substring of the label attached to the key, with the match being performed in a case-insensitive manner.
Crypto device	The key ID is a label attached to the key when it's loaded or generated into the device, and is specified using CRYPT_KEYID_NAME.
Database	The key ID is either the name or the email address of the key owner, and is matched in full in a case-insensitive manner.
HTTP	The key is implicitly specified by the URL, which can be given as either the CRYPT_KEYID_NAME or CRYPT_KEYID_EMAIL.
LDAP	The key ID is an X.500 distinguished name (DN), which is neither a name nor an email address but a peculiar construction which (in theory) uniquely identifies a key in the X.500 directory. Since a DN isn't really a name or an email address, it's possible to match an entry using either CRYPT_KEYID_NAME or CRYPT_KEYID_EMAIL.
	The key ID is matched in a manner which is controlled by the way the LDAP server is configured (usually the match is case-insensitive).
PGP	The key ID is a name with an optional email address which is usually given inside angle brackets. Since PGP keys usually combine the key owners name and email address into a single

Type	User ID Handling
	value, it's possible to match an email address using CRYPT_KEYID_NAME, and vice versa. The key ID is matched as a substring of any of the names and email addresses attached to the key, with the match being performed in a case-insensitive manner. This is the same as the matching performed by PGP. Note that, like PGP, this will return the first key in the keyset for which the name or email address matches the given key ID. This may result in unexpected matches if the key ID that you're using is a substring of a number of names or email addresses which are present in the keyring. Since email addresses are more likely to be unique than names, it's a good idea to specify the email address to guarantee a correct match.

Assuming you wanted to read Noki Crow's public key from a keyset, you would use:

```
CRYPT_HANDLE publicKey;

cryptGetPublicKey( cryptKeyset, &publicKey, CRYPT_KEYID_NAME, "Noki
   S.Crow" );
```

Note that the CRYPT_HANDLE is passed to **cryptGetPublicKey** by reference, as the function modifies it when it creates the public key context. If you knew that the keyset was a PGP keyring (which returns a CRYPT_CONTEXT rather than a CRYPT_CERTIFICATE) you could also use:

```
CRYPT_CONTEXT publicKeyContext;

cryptGetPublicKey( cryptKeyset, &publicKeyContext, CRYPT_KEYID_NAME,
   "Noki S.Crow" );
```

although the two are functionally equivalent. Reading a key from a crypto device works in an identical fashion:

```
CRYPT_HANDLE publicKey;

cryptGetPublicKey( cryptDevice, &publicKey, CRYPT_KEYID_NAME, "Noki
   S.Crow" );
```

The only real difference is that any encryption performed with the key is handled via the crypto device, although cryptlib hides all of the details so that the key looks and functions just like any other encryption context.

You can use **cryptGetPublicKey** not only on straight public-key keysets but also on private key keysets, in which case it will return the public portion of the private key or the key certificate associated with the key.

The other function which is used to obtain a key is **cryptGetPrivateKey**, which differs from **cryptGetPublicKey** in that it expects a password alongside the user ID if the key is being read from a keyset. This is required because private keys are usually stored encrypted and the function needs a password to decrypt the key. If the key is held in a crypto device (which requires a PIN or password when you open a session with it, but not when you read a key), you can pass in a null pointer in place of the password. For example if Noki Crow's email address was `noki@crow.com` and you wanted to read their private key, protected by the password "Password", from a keyset, you would use:

```
CRYPT_CONTEXT privateKeyContext;

cryptGetPrivateKey( cryptKeyset, &privateKeyContext,
   CRYPT_KEYID_EMAIL, "noki@crow.com", "Password" );
```

If you supply the wrong password to **cryptGetPrivateKey**, it will return CRYPT_-ERROR_WRONGKEY. You can use this to automatically handle the case where the key might not be protected by a password (for example if it's stored in a crypto device or a non-cryptlib keyset which doesn't protect private keys) by first trying the call without a password and then retrying it with a password if the first attempt fails

with CRYPT_ERROR_WRONGKEY. cryptlib caches key reads, so the overhead of the second key access attempt is negligible:

```
CRYPT_CONTEXT privateKeyContext;

/* Try to read the key without a password */
if( cryptGetPrivateKey( cryptKeyset, &privateKeyContext,
   CRYPT_KEYID_NAME, name, NULL ) == CRYPT_ERROR_WRONGKEY )
   {
   /* Ask the user for the keys' password and retry the read */
   password = ...;
   cryptGetPrivateKey( cryptKeyset, &privateKeyContext,
      CRYPT_KEYID_NAME, name, password );
   }
```

cryptGetPrivateKey always returns an encryption context.

General Keyset Queries

Where the keyset is implemented as a standard database, you can use cryptlib to perform general queries to obtain one or more certificates which fit a given match criterion. For example you could retrieve a list of all the keys which are set to expire within the next fortnight (to warn their owners that they need to renew them), or which belong to a company or a division within a company. You can also perform more complex queries such as retrieving all certificates from a division within a company which are set to expire within the next fortnight. cryptlib will return all certificates which match the query you provide, finally returning CRYPT_ERROR_-COMPLETE once all matching certificates have been obtained.

The general strategy for performing queries is as follows:

```
submit query
repeat
   read query result
while query status != CRYPT_COMPLETE
```

You can cancel a query in progress at any time by submitting a new query consisting of the command “cancel”.

Queries are submitted by setting the CRYPT_KEYSETINFO_QUERY attribute for a keyset, which tells it how to perform the query. Let’s look at a very simple query which is equivalent to a straight **cryptGetPublicKey**:

```
CRYPT_CERTIFICATE certificate;

cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY,
   "$email='noki@crow.com'", length );
do
   status = cryptGetPublicKey( keyset, &certificate, CRYPT_KEYID_NONE,
      NULL );
while( cryptStatusOK( status ) );
```

This will read each certificate corresponding to the given email address from the database (there should only be a single matching certificate present for the email address, so only one certificate should be returned). Note that the key ID is unused because the keys which are returned are selected by the initial query and not by the key identifier.

This example is an artificially simple one, it’s possible to submit queries of arbitrary complexity in the form of full SQL queries. Since the key databases which are being queried can have arbitrary names for the certificate attributes (corresponding to database columns), cryptlib provides a mapping from certificate attribute to database field names. An example of this mapping is shown in the code above, in which `$email` is used to specify the email address attribute, which may have a completely different name once it reaches the database backend. The certificate attribute names are as follows:

Attribute	Field
$C, $SP, $L, $O,	Certificate country, state or province, locality,

Attribute	Field
$OU, $CN	organisation, organisational unit, and common name.
$date	Certificate expiry date
$email	Certificate email address

You can use these attributes to build arbitrarily complex queries to retrieve particular groups of certificates from a key database. For example to retrieve all certificates issued for US users (obviously this is only practical with small databases) you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US'",
   length );
```

Extending this one stage further, you could retrieve all certificates issued to Californian users with:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US' AND
   $SP='CA'", length );
```

Going another step beyond this, you could retrieve all certificates issued to users in San Francisco:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US' AND
   $SP='CA' AND $L='San Francisco'", length );
```

Going even further than this, you could retrieve all certificates issued to users in San Francisco whose names begin with an ‘a’:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US' AND
   $SP='CA' AND $L='San Francisco' AND $CN LIKE 'A%'", length );
```

These queries will return the certificates in whatever order the underlying database returns them in. You can also specify that they be returned in a given order, for example to order the certificates in the previous query by user name you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US' AND
   $SP='CA' AND $L='San Francisco' ORDER BY $CN", length );
```

To return them in reverse order, you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$C='US' AND
   $SP='CA' AND $L='San Francisco' ORDER BY $CN DESCENDING", length );
```

The ability to selectively extract collections of certificates provides a convenient mechanism for implementing a heirarchical certificate database browsing capability. You can also use it to perform general-purposes queries and certificate extractions, for example to return all certificates which will expire within the next week (and which therefore need to be replaced or renewed), you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$date +
   1_week < today", length );
```

To sort the results in order or urgency of replacement, you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "$date +
   1_week < today ORDER BY $date", length );
```

To retrieve all certificates which don’t need replacement within the next week, you could negate the previous query to give:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "NOT $date +
   1_week < today", length );
```

As these examples show, cryptlibs keyset query capability provides the ability to perform arbitrary general-purpose queries on keysets.

Once a query has begun running, it can return a considerable number of certificates. If you try to initiate another query while the first one is in progress or perform a standard read, write, or delete operation, cryptlib will return a CRYPT_ERROR_-INCOMPLETE error to indicate that the query is still active. You can cancel the currently active query at any point by setting the CRYPT_KEYSETINFO_QUERY attribute to “cancel”:

```
cryptSetAttributeString( keyset, CRYPT_KEYSETINFO_QUERY, "cancel", 6
   );
```

This will clear the current query and prepare the keyset for another query or an alternative operation such as a key read, write, or delete.

Handling Multiple Certificates with the Same Name

Sometimes a keyset may contain multiple certificates issued to the same person. Whether this situation will occur varies by CA, some CA's won't issue multiple certificates with the same name, some will, and some may modify the name to eliminate conflicts (for example by adding unique ID values to the name or using middle initials to disambiguate names). If multiple certificates exist, you can use **cryptKeysetQuery** to read each in turn and try and find one which matches your requirements (for example you might be able to filter them based on key usage or some other parameter held in the certificate). The general idea here is to issue a query based on the name and then read each certificate which matches the query until you find an appropriate one:

```
cryptKeysetQuery();
while( cryptGetPublicKey() == CRYPT_OK && \
       certificate doesn't matches required usage )
      /* Continue */;
cryptSetAttributeString( "cancel" );
```

This use of general queries allows the maximum flexibility in selecting certificates in cases when multiple choices are present.

Writing a Key to a Keyset

Writing a key to a keyset isn't as complex as reading it since there's no need to specify the key identification information which is needed to read a key, however there are some restrictions on the type of key you can write to a keyset. Public-key keysets such as database and LDAP keysets store full key certificates, so the object which you write to these keysets must be a CRYPT_CERTIFICATE and not just a CRYPT_CONTEXT. In contrast, keysets such as cryptlib key files and smart cards can store public/private keys as well as certificates. If you try to write the incorrect type of object to a keyset (for example a private key to a public key keyset), cryptlib will return a CRYPT_ERROR_PARAM2 error to indicate that the object you are trying to add is of the incorrect type for this keyset.

You can't write a key to a crypto device because keys used with devices have to be either created or generated inside the device. Similarly, you can't create a key inside a standard cryptlib context and then move it to the device later on, since the security features of the device won't allow this, and you can't take a key created via a crypto device and write it to a keyset, because it can't be exported from the device. By using crypto hardware to handle your keys you're guaranteeing that the key is never exposed outside the hardware, keeping it safe from any malicious code which might be present in your system.

You can write a public key certificate to a keyset with **cryptAddPublicKey**, which takes as parameters the keyset and the key certificate to write:

```
cryptAddPublicKey( cryptKeyset, pubKeyCertificate );
```

Since all identification information is contained in the certificate, there's no need to specify any extra data such as the certificate owners name or email address.

If you try to write a key to a read-only keyset, cryptlib will return CRYPT_ERROR_-PERMISSION. If the certificate you are trying to write is already present in the keyset, cryptlib will return CRYPT_ERROR_DUPLICATE. If the keyset is a public-key keyset, you can use **cryptDeleteKey** to delete the existing certificate so you can write the new one in its place. If the keyset is a cryptlib key file, smart card, or crypto device, this would delete both the certificate and the key it corresponds to.

Writing a private key requires one extra parameter, the password which is used to encrypt the private key components. cryptlib will use the default encryption method (usually three-key triple DES) to encrypt the key with the given password, even if you're writing it to a supposedly secure medium like a smart card (even smart cards can be hacked, so the triple DES encryption offers an extra layer of security).

To write a private key to a keyset you would use the corresponding **cryptAddPrivateKey** function:

```
cryptAddPrivateKey( cryptKeyset, privKeyContext, password );
```

If you try to write a key to a read-only keyset, cryptlib will return CRYPT_ERROR_-PERMISSION. If the keyset already contains the private key, cryptlib will return CRYPT_ERROR_DUPLICATE and you will need to use **cryptDeleteKey** to delete it before you can add the new key.

Although cryptlib and PGP can work directly with private keys, other formats like X.509 certificates, S/MIME messages, and SSL require complex and convoluted naming and identification schemes for their keys. Because of this, you can't immediately use a newly-generated private key with these formats for anything other than signing a certification request or a self-signed certificate. To use it for any other purpose, you need to obtain an X.509 certificate which identifies the key. The process of obtaining a certificate and updating a keyset with it is covered in more detail in "Maintaining Keys and Certificates" on page 114. Once you've obtained the certificate, you can add it to the keyset and cryptlib will automatically associate it with the key when you try to read the key.

If you are working with a database keyset, you can also add a certificate revocation list (CRL) to the keyset. Since a CRL isn't an actual key, you can't read it back out of the keyset (there's nothing to read), but you can use it to check the revocation state of certificates. CRL's and their uses are explained in more detail in "Certificate Revocation Lists" on page 120.

Deleting a Key

Deleting a key with **cryptDeleteKey** works in the same manner as reading a key, with the key to delete being identified by a key ID in the usual manner. For example if you wanted to delete S.Crow's key from a keyset, you would use:

```
cryptDeleteKey( cryptKeyset, CRYPT_KEYID_NAME, "S.Crow" );
```

Deleting a key from a crypto device is identical:

```
cryptDeleteKey( cryptDevice, CRYPT_KEYID_NAME, "S.Crow" );
```

In the case of an LDAP directory, this will delete the entire entry, not just the certificate attribute or attributes for the entry. In the case of a cryptlib key file, smart card, or crypto device, this will delete the key and any certificates which may be associated with it. If you try to delete a key from a read-only keyset, cryptlib will return CRYPT_ERROR_PERMISSION. If the key you're trying to delete isn't present in the keyset, cryptlib will return CRYPT_ERROR_NOTFOUND.

Encryption and Decryption

Although envelope and keyset container objects provide an easy way to work with encrypted data, it's sometimes desirable to work at a lower level, either because it provides more control over encryption parameters or because it's more efficient than the use of the higher-level functions. The objects which you use for lower-level encryption functionality are encryption contexts. Internally, more complex objects such as envelope and certificate objects also use encryption contexts, although these are hidden and not accessible from the outside.

Creating/Destroying Encryption Contexts

To create an encryption context, you must specify the encryption algorithm and optionally the encryption mode you want to use for that context. The available encryption algorithms and modes are given in "Algorithms and Modes" on page 163. For example, to create and destroy an encryption context for DES you would use the following code:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, CRYPT_ALGO_DES );

/* Load key, perform en/decryption */

cryptDestroyContext( cryptContext );
```

The context will use the default encryption mode of CBC, which is the most secure and efficient encryption mode. If you want to use a different mode, you can set the context's CRYPT_CTXINFO_MODE attribute to specify the mode to use. For example to change the encryption mode used from CBC to CFB you would use:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_MODE, CRYPT_MODE_CFB );
```

In general you shouldn't need to change the encryption mode, the other cryptlib functions will automatically handle the mode choice for you. Public-key, hash, and MAC contexts work in the same way, except that they don't have different modes of use so the CRYPT_CTXINFO_MODE attribute can't be read or written for these types of contexts. The availability of certain algorithms and encryption modes in cryptlib does not mean that their use is recommended. Some are only present because they are needed for certain protocols or required by some standards.

Note that the CRYPT_CONTEXT is passed to **cryptCreateContext** by reference, as **cryptCreateContext** modifies it when it creates the encryption context. In almost all other cryptlib routines, CRYPT_CONTEXT is passed by value. The contexts which will be created are standard cryptlib contexts, to create a context which is handled via a crypto device such as a smart card or Fortezza card, you should use **cryptCreateDeviceObject**, which tells cryptlib to create a context in a crypto device. The use of crypto devices is explained in "Encryption Devices and Modules" on page 140.

cryptDestroyContext has a generic equivalent function **cryptDestroyObject** which takes a CRYPT_HANDLE parameter instead of a CRYPT_CONTEXT. This is intended for use with objects which are referred to using generic handles, but can also be used to specifically destroy encryption contexts — crytlib's object management routines will automatically sort out what to do with the handle or object.

Generating a Key into an Encryption Context

Once you've created an encryption context, the next step is to load a key into it. These keys will typically be either one-off session keys which are discarded after use, or long-term storage keys which are used to protect fixed data such as files or private keys. You can create a one-off session key with **cryptGenerateKey**:

```
cryptGenerateKey( cryptContext );
```

which will generate a key of a size which is appropriate for the encryption context. If you want to generate a key of a particular length, you can use the **cryptGenerateKeyEx** function which allows you specify the key size as its second parameter. For example to generate a 256-bit (32-byte) key you would use:

```
cryptGenerateKey( cryptContext, 32 );
```

Keys generated by cryptlib are useful when used with **cryptExportKey**/**cryptImportKey**. Since **cryptExportKey** usually encrypts the generated key using public-key encryption, you shouldn't make it too long or it'll be too big to be encrypted. Unless there's a specific reason for choosing the key length you should use the **cryptGenerateKey** function and let cryptlib choose the correct key length for you.

The only time when you may need to explicitly specify a key length is when you're using very short (in the vicinity of 512 bits) public keys to export Blowfish, RC2, RC4, or RC5 keys. In this case the public key isn't large enough to export the full-length keys for these algorithms, and **cryptExportKey** will return the error code CRYPT_ERROR_OVERFLOW to indicate that there's too much data to export. The solution is to either specify a shorter key length using **cryptGenerateKeyEx**, or, preferably, to use a longer public key. This is only a problem with very short public keys, when using the minimum recommended public key size of 1024 bits this situation will never occur.

Calling **cryptGenerateKey** only makes sense for conventional, public-key, or MAC contexts and will return the error code CRYPT_ERROR_NOTAVAIL for a hash encryption context to indicate that this operation is not available for hash algorithms. The generation of public/private key pairs has special requirements and is covered further on.

To summarise the steps so far, you can set up an encryption context in its simplest form so that it's ready to encrypt data with:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptGenerateKey( cryptContext );

/* Encrypt data */

cryptDestroyContext( cryptContext );
```

Once a key is generated into a context, you can't load or generate a new key over the top of it or change the encryption mode (for conventional encryption contexts). If you try to do this, cryptlib will return CRYPT_ERROR_INITED to indicate that a key is already loaded into the context.

Public/Private Key Generation

cryptlibs public/private key pair generation is mostly identical to conventional session key generation, with the exception that before you can generate a key into a context you need to set the CRYPT_CTXINFO_LABEL attribute which is later used to identify the key when it's written to or read from a keyset or a crypto device such as a smart card or a Fortezza card. If you try to generate a key into a context without first setting the key label, cryptlib will return CRYPT_ERROR_NOTINITED to indicate that the label hasn't been set yet. The process of generating a public/private key pair is then:

```
CRYPT_CONTEXT privKeyContext;

cryptCreateContext( &privKeyContext, CRYPT_ALGO_RSA );
cryptSetAttributeString( privKeyContext, CRYPT_CTXINFO_LABEL, label,
   labelLength );
cryptGenerateKey( privKeyContext );
```

As with conventional key generation, you can use **cryptGenerateKeyEx** to control the size of the generated key. You can also change the default encryption and signature keysizes using the cryptlib configuration options CRYPT_OPTION_-

PKC_KEYSIZE and CRYPT_OPTION_SIG_KEYSIZE as explained in "Miscellaneous Topics" on page 146.

Because the generation of larger public keys may take some time, cryptlib provides an asynchronous key generation capability which allows the key to be generated as a background task or thread on those systems which provide this capability. You can generate a key asynchronously with **cryptGenerateKeyAsync**, which works in the same way as **cryptGenerateKey**. You can check the status of an asynchronous key generation with **cryptAsyncQuery**, which will return CRYPT_ERROR_BUSY if the key generation operation is in progress or CRYPT_OK if the operation has completed. Any attempt to use the context while the key generation operation is still in progress will also return CRYPT_ERROR_BUSY:

```
cryptGenerateKeyAsync( privKeyContext );
while( 1 )
   {
   /* Perform other task(s) */
   /* ... */

   /* Check whether the keygen has completed */
   if( cryptAsyncQuery( privKeyContext ) != CRYPT_ERROR_BUSY )
      break;
   }
```

You can cancel the asynchronous key generation using **cryptAsyncCancel**.

Since the background key generation depends on how the operating system schedules threads, you shouldn't call **cryptAsyncQuery** immediately after calling **cryptGenerateKeyAsync** because the thread which performs the key generation may not have had time to run yet. The code example given above (which performs other work before querying the key generation progress) avoids any OS thread scheduling issues by performing another task while the OS starts the key generation thread in the background.

In general generating a (weak) 512-bit key is almost instantaneous, generating a 1024 bit key typically takes a few seconds, and generating a 2048 bit key takes anywhere from seconds to minutes depending on the algorithm type and machine speed.

Deriving a Key into an Encryption Context

Sometimes you will need to obtain a fixed-format encryption key for a context from a variable-length password or passphrase, or from any generic keying material. You can do this by deriving a key into a context rather than loading it directly. Deriving a key converts arbitrary-format keying information into the particular form required by the context, as well as providing extra protection against password-guessing attacks and other attacks which might take advantage of knowledge of the keying materials' format.

The key derivation process takes two sets of input data, the keying material itself (typically a password), and a salt value which is combined with the password to ensure that the key is different each time (so even if you reuse the same password multiple times, the key obtained from it will change each time. This ensures that even if one password-based key is compromised, all the others remain secure).

The salt attribute is identified by CRYPT_CTXINFO_KEYING_SALT and ranges in length from 64 bits (8 bytes) up to CRYPT_MAX_HASHSIZE. Using an 8-byte salt is a good choice. The keying information attribute is identified by CRYPT_-CTXINFO_KEYING_VALUE and can be of any length. To derive a key into a context you would use:

```
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_SALT,
   salt, saltLength );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_VALUE,
   passPhrase, passPhraseLength );
```

which takes the supplied passphrase and salt and converts them into an encryption key in a format suitable for use with the encryption context. Use of the key derivation capability is strongly recommended over loading keys directly into an

encryption context by setting the CRYPT_CTXINFO_KEY attribute since this often requires intimate knowledge of algorithm details such as how keys of different lengths are handled, how key bits are used, special considerations for key material, and so on.

Note that you have to set a salt value before you set the keying information attribute. If you don't supply a salt, cryptlib will return CRYPT_ERROR_NOTINITED when you try to supply the keying information to indicate that the salt hasn't been set yet. If you don't want to manage a unique salt value per key, you can set the salt to a fixed value (for example 64 bits of zeroes), although this isn't recommended since it means each use of the password will produce the same encryption key.

By default the key derivation process will repeatedly hash the input salt and keying information with the HMAC-SHA1 MAC function to generate the key, and will iterate the hashing process 500 times to make a passphrase-guessing attack more difficult[2]. If you want to change these values you can set the CRYPT_CTXINFO_-KEYING_ALGO and CRYPT_CTXINFO_KEYING_ITERATIONS attributes for the context before setting the salt and keying information attributes. For example to change the number of iterations to 1000 for extra security before setting the salt and key you would use:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_KEYING_ITERATIONS, 1000
   );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_SALT,
   salt, saltLength );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_VALUE,
   passPhrase, passPhraseLength );
```

cryptlib will then use this value when deriving the key. You can also change the default hash algorithm and iteration count using the cryptlib configuration options CRYPT_OPTION_KEYING_ALGO and CRYPT_OPTION_KEYING_-ITERATIONS as explained in "Miscellaneous Topics" on page 146.

To summarise the steps so far, you can set up an encryption context in its simplest form so that it's ready to encrypt data with:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_SALT,
   salt, saltLength );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_VALUE,
   passPhrase, strlen( passPhrase ) );

/* Encrypt data */

cryptDestroyContext( cryptContext );
```

Since public-key encryption uses a different type of key than other context types, you can't derive a key into a public or private key context.

Once a key is derived into a context, you can't load or generate a new key over the top of it or change the encryption mode (for conventional encryption contexts). If you try to do this, cryptlib will return CRYPT_ERROR_INITED to indicate that a key is already loaded into the context.

Loading Keys into Encryption Contexts

If necessary you can also manually load a raw key into an encryption context by setting the CRYPT_CTXINFO_KEY attribute. For example to load a raw 128-bit key "0123456789ABCDEF" into an IDEA conventional encryption context you would use:

```
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEY,
   "0123456789ABCDEF", 16 );
```

[2]It actually does a lot more than just hashing the passphrase, including performing processing steps designed to defeat various sophisticated attacks on the key-hashing process.

Unless you need to perform low-level key management yourself, you should avoid loading keys directly in this manner. The previous key load should really have been done by setting the CRYPT_CTXINFO_KEYING_SALT and CRYPT_CTXINFO_-KEYING_VALUE attributes to derive the key into the context.

For public-key encryption a key will typically have a number of components so you can't set the key directly. Instead you load the key components into a CRYPT_PKCINFO structure and then set this as the CRYPT_CTXINFO_KEY attribute:

```
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEY, &rsaKey,
   CRYPT_UNUSED );
```

More information on working with CRYPT_PKCINFO data structures is given in "Working with Public/Private Keys" on page 61.

Once a key is loaded into a context, you can't load or generate a new key over the top of it or change the encryption mode (for conventional encryption contexts). If you try to do this, cryptlib will return CRYPT_ERROR_INITED to indicate that a key is already loaded into the context.

If you need to reserve space for conventional and public/private keys, you can use the CRYPT_MAX_KEYSIZE and CRYPT_MAX_PKCSIZE defines to determine the mount of memory you need. No key used by cryptlib will ever need more storage than the settings given in these defines. Note that the CRYPT_MAX_PKCSIZE value specifies the maximum size of an individual key component, since public/private keys are usually composed of a number of components the overall size is larger than this.

Loading Initialisation Vectors

For conventional-key encryption contexts you can also load an initialisation vector (IV) into the context if the encryption mode being used supports an IV, although when you're using a context to encrypt data you can leave this to cryptlib to perform automatically when you call **cryptEncrypt** for the first time. IV's are required for the CBC, CFB, and OFB encryption modes. To load an IV you set the CRYPT_CTXINFO_IV attribute:

```
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv, ivSize );
```

To retrieve the IV which you have loaded or which has been generated for you by cryptlib you read the value of the attribute.

Trying to get or set the value of this attribute will return the error code CRYPT_-ERROR_NOTAVAIL for a hash, MAC, or public key encryption context or conventional encryption context with an encryption mode which doesn't use an IV to indicate that these operations are not available for this type of context.

If you need to reserve space for IV's, you can use the CRYPT_MAX_IVSIZE define to determine the mount of memory you need. No IV used by cryptlib will ever need more storage than the setting given in this define.

Working with Public/Private Keys

Since public/private keys typically have multiple components, you can't set them directly as a CRYPT_CTXINFO_KEY attribute. Instead, you load them into a CRYPT_PKCINFO structure and then set that as a CRYPT_CTXINFO_KEY_-COMPONENTS attribute. There are several CRYPT_PKCINFO structures, one for each class of public-key algorithm supported by cryptlib. The CRYPT_PKCINFO structures are described in "Data Structures" on page 173.

As with public/private key pair generation, you need to set the CRYPT_CTXINFO_-LABEL attribute to a unique value used to identify the key before you can load a key value. If you try to load a key into a context without first setting the key label, cryptlib will return CRYPT_ERROR_NOTINITED to indicate that the label hasn't been set yet.

Once a key is loaded into a context, you can't load or generate a new key over the top of it. If you try to do this, cryptlib will return CRYPT_ERROR_INITED to indicate that a key is already loaded into the context.

If you need to reserve space for public/private key components, you can use the CRYPT_MAX_PKCSIZE define to determine the mount of memory you need. No key used by cryptlib will ever need more storage than the settings given in these defines. Note that the CRYPT_MAX_PKCSIZE value specifies the maximum size of an individual key component, since public/private keys are usually composed of a number of components the overall size is larger than this.

Unless you explicitly need to load raw public/private key components into an encryption context, you should avoid loading keys directly in this manner and should instead use the key database access functions to load the key for you. These are described in "Key Databases" on page 40.

In addition, because the public key component manipulation functions need to perform low-level access to the CRYPT_PKCINFO data structures, they are implemented as C preprocessor macros and can't be translated into other languages such as Visual Basic and Delphi. If you're programming in a language other than C or C++, you should always use keyset objects to load and store keys rather than trying to load them using CRYPT_CTXINFO_KEY_COMPONENTS.

Loading Multibyte Integers

The multibyte integer strings which make up public and private keys are stored in big-endian format with the most significant digit first:

```
00000000000000000000000000000000xxxxxxxxxxxxxxxxxx
```

For example the number 123456789 would be stored in big-endian format as:

```
00000000000000000000000000000000000000000123456789
```

(with the remainder of the value padded with zeroes). In practice the numbers won't be stored with excessively long precision as they are in the above examples, so instead of being stored with 50 digits of precision of which 41 bytes contain zero padding, they would be stored with 9 digits of precision:

```
123456789
```

A multibyte integer therefore consists of two parameters, the data itself and the precision to which it is stored. When you load multibyte integer components into a CRYPT_PKCINFO structure you need to specify both of these parameters.

Before you can use the CRYPT_PKCINFO structure, you need to initialise it with `cryptInitComponents()`, which takes as parameter the type of the key, either CRYPT_KEYTYPE_PRIVATE or CRYPT_KEYTYPE_PUBLIC:

```
CRYPT_PKCINFO_RSA rsaKey;

cryptInitComponents( rsaKey, CRYPT_KEYTYPE_PRIVATE );
```

Now you can load the multibyte integer strings by using `cryptSetComponent()`, specifying the value to be loaded, the multibyte integer data, and the integer length in bits:

```
cryptSetComponent( rsaKey.n, modulus, 1024 );
cryptSetComponent( rsaKey.e, pubExponent, 17 );
cryptSetComponent( rsaKey.d, privExponent, 1024 );
```

Once all the parameters are set up, you can use the result as the CRYPT_CTXINFO_-KEY_COMPONENTS as explained above. Once you've finished working with the CRYPT_PKCINFO information, use `cryptDestroyComponents` to destroy the information:

```
cryptDestroyComponents( rsaKey );
```

The Diffie-Hellman, DSA, and Elgamal algorithms share the same key format and all use the CRYPT_PKCINFO_DLP structure to store their key components. DLP is

short for Discrete Logarithm Problem, the common underlying mathematical operation for the three cryptosystems.

To summarise the steps so far, you would load a public key into a DSA context with:

```
CRYPT_CONTEXT cryptContext;
CRYPT_PKCINFO_DLP dlpKey;

cryptCreateContext( cryptContext, CRYPT_ALGO_DSA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_LABEL, "DSA key",
   7 );
cryptInitComponents( dlpKey, CRYPT_KEYTYPE_PUBLIC );
cryptSetComponent( dlpKey.p, ... );
cryptSetComponent( dlpKey.g, ... );
cryptSetComponent( dlpKey.q, ... );
cryptSetComponent( dlpKey.y, ... );
cryptSetAttributeString ( cryptContext, CRYPT_CTXINFO_KEY_COMPONENTS,
   &dlpKey, sizeof( CRYPT_PKCINFO_DLP ) );
cryptDestroyComponents( dlpKey );
```

The context is now ready to be used to verify a DSA signature on a piece of data. If you wanted to load a DSA private key (which consists of one extra component), you would add:

```
cryptSetComponent( dlpKey.x, ... );
```

after the y component is loaded. This context can then be used to sign a piece of data.

Querying Encryption Contexts

A context has a number of attributes whose values you can get to obtain information about it. These attributes contain details such as the algorithm type and name, the key size (if appropriate), the key label (if this has been set), and various other details. The information attributes are:

Value	Type	Description
CRYPT_CTXINFO_ALGO CRYPT_CTXINFO_MODE	N	Algorithm and mode
CRYPT_CTXINFO_BLOCKSIZE	N	Cipher block size in bytes
CRYPT_CTXINFO_IVSIZE	N	Cipher IV size in bytes
CRYPT_CTXINFO_KEYING_-ALGO CRYPT_CTXINFO_KEYING_-ITERATIONS CRYPT_CTXINFO_KEYING_-SALT	N/S	The algorithm and number of iterations used to transform a user-supplied key or password into an algorithm-specific key for the context, and the salt value used in the transformation process
CRYPT_CTXINFO_KEYSIZE	N	Key size in bytes
CRYPT_CTXINFO_LABEL	S	Key label
CRYPT_CTXINFO_NAME_ALGO CRYPT_CTXINFO_NAME_MODE	S	Algorithm and mode name

For example to obtain the algorithm and mode used by an encryption context, you would use:

```
CRYPT_ALGO cryptAlgo;
CRYPT_MODE cryptMode;

cryptGetAttribute( cryptContext, CRYPT_CTXINFO_ALGO, &cryptAlgo );
cryptGetAttribute( cryptContext, CRYPT_CTXINFO_MODE, &cryptMode );
```

Although these attributes are listed as context attributes, they also apply to anything else which can act as a context action object, for example you can obtain algorithm,

mode, and key size values from a certificate since it can be used to encrypt or sign just like a context:

```
CRYPT_ALGO cryptAlgo;
CRYPT_MODE cryptMode;

cryptGetAttribute( cryptCertificate, CRYPT_CTXINFO_ALGO, &cryptAlgo );
cryptGetAttribute( cryptCertificate, CRYPT_CTXINFO_MODE, &cryptMode );
```

If any of the user-supplied attributes haven't been set and you try to read their value, cryptlib will return CRYPT_ERROR_NOTINITED.

Encrypting/Decrypting Data

Instead of indirectly encrypting and decrypting data using envelopes, you can also directly process it using an encryption context object. This eliminates the processing and encoding size overhead of envelopes, at the cost of slightly increased code complexity because of the need to duplicate work which is usually performed by the enveloping code.

Using Encryption Contexts to Encrypt/Decrypt Data

To encrypt or decrypt a block of data using an encryption context action object you use:

```
cryptEncrypt( cryptContext, buffer, length );
```

and:

```
cryptDecrypt( cryptContext, buffer, length );
```

Hash and MAC algorithms don't actually encrypt the data being hashed and can be called via **cryptEncrypt** or **cryptDecrypt**. They require a final call with the length set to 0 as a courtesy call to indicate to the hash or MAC function that this is the last data block and that the function should take whatever special action is necessary for this case:

```
cryptEncrypt( hashContext, buffer, length );
cryptEncrypt( hashContext, buffer, 0 );
```

If you call **cryptEncrypt** or **cryptDecrypt** after making the final call with the length set to 0, the function will return CRYPT_ERROR_COMPLETE to indicate that the hashing has completed and cannot be continued. Once the hashing is complete, the hash value is made available as the CRYPT_CTXINFO_HASHVALUE attribute which you can read in the usual manner:

```
unsigned char hash[ CRYPT_MAX_HASHSIZE ];
int hashLength;

cryptGetAttributeString( cryptContext, CRYPT_CTXINFO_HASHVALUE, hash,
   &hashLength );
```

The public-key algorithms encrypt a single block of data equal in length to the size of the public key being used. For example if you are using a 1024-bit public key then the length of the data to be encrypted should be 128 bytes. Preparation of the block of data to be encrypted requires special care; in general you should use high-level functions such as **cryptExportKey**/**cryptImportKey** and **cryptCreateSignature**/**cryptCheckSignature** rather than **cryptEncrypt** and **cryptDecrypt** when working with public-key algorithms. If the en/decryption operation fails due to incorrect public or private key parameters or incorrectly formatted input data, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed.

If you're using a block cipher in ECB or CBC mode, the encrypted data length must be a multiple of the block size. If the encrypted data length is not a multiple of the block size, the function will return CRYPT_ERROR_PARAM3 to indicate that the length is invalid. To encrypt a byte at a time you should use a stream encryption mode such as CFB or OFB, or better yet use an envelope which avoids the need to handle algorithm-specific details.

If the encryption context doesn't support the operation you are trying to perform (for example calling **cryptEncrypt** with a DSA public key), the function will return CRYPT_ERROR_NOTAVAIL to indicate that this functionality is not available.

If the key loaded into an encryption context doesn't allow the operation you are trying to perform (for example calling **cryptDecrypt** with an encrypt-only key), the function will return CRYPT_ERROR_PERMISSION to indicate that the context doesn't have the required key permissions to perform the requested operation. Similarly, if you're using a private key context which is tied to a certificate or crypto

device, the direct use of **cryptEncrypt** and **cryptDecrypt** could be used to bypass security constraints placed on the context (for example by changing the data formatting used with an encryption-only RSA context it's possible to misuse it to generate signatures even if the context is specifically intended for non-signature use). Because of this, if a context is tied to a certificate or a crypto device, it can't be used directly with these low-level functions but only with a higher-level function like **cryptCreateSignature** or with the enveloping code, which guarantee that a context can't be misused for a disallowed purpose. If you try to use a constrained context of this type directly, the function will return CRYPT_ERROR_PERMISSION to indicate that the context doesn't have the required permissions to perform the requested operation.

If an IV is required for the decryption and you haven't loaded one into the context by setting the CRYPT_CTXINFO_IV attribute, **cryptDecrypt** will return CRYPT_-ERROR_NOTINITED to indicate that you need to load an IV before you can decrypt the data. If the first 8 bytes of decrypted data are corrupted then you haven't set up the IV properly for the decryption. More information on setting up IV's is given in "Encryption and Decryption" on page 56.

Once an encryption context is set up, it can only be used for processing a single data stream in an operation such as encrypting data, decrypting data, or hashing a message. A context can't be reused to encrypt a second message after the first one has been encrypted, or to decrypt data after having encrypted data. This is because the internal state of the context is determined by the operation being performed with it, and performing two different operations with the same context causes the state from the first operation to affect the second operation. For example if you use an encryption context to encrypt two different files, cryptlib will see a single continuous data stream (since it doesn't know or care about the structure of the data being encrypted). As a result the second file is treated as a continuation of the first one, and can't be decrypted unless the context is used to decrypt the first file before decrypting the second one. Because of this you should always create a new encryption context for each discrete data stream you will be processing, and never reuse contexts to perform different operations. The one exception to this rule is when you're using cryptlib envelopes (described in "Enveloping " on page 22), where you can push a single encryption context into as many envelopes as you like. This is because an envelope takes its own copy of the encryption context, leaving the original untouched.

In practice this isn't strictly accurate, you can encrypt multiple independent data streams with a single context by loading a new IV for each new stream using the CRYPT_CTXINFO_IV attribute. If you don't understand how this would work then it's probably best to use a new context for each data stream.

Exchanging Keys

Although you can now encrypt and decrypt data with an encryption context, the key you're using is locked inside the context and (if you used **cryptGenerateKey** to create it) won't be known to you or the person you're trying to communicate with. To share the key with another party, you need to export it from the context in a secure manner and the other party needs to import it into an encryption context of their own. Because the key is a very sensitive and valuable resource, you can't just read it out of the context, but need to take special steps to protect the key once it leaves the context. This is taken care of by the key export/import functions.

Exporting a Key

To exchange a key with another party, you use the **cryptExportKey** and **cryptImportKey** functions in combination with a conventional or public-key encryption context or public key certificate. Let's say you've created a key in an encryption context `cryptContext` and want to send it to someone whose public key is in the encryption context `publicKeyContext` (you can also pass in a private key if you want, **cryptExportKey** will only use the public key components, although unless you're the US government it's not clear why you'd want to be in possession of someone elses private key). To do this you'd use:

```
CRYPT_CONTEXT publicKeyContext, cryptContext;
void *encryptedKey;
int encryptedKeyLength;

/* Generate a key */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptGenerateKey( cryptContext );

/* Allocate memory for the encrypted key */
encryptedKey = malloc( ... );

/* Export the key using a public-key encrypted blob */
cryptExportKey( encryptedKey, &encryptedKeyLength, publicKeyContext,
   cryptContext );
```

The resulting public-key encrypted blob is placed in the memory buffer pointed to by `encryptedKey`, and the length is stored in `encryptedKeyLength`. This leads to a small problem: How do you know how big to make the buffer? The answer is to use **cryptExportKey** to tell you. If you pass in a null pointer for `encryptedKey`, the function will set `encryptedKeyLength` to the size of the resulting blob, but not do anything else. You can then use code like:

```
cryptExportKey( NULL, &encryptedKeyLength, publicKeyContext,
   cryptContext );
encryptedKey = malloc( encryptedKeyLength );
cryptExportKey( encryptedKey, &encryptedKeyLength, publicKeyContext,
   cryptContext );
```

to create the exported key blob. Note that due to encoding issues for some algorithms the final exported blob may be one or two bytes smaller than the size which is initially reported, since the true size can't be determined until the key is actually exported.

Alternatively, you can just reserve a reasonably sized block of memory and use that to hold the encrypted key. "Reasonably sized" means a few Kb, a 4K block is plenty (an encrypted key blob for a 1024-bit public key is only about 200 bytes long).

You can also use a public key certificate to export a key. If, instead of a public key context, you had a key certificate contained in the certificate object `publicKeyCertificate`, the code for the previous example would become:

```
CRYPT_CERTIFICATE publicKeyCertificate;
CRYPT_CONTEXT cryptContext;
void *encryptedKey;
int encryptedKeyLength;
```

```
/* Generate a key */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptGenerateKey( cryptContext );

/* Allocate memory for the encrypted key */
encryptedKey = malloc( ... );

/* Export the key using a public-key encrypted blob */
cryptExportKey( encryptedKey, &encryptedKeyLength,
   publicKeyCertificate, cryptContext );
```

The use of key certificates is explained in "Certificate Management" on page 76.

If the encryption context contains too much data to encode using the given public key (for example trying to export an encryption context with a 600-bit key using a 512-bit public key) the function will return CRYPT_ERROR_OVERFLOW. As a rule of thumb a 1024-bit public key should be large enough to export the default key sizes for any encryption context.

If the encryption part of the export operation fails due to incorrect public or private key parameters, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed.

If the public key is stored in an encryption context with a certificate associated with it or in a key certificate, there may be constraints on the key usage which are imposed by the certificate. If the key can't be used for the export operation, the function will return CRYPT_ERROR_PERMISSION to indicate that the key isn't valid for this operation, you can find out more about the exact nature of the problem by reading the error-related attributes as explained in "Miscellaneous Topics" on page 146.

Exporting using Conventional Encryption

You don't need to use public-key encryption to export a key blob, it's also possible to use a conventional encryption context to export the key from another conventional encryption context. For example if you were using the key derived from the passphrase "This is a secret key" (which was also known to the other party) in an encryption context `keyContext` you would use:

```
CRYPT_CONTEXT sharedContext, keyContext;
void *encryptedKey;
int encryptedKeyLength;

/* Derive the export key into an encryption context */
cryptCreateContext( &keyContext, CRYPT_ALGO_3DES );
cryptSetAttributeString( keyContext, CRYPT_CTXINFO_KEYING_SALT, salt,
   saltLength );
cryptSetAttributeString( keyContext, CRYPT_CTXINFO_KEYING_VALUE, "This
   is a secret key", 20 );

/* Generate a key */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptGenerateKey( cryptContext );

/* Allocate memory for the encrypted key */
encryptedKey = malloc( ... );

/* Export the key using a conventionally encrypted blob */
cryptExportKey( encryptedKey, &encryptedKeyLength, keyContext,
   cryptContext );
```

You don't need to use a derived key to export the session key, you could have loaded the context in some other manner (for example from a smart card), but the sample code shown above, and further on for the key import phase, assumes that you'll be deriving the export/import key from a password.

This kind of key export isn't as convenient as using public keys since it requires that both sides know the encryption key in `keyContext` (or at least know how to derive it from some other keying material). One case where it's useful is when you want to encrypt data such as a disk file which will be decrypted later by the same person who

originally encrypted it. By prepending the key blob to the start of the encrypted file, you can ensure that each file is encrypted with a different session key (this is exactly what the cryptlib enveloping functions do). It also means you can change the password on the file by changing the exported key blob, without needing to decrypt and re-encrypt the entire file.

Importing a Key

Now that you've exported the key, the other party needs to import it. This is done using the **cryptImportKey** function and the private key corresponding to the public key used by the sender:

```
CRYPT_CONTEXT privateKeyContext, cryptContext;

/* Create a context for the imported key */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );

/* Import the key from the public-key encrypted blob */
cryptImportKey( encryptedKey, privateKeyContext, cryptContext );
```

Note the use of CRYPT_ALGO_3DES when creating the context for the imported key, this assumes that both sides have agreed in advance on the use of a common encryption algorithm to use (in this case triple DES). If the algorithm information isn't available, you'll have to negotiate the details in some other way (this is normally done for you by the enveloping code but isn't available at this level).

To summarise, sharing an encryption context between two parties using public-key encryption involves the following steps:

```
/* Party A creates the required encryption context and generates a key
   into it */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );
cryptGenerateKey( cryptContext );

/* Party A exports the key using party B's public key */
cryptExportKey( encryptedKey, &encryptedKeyLength, publicKeyContext,
   cryptContext );

/* Party B creates the encryption context to import the key into */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );

/* Party B imports the key using their private key */
cryptImportKey( encryptedKey, privateKeyContext, cryptContext );
```

If the decryption part of the import operation fails due to incorrect public or private key parameters, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed. If the input data or decrypted data is corrupt and the key couldn't be recovered, the function will return CRYPT_ERROR_BADDATA. In general CRYPT_ERROR_FAILED will be returned for incorrect public or private key parameters and CRYPT_ERROR_BADDATA will be returned if the input data has been corrupted. You can treat both of these as "key import failed" unless you want to include special-case error handling for them.

If the public key is stored in an encryption context with a certificate associated with it or in a key certificate, there may be constraints on the key usage which are imposed by the certificate. If the key can't be used for the import operation, the function will return CRYPT_ERROR_PERMISSION to indicate that the key isn't valid for this operation. You can find out more about the exact nature of the problem by reading the error-related attributes as explained in "Miscellaneous Topics" on page 146.

Importing using Conventional Encryption

If the key has been exported using conventional encryption, you can again use conventional encryption to import it. Using the same key derived from the passphrase "This is a secret key" which was used in the key export example, you would use:

```
CRYPT_CONTEXT keyContext, cryptContext;
```

```
/* Derive the import key into an encryption context */
cryptCreateContext( &keyContext, CRYPT_ALGO_3DES );
cryptSetAttributeString( keyContext, CRYPT_CTXINFO_KEYING_SALT, salt,
   saltLength );
cryptSetAttributeString( keyContext, CRYPT_CTXINFO_KEYING_VALUE, "This
   is a secret key", 20 );

/* Create a context for the imported key */
cryptCreateContext( &cryptContext, CRYPT_ALGO_3DES );

/* Import the key from the conventionally encrypted blob */
cryptImportKey( encryptedKey, keyContext, cryptContext );
```

Since the salt is a random value which changes for each key you derive, you won't know it in advance so you'll have to obtain it by querying the exported key object as explained below. Once you've queried the object, you can use the salt which is returned with the query information to derive the import key as shown in the above code.

Querying an Exported Key Object

So far it's been assumed that you know what's required to import the exported key blob you're given (that is, you know which type of processing to use to create the encryption context needed to import a conventionally encrypted blob). However sometimes you may not know this in advance, which is where the **cryptQueryObject** function comes in. **cryptQueryObject** is used to obtain information on the exported key blob which might be required to import it. You can also use **cryptQueryObject** to obtain information on signature blobs, as explained in "Signing Data" on page 72.

The function takes as parameters the object you want to query, and a pointer to a CRYPT_OBJECT_INFO structure which is described in "Data Structures" on page 173. The object type will be either a CRYPT_OBJECT_ENCRYPTED_KEY for a conventionally encrypted exported key, a CRYPT_OBJECT_PKCENCRYPTED_-KEY for a public-key encrypted exported key, or a CRYPT_OBJECT_-KEYAGREEMENT for a key-agreement key. If you were given an arbitrary object of an unknown type you'd use the following code to handle it:

```
CRYPT_OBJECT_INFO cryptObjectInfo;

cryptQueryObject( object, &cryptObjectInfo );
if( cryptObjectInfo.objectType == CRYPT_OBJECT_ENCRYPTED_KEY )
   /* Import the key using conventional encryption */
else
   if( cryptObjectInfo.objectType == CRYPT_OBJECT_PKCENCRYPTED_KEY ||
      cryptObjectInfo.objectType == CRYPT_OBJECT_KEYAGREEMENT )
     /* Import the key using public-key encryption */
   else
     /* Error */
```

Any CRYPT_OBJECT_INFO fields which aren't relevant for this type of object are set to null or zero as appropriate.

Once you've found out what type of object you have, you can use the other information returned by **cryptQueryObject** to process the object. For both conventional and public-key encrypted exported objects you can find out which encryption algorithm and mode were used to export the key using the `cryptAlgo` and `cryptMode` fields. For conventionally encrypted exported objects you can obtain the salt needed to derive the import key from the `salt` and `saltSize` fields.

Extended Key Export/Import

The **cryptExportKey** and **cryptImportKey** functions described above export and import keys in the cryptlib default format (which, for the technically inclined, is the Cryptographic Message Syntax format with key identifiers used to denote public keys). The default cryptlib format has been chosen to be independent of the underlying key format, so that it works equally well with any key type including X.509 certificates, PGP keys, and any other key storage format.

Alongside the default format, cryptlib supports the export and import of keys in other formats using **cryptExportKeyEx** and **cryptImportKeyEx**. **cryptExportKeyEx** works like **cryptExportKey** but takes an extra parameter which specifies the format to use for the exported keys. The formats are:

Format	Description
CRYPT_FORMAT_CMS CRYPT_FORMAT_SMIME	This is an older variation of the Cryptographic Message Syntax and is also known as S/MIME version 2 or 3 and PKCS #7. This format only allows public-key-based export, and the public key must be stored as an X.509 certificate.
CRYPT_FORMAT_CRYPTLIB	This is the default cryptlib format and can be used with any type of key. When used for public-key based key export, this format is also known as a newer variation of S/MIME version 3.

Apart from the format specifier and the restrictions on key types available with some formats, the extended export/import functions work identically to the standard export/import functions (in fact the two import functions map to the same function since cryptlib automatically determines the correct format to use and handles it appropriately).

Key Agreement

The Diffie-Hellman key agreement capability is currently disabled since, unlike RSA and conventional key exchange, there's no widely-accepted standard format for it (DH with SSL is barely supported and requires all of SSL to work, ssh v2 is barely supported, CMS is unsupported, and IKE requires all of IPSEC to work). If a widely-accepted standard emerges, cryptlib will use that format. Previous versions of cryptlib used a combination of PKCS #3, PKCS #5, and PKCS #7 formats and mechanisms to handle DH key agreement.

cryptlib supports a third kind of key export/import which doesn't actually export or import a key but merely provides a means of agreeing on a shared secret key with another party. You don't have to explicitly load of generate a session key for this one since the act of performing the key exchange will create a random, secret shared key. To use this form of key exchange, both parties call **cryptExportKey** to generate the blob to send to the other party, and then both in turn call **cryptImportKey** to import the blob sent by the other party.

The use of **cryptExportKey**/**cryptImportKey** for key agreement rather than key exchange is indicated by the use of a key agreement algorithm for the context which would normally be used to export the key. The key agreement algorithm used by cryptlib is the Diffie-Hellman (DH) key exchange algorithm, CRYPT_ALGO_DH. Creating a Diffie-Hellman context and loading a key into it is explained in "Working with Public/Private Keys" on page 61. In the following code the resulting Diffie-Hellman context is referred to as `dhContext`.

Since there's a two-way exchange of messages, both parties must create an identical "template" encryption context so **cryptExportKey** knows what kind of key to export. Lets assume that both sides know they'll be using Blowfish in CFB mode. The first step of the key exchange is therefore:

```
/* Create the key template */
cryptCreateContext( &cryptContext, CRYPT_ALGO_BLOWFISH );
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_MODE, CRYPT_MODE_CFB );

/* Export the key using the template */
cryptExportKey( encryptedKey, &encryptedKeyLength, dhContext,
   cryptContext );
cryptDestroyContext( cryptContext );
```

Note that there's no need to load a key into the template, since this is generated automatically as part of the export/import process. In addition the template is destroyed once the key has been exported, since there's no further use for it — it merely acts as a template to tell **cryptExportKey** what to do.

Both parties now exchange `encryptedKey` blobs, and then use:

```
cryptImportKey( encryptedKey, dhContext, cryptContext );
```

to create the `cryptContext` containing the shared key.

The agreement process requires that both sides export their own `encryptedKey` blobs before they import the other sides `encryptedKey` blob. A side-effect of this is that it allows additional checking on the key agreement process to be performed to guard against things like triple DES turning into 40-bit RC4 during transmission. If you try to import another parties `encryptedKey` blob without having first exported your own `encryptedKey` blob, **cryptImportKey** will return CRYPT_ERROR_NOTINITED.

If the en/decryption part of the export/import operation fails due to incorrect public or private key parameters, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed. If the input data or decrypted data is corrupt and the key couldn't be recovered, the function will return CRYPT_ERROR_BADDATA. In general CRYPT_ ERROR_FAILED will be returned for incorrect key parameters and CRYPT_ ERROR_BADDATA will be returned if the input data has been corrupted. You can treat both of these as "key export/import failed" unless you want to include special-case error handling for them.

Signing Data

Most public-key encryption algorithms can be used to generate digital signatures on data. A digital signature is created by signing the contents of a hash context with a private key to create a signature blob, and verified by checking the signature blob with the corresponding public key.

To do this, you use the **cryptCreateSignature** and **cryptCheckSignature** functions in combination with a public-key encryption context. Let's say you've hashed some data with an SHA-1 hash context `hashContext` and want to sign it with your private key in the encryption context `signatureKeyContext`. To do this you'd use:

```
CRYPT_CONTEXT signatureKeyContext, hashContext;
void *signature;
int signatureLength;

/* Create a hash context */
cryptCreateContext( &hashContext, CRYPT_ALGO_SHA );

/* Hash the data */
cryptEncrypt( hashContext, data, dataLength );
cryptEncrypt( hashContext, data, 0 );

/* Allocate memory for the signature */
signature = malloc( ... );

/* Sign the hash to create a signature blob */
cryptCreateSignature( signature, signatureLength, signatureKeyContext,
   hashContext );
cryptDestroyContext( hashContext );
```

The resulting signature blob is placed in the memory buffer pointed to by `signature`, and the length is stored in `signatureLength`. This leads to the same problem with allocating the buffer which was described for **cryptExportKey**, and the solution is again the same: You use **cryptCreateSignature** to tell you how big to make the buffer. If you pass in a null pointer for `signature`, the function will set `signatureLength` to the size of the resulting blob, but not do anything else. You can then use code like:

```
cryptCreateSignature( NULL, &signatureLength, signatureKeyContext,
   hashContext );
signature = malloc( signatureLength );
cryptCreateSignature( signature, &signatureLength,
   signatureKeyContext, hashContext );
```

to create the signature blob. Note that due to encoding issues for some algorithms the final exported blob may be one or two bytes smaller than the size which is initially reported, since the true size can't be determined until the signature is actually generated. Alternatively, you can just allocate a reasonably sized block of memory and use that to hold the signature. "Reasonably sized" means a few Kb, a 4K block is plenty (a signature blob for a 1024-bit public key is only about 200 bytes long).

Now that you've created the signature, the other party needs to check it. This is done using the **cryptCheckSignature** function and the public key or key certificate corresponding to the private key used to create the signature (you can also pass in a private key if you want, **cryptCheckSignature** will only use the public key components, although it's not clear why you'd be in possession of someone elses private key). To perform the check using a public key context you'd use:

```
CRYPT_CONTEXT sigCheckContext, hashContext;

/* Create a hash context */
cryptCreateContext( &hashContext, CRYPT_ALGO_SHA );

/* Hash the data */
cryptEncrypt( hashContext, data, dataLength );
cryptEncrypt( hashContext, data, 0 );
```

```
/* Check the signature using the signature blob */
cryptCheckSignature( signature, sigCheckContext, hashContext );
cryptDestroyContext( hashContext );
```

A signature check using a key certificate is similar, except that it uses a public key certificate object rather than a public key context. The use of key certificates is explained in "Certificate Management" on page 76.

If the en/decryption part of the signature create/check operation fails due to incorrect public or private key parameters, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed. If the signature is corrupt and couldn't be recovered, the function will return CRYPT_ERROR_BADDATA. In general CRYPT_ERROR_FAILED will be returned for incorrect public or private key parameters and CRYPT_ERROR_BADDATA will be returned if the signature has been corrupted. Finally, if the signature doesn't match the hash context, the function will return CRYPT_ERROR_SIGNATURE. You can treat all three of these as "signature generation/check failed" unless you want to include special-case error handling for them.

If the public key is stored in an encryption context with a certificate associated with it or in a key certificate, there may be constraints on the key usage which are imposed by the certificate. If the key can't be used for the signature or signature check operation, the function will return CRYPT_ERROR_PERMISSION to indicate that the key isn't valid for this operation, you can find out more about the exact nature of the problem by reading the error-related attributes as explained in "Miscellaneous Topics" on page 146.

Querying a Signature Object

Just as you can query exported key blobs, you can also query signature blobs using **cryptQueryObject**, which is used to obtain information on the signature. You can also use **cryptQueryObject** to obtain information on exported key blobs as explained in "Exchanging Keys" on page 66.

The function takes as parameters the object you want to query, and a pointer to a CRYPT_OBJECT_INFO structure which is described in "Data Structures" on page 173. The object type will be a CRYPT_OBJECT_SIGNATURE for a signature object. If you were given an arbitrary object of an unknown type you'd use the following code to handle it:

```
CRYPT_OBJECT_INFO cryptObjectInfo;

cryptQueryObject( object, &cryptObjectInfo );
if( cryptObjectInfo.objectType == CRYPT_OBJECT_SIGNATURE )
   /* Check the signature */
else
   /* Error */
```

Any CRYPT_OBJECT_INFO fields which aren't relevant for this type of object are set to null or zero as appropriate.

Once you've found out what type of object you have, you can use the other information returned by **cryptQueryObject** to process the object. The information which you need to obtain from the blob is the hash algorithm which was used to hash the signed data, which is contained in the `hashAlgo` field. To hash a piece of data before checking the signature on it you would use:

```
CRYPT_CONTEXT hashContext;

/* Create the hash context from the query info */
cryptCreateContext( &hashContext, cryptObjectInfo.hashAlgo );

/* Hash the data */
cryptEncrypt( hashContext, data, dataLength );
cryptEncrypt( hashContext, data, 0 );
```

Extended Signature Creation/Checking

The **cryptCreateSignatureEx** and **cryptCheckSignatureEx** functions described above create and verify signatures in the cryptlib default format (which, for the technically inclined, is the Cryptographic Message Syntax format with key identifiers used to denote public keys). The default cryptlib format has been chosen to be independent of the underlying key format, so that it works equally well with any key type including raw keys, X.509 certificates, PGP keys, and any other key storage format.

Alongside the default format, cryptlib supports the generation and checking of signatures in other formats using **cryptCreateSignatureEx** and **cryptCheckSignatureEx**. **cryptCreateSignatureEx** works like **cryptCreateSignature** but takes two extra parameters, the first of which specifies the format to use for the signature. The formats are:

Format	Description
CRYPT_FORMAT_CMS CRYPT_FORMAT_SMIME	This is an older variation of the Cryptographic Message Syntax and is also known as S/MIME version 2 or 3 and PKCS #7. This format only allows public-key-based export, and the public key must be stored as an X.509 certificate.
CRYPT_FORMAT_CRYPTLIB	This is the default cryptlib format and can be used with any type of key. When used for public-key based key export, this format is also known as a newer variation of S/MIME version 3.

The second extra parameter required by **cryptCreateSignatureEx** depends on the signature format being used. With CRYPT_FORMAT_CRYPTLIB this parameter isn't used and should be set to CRYPT_UNUSED. With CRYPT_FORMAT_CMS/CRYPT_FORMAT_SMIME, this parameter specifies optional additional information which is included with the signature. The only real difference between the CRYPT_FORMAT_CMS and CRYPT_FORMAT_SMIME signature format is that CRYPT_FORMAT_SMIME adds a few extra S/MIME-specific attributes which aren't added by CRYPT_FORMAT_CMS. This additional information includes things like the type of data being signed (so that the signed content can't be interpreted the wrong way), the signing time (so that an old signed message can't be reused), and any other information which the signer might consider worth including.

The easiest way to handle this extra information is to let cryptlib add it for you. If you set the parameter to CRYPT_USE_DEFAULT, cryptlib will generate and add the extra information for you:

```
void *signature;
int signatureLength;

cryptCreateSignatureEx( NULL, &signatureLength, CRYPT_FORMAT_CMS,
   signatureKeyContext, hashContext, CRYPT_USE_DEFAULT );
signature = malloc( signatureLength );
cryptCreateSignatureEx( signature, &signatureLength, CRYPT_FORMAT_CMS,
   signatureKeyContext, hashContext, CRYPT_USE_DEFAULT );
```

If you need more precise control over the extra information, you can specify it yourself in the form of a CRYPT_CERTTYPE_CMS_ATTRIBUTES certificate object, which is described in more detail in "Further Certificate Objects" on page 124. By default cryptlib will include the default signature attributes CRYPT_CERTINFO_CMS_SIGNINGTIME and CRYPT_CERTINFO_CMS_-CONTENTYPE for you if you don't specify it yourself, and for S/MIME signatures it will also include CRYPT_CERTINFO_CMS_SMIMECAPABILITIES. You can disable this automatic including with the cryptlib configuration option CRYPT_OPTION_CMS_DEFAULTATTRIBUTES/CRYPT_OPTION_SMIME_-

DEFAULTATTRIBUTES as explained in “Miscellaneous Topics” on page 146, this will simplify the signature somewhat and reduce its size and processing overhead:

```
CRYPT_CERTIFICATE signatureAttributes;
void *signature;
int signatureLength;

/* Create the signature attribute object */
cryptCreateCert( &signatureAttributes, CRYPT_CERTTYPE_CMS_ATTRIBUTES
   );
/* ... */

/* Create the signature including the attributes as extra information
   */
cryptCreateSignatureEx( NULL, &signatureLength, CRYPT_FORMAT_CMS,
   signatureKeyContext, hashContext, signatureAttributes );
signature = malloc( signatureLength );
cryptCreateSignatureEx( signature, &signatureLength, CRYPT_FORMAT_CMS,
   signatureKeyContext, hashContext, signatureAttributes );
cryptDestroyCert( signatureAttributes );
```

In general if you’re sending signed data to a recipient who is also using cryptlib-based software, you should use the default cryptlib signature format which is more flexible in terms of key handling and far more space-efficient (CMS/SMIME signatures are typically ten times the size of the default cryptlib format while providing little extra information, and have a much higher processing overhead than cryptlib signatures).

Extended signature checking follows the same pattern as extended signature generation, with the extra parameter to the function being a pointer to the location which receives the additional information included with the signature. With the CRYPT_FORMAT_CRYPTLIB format type, there’s no extra information present and the parameter should be set to null. With CRYPT_FORMAT_CMS/ CRYPT_FORMAT_SMIME, you can also set the parameter to null if you’re not interested in the additional information, and cryptlib will discard it after using it as part of the signature checking process (this is required even if the information isn’t used). If you are interested in the additional information, you should set the parameter to a pointer to a CRYPT_CERTIFICATE value which cryplib will create for you and populate with the additional signature information. If the signature check succeeds, you can work with the resulting CRYPT_CERTIFICATE as you would with any certificate object:

```
CRYPT_CERTIFICATE signatureAttributes;
int status;

status = cryptCheckSignatureEx( signature, sigCheckCertificate,
   hashContext, &signatureAttributes );
if( cryptStatusOK( status ) )
   {
   /* Work with extra signature information in signatureAttributes */
   /* ... */

   /* Clean up */
   cryptDestroyCert( signatureAttributes );
   }
```

Certificate Management

Although an encryption context can be used to store basic key components, it's not capable of storing any additional information such as the key owners name, usage restrictions, and key validity information. This type of information is stored in a key certificate, which is encoded according to the X.509 standard and sundry amendments, corrections, extensions, profiles, and related standards. A certificate consists of the encoded public key, information to identify the owner of the certificate, other data such as usage and validity information, and a digital signature which binds all this information to the key.

There are a number of different types of certificate objects, including actual certificates, certification requests, certificate revocation lists (CRL's), certification authority (CA) certificates, certificate chains, attribute certificates, and others. For simplicity the following text refers to all of these items using the general term "certificate". Only where a specific type of item such as a CA certificate or a certification request is required will the actual name be used.

cryptlib stores all of these items in a generic CRYPT_CERTIFICATE container object into which you can insert various items such as identification information and key attributes, as well as public-key encryption contexts or other certificate objects. Once everything has been added, you can fix the state of the certificate by signing it, after which you can't change it except by starting again with a fresh certificate object.

Overview of Certificates

Public key certificates are objects which bind information about the owner of a public key to the key itself. The binding is achieved by having the information in the certificate signed by a certification authority (CA) which protects the integrity of the certificate information and allows it to be distributed over untrusted channels and stored on untrusted systems.

You can request a certificate from a CA with a certification request, which encodes a public key and identification information and binds them together for processing by the CA. The CA responds to a certificate request with a signed certificate.

You can also cancel (or revoke) an existing certificate with a certificate revocation (traditionally referred to as a certificate revocation list or CRL), although proper CRL's were never terribly practical, often have little support in actual implementations, and will probably be superseded with online validation techniques ranging from phonecalls through to full online validation protocols.

Certificates and Standards Compliance

The key certificates used by most software today were originally specified in the CCITT (now ITU) X.509 standard, and have been extended via assorted ISO, ANSI, ITU, IETF, and national standards (generally referred to as "X.509 profiles"), along with sundry amendments, meeting notes, draft standards, committee drafts, working drafts, and other work-in-progress documents. X.509 version 1 (X.509v1) defined the original, very basic certificate format, the latest version of the standard is version 3 (X.509v3) which defines all manner of extensions and additions and is still in the process of being finalised and profiled. Compliance with the various certificate standards varies greatly. Most implementations manage to get the decade-old X.509v1 more or less correct, and cryptlib includes special code to allow it to process many incorrectly-formatted X.509v1-style certificates as well as all correctly formatted ones. However compliance with X.509v3 profiles is extremely patchy. Because of this, it is strongly recommended that you test the certificates you plan to produce with cryptlib against any other software you want to interoperate with. Although cryptlib produces certificates which comply fully with X.509v3 and related standards and recommendations, many other programs (including several common

web browsers and servers) either can't process these certificates at all or will process them incorrectly.

To bypass this problem, cryptlib provides the ability to selectively disable various levels of X.509v3 compliance in order to produce certificates which can be loaded by other software. To check interoperability, start with a full X.509v3 certificate and then gradually disable more and more X.509v3 features until the certificate can be processed by the other software. Note that even if the other software loads your certificate, it may not process the information contained in it correctly, so you should verify that it's handling it in the way you expect it to.

If you need to interoperate with a variety of other programs, you may need to find the lowest common denominator which all programs can accept, which is usually X.509v1, sometimes with one or two X.509v3 extensions. Alternatively, you can issue different certificates for different software, a technique which is currently used by some CA's which have a different certificate issuing process for Netscape, MSIE, and everything else.

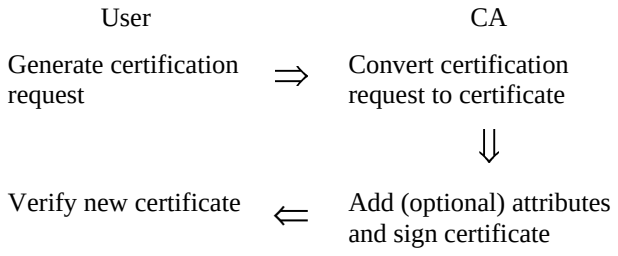
Much current certificate management software produces an amazing collection of garbled, invalid, and just plain broken certificates which will be rejected by cryptlib in its default mode of operation. As with certificate generation, it's possible to disable various portions of the certificate checking code in order to allow these certificates to be processed. If a certificate fails to load you can try disabling more and more certificate checking in cryptlib until the certificate can be loaded, although disabling these checks will also void any guarantees about correct certificate handling.

To provide maximum compatibility with existing implementations, you should set the configuration options CRYPT_OPTION_CERT_ENCODE_VALIDITYNESTING to true and CRYPT_OPTION_CERT_ENCODECRITICAL, CRYPT_OPTION_-CERT_DECODECRITICAL CRYPT_OPTION_CERT_DECODE_-VALIDITYNESTING, CRYPT_OPTION_CERT_CHECKENCODING, and CRYPT_OPTION_CERT_FIXSTRINGS to false. To provide maximum compliance with the standards which cover certificates, you should set the configuration options CRYPT_OPTION_CERT_ENCODE_VALIDITYNESTING, CRYPT_OPTION_-CERT_DECODE_VALIDITYNESTING, CRYPT_OPTION_CERT_-ENCODECRITICAL, CRYPT_OPTION_CERT_DECODECRITICAL, CRYPT_OPTION_CERT_CHECKENCODING, and CRYPT_OPTION_CERT_-FIXSTRINGS to true.

Finally, implementations are free to stuff anything they feel like into certain areas of the certificate. cryptlib goes to some lengths to take this into account and process the certificate no matter what data it finds in there, however sometimes it may find something that it can't handle. If you require support for special certificate components (either to generate them or to process them), please contact the cryptlib developers. Support for reasonably normal certificate extensions and peculiarities can usually be added within a fortnight of receiving the requirements for the implementation.

The Certification Process

Obtaining a public key certificate involves generating a public key, creating a certificate request from it, transmitting it to a CA who converts the certification request into a certificate and signs it, and finally retrieving the completed certificate from the CA:



These steps can be broken down into a number of individual operations. The first step, generating a certification request, involves the following:

```
generate public/private key pair;
create certificate object;
add public key to certificate object;
add identification information to certificate object;
sign certificate object with private key;
export certification request for transmission to CA;
destroy certificate object;
```

The CA receives the certification request and turns it into a certificate as follows:

```
import certification request;
check validity and signature on certification request;
create certificate object;
add certification request to certificate object;
add any extra information (eg key usage constraints) to certificate
   object;
sign certificate object;
export certificate for transmission to user;
destroy certificate objects;
```

Finally, the user receives the signed certificate from the CA and processes it as required, typically writing it to a public key keyset or updating a private key keyset:

```
import certificate;
check validity and signature on certificate;
write certificate to keyset;
destroy certificate object;
```

The details on performing these operations are covered in the following sections.

Creating/Destroying Certificate Objects

Certificates are accessed as certificate objects which work in the same general manner as the other container objects used by cryptlib. You create the certificate object with **cryptCreateCert**, specifying the type of certificate you want to create. Once you've finished with the object, you use **cryptDestroyCert** to destroy it:

```
CRYPT_CERTIFICATE cryptCert;

cryptCreateCert( &cryptCert, certificateType );

/* Work with the certificate */

cryptDestroyCert( cryptCert );
```

The available certificate types are:

Certificate Type	Description
CRYPT_CERTTYPE_ATTRCERT	Attribute certificate.
CRYPT_CERTTYPE_CERTCHAIN	Certificate chain
CRYPT_CERTTYPE_CERTIFICATE	Certificate or CA certificate.
CRYPT_CERTTYPE_CERTREQUEST	Certification request
CRYPT_CERTTYPE_CRL	Certificate revocation.

Note that the CRYPT_CERTIFICATE is passed to **cryptCreateCert** by reference, as the function modifies it when it creates the certificate object. In all other routines, CRYPT_CERTIFICATE is passed by value.

You can also create a certificate object by reading a certificate from a public key database, as explained in "Key Databases" on page 40. Unlike **cryptCreateCert**, this will read a complete certificate into a certificate object, while **cryptCreateCert** only creates a certificate template which still needs various details such as the public key and key owners name filled in.

A third way to create a certificate object is to import an encoded certificate using **cryptImportCert**, which is explained in more detail below. Like the public key read functions, this imports a complete certificate into a certificate object.

Working with Certificate Attributes

Certificate objects contain a number of basic attributes and an optional collection of often complex data structures and components. cryptlib provides a variety of mechanisms for working with them. The attributes in a certificate object can be broken up into three basic types:

1. Basic certificate attributes such as the public key and timestamp/validity information.
2. Identification information such as the certificate subject and issuer name.
3. Certificate extensions which can contain almost anything. These are covered in "Certificate Extensions" on page 97.

Although cryptlib provides the ability to manipulate all of these attributes, in practice you only need to handle a small subset of them yourself. The rest will be set to sensible defaults by cryptlib.

Apart from this, certificate attributes are handled in the standard way described in "Working with Object Attributes" on page 17.

Certificate Structures

Certificates, attribute certificates, certification requests, and CRL's have their own, often complex, structures which are encoded and decoded for you by cryptlib. Although cryptlib provides the ability to control the details of each certificate object in great detail if you require this, in practice you should leave the certificate management to cryptlib. If you don't fill in the non-mandatory fields, cryptlib will fill them in for you with default values when you sign the certificate object.

Certificate chains are composite objects which contain within them one or more complete certificates. These are covered in more detail in "Certificate Chains" on page 114.

Attribute Certificate Structure

An X.509 attribute certificate has the following structure:

Field	Description
Version	The version number defines the attribute certificate version and is filled in automatically by cryptlib when the certificate is signed.
Owner	The owner identifies the owner of the attribute certificate and is explained in more detail further on. If you add a certificate request using CRYPT_CERTINFO_CERTREQUEST or a certificate using CRYPT_CERTINFO_USERCERTIFICATE, this field will be filled in for you.

Field	Description
	This is a composite field which you must fill in yourself (unless it has already been filled in from a certification request or certificate).
Issuer	The issuer name identifies the attribute certificate signer (usually an authority, the attribute-certificate version of a CA), and is filled in automatically by cryptlib when the certificate is signed.
Signature	The signature algorithm identifies the algorithm used to sign the attribute certificate, and is filled in automatically by cryptlib when the certificate is signed.
SerialNumber	The serial number is unique for each attribute certificate issued by an authority, and is filled in automatically by cryptlib when the certificate is signed. You can obtain the value of this field with CRYPT_CERTINFO_SERIALNUMBER, but you can't set it. If you try to set it, cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you don't have permission to set this field. The serial number is returned as a binary string and not as a numeric value, since it is often 15-20 bytes long. cryptlib doesn't use strict sequential numbering for the certificates it issues since this would make it very easy for a third party to determine how many certificates a CA is issuing at any time.
Validity	The validity period defines the period of time over which an attribute certificate is valid. CRYPT_-CERTINFO_VALIDFROM specifies the validity start period, and CRYPT_CERTINFO_VALIDTO specifies the validity end period, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer). If you don't set these, cryptlib will set them for you when the attribute certificate is signed so that the certificate validity starts on the day of issue and ends one year later. You can change the default validity period using the cryptlib configuration option CRYPT_OPTION_-CERT_VALIDITY as explained in "Miscellaneous Topics" on page 146. By default, cryptlib will enforce validity period nesting when generating an attribute certificate, so that the validity period of an attribute certificate will be constrained to lie within the validity period of the authority certificate which signed it. If this isn't done, some software will treat the certificate as being invalid, or will regard it as having expired once the authority certificate which signed it expires. You can change the enforcement of validity period nesting using the cryptlib configuration options CRYPT_-OPTION_CERT_ENCODE_VALIDITYNESTING (to control the creation of certificates with nested validity) and CRYPT_OPTION_CERT_DECODE_-VALIDITYNESTING (to control the processing of certificates with nested validity) as explained in

Field	Description
	"Miscellaneous Topics" on page 146.
Attributes	The attributes field contains a collection of attributes for the certificate owner. Since no standard attributes had been defined at the time of the last X.509 attribute certificate committee draft, cryptlib doesn't currently support attributes in this field. When attributes are defined, cryptlib will support them.
IssuerUniqueID	The issuer unique ID was added in X.509v2, but its use has been discontinued. If this string field is present in existing attribute certificates you can obtain its value using CRYPT_CERTINFO_ISSUERUNIQUEID, but you can't set it. If you try to set it, cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you have no permission to set this field.
Extensions	Certificate extensions allow almost anything to be added to an attribute certificate and are covered in more detail in "Certificate Extensions" on page 97.

Certificate Structure

An X.509 certificate has the following structure:

Field	Description
Version	The version number defines the certificate version and is filled in automatically by cryptlib when the certificate is signed. It is used mainly for marketing purposes to claim that software is X.509v3 compliant (even when it isn't).
SerialNumber	The serial number is unique for each certificate issued by a CA, and is filled in automatically by cryptlib when the certificate is signed. You can obtain the value of this field with CRYPT_CERTINFO_-SERIALNUMBER, but you can't set it. If you try to set it, cryptlib will return CRYPT_ERROR_-PERMISSION to indicate that you don't have permission to set this field. The serial number is returned as a binary string and not as a numeric value, since it is often 15-20 bytes long. cryptlib doesn't use strict sequential numbering for the certificates it issues since this would make it very easy for a third party to determine how many certificates a CA is issuing at any time.
SignatureAlgorithm	The signature algorithm identifies the algorithm used to sign the certificate, and is filled in automatically by cryptlib when the certificate is signed.
IssuerName	The issuer name identifies the certificate signer (usually a CA), and is filled in automatically by cryptlib when the certificate is signed.
Validity	The validity period defines the period of time over which a certificate is valid. CRYPT_CERTINFO_-VALIDFROM specifies the validity start period, and CRYPT_CERTINFO_VALIDTO specifies the validity end period, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp

Field	Description
	value (in C and C++ this is a `time_t`, usually a signed long integer). If you don't set these, cryptlib will set them for you when the certificate is signed so that the certificate validity starts on the day of issue and ends one year later. You can change the default validity period using the cryptlib configuration option CRYPT_OPTION_CERT_VALIDITY as explained in "Miscellaneous Topics" on page 146.
	By default, cryptlib will enforce validity period nesting when generating a certificate, so that the validity period of a certificate will be constrained to lie within the validity period of the CA certificate which signed it. If this isn't done, some software will treat the certificate as being invalid, or will regard it as having expired once the CA certificate which signed it expires. You can change the enforcement of validity period nesting using the cryptlib configuration options CRYPT_OPTION_CERT_ENCODE_-VALIDITYNESTING (to control the creation of certificates with nested validity) and CRYPT_OPTION_CERT_DECODE_-VALIDITYNESTING (to control the processing of certificates with nested validity) as explained in "Miscellaneous Topics" on page 146.
SubjectName	The subject name identifies the owner of the certificate and is explained in more detail further on. If you add the subject public key info from a certification request using CRYPT_CERTINFO_CERTREQUEST, this field will be filled in for you.
	This is a composite field which you must fill in yourself (unless it has already been filled in from a certification request).
SubjectPublicKey-Info	The subject public key info contains the public key for this certificate. You can specify the public key with CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, and provide either an encryption context or a certificate object which contains a public key. You can also add a certification request with CRYPT_CERTINFO_-CERTREQUEST, which fills in the subject public key info, subject name, and possibly some certificate extensions.
	This is a numeric field which you must fill in yourself.
IssuerUniqueID SubjectUniqueID	The issuer and subject unique ID were added in X.509v2, but their use has been discontinued. If these string fields are present in existing certificates you can obtain their values using CRYPT_CERTINFO_-ISSUERUNIQUEID and CRYPT_CERTINFO_-SUBJECTUNIQUEID, but you can't set them. If you try to set them, cryptlib will return CRYPT_ERROR_-PERMISSION to indicate that you have no permission to set these fields.
Extensions	Certificate extensions were added in X.509v3. Extensions allow almost anything to be added to a certificate and are covered in more detail in "Certificate Extensions" on page 97.

Certification Request Structure

A PKCS #10 certification request has the following structure:

Field	Description
Version	The version number defines the certification request version and is filled in automatically by cryptlib when the request is signed.
SubjectName	The subject name identifies the owner of the certification request and is explained in more detail further on. This is a composite field which you must fill in yourself.
SubjectPublicKey-Info	The subject public key info contains the public key for this certification request. You can specify the public key with CRYPT_CERTINFO_-SUBJECTPUBLICKEYINFO, and provide either an encryption context or a certificate object which contains a public key. This is a composite field which you must fill in yourself.
Extensions	Extensions allow almost anything to be added to a certification request and are covered in more detail in “Certificate Extensions” on page 97.

PKCS #10 certification requests can have one of two encodings, the standard one which is used by most implementations, and an alternative encoding which arises from an error in the PKCS #10 specification (the default encoding encodes an empty set of extensions as a zero-length sequence while the alternative encoding omits the extensions field if none are present). Unfortunately there is no way to tell which of the two encodings will be accepted by a CA, but most CA’s accept the form which is used by cryptlib by default (cryptlib itself will accept either form). To enable the use of the alternative encoding, you can use the cryptlib configuration option CRYPT_OPTION_CERT_PKCS10ALT as explained in “Miscellaneous Topics” on page 146.

cryptlib will also read Netscape SignedPublicKeyAndChallenge records, converting the data in them into the equivalent certification request information so that the resulting object appears as a standard certification request. Since the SignedPublicKeyAndChallenge doesn’t include anything other than a public key, you will need to manually add a subject name and any necessary extensions when you add the certification request information to a certificate. In addition because the object created from a SignedPublicKeyAndChallenge isn’t a genuine certification request object, you can’t re-export it from the object as a certification request or save it to a keyset.

CRL Structure

An X.509 CRL has the following structure:

Field	Description
Version	The version number defines the CRL version and is filled in automatically by cryptlib when the CRL is signed.
SignatureAlgorithm	The signature algorithm identifies the algorithm used to sign the CRL, and is filled in automatically by cryptlib when the CRL is signed.

Field	Description
IssuerName	The issuer name identifies the CRL signer, and is filled in automatically by cryptlib when the CRL is signed.
ThisUpdate NextUpdate	The update time specifies when the CRL was issued, and the next update time specifies when the next CRL will be issued. CRYPT_CERTINFO_THISUPDATE specifies the current CRL issue time, and CRYPT_CERTINFO_NEXTUPDATE specifies the next CRL issue time, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer). If you don't set these, cryptlib will set them for you when the CRL is signed so that the issue time is the day of issue and the next update time is 90 days later. You can change the default update interval using the cryptlib configuration option CRYPT_OPTION_CERT_-UPDATEINTERVAL as explained in "Miscellaneous Topics" on page 146.
UserCertificate	The user certificate identifies the certificates which are being revoked in this CRL. The certificates must be ones which were issued using the CA certificate which is being used to issue the CRL. If you try to revoke a certificate which was issued using a different CA certificate, cryptlib will return a CRYPT_ERROR_-INVALID error when you add the certificate or sign the CRL to indicate that the certificate can't be revoked using this CRL. You can specify the certificates to be revoked with CRYPT_CERTINFO_-USERCERTIFICATE. This is a numeric field, and the only one which you must fill in yourself.
RevocationDate	The revocation date identifies the date on which a certificate was revoked. You can specify the revocation date with CRYPT_CERTINFO_-REVOCATIONDATE, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer). If you don't set it, cryptlib will set it for you to the date on which the CRL was signed. The revocation date you specify applies to the last certificate added to the list of revoked certificates. If no certificates have been added yet, it will be used as a default date which applies to all certificates for which no revocation date is explicitly set.

Basic Certificate Management

With the information from the previous section, it's now possible to start creating basic certificate objects. To create a certification request, you would do the following:

```
CRYPT_CERTIFICATE cryptCertRequest;
void *certRequest;
int certRequestLength;
```

```
/* Create a certification request and add the public key to it */
cryptCreateCert( cryptCertRequest, CRYPT_CERTTYPE_CERTREQUEST );
cryptSetAttribute( cryptCertRequest,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, pubKeyContext );

/* Add identification information */
/* ... */

/* Sign the certification request with the private key and export it
   */
cryptSignCert( cryptCertRequest, privKeyContext );
cryptExportCert( NULL, &certRequestLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertRequest );
certRequest = malloc( certRequestLength );
cryptExportCert( certRequest, &certRequestLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertRequest );

/* Destroy the certification request */
cryptDestroyCert( cryptCertRequest );
```

This simply takes a public key, adds some identification information to it (the details of this will be covered later), signs it, and exports the encoded certification request for transmission to a CA. Since cryptlib will only copy across the appropriate key components, there's no need to have a separate public and private key context, you can add the same private key context which you'll be using to sign the certification request to supply the CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO information and cryptlib will use the appropriate data from it.

To process the certification request and convert it into a certificate, the CA does the following:

```
CRYPT_CERTIFICATE cryptCert, cryptCertRequest;
void *cert;
int certLength;

/* Import the certification request and check its validity */
cryptImportCert( certRequest, &cryptCertRequest );
cryptCheckCert( cryptCertRequest, CRYPT_UNUSED );

/* Create a certificate and add the information from the certification
   request to it */
cryptCreateCert( &cryptCert, CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCert, CRYPT_CERTINFO_CERTREQUEST,
   cryptCertRequest );

/* Sign the certificate with the CA's private key and export it */
cryptSignCert( cryptCert, caPrivateKey );
cryptExportCert( NULL, &certLength, CRYPT_CERTFORMAT_CERTIFICATE,
   cryptCert );
cert = malloc( certLength );
cryptExportCert( cert, &certLength, CRYPT_CERTFORMAT_CERTIFICATE,
   cryptCert );

/* Destroy the certificate and certification request */
cryptDestroyCert( cryptCert );
cryptDestroyCert( cryptCertRequest );
```

In this case the CA has put together a very minimal X.509v1 certificate which can be processed by most software but which is rather limited in the amount of control which the CA and end user has over the certificate, since no constraint or usage control information has been added to the certificate. By default cryptlib actually adds the necessary fields for a full X.509v3 certificate, but this won't contain all the information which would be available if the CA explicitly creates an X.509v3 certificate. Creating full X.509v3 certificates involves the use of certificate extensions and is covered in more detail later.

To check the signed certificate returned from the CA and add it to a keyset, the user does the following:

```
CRYPT_CERTIFICATE cryptCert;

/* Import the certificate and check its validity */
cryptImportCert( cert, &cryptCert );
```

```
cryptCheckCert( cryptCert, caCertificate );

/* Add the certificate to a keyset */
/* ... */

/* Destroy the certificate */
cryptDestroyCert( cryptCert );
```

To obtain information about the key contained in a certificate you can read the appropriate attributes just like an encryption context, for example CRYPT_CTXINFO_ALGO will return the encryption/signature algorithm type, CRYPT_CTXINFO_NAME_ALGO will return the algorithm name, and CRYPT_CTXINFO_KEYSIZE will return the key size.

Certificate Identification Information

Traditionally, certificate objects have been identified by a construct called an X.500 Distinguished Name (DN). In ISO/ITU terminology, the DN defines a path through an X.500 directory information tree (DIT) via a sequence of Relative Distinguished Name (RDN) components which in turn consist of a set of one or more Attribute Value Assertions (AVA's) per RDN. The description then goes on in this manner for another hundred-odd pages, and includes diagrams which are best understood when held upside down in front of a mirror.

To keep things manageable, cryptlib goes to some lengths to hide the complexity involved by handling the processing of DN's for you. A cryptlib DN can contain the following text string components:

Component	Description
CountryName (C)	The two-letter international country code (specified in ISO 3166 in case you ever need to look it up). Examples of country codes are 'US' and 'NZ'. You can specify the country with CRYPT_CERTINFO_COUNTRYNAME. This is a field which you must fill in.
Organization (O)	The organisation for which the certificate will be issued. Examples of organisations are 'Microsoft Corporation' and 'Verisign, Inc'. You can specify the organisation with CRYPT_CERTINFO_-ORGANIZATIONNAME.
OrganisationalUnit-Name (OU)	The division of the organisation for which the certificate will be issued. Examples of organisational units are 'Sales and Marketing' and 'Purchasing'. You can specify the organisational unit with CRYPT_CERTINFO_-ORGANIZATIONALUNITNAME.
StateOrProvinceName (SP)	The state or province in which the certificate owner is located. Examples of state or province names are 'Utah', 'Steyrmark', and 'Puy de Dôme'. You can specify the state or province with CRYPT_-CERTINFO_STATEORPROVINCENAME.
LocalityName (L)	The locality in which the certificate owner is located. Examples of localities are 'San Jose', 'Seydisfjördur', and 'Mönchengladbach'. You can specify the localitywith CRYPT_CERTINFO_-LOCALITYNAME.
CommonName (CN)	The name of the certificate owner, which can be either a person such as 'John Doe', a business role such as 'Accounts Manager', or even an entity like

Component	Description
	'Laser Printer #6'. You can specify the common name with CRYPT_CERTINFO_-COMMONNAME. This is a field which you must fill in.

All DN components except the country name are limited to a maximum of 64 characters (this is a requirement of the X.500 standard which defines the certificate format and use). cryptlib provides the CRYPT_MAX_TEXTSIZE constant for this limit. Note that this defines the number of characters and not the number of bytes, so that a Unicode string could be several times as long in bytes as it would be in characters, depending on which data type the system uses to represent Unicode characters.

The complete DN can be used for a personal key used for private purposes (for example to perform home banking or send private email) or for a key used for business purposes (for example to sign business agreements). The difference between the two key types is that a personal key will identify someone as a private individual, whereas a business key will identify someone terms of the organisation for which they work.

A DN must always contain a country name and a common name, and should generally also contain one or more of the other components. If a DN doesn't contain at least the two minimum components, cryptlib will return CRYPT_ERROR_NOTINITED with an extended error indicating the missing component when you try to sign the certificate object.

Some software generates certification requests (and certificates) with incorrectly encoded DN's. By default cryptlib will correct the encoding when it generates a certificate from a certification request, however a subset of the software will check that it receives back the same invalid encoding in the certificate which it generated in the certification request, and reject the certificate with the correct encoding. To disable the correction of the encoding, you can use the cryptlib configuration option CRYPT_OPTION_CERT_FIXSTRINGS as explained in "Miscellaneous Topics" on page 146.

Realising that DN's are too complex and specialised to handle many types of current certificate usage, more recent revisions of the X.509 standard were extended to include a more generalised name format called a GeneralName, which is explained in more detail in "Extended Certificate Identification Information" on page 88.

DN Structure for Business Use

For business use, the DN should include the country code, the organisation name, an optional organisational unit name, and the common name. An example of a DN structured for business use would be:

C = US
O = Cognitive Cybernetics Incorporated
OU = Research and Development
CN = Paul Johnson

This is a key which is used by an individual within an organisation. It might also describe a role within the organisation, in this case a class of certificate issuer in a CA:

C = DE
O = Individual Network Certification Authority
CN = Class 1 CA

It might even describe an entity with no direct organisational role:

C = AT
O = Erste Allgemeine Verunsicherung
CN = Mail Gateway

In this last case the certificate might be used by the mail gateway machine to authenticate data transmitted through it.

DN Structure for Private Use

For private, non-business use, the DN should include the country code, an optional state or province name, the locality name, and the common name. An example of a DN structured for private use would be:

C = US
SP = California
L = El Cerrito
CN = Dave Taylor

Other DN Structures

It's also possible to combine components of the above DN structures, for example if an organisation has divisions in multiple states you might want to include the state or province name component in the DN:

C = US
SP = Michigan
O = Last National Bank
CN = Personnel Manager

Another example would be:

C = US
L = Area 51
O = Hanger 18
OU = X.500 Standards Designers
CN = John Doe

Working with Distinguished Names

Now that the details of DN's have been covered, you can use them to add identification information to certification requests and certificates. For example to add the business DN shown earlier to a certification request you would use:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create certification request and add other components */
/* ... */

/* Add identification information */
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_COUNTRYNAME,
   "US", 2 );
cryptSetAttributeString( cryptCertRequest,
   CRYPT_CERTINFO_ORGANIZATIONNAME, "Cognitive Cybernetics
   Incorporated", 34 );
cryptSetAttributeString( cryptCertRequest,
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, "Research and Development",
   24 );
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_COMMONNAME,
   "Paul Johnson", 12 );

/* Sign certification request and transmit to CA */
/* ... */
```

The same process applies for adding other types of identification information to a certification request or certificates. Note that cryptlib sorts the DN components into the correct order when it creates the certification request or certificate, so there's no need to specify them in strict order as in the above code.

Extended Certificate Identification Information

In the early to mid 1990's when it became clear that the Internet was going to be the driving force behind certificate technology, X.509 was amended to allow a more

general-purpose type of identification than the complex and specialised DN. This new form was called the GeneralName, since it provided far more flexibility than the original DN. A GeneralName can contain an email address, a URL, an IP address, an alternative DN which doesn't follow the strict rules for the main certificate DN (it could for example contain a postal or street address), less useful components like X.400 and EDI addressing information, and even user-defined information which might be used in a certificate, for example medical patient, taxpayer, or social security ID's.

As with DN's, cryptlib goes to some lengths to hide the complexity involved in handling GeneralNames (recall the previous technical description of a DN, and then consider that this constitutes only a small portion of the entire GeneralName). Like a DN, a GeneralName can contain a number of components. Unless otherwise noted, the components are all text strings.

Component	Description
DirectoryName	A DN which can contain supplementary information which doesn't fit easily into the main certificate DN. You can specify this value with CRYPT_CERTINFO_DIRECTORYNAME.
DNSName	An Internet hosts' fully-qualified domain name. You can specify this value with CRYPT_CERTINFO_DNSNAME.
EDIPartyName.Name-Assigner EDIPartyName.Party-Name	An EDI assigner-and-value pair with the EDI name assigner specified by CRYPT_CERTINFO_-EDIPARTYNAME_NAMEASSIGNER and the party name specified by CRYPT_CERTINFO_-EDIPARTYNAME_PARTYNAME.
IPAddress	An IP address as per RFC 791, containing a 4-byte binary string in network byte order. You can specify this value with CRYPT_CERTINFO_-IPADDRESS.
OtherName.TypeID OtherName.Value	A user-defined type-and-value pair with the type specified by CRYPT_CERTINFO_-OTHERNAME_TYPEID and the value specified by CRYPT_CERTINFO_OTHERNAME_VALUE. The type is an ISO object identifier and the corresponding value is a binary string which can contain anything, identified by the object identifier (if you know what this is then you should also know how to obtain one).
RegisteredID	An object identifier (again, if you know what this is then you should know how to obtain one). You can specify this value with CRYPT_CERTINFO_-REGISTEREDID.
RFC822Name	An email address. You can specify this value with CRYPT_CERTINFO_RFC822NAME. For compatibility with the older (obsolete) PKCS #9 emailAddress attribute, cryptlib will also accept CRYPT_CERTINFO_EMAIL to specify this field.
UniformResource-Identifier	A URL for either FTP, HTTP, or LDAP access as per RFC 1738. You can specify this value with CRYPT_CERTINFO_-UNIFORMRESOURCEIDENTIFIER.

Of the above GeneralName components, the most useful ones are the RFC822Name (to specify an email address), the DNSName (to specify a server address), and the

UniformResourceIdentifier (to specify a home page or FTP directory). Somewhat less useful is the DirectoryName, which can specify additional information which doesn't fit easily into the main certificate DN. The other components should be avoided unless you have a good reason to require them (that is, don't use them just because they're there).

Working with GeneralNames

Now that the details of GeneralNames have been covered, you can use them to add additional identification information to certificate requests and certificates. For example to add an email address and home page URL to the certification request shown earlier you would use:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create certification request and add other components */
/* ... */

/* Add identification information */
/* ... */

/* Add additional identification information */
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_RFC822NAME,
   "paul@cci.com", 12 );
cryptSetAttributeString( cryptCertRequest,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER,
   "http://www.cci.com/~paul", 23 );

/* Sign certification request and transmit to CA */
/* ... */
```

Although GeneralNames are commonly used to identify a certificates owner just like a DN, they are in fact a certificate extension rather than a basic attribute, and are covered in more detail in "Certificate Extensions" on page 97.

Certificate Fingerprints

Certificates are sometimes identified through "fingerprints" which constitute either an MD5 or SHA-1 hash of the certificate data (the most common form is an MD5 hash). You can obtain a certificates fingerprint by reading its CRYPT_CERTINFO_-FINGERPRINT attribute, which yields the default (MD5) fingerprint for the certificate. You can also explicitly query a particular fingerprint type with CRYPT_-CERTINFO_FINGERPRINT_MD5 and CRYPT_CERTINFO_FINGERPRINT_-SHA:

```
unsigned char fingerprint[ CRYPT_MAX_HASHSIZE ]
int fingerprintSize;

cryptGetAttributeString( certificate, CRYPT_CERTINFO_FINGERPRINT,
   &fingerprint, &fingerprintSize );
```

This will return the certificate fingerprint.

Importing/Exporting Certificates

If you have an encoded certificate which was obtained elsewhere, you can import it into a certificate object using **cryptImportCert**. There are more than a dozen mostly incompatible formats for communicating certificates, of which cryptlib will handle all the generally useful and known ones. This includes straight binary certification requests, certificates, attribute certificates, and CRL's (usually stored with a .der file extension when they are saved to disk), PKCS #7 certificate chains, and Netscape certificate sequences. Certificates can also be protected with base64 armouring and BEGIN/END CERTIFICATE delimiters, which is the format used by some web browsers and other applications. When transferred via HTTP using the Netscape-specific format, certificates, certificate chains, and Netscape certificate sequences are identified with have the MIME content types `application/x-x509-user-cert`, `application/x-x509-ca-cert`, and `application/x-x509-`

email-cert, depending on the certificate type (cryptlib doesn't use the MIME content type since the certificate itself provides a far more reliable indication of its intended use than the easily-altered MIME content type).. Portions of certificate requests can also be communicated as Netscape SignedPublicKeyAndChallenge objects, which are imported by cryptlib as incomplete certification requests. Finally, certification requests and certificate chains can be encoded with the MIME / S/MIME content types application/pkcs-signed-data, application/x-pkcs-signed-data, application/pkcs-certs-only, application/x-pkcs-certs-only, application/pkcs10, or application/x-pkcs10. These are usually stored with a .p7c extension (for pure certificate chains), a .p7s extension (for signatures containing a certificate chain), or a .p10 extension (for certification requests) when they are saved to disk.

cryptlib will import any of the previously described certificate formats if they are encoded in this manner. To import a certificate object you would use:

```
CRYPT_CERTIFICATE cryptCert;

/* Import the certificate object from the encoded certificate */
cryptImportCert( certificate, &cryptCert );
```

Note that the CRYPT_CERTIFICATE is passed to **cryptImportCert** by reference, as the function modifies it when it creates the certificate object.

Many certificates produced by current software have incorrect or invalid encodings, and **cryptImportCert** will reject them with a CRYPT_ERROR_BADDATA error. You can disable the checking of encoding validity by setting the cryptlib configuration option CRYPT_OPTION_CERT_CHECKENCODING to false, which will often allow the certificate to be imported (cryptlib goes to a great deal of effort to try to handle incorrectly encoded certificates). However the correct processing of certificates which require disabling of validity checking in order to be handled cannot be guaranteed (in other words disabling this check voids your warranty).

Some certificate objects may contain unrecognised critical extensions (certificate extensions are covered in "Certificate Extensions" on page 97) which require that the certificate be rejected by cryptlib. If a certificate contains an unrecognised critical extension, cryptlib will return a CRYPT_ERROR_PERMISSION error to indicate that you have no permission to use this object. Since the use of critical extensions can lead to logical consistency problems in some instances, cryptlib provides the ability to disable the rejection of these extensions with the cryptlib option CRYPT_OPTION_CERT_DECODE_CRITICAL as explained in "Miscellaneous Topics" on page 146. This option affects the ability to rely on certificate attributes, so you should only use it if you're familiar with the implications and intended uses of critical extensions.

All the parameters and information needed to create the certificate object are a part of the certificate, and **cryptImportCert** takes care of initialising the certificate object and setting up the attributes and information inside it. The act of importing a certificate simply decodes the information and initialises a certificate object, it doesn't check the signature on the certificate. To check the certificates signature you need to use **cryptCheckCert**, which is explained further on.

There may be instances in which you're not exactly certain of the type of certificate object you have imported (for example importing a file with a .der extension could create a certificate request, a certificate, an attribute certificate, or a certificate chain object depending on the file contents). In order to determine the exact type of the object, you can read its CRYPT_CERTINFO_CERTTYPE attribute:

```
CRYPT_CERTTYPE_TYPE certType;

cryptGetAttribute( certificate, CRYPT_CERTINFO_CERTTYPE, &certType );
```

This will return the type of the imported object.

You can export a signed certificate from a certificate object using **cryptExportCert**:

```
CRYPT_CERTIFICATE cryptCert;
```

```
void *certificate;
int certificateLength

/* Allocate memory for the encoded certificate */
certificate = malloc( ... );

/* Export the encoded certificate from the certificate object */
cryptExportCert( certificate, &certificateLength, certFormatType,
   cryptCert );
```

cryptlib will export certificates in any of the formats in which it can import them. The available `certFormat` types are:

Format Type	Description
CRYPT_CERTFORMAT_- CERTCHAIN	A certificate encoded as a PKCS #7 certificate chain.
CRYPT_CERTFORMAT_- CERTIFICATE	A certification request, certificate, or CRL in binary data format. The certificate object is encoded according to the ASN.1 distinguished encoding rules. This is the normal certificate encoding format.
CRYPT_CERTFORMAT_- SMIME_CERTIFICATE	As CRYPT_CERTFORMAT_- CERTIFICATE but with MIME / S/MIME encoding of the binary data. This format is only usable with certification requests and certificate chains as the encoding for the other certificate object types isn't defined.
CRYPT_CERTFORMAT_- TEXT_CERTCHAIN	As CRYPT_CERTFORMAT_CERTCHAIN but with base64 armouring of the binary data.
CRYPT_CERTFORMAT_- TEXT_CERTIFICATE	As CRYPT_CERTFORMAT_- CERTIFICATE but with base64 armouring of the binary data.

If the object you are exporting is a complete certificate chain rather than an individual certificate then these options work somewhat differently. The details of exporting certificate chains are covered in "Certificate Chains" on page 114.

The resulting encoded certificate is placed in the memory buffer pointed to by `certificate`, and the length is stored in `certificateLength`. This leads to a small problem: How do you know how big to make the buffer? The answer is to use **cryptExportCert** to tell you. If you pass in a null pointer for `certificate`, the function will set `certificateLength` to the size of the resulting encoded certificate, but not do anything else. You can then use code like:

```
cryptExportCert( NULL, &certificateLength, certFormatType, cryptCert
   );
certificate = malloc( certificateLength );
cryptExportCert( certificate, &certificateLength, certFormatType,
   cryptCert );
```

to create the encoded certificate.

Alternatively, you can just reserve a reasonably sized block of memory and use that to hold the encoded certificate. "Reasonably sized" means a few Kb, a 4K block is plenty (a certificate for a 1024-bit key without certificate extensions is typically about 700 bytes long if encoded using any of the binary formats, or 900 bytes long if encoded using any of the text formats).

If the certificate is one which you've created yourself rather than importing it from an external source, you need to add various data items to the certificate and then sign it before you can export it. If you try to export an incompletely prepared certificate such as a certificate in which some required fields haven't been filled in or one which hasn't been signed, **cryptExportCert** will return the error CRYPT_ERROR_-

NOTINITED to tell you that the certificate information hasn't been completely set up.

Signing/Verifying Certificates

Once a certificate object contains all the information you want to add to it, you need to sign it in order to transform it into its final state in which the data in it can be written to a keyset (if the objects final state is a key certificate or CA certificate) or exported from the object. Before you sign the certificate, the information within it exists only in a very generic and indeterminate state. After signing it, the information is turned into a fixed certificate, CA certificate, certification request, or CRL, and no further changes can be made to it.

You can sign the information in a certificate object with **cryptSignCert**:

```
CRYPT_CONTEXT privateKeyContext;

/* Sign the certificate object */
cryptSignCert( cryptCert, privateKeyContext );
```

There are some restrictions on the types of keys which can be used to sign certificate objects. These restrictions are imposed by the way in which certificates and certificate-related items are encoded, and are as follows:

Certificate Type	Can be Signed By
Attribute certificate	Private key associated with an authority certificate.
Certificate	Private key associated with a CA certificate. This can also be a self-signed certificate, but a lot of software will then decide that the resulting certificate is a CA certificate even though it isn't.
CA certificate	Private key associated with a CA certificate (when one CA certifies another) or the private key from which the certificate being signed was created (when the CA certifies itself).
Certification request	Private key associated with the certification request.
Certificate chain	Private key associated with a CA certificate.
CRL	Private key associated with the CA certificate which was used to issue the certificates which are being revoked.

Once a certificate item has been signed, it can no longer be modified or updated using the usual certificate manipulation functions, and any attempt to update information in it will return CRYPT_ERROR_PERMISSION to indicate that you have no permission to modify the object. If you want to add or delete data to or from the certificate item, you have to start again with a new certificate object. You can determine whether a certificate item has been signed and can therefore no longer be changed by reading its CRYPT_CERTINFO_IMMUTABLE attribute:

```
int isImmutable;

cryptGetAttribute( certificate, CRYPT_CERTINFO_IMMUTABLE, &isImmutable
   );
```

If the result is set to true (a nonzero value), the certificate item can no longer be changed.

If you're creating a self-signed certificate signed by a raw private key with no certificate information associated with it, you need to set the CRYPT_CERTINFO_-SELFSIGNED attribute before you sign it otherwise cryptlib will flag the attempt to sign using a non-certificate key as an error. Non-certificate private keys can only be

used to create self-signed certificates (if CRYPT_CERTINFO_SELFSIGNED is set) or certification requests.

If the object being signed contains unrecognised extensions, cryptlib will not include them in the signed object (signing extensions of unknown significance is a risky practice for a CA, which in most jurisdictions would be held liable for any arising problems). If you If you want to be able to sign unrecognised extensions, you can enable this with the cryptlib configuration option CRYPT_OPTION_CERT_-SIGNUNRECOGNISEDATTRIBUTES as explained in "Miscellaneous Topics" on page 146.

By default cryptlib will create an X.509v3 certificate when you sign a certificate object to create a certificate, automatically adding or updating the standard X.509v3 attributes for you — cryptlib will automatically adjust the certificate attributes to ensure that any certificate you create contains appropriate and valid basic attributes. If you want to create X.509v1 certificates or disable the automatic attribute handling in order to obtain precise control over the basic attributes (for example to set the attributes to nonstandard values), you can disable the automatic creation of X.509v3 certificates with the cryptlib configuration option CRYPT_OPTION_CERT_-CREATEV3CERT as explained in "Miscellaneous Topics" on page 146.

As usual, you should experiment with the signature(s) you use to determine which ones work with the software you need to interoperate with, and how the software interprets the signatures you create.

You can verify the signature on a certificate object using **cryptCheckCert** and the public key or key certificate corresponding to the private key which was used to sign the certificate (you can also pass in a private key if you want, **cryptCheckCert** will only use the public key components, although you shouldn't really be in possession of someone elses private key). To perform the check using a public key context you'd use:

```
CRYPT_CONTEXT publicKeyContext;

/* Check the signature on the certificate object information using the
   public key */
cryptCheckCert( cryptCert, publicKeyContext );
```

A signature check using a key certificate is similar, except that it uses a certificate object rather than a public key context.

If the certificate object is self-signed, you can pass in CRYPT_UNUSED as the second parameter and **cryptCheckCert** will use the key contained in the certificate object to check its validity. You can determine whether a certificate object is self-signed by reading its CRYPT_CERTINFO_SELFSIGNED attribute. Certification requests are always self-signed, and certificate chains count as self-signed if they contain a self-signed top-level certificate which can be used to recursively check the rest of the chain. If the certificate object is a CA certificate which is signing itself (in other words if it's a self-signed cert), you can also pass the certificate as the second parameter in place of CRYPT_UNUSED, this has the same effect since the certificate is both the signed and signing object.

If the certificate is invalid (for example because it has expired or because some certificate usage constraint hasn't been met), cryptlib will return CRYPT_ERROR_-INVALID to indicate that the certificate isn't valid. This value is returned regardless of whether the signature check succeeds or fails. You can find out the exact nature of the problem by reading the extended error attributes as explained further on.

If the signature create/check operation fails due to incorrect public or private key parameters, the function will return CRYPT_ERROR_FAILED to indicate that the operation failed. If the signature on the certificate object is corrupt and couldn't be processed, the function will return CRYPT_ERROR_BADDATA. In general CRYPT_ERROR_FAILED will be returned for incorrect public or private key parameters and CRYPT_ERROR_BADDATA will be returned if the signature has been corrupted. Finally, if the certificate data has been changed and the signature invalidated, the function will return CRYPT_ERROR_SIGNATURE. You can treat

all three of these as “signature generation/check failed” unless you want to include special-case error handling for them.

If the signing/signature check key is stored in an encryption context with a certificate associated with it or in a key certificate, there may be constraints on the key usage which are imposed by the certificate. If the key can’t be used for the signature or signature check operation, the function will return CRYPT_ERROR_INVALID to indicate that the key isn’t valid for this operation. You can find out more about the exact nature of the problem by reading the extended error attributes as explained further on.

Certificate Trust Management

In order to provide extended control over certificate usage, cryptlib allows you to both further restrict the usage given in the certificates CRYPT_CERTINFO_-KEYUSAGE attribute and to specify whether a given certificate should be implicitly trusted, avoiding the requirement to process a (potentially large) chain of certificates in order to determine the certificates validity.

Controlling Certificate Usage

You can control the way a certificate can be used by setting its CRYPT_-CERTINFO_TRUSTED_USAGE attribute, which provides extended control over the usage types which a certificate is trusted for. This attribute works by further restricting the usage specified by the CRYPT_CERTINFO_KEYUSAGE attribute, acting as a mask for the standard key usage so that a given usage is only permitted if it’s allowed by both the key usage and trusted usage attributes. If the trusted usage attribute isn’t present (which is the default setting) then all usage types specified in the key usage attribute are allowed.

For example assume a certificates key usage attribute is set to CRYPT_-KEYUSAGE_DIGITALSIGNATURE and CRYPT_KEYUSAGE_-KEYENCIPHERMENT. By setting the trusted usage attribute to CRYPT_-KEYUSAGE_DIGITALSIGNATURE only, you can tell cryptlib that you only trust the certificate to be used for signatures, even though the certificates standard usage would also allow encryption. This means that you can control precisely how a certificate is used at a level beyond that provided by the certificate itself.

Implicitly Trusted Certificates

To handle certificate validation trust issues, cryptlib has a built-in trust manager which records whether a given CA’s or end users certificate is implicitly trusted. When cryptlib gets to a trusted certificate during the certificate validation process (for example as it’s validating the certificates in a certificate chain), it knows that it doesn’t have to go any further in trying to get to an ultimately trusted certificate. If you installed the default cryptlib certificates when you installed cryptlib itself then you’ll have a collection of top-level certificates from the world’s largest CA’s already present and marked as trusted by cryptlib, so that if cryptlib is asked to process a certificate chain ending in one of these trusted CA certificates, the cryptlib trust manager will determine that the top-level certificate is implicitly trusted and use it to verify the lower-level certificates in the chain.

The trust manager provides a convenient mechanism for managing not only CA certificates but also any certificates which you decide you can trust implicitly, for example if you’ve obtained a certificate from a trusted source such as direct communication with the owner or from a trusted referrer, you can mark the certificate as trusted even if it doesn’t have a full chain of CA certificates in tow. This is a natural certificate handling model in many situations (for example trading partners with an existing trust relationship), and avoids the complexity and expense of using an external CA to verify something which both parties know already. When scaled up to thousands of users (and certificates), this can provide a considerable savings

both in terms of managing the certification process and in the cost of obtaining and renewing huge numbers of certificates each year.

Working with Trust Settings

You can get and set a certificates trusted usage using CRYPT_CERTINFO_-TRUSTED_USAGE, which takes as value the key usage(s) for which the certificate is trusted. To mark a certificate as trusted only for encryption, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_TRUSTED_USAGE,
   CRYPT_KEYUSAGE_KEYENCIPHERMENT );
```

This setting will now be applied automatically to the certificates usage permissions, so that even if its CRYPT_CERTINFO_KEYUSAGE attribute allowed signing and encryption, the CRYPT_CERTINFO_TRUSTED_USAGE attribute would restrict this to only allow encryption.

To remove any restrictions and allow all usages specified by CRYPT_CERTINFO_-KEYUSAGE, delete the CRYPT_CERTINFO_TRUSTED_USAGE attribute, which allows the full range of usage types which are present in CRYPT_CERTINFO_-KEYUSAGE:

```
cryptDeleteAttribute( cryptCert, CRYPT_CERTINFO_TRUSTED_USAGE
```

You can get and set a certificates implicitly trusted status using the CRYPT_-CERTINFO_TRUSTED_IMPLICIT attribute, which takes as value a boolean flag which indicates whether the certificate is implicitly trusted or not. To mark a certificate as trusted, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 1 );
```

To check whether a certificate is trusted you would use:

```
int isTrusted;

cryptGetAttribute( certificate, CRYPT_CERTINFO_TRUSTED_IMPLICIT,
   &isTrusted );
```

If the result is set to true (a nonzero value) then the certificate is implicitly trusted by cryptlib. In practice you won't need to bother with this checking, since cryptlib will do it for you when it verifies certificate chains.

The certificate trust settings are part of cryptlib's configuration options, which are explained in more detail in "Miscellaneous Topics" on page 146. Like all configuration options, changes to the trust settings only remain in effect during the current session with cryptlib unless you explicitly force them to be committed to permanent storage by resetting the configuration changed flag. For example if you change the trust settings for various certificates and want the new trust values to be applied when you use cryptlib in the future, you would use code like:

```
/* Mark various certificates as trusted and one as untrusted */
cryptSetAttribute( certificate1, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 1 );
cryptSetAttribute( certificate2, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 1 );
cryptSetAttribute( certificate3, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 1 );
cryptSetAttribute( certificate4, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 0 );

/* Save the new settings to permanent storage */
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_CONFIGCHANGED, FALSE );
```

Marking a certificate as untrusted doesn't mean that it can never be trusted, but merely that it's actual trust status is currently unknown. If the untrusted certificate is signed by a trusted CA certificate (possibly several levels up a certificate chain) then the certificate will be regarded as trusted when cryptlib checks the certificate chain. In practice an untrusted certificate is really a certificate whose precise trust level has yet to be determined rather than a certificate which is explicitly not trusted.

Certificate Errors

The standard cryptlib error codes aren't capable of returning full details on the wide variety of possible error conditions which can be encountered when processing or

working with certificate objects, particularly when it comes to violations of certificate usage constraints. If there is a problem with a certificate, cryptlib will return CRYPT_ERROR_INVALID. In order to obtain more information on problems which are encountered when you process a certificate you can read a certificates CRYPT_ATTRIBUTE_ERRORLOCUS attribute to obtain the locus of the error (the certificate component which caused the problem) and the CRYPT_ATTRIBUTE_-ERRORTYPE attribute to obtain the error type. For example to obtain more information on why an attempt to sign a certificate failed you would use:

```
CRYPT_CERTINFO_TYPE errorLocus;
CRYPT_ERRTYPE_TYPE errorType;

status = cryptSignCert( cryptCert, cryptCAKey );
if( cryptStatusError( status ) )
   {
   cryptGetAttribute( cryptCert, CRYPT_ATTRIBUTE_ERRORLOCUS,
      &errorLocus );
   cryptGetAttribute( cryptCert, CRYPT_ATTRIBUTE_ERRORTYPE, &errorType
      );
   }
```

More information on working with the extended error information is given in “Error Handling” on page 158.

Certificate Extensions

Certificate extensions form by far the most complicated portion of certificates. By default, cryptlib will add appropriate certificate extension attributes to certificates for you if you don't add any, but sometimes you may want to add or change these yourself. cryptlib supports extensions in two ways, through the usual add/get/delete attribute mechanism for extensions it recognises, and through **cryptAddCertExtension**, **cryptGetCertExtension**, and **cryptDeleteCertExtension** for general extensions it doesn't recognise. The general extension handling mechanism allows you to add, query, and delete any kind of extension to a certificate, including ones you define yourself.

Extension Structure

X.509 version 3 introduced a mechanism by which additional information could be added to certificates through the use of certificate extensions. The X.509 standard defined a number of extensions, and over time other standards organisations defined their own additions and amendments to these extensions. In addition private organisations, businesses, and individuals have all defined their own extensions, some of which (for example the extensions from Netscape and Microsoft) have seen a reasonably wide amount of use.

An extension consists of three fields, which are:

Field	Description
Type	The extension type, a unique identifier called an object identifier. This is given as a sequence of numbers which trace a path through an object identifier tree. For example the object identifier for the keyUsage extension is 2 5 29 15. The object identifier for cryptlib is 1 3 6 1 4 1 3029 32.
Critical Flag	A flag which defines whether the extension is important enough that it must be processed by an application. If the critical flag is set and an application doesn't recognise the extension, it will reject the certificate. Extensions should only be marked critical if this is required to prevent the unsafe use of a certificate, although it's recommended that certain basic extensions such as keyUsage and basicConstraints always be marked critical. Unfortunately since some standards (including X.509 itself) allow implementations to selectively ignore non-critical extensions, and support for extensions is often haphazard, it may be necessary to mark an extension as critical in order to ensure that other implementions process it. As usual, you should check to see whether your intended target correctly processes the extensions you plan to use.
Value	The extension data.

For the extensions which cryptlib recognises and processes automatically, the handling of the critical flag is automatic. Since some implementations will reject a certificate which contains a critical extension, you can turn off the encoding of the critical flag by setting CRYPT_OPTION_ENCODE_CRITICAL to false. By default, cryptlib will encode the critical flag where this is recommended by the relevant standards.

For extensions which cryptlib doesn't handle itself, you need to set the critical flag yourself when you add the extension data using **cryptAddCertExtension**.

Working with Extension Attributes

cryptlib can identify attributes in extensions in one of three ways:

1. Through an extension identifier which denotes the entire extension. For example CRYPT_CERTINFO_CERTPOLICIES denotes the certificatePolicies extension.

2. Through an attribute identifier which denotes a particular attribute within an extension. For example CRYPT_CERTINFO_CERTPOLICY denotes the policyIdentifier attribute contained within the certificatePolicies extension.

   Some extensions only contain a single field, in which case the extension identifier is the same as the attribute identifier. For example the CRYPT_CERTINFO_-KEYUSAGE extension contains a single attribute which is also identified by CRYPT_CERTINFO_KEYUSAGE.

3. Through an extension cursor mechanism which allows you to step through a set of extensions extension by extension or field by field. This is explained in more detail below.

You can use the extension identifier to determine whether a particular extension is present with **cryptGetAttribute** (it will return CRYPT_ERROR_NOTFOUND if the extension isn't present), to delete an entire extension with **cryptDeleteAttribute**, and to position the extension cursor at a particular extension.

Attributes within extensions are handled in the usual manner, for example to retrieve the value of the basicConstraints CA field (which determines whether a certificate is a CA certificate) you would use:

```
int isCA;

cryptGetAttribute( certificate, CRYPT_CERTINFO_CA, &isCA );
```

To determine whether the entire basicConstraints extension is present, you would use:

```
int basicConstraintsPresent;

status = cryptGetAttribute( certificate,
   CRYPT_CERTINFO_BASICCONSTAINTS, &basicConstraintsPresent );
if( cryptStatusOK( status ) )
   /* basicConstraints extension is present */
```

You don't have to worry about the structure of individual extensions since cryptlib will handle this for you. For example to make a certificate a CA certificate, all you need to do is:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CA, 1 );
```

and cryptlib will construct the basicConstraints extension for you and set up the CA attribute as required (in fact because the basicConstraints extension is a fundamental X.509v3 extension, cryptlib will always add this by default even if you don't explicitly specify it).

If an attribute has already been assigned a value, an attempt to assign a new value to it will return CRYPT_ERROR_INITED, and you must explicitly delete it before you can assign it a new value.

Extension Cursor Management

Extensions and extension attributes can also be managed through the use of an extension cursor which cryptlib maintains for each certificate object. You can set or move the cursor by extension or by extension attribute, either to an absolute position or relative to the current position.

You move the extension/attribute cursor by setting certificate attributes which tell cryptlib where or how to move the cursor. These attributes, identified by CRYPT_CERTINFO_CURRENT_EXTENSION for the extension and CRYPT_CERTINFO_CURRENT_FIELD for the attribute field within the extension, move the cursor to the particular extension or attribute. The attribute value which

you specify is the extension or extension attribute ID which you want to move the cursor to. For example:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   CRYPT_CERTINFO_CA );
```

would move the extension cursor to the start of the extension containing the given extension field (in this case the start of the basicConstraints extension). In contrast:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_FIELD,
   CRYPT_CERTINFO_CA );
```

would move the cursor to the extension attribute (in this case the CA attribute in the basicConstraints extension).

This type of cursor positioning is absolute cursor positioning, since it moves the cursor to an absolute position in the extensions. You can also use relative cursor positioning which positions the cursor relative to its current position. In this case instead of specifying an extension or extension attribute ID as the value, you specify a movement code which indicates how you want the cursor moved. The movement codes are:

Code	Description
CRYPT_CURSOR_FIRST	Move the cursor to the first extension or the first attribute in the extension.
CRYPT_CURSOR_LAST	Move the cursor to the last extension or the last attribute in the extension.
CRYPT_CURSOR_NEXT	Move the cursor to the next extension or the next attribute in the extension.
CRYPT_CURSOR_PREV	Move the cursor to the previous extension or the previous attribute in the extension.

For example to move the cursor to the start of the first extension, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   CRYPT_CURSOR_FIRST );
```

To advance the cursor to the start of the next extension, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   CRYPT_CURSOR_NEXT );
```

To advance the cursor to the next attribute in the extension, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_FIELD,
   CRYPT_CURSOR_NEXT );
```

Once you have the cursor position, you can work with the extension or extension attribute at the cursor position. For example to delete the entire extension at the current cursor position you would use:

```
cryptDeleteAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION );
```

Deleting the extension at the cursor position will move the cursor to the start of the extension which follows the deleted one, or to the start of the previous extension if the one being deleted was the last one present. This means you can delete every extension simply by repeatedly deleting the extension under the cursor.

To obtain the extension ID or extension attribute ID for the current cursor position, you would use:

```
CRYPT_CERTINFO_TYPE extensionAttributeID;

cryptGetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   &extensionAttributeID );
```

This example obtains the extension ID, to obtain the extension attribute ID you would substitute CRYPT_CERTINFO_CURRENT_FIELD in place of CRYPT_-CERTINFO_CURRENT_EXTENSION.

The extension cursor provides a convenient mechanism for stepping through every extension which is present in a certificate object. For example to iterate through every extension you would use:

```
if( cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      CRYPT_ATTRIBUTE_TYPE extensionID;

      /* Get the ID of the extension under the cursor */
      cryptGetAttribute( certificate,
         CRYPT_CERTINFO_CURRENT_EXTENSION, &extensionID );
      }
   while( cryptSetAttribute( certificate,
      CRYPT_CERTINFO_CURRENT_EXTENSION, CRYPT_CURSOR_NEXT ) ==
      CRYPT_OK );
```

To extend this a stage further and iterate through every attribute in every extension in the certificate object, you would use:

```
if( cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_EXTENSION,
   CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      do
         {
         CRYPT_ATTRIBUTE_TYPE extensionAttributeID;

         /* Get the attribute of the extension attribute under the
            cursor */
         cryptGetAttribute( certificate, CRYPT_CERTINFO_CURRENT_FIELD,
            &extensionFieldID );
         }
      while( cryptSetAttribute( certificate,
         CRYPT_CERTINFO_CURRENT_FIELD, CRYPT_CURSOR_NEXT ) == CRYPT_OK
         );
      }
   while( cryptSetAttribute( certificate,
      CRYPT_CERTINFO_CURRENT_EXTENSION, CRYPT_CURSOR_NEXT ) ==
      CRYPT_OK );
```

Note that iterating attribute by attribute works within the current extension, but won't jump from one extension to the next — to do that, you need to iterate by extension.

Composite Extension Attributes

Some extension attributes are composite attributes which have further sub-components within them. These attribute s are ones which contain complete GeneralNames and/or DN's and are handled in a manner similar to that used for handling the extension cursor: You use **cryptSetAttribute** to identify the attribute which contains the GeneralName or DN you want to work with, specifying a numeric value of CRYPT_UNUSED since this parameter isn't needed, and then get, set, or delete attributes as usual:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_PERMITTEDSUBTREES,
   CRYPT_UNUSED );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_RFC822NAME,
   rfc822Name, rfc822NameLength );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_DNSNAME, dnsName,
   dnsNameLength );
```

This code first identifies the nameConstraints permittedSubtrees GeneralName as the one to be modified, and then sets the GeneralName attributes as usual. If you want to set the DN (which is itself a composite field) within the GeneralName, you need to perform a two-level selection, once to select the GeneralName to modify, and the second time to select the DN within the GeneralName. Once this is done, you can set, query, or delete DN components as usual:

```
/* Select the permittedSubtrees GeneralName, then select the
   DirectoryName DN within the GeneralName */
cryptSetAttribute( certificate, CRYPT_CERTINFO_PERMITTEDSUBTREES,
   CRYPT_UNUSED );
```

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_DIRECTORYNAME,
   CRYPT_UNUSED );

/* Set DN components */
cryptSetAttributeString( certificate, CRYPT_CERTINFO_COUNTRYNAME,
   countryName, countryNameLength );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_LOCALITYNAME,
   localityName, localityNameLength );
```

This code first identifies the nameConstraints permittedSubtrees GeneralName as the one to be modified, then selects the DN within the GeneralName, and finally sets the DN components as usual. cryptlib uses this mechanism to access all DN's and GeneralNames, although this is usually hidden from you — when you modify a certificate, attribute certificate, or certification requests DN, cryptlib automatically uses the subject DN if you don't explicitly specify it, and when you modify the GeneralName cryptlib uses the subject altName if you don't explicitly specify it. In this way you can work with subject names and altNames without having to know about the DN and GeneralName selection mechanism.

Once you've selected a different GeneralName and/or DN, it remains selected until you select a different one, so if you wanted to move back to working with the subject name after performing the operations shown above you'd need to use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_SUBJECTNAME,
   CRYPT_UNUSED );
```

otherwise attempts to add, delete, or query further DN (or GeneralName) attributes will apply to the selected nameConstrainst excludedSubtrees attribute instead of the subject name.

X.509 Extensions

X.509 version 3 and assorted additional standards and revisions specify a large number of extensions, all of which are handled by cryptlib. In addition there are a number of proprietary and vendor-specific extensions which are also handled by cryptlib.

In the following descriptions only the generally useful attributes have been described. The full range of attributes is enormous, requires several hundred pages of standards specifications to describe them all, and will probably never be used in real life. These attributes are marked with "See certificate standards documents" to indicate that you should refer to other documents to obtain information about their usage (this is also a good indication that you shouldn't really be using this attribute).

Alternative Names

The subject and issuer altNames are used to specify all the things which aren't suitable for the main certificate DN's. The issuer altName is identified by CRYPT_CERTINFO_ISSUERALTNAME and the subject altName is identified by CRYPT_CERTINFO_SUBJECTALTNAME. Both consist of a single GeneralName whose use is explained in "Extended Certificate Identification Information" on page 88. This extension is valid in certificates, certification requests, and CRL's, and can contain one of each type of GeneralName component.

Basic Constraints

This is a standard extension identified by CRYPT_CERTINFO_-BASICCONSTRAINTS and is used to specify whether a certificate is a CA certificate or not. If you don't set this extension, cryptlib will set it for you and mark the certificate as a non-CA certificate. This extension is valid in certificates, attribute certificates, and certification requests, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CA Whether the certificate is a CA certificate or not. When used with attribute certificates, the CA is called an authority, so cryptlib will also accept the alternative CRYPT_CERTINFO_AUTHORITY which has the same meaning as CRYPT_CERTINFO_CA. If this attribute isn't set, the certificate is treated as a non-CA certificate.	Boolean
CRYPT_CERTINFO_PATHLENCONSTRAINT See certificate standards documents.	Numeric

For example to mark a certificate as a CA certificate you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CA, 1 );
```

Certificate Policies, Policy Mappings, and Policy Constraints

The certificate policy extensions allow a CA to provide information on the policies governing a certificate, and to control the way in which a certificate can be used. For example it allows you to check that each certificate in a certificate chain was issued under a policy you feel comfortable with (certain security precautions taken, vetting of employees, physical security of the premises, and so on). The certificate policies attribute is identified by CRYPT_CERTINFO_CERTIFICATEPOLICIES and is valid in certificates.

The certificate policies attribute is a complex extension which allows for all sorts of qualifiers and additional modifiers. In general you should only use the policyIdentifier attribute in this extension, since the other attributes are difficult to support in user software and seem to be ignored by most implementations:

Attribute/Description	Type
CRYPT_CERTINFO_CERTPOLICYID The object identifier which identifies the policy under which this certificate was issued.	String
CRYPT_CERTINFO_CERTPOLICY_CPSURI The URL for the certificate practice statement (CPS) for this certificate policy.	String
CRYPT_CERTINFO_CERTPOLICY_ORGANIZATION	String
CRYPT_CERTINFO_CERTPOLICY_NOTICENUMBERS	Numeric
CRYPT_CERTINFO_CERTPOLICY_EXPLICITTEXT These attributes contain further qualifiers, modifiers, and text information which amend the certificate policy information. Refer to certificate standards documents for more information on these attributes.	String

Since various CA's which would like to accept each others certificates may have differing policies, there is an extension which allows a CA to map its policies to those of another CA. The policyMappings extension provides a means of mapping one policy to another (that is, for a CA to indicate that policy A, under which it is issuing a certificate, is equivalent to policy B, which is required by the certificate user). This extension is is identified by CRYPT_CERTINFO_POLICYMAPPINGS and is valid in certificates:

Attribute/Description	Type
CRYPT_CERTINFO_ISSUERDOMAINPOLICY The object identifier for the source (issuer) policy.	String
CRYPT_CERTINFO_SUBJECTDOMAINPOLICY The object identifier for the destination (subject) policy.	String

A CA can also specify acceptable policy constraints for use in certificate chain validation. The policyConstraints extension is identified by CRYPT_CERTINFO_-POLICYCONSTRAINTS and is valid in certificates:

Attribute/Description	Type
CRYPT_CERTINFO_REQUIREEXPLICITPOLICY See certificate standards documents.	Numeric
CRYPT_CERTINFO_INHIBITPOLICYMAPPING See certificate standards documents.	Numeric

CRL Distribution Points and Authority Information Access

These extensions specify how to obtain CRL information and information on the CA which issued a certificate. The cRLDistributionPoint extension is valid in certificates and is identified by CRYPT_CERTINFO_CRLDISTRIBUTIONPOINT:

Attribute/Description	Type
CRYPT_CERTINFO_CRLDIST_FULLNAME The location at which CRL's may be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_CRLDIST_REASONS	Numeric
CRYPT_CERTINFO_CRLDIST_CRLISSUER See certificate standards documents.	GeneralName

Note that the CRYPT_CERTINFO_CRLDIST_REASONS attribute has the same allowable set of values as the cRLReasons reasonCode, but in this case is given as a series of bit flags rather than the reasonCode numeric value (because X.509 says so, that's why). Because of this you must use CRYPT_CRLREASONFLAGS_*name* instead of CRYPT_CRLREASON_*name* when getting and setting these values.

The authorityInfoAccess extension is valid in certificates and CRL's and is identified by CRYPT_CERTINFO_AUTHORITYINFOACCESS:

Attribute/Description	Type
CRYPT_CERTINFO_AUTHORITYINFO_OCSP The location at which certificate status information can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AUTHORITYINFO_CAISSUERS The location at which information on CA's located above the CA which issued this certificate can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName

Directory Attributes

This extension, identified by CRYPT_CERTINFO_-SUBJECTDIRECTORYATTRIBUTES, allows additional X.500 directory attributes to be specified for a certificate. This extension is valid in certificates, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SUBJECTDIR_TYPE The object identifier which identifies the type of the directory attribute.	String
CRYPT_CERTINFO_SUBJECTDIR_VALUES The value of the directory attribute.	String

Key Usage, Extended Key Usage, and Netscape cert-type

These extensions specify the allowed usage for the key contained in this certificate. The keyUsage attribute is a standard extension identifed by CRYPT_CERTINFO_-KEYUSAGE and is used to specify general-purpose key usage such as key encryption, digital signatures, and certificate signing. If you don't set this attribute,

cryptlib will set it for you to a value appropriate for the key type (for example a key for a signature-only algorithm such as DSA will be marked as a signature key).

The extKeyUsage attribute is identified by CRYPT_CERTINFO_EXTKEYUSAGE and is used to specify additional special-case usage such as code signing and SSL server authentication.

The Netscape cert-type field is a vendor-specific attribute identified by CRYPT_CERTINFO_NS_CERTTYPE and was used to specify certain types of web browser-specific certificate usage before the extKeyUsage attribute was fully specified. This attribute has now been superseded by extKeyUsage, but is still found in a number of certificates.

The keyUsage extension has a single numeric attribute with the same identifier as the extension itself (CRYPT_CERTINFO_KEYUSAGE). This extension is valid in certificates and certification requests, and contains a bit flag which can contain any of the following values:

Value	Description
CRYPT_KEYUSAGE_-DATAENCIPHERMENT	The key can be used for data encryption. This implies using public-key encryption for bulk data encryption, which is almost never done.
CRYPT_KEYUSAGE_-DIGITALSIGNATURE	The key can be used for digital signature generation and verification. This is the standard flag to set for digital signature use.
CRYPT_KEYUSAGE_-ENCIPHERONLY CRYPT_KEYUSAGE_-DECIPHERONLY	These flags modify the keyAgreement flag to allow the key to be used for only one part of the key agreement process.
CRYPT_KEYUSAGE_-KEYAGREEMENT	The key can be used for key agreement. This is the standard flag to set for key-agreement algorithms such as Diffie-Hellman.
CRYPT_KEYUSAGE_-KEYCERTSIGN CRYPT_KEYUSAGE_-CRLSIGN	The key can be used to sign certificates and CRL's. Using these flags requires the basicConstraint CA value to be set.
CRYPT_KEYUSAGE_-KEYENCIPHERMENT	The key can be used for key encryption/key transport. This is the standard flag to set for encryption use.
CRYPT_KEYUSAGE_-NONREPUDIATION	The key can be used for nonrepudiation purposes. Note that this use is subtly different to CRYPT_KEYUSAGE_-DIGITALSIGNATURE, so you shouldn't set this unless you really have created the key within a nonrepudiation framework.

For example to mark the key in a certificate as being usable for digital signatures and encryption you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_KEYUSAGE,
   CRYPT_KEYUSAGE_DIGITALSIGNATURE | CRYPT_KEYUSAGE_KEYENCIPHERMENT );
```

The extKeyUsage attribute contains a collection of one or more values which specify a specific type of extended usage which extends beyond the general keyUsage.

This extension is used by applications to determine whether a certificate is meant for a particular purpose such as timestamping or code signing. The extension is valid in certificates and certification requests and can contain any of the following values:

Value	Used in
CRYPT_CERTINFO_EXTKEY_- CODESIGNING	Code-signing certificate.
CRYPT_CERTINFO_EXTKEY_- DIRECTORYSERVICE	Directory service certificate.
CRYPT_CERTINFO_EXTKEY_- EMAILPROTECTION	email encryption/signing certificate.
CRYPT_CERTINFO_EXTKEY_- IPSECENDSYSTEM CRYPT_CERTINFO_EXTKEY_- IPSECTUNNEL CRYPT_CERTINFO_EXTKEY_- IPSECUSER	Various IPSEC certificates.
CRYPT_CERTINFO_EXTKEY_- MS_CERTTRUSTLISTSIGNING CRYPT_CERTINFO_EXTKEY_- MS_TIMESTAMPSIGNING	Microsoft certificate trust list signing and timestamping certificate, used for AuthentiCode signing.
CRYPT_CERTINFO_EXTKEY_- MS_ENCRYPTEDFILESYSTEM	Microsoft encrypted filesystem certificate.
CRYPT_CERTINFO_EXTKEY_- MS_INDIVIDUALCODESIGNING CRYPT_CERTINFO_EXTKEY_- MS_COMMERCIALCODESIGNING	Microsoft individual and commercial code-signing certificate, used for AuthentiCode signing.
CRYPT_CERTINFO_EXTKEY_- MS_SERVERGATEDCRYPTO	Microsoft server-gated crypto (SGC) certificate, used to enable strong encryption on non-US servers.
CRYPT_CERTINFO_EXTKEY_- NS_SERVERGATEDCRYPTO	Netscape server-gated crypto (SGC) certificate, used to enable strong encryption on non-US servers.
CRYPT_CERTINFO_EXTKEY_- SERVERAUTH CRYPT_CERTINFO_EXTKEY_- CLIENTAUTH	SSL server and client authentication certificate.
CRYPT_CERTINFO_EXTKEY_- TIMESTAMPING	Timestamping certificate.
CRYPT_CERTINFO_EXTKEY_- VS_SERVERGATEDCRYPTO_CA	Verisign server-gated crypto CA certificate, used to sign SGC certificates.

For example to mark the key in a certificate as being used for SSL server authentication you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_EXTKEY_SERVERAUTH,
   CRYPT_UNUSED );
```

Like the keyUsage extension, the Netscape cert-type extension has a single numeric attribute with the same identifier as the extension itself (CRYPT_CERTINFO_NS_-CERTTYPE). This extension is valid in certificates and certification requests and contains a bit flag which can contain any of the following values:

Value	Used in
CRYPT_NS_CERTTYPE_- OBJECTSIGNING	Object signing certificate (equivalent to Microsoft's AuthentiCode use).

CRYPT_NS_CERTTYPE_SMIME	S/MIME email encryption/signing certificate.
CRYPT_NS_CERTTYPE_SSLCLIENT CRYPT_NS_CERTTYPE_SSLSERVER	SSL client and server certificate.
CRYPT_NS_CERTTYPE_SSLCA CRYPT_NS_CERTTYPE_SMIMECA CRYPT_NS_CERTTYPE_OBJECTSIGNINGCA	CA certificates corresponding to the above certificate types. Using these flags requires the basicConstraint CA value to be set.

To mark a key in a certificate as being used for SSL server authentication as in the previous example you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_NS_CERTTYPE,
   CRYPT_NS_SSLSERVER );
```

Name Constraints

The nameConstraints extension is used to constrain the certificates' subjectName and subject altName to lie inside or outside a particular DN subtree or substring, with the excludedSubtrees attribute taking precedence over the permittedSubtrees attribute. The principal use for this extension is to allow control of the certificate namespace, so that a CA can restrict the ability of any CA's it certifies to issue certificates outside a very restricted domain (for example corporate headquarters might constrain a divisional CA to only issue certificates for its own business division). This extension is identified by CRYPT_CERTINFO_NAMECONSTRAINTS, and is valid in certificates:

Attribute/Description	Type
CRYPT_CERTINFO_PERMITTEDSUBTREES The subtree within which the subjectName and subject altName of any issued certificates must lie.	GeneralName
CRYPT_CERTINFO_EXCLUDEDSUBTREES The subtree within which the subjectName and subject altName of any issued certificates must not lie.	GeneralName

Due to ambiguities in the encoding rules for strings contained in DN's, it is possible to avoid the excludedSubtrees for DN's by choosing unusual (but perfectly valid) string encodings which don't appear to match the excludedSubtrees. Because of this you should rely on permittedSubtrees rather than excludedSubtrees for DN constraint enforcement.

The nameConstraints are applied to both the certificate subject name and the subject altName. For example if a CA run by Cognitive Cybernetics Incorporated wanted to issue a certificate to a subsidiary CA which was only permitted to issue certificates for Cognitive Cybernetics' marketing division, it would set DN name constraints with:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_PERMITTEDSUBTREES,
   CRYPT_UNUSED );
cryptSetAttribute( certificate, CRYPT_CERTINFO_DIRECTORYNAME,
   CRYPT_UNUSED );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_COUNTRYNAME,
   "US", 2 );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_ORGANIZATIONNAME,
   "Cognitive Cybernetics Incorporated", 32 );
cryptSetAttributeString( certificate,
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, "Marketing", 9 );
```

This means that the subsidiary CA can only issue certificates to employees of the marketing division. Note that since the excludedSubtrees field is a GeneralName, the

DN is selected through a two-level process, first to select the excludedSubtrees GeneralName and then to select the DN within the GeneralName.

GeneralName components which have a flat structure (for example email addresses) can have constraints specified through the '*' wildcard. For example to extend the above constraint to also include email addresses, the issuing CA would set a name constraint with:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_PERMITTEDSUBTREES,
   CRYPT_UNUSED );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_RFC822NAME,
   "*@marketing.cci.com", 19 );
```

This means that the subsidiary CA can only issue certificates with email addresses within the marketing division. Note again the selection of the excludedSubtrees GeneralName followed by the setting of the email address (if the GeneralName is still selected from the earlier code, there's no need to re-select it at this point).

Private Key Usage Period

This extensions specifies the date on which the private key for this certificate expires. This extension is identified by CRYPT_CERTINFO_-PRIVATEKEYUSAGEPERIOD and is valid in certificates. This is useful where a certificate needs to have a much longer lifetime than the private key it corresponds to, for example a long-term signature might have a lifetime of 10-20 years, but the private key used to generate it should never be retained for such a long period. The privateKeyUsagePeriod extension is used to specify a (relatively) short lifetime for the private key while allowing for a very long lifetime for the signatures it generates:

Attribute/Description	Type
CRYPT_CERTINFO_PRIVATEKEY_NOTBEFORE	Binary data
CRYPT_CERTINFO_PRIVATEKEY_NOTAFTER	Binary data

The private key usage period defines the period of time over which the private key for a certificate object is valid. CRYPT_CERTINFO_-PRIVATEKEY_NOTBEFORE specifies the validity start period, and CRYPT_CERTINFO_PRIVATEKEY_NOTAFTER specifies the validity end period, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer).

Subject and Authority Key Identifiers

These extensions are used to provide additional identification information for a certificate, and are usually generated automatically by certificate management code. For this reason the extensions are marked as read-only.

The authorityKeyIdentifier is identified by CRYPT_CERTINFO_-AUTHORITYKEYIDENTIFIER and has the following fields:

Attribute/Description	Type
CRYPT_CERTINFO_AUTHORITY_KEYIDENTIFIER	Binary data

Binary data identifying the public key in the certificate which was used to sign this certificate.

CRYPT_CERTINFO_AUTHORITY_CERTISSUER	GeneralName
CRYPT_CERTINFO_AUTHORITY_-CERTSERIALNUMBER	Binary data

The issuer name and serial number for the certificate which was used to sign this certificate. The serial number is treated as a binary string and not as a numeric value, since it is often 15-20 bytes long.

There are a number of incompatible standards definitions for key identifiers, and many implementations augment these by inventing their own formats on top of the

standard ones. Because of this, cryptlib will not by default try to decode the authorityKeyIdentifier, but will treat it as a single opaque blob with an unknown (or at least irrelevant) internal structure, so that the CRYPT_CERTINFO_-AUTHORITY_*xxx* attributes won't be present. If you want cryptlib to try and decode the authorityKeyIdentifier attributes, you can disable treating the extension data as an opaque blob with the cryptlib configuration option CRYPT_OPTION_CERT_-KEYIDENTIFIERBLOB as explained in "Miscellaneous Topics" on page 146.

The subjectKeyIdentifier is identified by CRYPT_CERTINFO_-SUBJECTKEYIDENTIFIER and contains binary data identifying the public key in the certificate.

CRL Extensions

CRL's have a number of CRL-specific extensions which are described below.

CRL Reasons, CRL Numbers, Delta CRL Indicators

These extensions specify various pieces of information about CRL's. The reasonCode extension is used to indicate why a CRL was issued. The cRLNumber extension provides a serial number for CRL's. The deltaCRLIndicator indicates a delta CRL which contains changes between a base CRL and a delta-CRL (this is used to reduce the overall size of CRL's).

The reasonCode extension is identified by CRYPT_CERTINFO_CRLREASON and is valid in CRL's. The extension has a single numeric field with the same identifier as the extension itself (CRYPT_CERTINFO_CRLREASON) which contains a bit flag which can contain one of the following values:

Value	Description
CRYPT_CRLREASON_-AFFILIATIONCHANGED	The affiliation of the certificate owner has changed, so that the subjectName or subject altName is no longer valid.
CRYPT_CRLREASON_-CACOMPROMISE	The CA which issued the certificate was compromised.
CRYPT_CRLREASON_-CERTIFICATEHOLD	The certificate is to be placed on hold pending further communication from the CA (the further communication may be provided by the holdInstructionCode extension).
CRYPT_CRLREASON_-CESSATIONOFOPERATION	The certificate owner has ceased to operate in the role which requires the use of the certificate.
CRYPT_CRLREASON_-KEYCOMPROMISE	The key for the certificate was compromised.
CRYPT_CRLREASON_-REMOVEFROMCRL	The certificate should be removed from the certificate revocation list.
CRYPT_CRLREASON_-SUPERSEDED	The certificate has been superseded.
CRYPT_CRLREASON_-UNSPECIFIED	No reason for the CRL. You should avoid including a reasonCode at all rather than using this code.

To indicate that a certificate is being revoked because the key it corresponds to has been compromised, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CRLREASON,
   CRYPT_CRLREASON_KEYCOMPROMISE );
```

The cRLNumber extension is identified by CRYPT_CERTINFO_CRLNUMBER and is valid in CRL's. The extension has a single attribute with the same identifier as the extension itself (CRYPT_CERTINFO_CRLNUMBER) which contains a monotonically increasing sequence number for each CRL issued. This allows an application to check that it has received and processed each CRL which was issued.

The deltaCRLIndicator extension is identified by CRYPT_CERTINFO_-DELTACRLINDICATOR and is valid in CRL's. The extension has a single attribute with the same identifier as the extension itself (CRYPT_CERTINFO_-DELTACRLINDICATOR) which contains the cRLNumber of the base CRL from which this delta CRL is being constructed (see certificate standards documents for more information on delta CRL's).

Hold Instruction Code

This extension contains a code which specifies what to do with a certificate which has been placed on hold through a CRL (that is, its revocation reasonCode is CRYPT_CRLREASON_CERTIFICATEHOLD). The extension is identified by CRYPT_CERTINFO_HOLDINSTRUCTIONCODE, is valid in CRL's, and can contain one of the following values:

Value	Description
CRYPT_HOLDINSTRUCTION_-CALLISSUER	Call the certificate issuer for details on the certificate hold.
CRYPT_HOLDINSTRUCTION_NONE	No hold instruction code. You should avoid including a holdInstructionCode at all rather than using this code.
CRYPT_HOLDINSTRUCTION_-REJECT	Reject the transaction which the revoked/held certificate was to be used for.

As the hold code descriptions indicate, this extension was developed mainly for use in the financial industry. To indicate that someone should call the certificate issuer for further information on a certificate hold, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_HOLDINSTRUCTIONCODE,
   CRYPT_HOLDINSTRUCTION_CALLISSUER );
```

Invalidity Date

This extension contains the date on which the private key for a certificate became invalid. The extension is identified by CRYPT_CERTINFO_INVALIDITYDATE and is valid in CRL's:

Attribute/Description	Type
CRYPT_CERTINFO_INVALIDITYDATE The date on which the key identified in a CRL became invalid, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer).	Binary data

Note that a CRL contains both its own date and a date for each revoked certificate, so this extension is only useful if there's some reason for communicating the fact that a key compromise occurred at a time other than the CRL issue time or the certificate revocation time.

Issuing Distribution Point and Certificate Issuer

These extensions specify the CRL distribution point for a CRL and provide various pieces of additional information about the distribution point. The issuingDistributionPoint specifies the distribution point for a CRL, and the

certificateIssuer specifies the issuer for an indirect CRL as indicated by the issuingDistributionPoint extension.

The issuingDistributionPoint extension is identified by CRYPT_CERTINFO_-ISSUINGDISTRIBUTIONPOINT and is valid in CRL's:

Attribute/Description	Type
CRYPT_CERTINFO_ISSUINGDIST_FULLNAME The location at which CRL's may be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_ISSUINGDIST_USERCERTSONLY	Boolean
CRYPT_CERTINFO_ISSUINGDIST_CACERTSONLY	Boolean
CRYPT_CERTINFO_ISSUINGDIST_SOMEREASONSONLY	Numeric
CRYPT_CERTINFO_ISSUINGDIST_INDIRECTCRL See certificate standards documents.	Boolean

Note that the CRYPT_CERTINFO_ISSUINGDIST_SOMEREASONSONLY attribute has the same allowable set of values as the cRLReasons reasonCode, but in this case is given as a series of bit flags rather than the reasonCode numeric value (because X.509 says so, that's why). Because of this you must use CRYPT_-CRLREASONFLAGS_*name* instead of CRYPT_CRLREASON_*name* when getting and setting these values.

The certificateIssuer extension contains the certificate issuer for an indirect CRL. The extension is identified by CRYPT_CERTINFO_CERTIFICATEISSUER and is valid in CRL's:

Attribute/Description	Type
CRYPT_CERTINFO_CERTIFICATEISSUER See certificate standards documents.	GeneralName

Digital Signature Legislation Extensions

Various digital signature laws specify extensions beyond the X.509v3 ones which are described below.

Certificate Generation Date

The German signature law specifies an extension containing the date at which the certificate was generated. This is necessary for postdated certificates to avoid problems if the CA's key is compromised between the time the certificate is issued and the time it takes effect. The extension is identified by CRYPT_CERTINFO_-SIGG_DATEOFCERTGEN and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_DATEOFCERTGEN The date on which the certificate was issued, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer).	Time

Other Restrictions

The German signature law specifies an extension containing any other general free-form restrictions which may be imposed on the certificate. The extension is identified by CRYPT_CERTINFO_SIGG_RESTRICTION and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_RESTRICTION Text containing any further restrictions not already handled via certificate policies or constraints.	String

Reliance Limit

The German signature law specifies an extension containing a reliance limit for the certificate, which specifies the (recommended) monetary reliance limit for the certificate. The extension is identified by CRYPT_CERTINFO_SIGG_-MONETARYLIMIT and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_MONETARY_CURRENCY The three-letter currency code.	String
CRYPT_CERTINFO_SIGG_MONETARY_AMOUNT The amount, specified as an integer in the range 1…200.	Integer
CRYPT_CERTINFO_SIGG_MONETARY_EXPONENT The exponent for the amount, specified as an integer 1…200, so that the actual value is amount $\times 10^{exponent}$.	Integer

Signature Delegation

The German signature law specifies an extension containing details about signature delegation, in which one party may sign on behalf of another (for example someone's secretary signing correspondence on their behalf). The extension is identified by CRYPT_CERTINFO_SIGG_PROCURATION and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_PROCURE_- TYPEOFSUBSTITUTION The type of signature delegation being performed (for example "Signed on behalf of").	String
CRYPT_CERTINFO_SIGG_PROCURE_SIGNINGFOR The identity of the person or organisation the signer is signing on behalf of.	GeneralName

SET Extensions

SET specifies a number of extensions beyond the X.509v3 ones which are described below.

SET Card Required and Merchant Data

These extensions specify various pieces of general information used in the SET electronic payment protocol.

The cardRequired extension contains a flag indicating whether a card is required for a transaction. The extension is identified by CRYPT_CERTINFO_SET_-CERTCARDREQUIRED, and is valid in certificates and certification requests. The extension contains a single boolean attribute with the same identifier as the extension itself (CRYPT_CERTINFO_SET_CARDREQUIRED) which is explained in the SET standards documents.

The merchantData extension contains further information on a merchant. The extension is identified by CRYPT_CERTINFO_SET_MERCHANTDATA and is valid in certificates and certification requests:

Attribute/Description	Type
CRYPT_CERTINFO_SET_MERACQUIRERBIN	String
CRYPT_CERTINFO_SET_MERAUTHFLAG	Boolean
CRYPT_CERTINFO_SET_MERCOUNTRY	Numeric
CRYPT_CERTINFO_SET_MERID	String
Merchants 6-digit BIN, authorisation flag, ISO country code, and merchant ID.	
CRYPT_CERTINFO_SET_MERCHANTCITY	String
CRYPT_CERTINFO_SET_MERCHANTCOUNTRYNAME	String
CRYPT_CERTINFO_SET_MERCHANTLANGUAGE	String
CRYPT_CERTINFO_SET_MERCHANTNAME	String
CRYPT_CERTINFO_SET_MERCHANTPOSTALCODE	String
CRYPT_CERTINFO_SET_MERCHANTSTATEPROVINCE	String
Merchants language, name, city, state or province, postal code, and country name.	

SET Certificate Type, Hashed Root Key, and Tunneling

These extensions specify various pieces of certificate management information used in the SET electronic payment protocol.

The certificateType extension contains the SET certificate type. The extension is identified by CRYPT_CERTINFO_SET_CERTIFICATETYPE and is valid in certificates and certification requests. The extension contains has a single bit flag attribute with the same identifier as the extension itself (CRYPT_CERTINFO_SET_-CERTIFICATETYPE) and can contain any of the following values which are explained in the SET standards documentation:

Value
CRYPT_SET_CERTTYPE_ACQ
CRYPT_SET_CERTTYPE_BCA
CRYPT_SET_CERTTYPE_CARD
CRYPT_SET_CERTTYPE_CCA
CRYPT_SET_CERTTYPE_GCA
CRYPT_SET_CERTTYPE_MCA
CRYPT_SET_CERTTYPE_MER
CRYPT_SET_CERTTYPE_PCA
CRYPT_SET_CERTTYPE_PGWY
CRYPT_SET_CERTTYPE_RCA

The hashedRootKey extension contains a thumbprint (SET-speak for a hash) of a SET root key. The extension is identified by CRYPT_CERTINFO_SET_-HASHEDROOTKEY and is valid in certificates and certification requests. The extension contains a single attribute:

Attribute/Description	Type
CRYPT_CERTINFO_SET_ROOTKEYTHUMBPRINT	Binary data
Binary string containing the root key thumbprint (see the SET standards documents).	

You can the obtain key hash which is required for the thumbprint from another certificate by reading its CRYPT_CERTINFO_SUBJECTKEYIDENTIFIER attribute and then adding it to the certificate you're working with as the CRYPT_-CERTINFO_SET_ROOTKEYTHUMBPRINT attribute. cryptlib will perform the further work required to convert this attribute into the root key thumbprint.

The tunneling extension contains a tunneling indicator and algorithm identifier. The extension is identified by CRYPT_CERTINFO_SET_TUNNELING and is valid in certificates and certification requests.

Attribute/Description	Type
CRYPT_CERTINFO_SET_TUNNELINGFLAG	Boolean
CRYPT_CERTINFO_SET_TUNNELINGALGID	String
See SET standards documents.	

Vendor-specific Extensions

A number of vendors have defined their own extensions which extend or complement the X.509 ones. These are described below.

Netscape Certificate Extensions

Netscape defined a number of extensions which mostly predate the various X.509v3 extensions which now provide the same functionality. The various Netscape certificate extensions are:

Extension/Description	Type
CRYPT_CERTINFO_NS_BASEURL	String
A base URL which, if present, is added to all partial URL's in Netscape extensions to create a full URL.	
CRYPT_CERTINFO_NS_CAPOLICYURL	String
The URL at which the certificate policy under which this certificate was issued can be found.	
CRYPT_CERTINFO_NS_CAREVOCATIONURL	String
The URL at which the revocation status of a CA certificate can be checked.	
CRYPT_CERTINFO_NS_CERTRENEWALURL	String
The URL at which a form allowing renewal of this certificate can be found.	
CRYPT_CERTINFO_NS_COMMENT	String
A comment which should be displayed when the certificate is viewed.	
CRYPT_CERTINFO_NS_REVOCATIONURL	String
The URL at which the revocation status of a server certificate can be checked.	
CRYPT_CERTINFO_NS_SSLSERVERNAME	String
A wildcard string containing a shell expression which matches the hostname of the SSL server using this certificate.	

Note that each of these entries represent a separate extension containing a single text string, they have merely been listed in a single table for readability. You should avoid using these extensions if possible and instead use one of the standard X.509v3 extensions.

Thawte Certificate Extensions

Thawte Consulting have defined an extension which allows the use of certificates with secure extranets. This extension is identified by CRYPT_CERTINFO_-STRONGEXTRANET and is valid in certificates and certification requests:

Attribute/Description	Type
CRYPT_CERTINFO_STRONGEXTRANET_ZONE	Numeric
CRYPT_CERTINFO_STRONGEXTRANET_ID	Binary data
Extranet zone and ID.	

Maintaining Keys and Certificates

Although cryptlib and PGP can work directly with private keys, other formats like X.509 certificates, S/MIME messages, and SSL require complex and convoluted naming and identification schemes for their keys. Because of this, you can't immediately use a newly-generated private key with these formats for anything other than signing a certification request or a self-signed certificate. To use it for any other purpose, you need to obtain an X.509 certificate which identifies the key.

This presents something of a problem, since the certificate isn't generally available when the key is generated and written to a cryptlib key file, smart card, or crypto device. To resolve this, cryptlib provides a means of updating a keyset or device with additional information which amends the basic public/private key data. This additional information can be a key certificate or a full certificate chain from a trusted root CA down to the key certificate. This chapter covers the details of obtaining a certificate or certificate chain and attaching it to a private key.

In addition to creating keys, you may occasionally need to revoke them. Revoked keys are handled via certificate revocation lists (CRL's), which work like 1970's-vintage credit card blacklists by providing users with a list of certificates which shouldn't be honoured any more. Revocations can only be issued by a CA, so to revoke a certificate you either have to be a CA or have the cooperation of a CA. This chapter covers the details of creating and issuing CRL's.

Updating a Private Key with Certificate Information

Once a public/private key pair is saved to a private key keyset, cryptlib allows extra certificate information to be retroactively added to the keyset. For example the process of creating a keyset containing a certificate and private key is:

```
generate public/private key pair;
write key pair to keyset;
submit certification request request to certificate authority;
receive certificate from certification authority;
update keyset to include certificate;
```

If the key pair is being generated in a crypto device such as a smart card or Fortezza card, this process is:

```
generate public/private key pair;
submit certification request request to certificate authority;
receive certificate from certification authority;
update device to include certificate;
```

This example assumes that the certificate is immediately available from a CA, which is not always the case. The full range of possibilities are covered in more detail further on.

The update process involves adding the certificate information to the keyset or device, which updates it with the certificate object (either a certificate or a certificate chain):

```
cryptAddPublicKey( cryptKeyset, cryptCertificate );
```

The certificate object which is being written must match a private key stored in the keyset or device. If it doesn't match a private key, cryptlib will return a CRYPT_-ERROR_PARAM2 error to indicate that the information in the certificate object being added is incorrect.

If the keyset you're updating is a smart card keyset, you should ensure that the card has enough capacity to store the combined certificate object and private key. cryptlib will check whether enough room is available to write both components, and return a CRYPT_ERROR_OVERFLOW error if there isn't enough capacity to store the updated key, in which case you need to copy the private key to a card with more storage capacity and update it there. The private key itself typically requires 500-2K bytes of storage, the certificate object expands this by the size of the encoded

certificate components (typically a few hundred bytes for a certification request, 500-1K bytes for a certificate, and up to 10K for a certificate chain), so the smaller memory cards won't have enough capacity to store a key and certificate, and fairly serious cards are required to store certificate chains.

Changing a Private Key Password

Changing the password on a private key file involves reading the key from a keyset using the old password, deleting the key from the keyset, and writing the in-memory copy back again using the new password:

```
read key from keyset using old password;
delete key from keyset;
re-write key to keyset using new password;
```

All cryptlib key file updates are atomic all-or-nothing operations, which means that if the computer crashes between deleting the old key and writing the new one, the old key will still be present when the machine is rebooted (specifically, all changes are committed when the keyset is closed, which minimises the risk of losing data due to a system crashe or power outage in the middle of a long sequence of update operations).

To update a private key with a new password, you'd use code like:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_CONTEXT cryptKey;

/* Read the key from the keyset using the old password */
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE, keysetName,
   CRYPT_KEYOPT_NONE );
cryptGetPrivateKey( cryptKeyset, &cryptKey, CRYPT_KEYID_NAME, label,
   oldPassword );

/* Delete the current copy of the key from the keyset */
cryptDeleteKey( cryptKeyset, label );

/* Write the key back to the keyset using the new password */
cryptAddPrivateKey( cryptKeyset, cryptKey, newPassword );
cryptKeysetClose( cryptKeyset );
```

The Certification Process

Creating a private key and associated certificate involves two separate processes: generating the public/private key pair, and obtaining a certificate for the public key which is then attached to the public/private key. The key generation process is:

```
generate public/private key pair;
write key pair to keyset;
```

For a crypto device such as a smart card or Fortezza card, the key is generated inside the device, so this step simplifies to:

```
generate public/private key pair;
```

The generated key is already stored inside the device, so there's no need to explicitly write it to any storage media.

The certification process varies somewhat, a typical case has already been presented earlier:

```
create certification request;
submit certification request to certificate authority;
receive certificate from certification authority;
update keyset or device to include certificate;
```

Now that the general outline has been covered, we can look at the individual steps in more detail. Generating a public/private key pair and saving it to a keyset is relatively simple:

```
CRYPT_CONTEXT cryptContext;
CRYPT_KEYSET cryptKeyset;
```

```
/* Create an RSA public-key context, set a label for it, and generate
   a key into it */
cryptCreateContext( &cryptContext, CRYPT_ALGO_RSA );
cryptSetAttributeString( &cryptContext, CRYPT_CTXINFO_LABEL, "Private
   key", 11 );
cryptGenerateKey( cryptContext );

/* Save the generated public/private key pair to a keyset */
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE, fileName,
   CRYPT_KEYOPT_CREATE );
cryptAddPrivateKey( cryptKeyset, cryptContext, password );
cryptKeysetClose( cryptKeyset );

/* Clean up */
cryptDestroyContext( cryptContext );
```

The process for a device is identical except that the keyset write is omitted, since the key is already held inside the device.

In practice you'd probably use **cryptGenerateKeyAsync** so the user can perform other actions while the key is being generated. Typically you'd run the key generation (via **cryptGenerateKeyAsync**) and the certification request creation in parallel so that by the time the certificate details have been filled in the key is ready for use.

At the same time as you create and save the public/private key pair, you would create a certification request:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create a certification request */
cryptCreateCert( cryptCertRequest, CRYPT_CERTTYPE_CERTREQUEST );

/* Fill in the certification request details */
/* ... */
```

The next step depends on the speed with which the certification request can be turned into a certificate. If the CA's turnaround time is very quick (for example if it's operated in-house) then you can submit the request directly to the CA to convert it into a certificate. In this case you can keep the keyset which you wrote the key to open and update it immediately with the certificate:

```
CRYPT_CERTIFICATE cryptCert;

/* Send the certification request to the CA and obtain the returned
   certificate */
/* ... */

/* Import the certificate and check its validity */
cryptImportCert( certificate, &cryptCert );
cryptCheckCert( cryptCert, caCertificate );

/* Update the still-open keyset with the certificate */
cryptAddPublicKey( cryptKeyset, cryptCert );

/* Clean up */
cryptKeysetClose( cryptKeyset );
cryptDestroyCert( cryptCert );
```

Since a device acts just like a keyset for certificate updates, you can write a certificate to a device in the same manner.

If, as will usually be the case, the certification turnaround time is somewhat longer, you will need to wait awhile to receive the certificate back from the CA. Once the certificate arrives from the CA, you update the keyset as before:

```
CRYPT_CERTIFICATE cryptCert;
CRYPT_KEYSET cryptKeyset;

/* Obtain the returned certificate from the CA */
/* ... */

/* Import the certificate and check its validity */
cryptImportCert( certificate, &cryptCert );
```

```
cryptCheckCert( cryptCert, caCertificate );

/* Open the keyset for update and add the certificate */
cryptKeysetOpen( &cryptKeyset, CRYPT_KEYSET_FILE, fileName,
   CRYPT_KEYOPT_NONE );
cryptAddPublicKey( cryptKeyset, cryptCert );
cryptKeysetClose( cryptKeyset );

/* Clean up */
cryptDestroyCert( cryptCert );
```

Again, device updates work in the same manner.

A final case involves self-signed certificates. In this case you can immediately update the (still-open) keyset with the self-signed certificate without any need to go through the usual certification process:

```
CRYPT_CERTIFICATE cryptCert;

/* Create a self-signed certificate */
cryptCreateCert( cryptCert, CRYPT_CERTTYPE_CERTIFICATE );
/* ... */

/* Sign the certificate with the private key and update the still-open
   keyset with it*/
cryptSetAttribute( cryptCert, CRYPT_CERTINFO_SELFSIGNED, 1 );
cryptSignCert( cryptCert, cryptContext );
cryptAddPublicKey( cryptKeyset, cryptCert );

/* Clean up */
cryptKeysetClose( cryptKeyset );
cryptDestroyCert( cryptCert );
```

Certificate Chains

Because of the lack of availability of a general-purpose certificate directory, many security protocols (most notable S/MIME and SSL) transmit not individual certificates but entire certificate chains which contain a complete certificate path from the end users certificate up to some widely-trusted CA certificate (referred to as a root CA certificate if it's a self-signed CA certificates) whose trust will be handled for you by cryptlib's trust manager. cryptlib supports the creation, import, export, and checking of certificate chains as CRYPT_CERTTYPE_CERTCHAIN objects, with individual certificates in the chain being accessed as if they were standard certificates contained in a CRYPT_CERTTYPE_CERTIFICATE object.

Working with Certificate Chains

Individual certificates in a chain are addressed through a certificate cursor which functions in the same way as the extension cursor discussed in "Extension Cursor Management" on page 98. Although a certificate chain object appears as a single object, it consists internally of a collection of certificates of which the first in the chain is the end-users certificate and the last is a root CA certificate or at least an implicitly trusted CA certificate.

You can move the certificate cursor using the CRYPT_CERTINFO_CURRENT_-CERTIFICATE certificate attribute and the following cursor movement codes:

Code	Description
CRYPT_CURSOR_FIRST	Move the cursor to the first certificate in the chain.
CRYPT_CURSOR_LAST	Move the cursor to the last certificate in the chain.
CRYPT_CURSOR_NEXT	Move the cursor to the next certificate in the chain.
CRYPT_CURSOR_PREV	Move the cursor to the previous certificate in

Code	Description
	the chain.

For example to move the cursor to the first (end-user) certificate in the chain, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_CERTIFICATE,
   CRYPT_CURSOR_FIRST );
```

To advance the cursor to the next certificate, you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CURRENT_CERTIFICATE,
   CRYPT_CURSOR_NEXT );
```

The certificate cursor and the extension/extension attribute cursor are two completely independent objects, so moving the certificate cursor from one certificate to another doesn't affect the extension cursor setting for each certificate. If you select a particular extension in a certificate, then move to a different certificate and select an extension in that, and then move back to the first certificate, the original extension will still be selected.

Once you've selected a particular certificate in the chain, you can work with it as if it were the only certificate contained in the certificate object. The initially selected certificate is the end-users certificate at the start of the chain. For example to read the commonName from the subject name for the end-users certificate and for the next certificate in the chain you would use:

```
char commonName[ CRYPT_MAX_TEXTSIZE + 1 ];
int commonNameLength;

/* Retrieve the commonName from the end-users certificate */
cryptGetAttributeString( cryptCertChain, CRYPT_CERTINFO_COMMONNAME,
   commonName, &commonNameLength );
commonName[ commonNameLength ] = '\0';

/* Move to the next certificate in the chain */
cryptSetAttribute( cryptCertChain, CRYPT_CERTINFO_CURRENT_CERTIFICATE,
   CRYPT_CURSOR_NEXT );

/* Retrieve the commonName from the next certificate */
cryptGetAttributeString( cryptCertChain, CRYPT_CERTINFO_COMMONNAME,
   commonName, &commonNameLength );
commonName[ commonNameLength ] = '\0';
```

Apart from this, certificate chains work just like certificates — you can import them, export them, verify the signatures on them (which verifies the entire chain of certificates until a trusted certificate is reached), and write them to and read them from private key keysets in exactly the same manner as an individual certificate. You can also write them to public key keysets, although what is written is the currently selected certificate rather than the entire chain, since the keyset stores individual certificates and not composite objects like certificate chains.

Signing Certificate Chains

When you sign a single subject certificate using **cryptSignCert**, a small amount of information is copied from the issuer certificate to the subject certificate as part of the signing process, and the result is a single, signed subject certificate. In contrast signing a single subject certificate contained in a certificate chain object results in the signing certificates (either a single issuer certificate or an entire chain of certificates) being copied over to the certificate chain object so that the signed certificate ends up as part of a complete chain. The exact details are as follows:

Object to sign	Signing object	Result
Certificate	Certificate	Certificate
Certificate	Certificate chain	Certificate
Certificate chain	Certificate	Certificate chain, length = 2

Object to sign	Signing object	Result
Certificate chain	Certificate chain	Certificate chain, length = length of signing chain + 1

For example the following code produces a single signed certificate:

```
CRYPT_CERTIFICATE cryptCert;

/* Build a certificate from a cert request */
cryptCreateCert( &cryptCert, CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCert, CRYPT_CERTINFO_CERTREQUEST,
   cryptCertRequest );

/* Read a private key with cert chain from a private key keyset */
/* ... */

/* Sign the certificate */
cryptSignCert( cryptCert, caPrivateKey );
```

In contrast the following code produces a complete certificate chain, since the object being created is a CRYPT_CERTTYPE_CERTCHAIN (which can hold a complete chain) rather than a CRYPT_CERTTYPE_CERTIFICATE (which only holds a single certificate):

```
CRYPT_CERTIFICATE cryptCertChain;

/* Build a certificate from a cert request */
cryptCreateCert( &cryptCertChain, CRYPT_CERTTYPE_CERTCHAIN );
cryptSetAttribute( cryptCertChain, CRYPT_CERTINFO_CERTREQUEST,
   cryptCertRequest );

/* Read a private key with cert chain from a private key keyset */
/* ... */

/* Sign the certificate chain */
cryptSignCert( cryptCertChain, caPrivateKey );
```

By specifying the object type to be signed, you can choose between creating a single signed certificate or a complete certificate chain.

Checking Certificate Chains

When verifying a certificate chain with **cryptCheckCert**, you don't have to supply an issuer certificate since the chain should contain all the issuer certificates up to one which is trusted by cryptlib:

```
CRYPT_CERTIFICATE cryptCertChain;

/* Verify an entire cert chain */
cryptCheckCert( cryptCertChain, CRYPT_UNUSED );
```

As with self-signed certificates, you can also pass in the cert chain as the signing certificate instead of using CRYPT_UNUSED, this has the same effect since the cert chain is both the signed and signing object.

If a certificate in the chain is invalid or the chain doesn't contain a trusted certificate at some point in the chain, cryptlib will return an appropriate error code and leave the invalid certificate as the currently selected one, allowing you to obtain information about the nature of the problem by reading the error information attributes as explained in "Error Handling" on page 158.

If the error encountered is the fact that the chain doesn't contain a trusted certificate somewhere along the line, cryptlib will either mark the top-level certificate as having a missing CRYPT_CERTINFO_TRUSTEDUSAGE attribute if it's a CA root certificate (that is, there's a root certificate present but it isn't trusted) or mark the chain a whole as having a missing certificate if there's no CA root certificate present and no trusted certificate present either. Certificate trust management is explained in more detail in "Certificate Management" on page 76.

Certificate chain validation is an extremely complex process which takes into account an enormous amount of validation information which may be spread across an entire

certificate chain. For example in a chain of 10 certificates, the 3rd certificate from the root may place a constraint which doesn't take effect until the 7th certificate from the root is reached. Because of this, a reported validation problem isn't necessary related to a given certificate and its immediate issuing certificate, but may have been caused by a different certificate a number of steps further along the chain.

Some certificate chains may not contain or be signed by a trusted CA certificate, but may end in a root CA certificate with an unknown trust level. Since the cryptlib trust manager can't provide any information about this certificate, it won't be possible to verify the chain. If you want to explicitly trust the root CA certificate, you can use the cryptlib configuration option CRYPT_OPTION_CERT_TRUSTCHAINROOT to force cryptlib to explicitly trust the CA root certificate, but this isn't recommended since it bypasses the normal trust management mechanisms.

Exporting Certificate Chains

As is the case when signing certificates and certificate chains, cryptlib gives you a high degree of control over what part of the chain you want to export. By specifying an export format of CRYPT_CERTFORMAT_CERTIFICATE or CRYPT_-CERTFORMAT_CERTCHAIN, you can control whether a single certificate or an entire chain is exported. The exact details are as follows:

Object type	Export format	Result
Certificate	Certificate	Certificate
Certificate	Certificate chain	Certificate chain, length = 1
Certificate chain	Certificate	Currently selected certificate in the chain
Certificate chain	Certificate chain	Certificate chain

For example the following code exports the currently selected certificate in the chain as a single certificate:

```
CRYPT_CERTIFICATE cryptCertChain;
void *certificate;
int certificateLength;

/* Allocate memory for the encoded certificate */
certificate = malloc( ... );

/* Export the currently selected certificate from the certificate
   chain */
cryptExportCert( certificate, &certificateLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertChain );
```

In contrast the following code exports the entire certificate chain:

```
CRYPT_CERTIFICATE cryptCertChain;
void *certChain;
int certChainLength;

/* Allocate memory for the encoded certificate chain */
certChain = malloc( ... );

/* Export the entire certificate chain */
cryptExportCert( certChain, &certChainLength,
   CRYPT_CERTFORMAT_CERTCHAIN, cryptCertChain );
```

Certificate Revocation Lists

Once a certificate has been issued, you may need to revoke it before its expiry date if the private key it corresponds to is lost or stolen, or if the details given in the certificate (for example your job role or company affiliation) change. Certificate revocation is done through a certificate revocation list (CRL) which contains references to one or more certificates which have been revoked by a CA. cryptlib supports the creation, import, export, and checking of CRL's as CRYPT_-CERTTYPE_CRL objects, with individual revocation entries accessed as if they were

standard certificate components. Note that these entries are merely references to revoked certificates and not the certificates themselves, so all they contain is a certificate reference, the date of revocation, and possibly various optional extras such as the reason for the revocation.

Working with CRL's

Individual revocation entries in a CRL are addressed through a certificate cursor which functions in the same way as the certificate chain cursor discussed in "Working with Certificate Chains" on page 117. Although a CRL appears as a single object, it consists internally of a collection of certificate revocation entries which you can move through using the following cursor movement codes:

Code	Description
CRYPT_CURSOR_FIRST	Move the cursor to the first entryin the CRL.
CRYPT_CURSOR_LAST	Move the cursor to the last entry in the CRL.
CRYPT_CURSOR_NEXT	Move the cursor to the next entry in the CRL.
CRYPT_CURSOR_PREV	Move the cursor to the previous entry in the CRL.

For example to move the cursor to the first entry in the CRL, you would use:

```
cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CURRENT_CERTIFICATE,
   CRYPT_CURSOR_FIRST );
```

To advance the cursor to the next entry, you would use:

```
cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CURRENT_CERTIFICATE,
   CRYPT_CURSOR_NEXT );
```

Since each revocation entry can have its own attributes, moving the entry cursor from one entry to another can change the attributes which are visible. This means that if you're working with a particular entry, the attributes for that entry will be visible, but attributes for other entries won't be. To complicate this further, CRL's can also contain global attributes which apply to, and are visible for, all entries in the CRL. cryptlib will automatically handle these for you, allowing access to all attributes (both per-entry and global) which apply to the currently selected revocation entry.

Creating CRL's

To create a CRL, you first create the CRL certificate object as usual and then push one or more certificates to be revoked into it.

```
CRYPT_CERTIFICATE cryptCRL;

/* Create the (empty) CRL */
cryptCreateCert( &cryptCRL, CRYPT_CERTTYPE_CRL );

/* Add the certificates to be revoked */
cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CERTIFICATE, revokedCert1
   );
cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CERTIFICATE, revokedCert2
   );
/* ... */
cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CERTIFICATE, revokedCertN
   );

/* Sign the CRL */
cryptSignCertificate( cryptCRL, caPrivateKey );
```

As has already been mentioned, you must be a CA in order to issue a CRL, and you can only revoke certificates which you have issued using the certificate used to sign the CRL (you can't, for example, revoke a certificate issued by another CA, or revoke a certificate issued with one CA certificate using a different CA certificate). If you try to add certificates issued by multiple CA's to a CRL, or try to sign a CRL with a CA certificate which differs from the one which signed the certificates in the CRL,

cryptlib will return a CRYPT_ERROR_INVALID error to indicate that the certificate you are trying to add to the CRL or sign the CRL with is from the wrong CA. To reiterate: Every certificate in a given CRL must have been issued using the CA certificate which is used to sign the CRL. If your CA uses multiple certificates (for example a Class 1 certificate, a Class 2 certificate, and a Class 3 certificate) then it must issue one CRL for each certificate class. cryptlib will perform the necessary checking for you to ensure you don't issue an invalid CRL.

Advanced CRL Creation

The code shown above creates a relatively straightforward, simple CRL with no extra information included with the revocation. You can also include extra attributes such as the time of the revocation (which may differ from the time the CRL was issued, if you don't specify a time cryptlib will use the CRL issuing time), the reason for the revocation, and the various other CRL-specific information as described in "Certificate Extensions" on page 97.

If you set a revocation time with no revoked certificates present in the CRL, cryptlib will use this time for any certificates you add to the CRL for which you don't explicitly set the revocation time (so you can use this to set a default revocation time for any certificates you add). If you set a revocation time and there are revoked certificates present in the CRL, cryptlib will set the time for the currently selected certificate, which will be either the last one added or the one selected with the certificate cursor commands.

For example to revoke a list of certificates, setting the revocation date for each one individually, you would use:

```
CRYPT_CERTIFICATE cryptCRL;

while( moreCerts )
   {
   CRYPT_CERTIFICATE revokedCert;
   time_t revocationTime;

   /* Get the certificate to revoke and its revocation time */
   revokedCert = ...;
   revocationTime = ...;

   /* Add them to the CRL */
   cryptSetAttribute( cryptCRL, CRYPT_CERTINFO_CERTIFICATE,
      revokedCert );
   cryptSetAttributeString( cryptCRL, CRYPT_CERTINFO_REVOCATIONDATE,
      &revocationTime, sizeof( time_t ) );

   /* Clean up */
   cryptDestroyCert( revokedCert );
   }
```

You can also add additional attributes such as the reason for the revocation to each revoked certificate, a number of standards recommend that a reason is given for each revocation. The revocation codes are specified in "Certificate Extensions" on page 97.

CRL's can be signed, verified, imported, and exported just like other certificate objects.

Checking Certificates against CRL's

Verifying a certificate against a CRL with **cryptCheckCert** works just like a standard certificate check, with the second parameter being the CRL which the certificate is being checked against:

```
CRYPT_CERTIFICATE cryptCRL;

/* Check the certificate against the CRL */
cryptCheckCert( cryptCert, cryptCRL );
```

If the certificate has been revoked, cryptlib will return CRYPT_ERROR_INVALID and leave the certificates revocation entry in the CRL as the selected one, allowing you to obtain further information on the revocation (for example the revocation date or reason):

```
time_t revocationTime;
int revocationReason;

status = cryptCheckCert( cryptCert, cryptCRL );
if( status == CRYPT_INVALID )
   {
   int revocationTimeLength;

   /* The certificate has been revoked, get the revocation time and
      reason */
   cryptGetAttributeString( cryptCRL, CRYPT_CERTINFO_REVOCATIONDATE,
      &revocationTime, &revocationTimeLength );
   cryptGetAttribute( cryptCRL, CRYPT_CERTINFO_CRLREASON,
      &revocationReason );
   }
```

Note that the revocation reason is an optional CRL component, so this may not be present in the CRL (it rarely is in current CRL's). If the revocation reason isn't present, cryptlib will return CRYPT_ERROR_NOTFOUND.

Automated CRL Checking

As you can see from the description of the revocation checking process above, it quickly becomes unmanageable as the number of CRL's and the size of each CRL increases, since what should be a simple certificate validation check now involves checking the certificate against any number of CRL's (CRL's are generally regarded as a rather unsatisfactory solution to the problem of certificate revocation, but we're stuck with them for the forseeable future).

In order to ease this complex and long-winded checking process, cryptlib provides the ability to automatically check a certificate against CRL's stored in a cryptlib database keyset. To do this you first need to write the CRL or CRL's to the keyset as if they were normal certificates, as explained in "Key Databases" on page 40. cryptlib will take each complete CRL and record all of the individual revocations contained in it for later use.

Once you have a keyset containing revocation information, you can use it to check the validity of a certificate using **cryptCheckCert**, giving the keyset as the second parameter:

```
CRYPT_KEYSET cryptKeyset;

/* Check the certificate using the keyset */
cryptCheckCert( cryptCert, cryptKeyset );
```

As with the check against a CRL, cryptlib will return CRYPT_ERROR_INVALID if the certificate has been revoked.

This form of automated checking considerably simplifies the otherwise arbitrarily complex CRL checking process since cryptlib can handle the check with a simple keyset query rather than having to locate and search any number of CRL's.

Further Certificate Objects

Alongside standard certificates, CRL's, and certificate chains, cryptlib provides the ability to work with other certificate-like objects which are used during the certification process and in standards such as S/MIME which work with certificates. These certificate-like objects aren't signed like certificates but share most of the properties of certificates and are manipulated in the same way as certificates.

The available certificate-like object types are:

Certificate Type	Description
CRYPT_CERTTYPE_CMS_- ATTRIBUTES	CMS/SMIME/PKCS #7 signature attributes

These objects are created and destroyed in the standard manner usign **cryptCreateCert** and **cryptDestroyCert**.

Certificate-like Object Structure

Like the standard certificate types, certificate-like objects have their own internal structures which are encoded and decoded for you by cryptlib. Although cryptlib provides the ability to control each certificate-like object in great detail if you require this, in practice you should leave the handling of the details to cryptlib. If you don't fill in the non-mandatory fields, cryptlib will fill them in for you before it uses the object.

CMS Attributes

CMS/SMIME/PKCS #7 signing attributes have the following structure:

Field	Description
Attributes	Signing attributes which allow extra information to be included alongside signatures. These attributes work like certificate extensions and are described in more detail further on.

CMS Attributes

The S/MIME standards specify various attributes which can be included with signatures. In addition there are a variety of proprietary and vendor-specific attributes which are also handled by cryptlib. In the following description only the generally useful fields have been described, the full range of fields is enormous and requires a number of standards specifications (often followed by cries for help on mailing lists) to interpret them. These fields are marked with "See S/MIME standards documents" to indicate that you should refer to other documents to obtain information about their use (this is also a good indication that you shouldn't really be using this attribute).

Content Type

This is a standard CMS attribute identified by CRYPT_CERTINFO_CMS_-CONTENTTYPE and is used to specify the type of data which is being signed. This is used because some signed information could be interpreted in different ways depending on the data type it's supposed to represent (for example something viewed as encrypted data could be interpreted quite differently if viewed as plain data). If you don't set this attribute, cryptlib will set it for you and mark the signed content as plain data.

The content type CMS attribute can contain one of the following CRYPT_-CONTENT_TYPE values:

Value	Description
CRYPT_CONTENT_CMS_-DATA	Plain data.
CRYPT_CONTENT_CMS_-SIGNEDDATA	Signed data.
CRYPT_CONTENT_CMS_-ENVELOPEDDATA	Data encrypted using a password or public-key or conventional encryption.
CRYPT_CONTENT_CMS_-SIGNEDANDENVELOPED-DATA	Data which is both signed and enveloped (this is an obsolete composite content type which shouldn't be used).
CRYPT_CONTENT_CMS_-DIGESTEDDATA	Hashed data.
CRYPT_CONTENT_CMS_-ENCRYPTEDDATA	Data encrypted directly with a session key.
CRYPT_CONTENT_CMS_-SPCINDIRECTDATA-CONTEXT	Indirectly signed data used in Authenticode signatures.

The distinction between the different types arises from the way they are specified in the standards documents, as a rule of thumb if the data being signed is encrypted then use CRYPT_CERTINFO_CMS_ENVELOPEDDATA (rather than CRYPT_CERTINFO_CMS_ENCRYPTEDDATA, which is slightly different), if it's signed then use CRYPT_CERTINFO_CMS_SIGNEDDATA, and if it's anything else then use CRYPT_CERTINFO_CMS_DATA. For example to identify the data you're signing as encrypted data, you would use:

```
cryptSetAttribute( cmsAttributes, CRYPT_CERTINFO_CMS_CONTENTTYPE,
   CRYPT_CONTENT_CMS_ENVELOPEDDATA );
```

If you're generating the signature via the cryptlib enveloping code then cryptlib will set the correct type for you so there's no need to set it yourself.

Countersignature

This CMS attribute contains a second signature which countersigns one of the signatures on the data (that is, it signs the other signature rather than the data). The attribute is identified by CRYPT_CERTINFO_CMS_COUNTERSIGNATURE:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_COUNTERSIGNATURE See S/MIME standards documents.	Binary data

Message Digest

This read-only CMS attribute is used as part of the signing process and is generated automatically by cryptlib. The attribute is identified by CRYPT_CERTINFO_-CMS_MESSAGEDIGEST:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_MESSAGEDIGEST The hash of the content being signed.	Binary data

Signing Time

This is a standard CMS attribute identified by CRYPT_CERTINFO_CMS_-SIGNINGTIME and is used to specify the time at which the signature was generated. If you don't set this attribute, cryptlib will set it for you.

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGNINGTIME The time at which the signature was generated, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer).	Binary data

Extended CMS Attributes

The attributes given above are the standard CMS attributes. Extending beyond this are further attributes which are defined in additional standards documents and which apply mostly to S/MIME messages, as well as vendor-specific and proprietary attributes. Before you use these additional attributes you should ensure that any software you plan to interoperate with can process them, since currently almost nothing will recognise them (for example it's not a good idea to put a security label on your data and expect other software to handle it correctly).

AuthentiCode Attributes

AuthentiCode code-signing uses a number of of attributes which apply to signed executable content. These attributes are listed below.

The agency information CMS attribute, identified by CRYPT_CERTINFO_CMS_-SPCAGENCYINFO, is used to provide extra information about the signer of the data and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SPCAGENCYURL The URL of a web page containing more information about the signer.	String

The statement type CMS attribute, identified by CRYPT_CERTINFO_CMS_-SPCSTATEMENTTYPE, is used to identify whether the content was signed by an individual or a commercial organisation, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SPCSTMT_INDIVIDUAL-CODESIGNING The data was signed by an individual.	String
CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIAL-CODESIGNING The data was signed by a commercial organisation.	

For example to indicate that the data was signed by an individual, you would use:

```
cryptSetAttribute( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIALCODESIGNING, CRYPT_UNUSED );
```

The opus information CMS attribute is an obsolete attribute which is included in signatures for backwards-compatibility reasons. It is identified by CRYPT_-CERTINFO_CMS_SPCOPUSINFO and appears as a single opaque object which has a numeric value of 'true' (any nonzero value) to indicate that it is present in the

collection of attributes. When you want to add this attribute, you should add it as a numeric component with a value of 'true'.

For example to create an AuthentiCode signature as a commercial organisation you would use:

```
CRYPT_CERTIFICATE cmsAttributes;

/* Create the CMS attribute object and add the AuthentiCode attributes
   */
cryptCreateCert( &cmsAttributes, CRYPT_CERTTYPE_CMS_ATTRIBUTES );
cryptSetAttributeString( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCAGENCYURL,
   "http://homepage.organisation.com", 32 );
cryptSetAttribute( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIALCODESIGNING, CRYPT_UNUSED );
cryptSetAttribute( cmsAttributes, CRYPT_CERTINFO_CMS_SPCOPUSINFO, 1 );

/* Add the content-type required for AuthentiCode data */
cryptSetAttribute( cmsAttributes, CRYPT_CERTINFO_CMS_CONTENTTYPE,
   CRYPT_CONTENT_SPCINDIRECTDATACONTEXT );

/* Sign the data with the attributes included */
cryptCreateSignatureEx( ... );

cryptDestroyCert( cmsAttributes );
```

The other attributes used when signing are standard attributes which will be added automatically for you by cryptlib.

Content Hints

This CMS attribute can be supplied in the outer layer of a multi-layer message to provide information on what the innermost layer of the message contains. The attribute is identified by CRYPT_CERTINFO_CMS_CONTENTHINTS and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_CONTENTHINT_-DESCRIPTION A human-readable description which may be useful when processing the content.	String
CRYPT_CERTINFO_CMS_CONTENTHINT_TYPE The type of the innermost content, specified as a CRYPT_CONTENT_-*content-type* value.	Numeric

Mail List Expansion History

This CMS attribute contains information on what happened to a message when it was processed by mailing list software. It is identified by CRYPT_CERTINFO_CMS_-MLEXPANSIONHISTORY and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_MLEXP_ENTITYIDENTIFIER See S/MIME standards documents.	Binary data
CRYPT_CERTINFO_CMS_MLEXP_TIME The time at which the mailing-list software processed the message, expressed in local time and using the standard ANSI/ISO C seconds since 1970 format. This is a binary data field, with the data being the timestamp value (in C and C++ this is a `time_t`, usually a signed long integer).	Binary data
CRYPT_CERTINFO_CMS_MLEXP_NONE CRYPT_CERTINFO_CMS_MLEXP_INSTEADOF CRYPT_CERTINFO_CMS_MLEXP_INADDITIONTO This field can have one of the three values specified above, and is used to indicate a receipt policy which overrides the one given in the original message. See the S/MIME standards documents for more information.	— General-Name

Receipt Request

This CMS attribute is used to request a receipt from the recipient of a message and is identified by CRYPT_CERTINFO_CMS_RECEIPT_REQUEST. As with the security label attribute, you shouldn't rely on the recipient of a message being able to do anything with this information, which consists of the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_RECEIPT_-CONTENTIDENTIFIER A magic value used to identify a message, see the S/MIME standards documents for more information.	Binary data
CRYPT_CERTINFO_CMS_RECEIPT_FROM CRYPT_CERTINFO_CMS_RECEIPT_TO An indication of who receipts should come from and who they should go to, see the S/MIME standards documents for more information.	Numeric General-Name

Security Label, Equivalent Label

These CMS attributes specify security information for the content contained in the message, allowing recipients to decide how they should process it, for example an implementation could refuse to display a message to a recipient who isn't cleared to see it (this assumes that the recipient software is implemented at least in part using tamper-resistant hardware, since a pure software implementation could be set up to ignore the security label). These attributes originate (in theory) in X.400 and (in practice) in DMS, the US DoD secure email system, and virtually no implementations outside this area understand them so you shouldn't rely on them to ensure proper processing of a message.

The basic security label on a message is identified by CRYPT_CERTINFO_CMS_-SECURITYLABEL. Since different organisations have different ways of handling security policies, their labelling schemes may differ, so the equivalent labels CMS attribute, identified by CRYPT_CERTINFO_CMS_EQUIVALENTLABEL, can be used to map from one to the other. These contain the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SECLABEL_POLICY The object identifier for the security policy which the security label is issued under.	String
CRYPT_CERTINFO_CMS_SECLABEL_-CLASSIFICATION The security classification for the content identified relative to the security policy being used. There are six standard classifications (described below) and an extended number of user-defined classifications, for more information see the S/MIME standards documents and X.411.	Numeric
CRYPT_CERTINFO_CMS_SECLABEL_PRIVACYMARK A privacy mark value which unlike the security classification isn't used for access control to the message contents. See S/MIME standards documents for more information.	Numeric
CRYPT_CERTINFO_CMS_SECLABEL_CATTYPE	String
CRYPT_CERTINFO_CMS_SECLABEL_CATVALUE See S/MIME standards documents.	Binary data

The security classification can have one of the following predefined values (which are relative to the security policy and whose interpretation can vary from one organisation to another), or policy-specific, user-defined values which lie outside this range:

Value
CRYPT_CLASSIFICATION_UNMARKED
CRYPT_CLASSIFICATION_UNCLASSIFIED
CRYPT_CLASSIFICATION_RESTRICTED
CRYPT_CLASSIFICATION_CONFIDENTIAL
CRYPT_CLASSIFICATION_SECRET
CRYPT_CLASSIFICATION_TOP_SECRET

S/MIME Capabilities

This CMS attribute provides additional information about the capabilities and preferences of the sender of a message, allowing them to indicate their preferred encryption algorithm(s) and . The attribute is identified by CRYPT_CERTINFO_-CMS_SMIMECAPABILITIES and can contains any of the following values:

Value	Description
CRYPT_CERTINFO_CMS_-SMIMECAP_3DES CRYPT_CERTINFO_CMS_-SMIMECAP_CAST128 CRYPT_CERTINFO_CMS_-SMIMECAP_DES CRYPT_CERTINFO_CMS_-SMIMECAP_IDEA CRYPT_CERTINFO_CMS_-SMIMECAP_RC2 CRYPT_CERTINFO_CMS_-SMIMECAP_RC5 CRYPT_CERTINFO_CMS_-SMIMECAP_SKIPJACK	The sender supports the use of these algorithms. When encoding them, cryptlib will order them by algorithm strength so that triple DES will be preferred over Skipjack which will be preferred over DES.

CRYPT_CERTINFO_CMS_- SMIMECAP_- PREFERSIGNEDDATA	The sender would prefer to be sent signed data.
CRYPT_CERTINFO_CMS_- SMIMECAP_- CANNOTDECRYPTANY	The sender can't handle any form of encrypted data.

To indicate that you can support messages encrypted with triple DES and Cast-128, you would use::

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CMS_SMIMECAP_3DES,
   CRYPT_UNUSED );
cryptSetAttribute( certificate, CRYPT_CERTINFO_CMS_SMIMECAP_CAST128,
   CRYPT_UNUSED );
```

If you're using CRYPT_FORMAT_SMIME data, cryptlib will automatically add the appropriate attributes for you so there's normally no need to set these attributes yourself.

Signing Certificate

This CMS attribute provides additional information about the certificate used to sign a message, is identified by CRYPT_CERTINFO_SIGNINGCERTIFICATE, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGNINGCERT_CERTS The SHA-1 hash of the signing certificate	Binary data
CRYPT_CERTINFO_CMS_SIGNINGCERT_POLICIES The object identifier for the policy which applies to the signing certificate	String

S/MIME

S/MIME is a standard format for transferring signed, encrypted, or otherwise processed data as a MIME-encoded message (for example as email or embedded in a web page). The MIME-encoding is only used to make the result palatable to mailers, it's also possible to process the data without the MIME encoding.

The exact data formatting and terminology used requires a bit of further explanation. In the beginning there was PKCS #7, a standard format for signed, encrypted, or otherwise processed data. When the earlier PEM secure mail standard failed to take off, PKCS #7 was wrapped up in MIME encoding and christened S/MIME version 2. Eventually PKCS #7 was extended to become the Cryptographic Message Syntax (CMS), and when that's wrapped in MIME it's called S/MIME version 3.

In practice it's somewhat more complicated than this since there's significant blurring between S/MIME version 2 and 3 (and PKCS #7 and CMS). The main effective difference between the two is that PKCS #7/SMIME version 2 are completely tied to X.509 certificates, certification authorities, certificate chains, and other paraphernalia, CMS can be used without requiring all these extras if necessary, and S/MIME version 3 restricts CMS back to requiring X.509 for S/MIME version 2 compatibility.

The cryptlib native format is CMS used in the configuration which doesn't tie it to the use of certificates (so it'll work with PGP keys, raw public/private keys, and other keying information as well as with X.509 certificates). In addition to this format, cryptlib also supports the S/MIME format which is tied to X.509 — this is just the cryptlib native format restricted so that the full range of key management options aren't available. If you want to interoperate with other implementations, you should use this format since many implementations can't work with the newer key management options which were added in CMS.

You can specify the use of the restricted CMS/SMIME format with the formatting specifier CRYPT_FORMAT_CMS or CRYPT_FORMAT_SMIME (they're almost identical, the few minor differences are explained further on), which tells cryptlib to use the restricted CMS/SMIME rather than the (default) unrestricted CMS format. You can use the format specifiers with **cryptExportKeyEx** and **cryptCreateSignatureEx** (which take as their third argument the format specifier) and with **cryptCreateEnvelope**. The use of this format with the mid-level encryption and signature functions is explained in more detail in "Encrypting/Decrypting Data" on page 64 and "Signing Data" on page 72.

S/MIME Enveloping

Although it's possible to use the S/MIME format directly with the mid-level signature and encryption functions, S/MIME requires a considerable amount of extra processing above and beyond that required by cryptlib's default format, so it's easiest to let cryptlib take care of this extra work for you by using the enveloping functions to process S/MIME data.

To create an envelope which uses the S/MIME format, call **cryptCreateEnvelope** as usual but specify a format type of CRYPT_FORMAT_SMIME instead of the usual CRYPT_FORMAT_CRYPTLIB:

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_SMIME );

/* Perform enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

Creating the envelope in this way restricts cryptlib to using the standard X.509-based S/MIME data format instead of the more flexible data format which is used for envelopes by default.

Encrypted Enveloping

S/MIME supports password-based enveloping in the same way as ordinary cryptlib envelopes (in fact the two formats are identical). Public-key encrypted enveloping is supported only when the public key is held in an X.509 certificate, because of this restriction the private decryption key must also have a certificate attached to it. Apart from these restrictions, public-key based S/MIME enveloping also works the same way as standard cryptlib enveloping. For example to encrypt data using the key contained in an X.509 certificate you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_SMIME );

/* Add the certificate */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PUBLICKEY, certificate
   );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Since the certificate will originally come from a keyset, a simpler alternative to reading the certificate yourself and explicitly adding it to the envelope is to let cryptlib do it for you by first adding the keyset to the envelope and then specifying the email address of the recipient or recipients of the message with the CRYPT_-ENVINFO_RECIPIENT attribute:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_SMIME );

/* Add the encryption keyset and recipient email address */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_ENCRYPT,
   cryptKeyset );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_RECIPIENT,
   "person@company.com", 18 );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

For each message recipient which you add, cryptlib will look up key in the encryption keyset and add the appropriate information to the envelope to encrypt the message to that person. This is the recommended way of handling public-key encrypted enveloping, since it lets cryptlib handle the certificate details for you and makes it possible to manage problem areas such as cases where the same email address is present in multiple certificates of which only one is valid for message encryption. If you wanted to handle this case yourself, you'd have to use **cryptEnvelopeQuery** to search the duplicate certificates yourself as described in "Handling Multiple Certificates with the Same Name" on page 56.

Deenveloping works as for standard enveloping:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_ENVINFO_TYPE requiredAttribute;
```

```
int bytesCopied, status;

/* Create the envelope and add the private key keyset and data */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_KEYSET_DECRYPT,
   privKeyKeyset );
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );

/* Find out what we need to continue and, if it's a private key, add
   the password to recover it */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_CURRENT_COMPONENT,
   &requiredAttribute );
if( requiredResource != CRYPT_ENVINFO_PRIVATEKEY )
   /* Error */
cryptAddAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Pop the data and clean up */
cryptPopData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptDestroyEnvelope( cryptEnvelope );
```

More information on public-key encrypted enveloping, including its use with crypto devices such as smart cards and Fortezza cards, is given in “Advanced Enveloping” on page 32.

Digitally Signed Enveloping

Like public-key encrypted enveloping, digitally signed enveloping works just like standard enveloping except that the signing key is restricted to one which has a full chain of X.509 certificates (or at least a single certificate) attached to it. For example if you wanted to sign data using a private key contained in `sigKeyContext`, you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_SMIME );

/* Add the signing key */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

When you sign data in this manner, cryptlib includes any certificates attached to the signing key alongside the message. Although you can sign a message using a key with a single certificate attached to it, it’s safer to use one which has a full certificate chain associated with it because including only the key certificate with the message requires that the recipient locate any other certificates which are required to verify the signature. Since there’s no easy way to do this, signing a message using only a standalone certificate can cause problems when the recipient tries to verify the signature.

Verifying the signature on the data works slightly differently from the normal signature verification process since the signed data already carries with it the complete certificate chain required to verify the signature. This means that you don’t have to push a signature verification keyset or key into the envelope to verify the signature because the required certificate is already included with the data:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;
```

```
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

/* Determine the result of the signature check */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_RESULT,
   &sigCheckStatus );

cryptDestroyEnvelope( cryptEnvelope );
```

Since the certificate is included with the data, anyone could alter the data, re-sign it with their own certificate, and then attach their certificate to the data. To avoid this problem, cryptlib provides the ability to verify the chain of certificates, which works in combination with cryptlib's certificate trust manager. You can obtain the certificate object containing the signing certificate chain with:

```
CRYPT_CERTIFICATE cryptCertChain;

cryptGetAttribute( cryptEnvelope, CRYPT_SIGNATURE, &cryptCertChain );
```

You can work with this certificate chain as usual, for example you may want to display the certificates and any related information to the user. At the least, you should verify the chain using **cryptCheckCert**. More details on working with certificate chains are given in "Maintaining Keys and Certificates" on page 114, and details on signed enveloping (including its use with crypto devices like smart cards and Fortezza cards) are given in "Advanced Enveloping" on page 32.

Detached Signatures

So far, the signature for the signed data has always been included with the data itself, allowing it to be processed as a single blob. cryptlib also provides the ability to create detached signatures in which the signature is held separate from the data. This leaves the data being signed unchanged and produces a standalone signature as the result of the encoding process.

To specify that an envelope should produce a detached signature rather than standard signed data, you should add a CRYPT_ENVINFO_DETACHED_SIGNATURE component to the envelope with the value set to 'true' (any nonzero value) before you push in any data

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHED_SIGNATURE, 1
   );
```

Apart from that, the creation of detached signatures works just like the creation of standard signed data, with the result of the enveloping process being the standalone signature (without the data attached):

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_SMIME );

/* Add the signing key */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the detached signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, detachedSignature,
   detachedSignatureBufferSize, &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Verifying a detached signature requires an extra processing step, since the signature is no longer bundled with the data. First, you need to push in the detached signature (to tell cryptlib what to do with any following data). After you've pushed in the signature and followed it up with the usual zero-byte push to wrap up the processing, you need to push in the data which was signed by the detached signature as the second processing step:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;

cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_AUTO );

/* Push in the detached signature */
cryptPushData( cryptEnvelope, detachedSignature, detachedSigLength );
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Push in the data */
cryptPushData( cryptEnvelope, data, dataLength );
cryptPushData( cryptEnvelope, NULL, 0, NULL );

/* Determine the result of the signature check */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_RESULT,
   &sigCheckStatus );

cryptDestroyEnvelope( cryptEnvelope );
```

Since the data wasn't enveloped to begin with, there's nothing to deenvelope which means there's nothing to pop out of the envelope (apart from the signing certificate chain which you can obtain as before by reading the CRYPT_ENVINFO_-SIGNATURE attribute).

In case you're not sure whether a signature includes data or not, you can query its status by checking the value of the CRYPT_ENVINFO_DETACHED_SIGNATURE attribute after you've pushed in the signature:

```
int isDetachedSignature;

/* Push in the signed enveloped data */
cryptPushData( cryptEnvelope, signedData, signedDataLength,
   &bytesCopied );

/* Check the signed data type */
cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHED_SIGNATURE, &
   isDetachedSignature );
if( isDetachedSignature )
   /* Detached signature */
else
   /* Signed data + signature */
```

Extra Signature Information

S/MIME signatures can include with them extra information such as the time at which the message was signed. Normally cryptlib will add and verify this information for you automatically, however you can also handle it yourself if you require extra control over what's included with the signature. The extra information is specified as a CRYPT_CERTTYPE_CMS_ATTRIBUTES certificate object which is described in more detail in "Further Certificate Objects" on page 124. To include this information with the signature you should add it to the envelope alongside the signing key as CRYPT_ENVINFO_SIGNATURE_EXTRADATA:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_CERTIFICATE cmsAttributes;

/* Create the CMS attribute object */
cryptCreateCert( &cmsAttributes, CRYPT_CERTTYPE_CMS_ATTRIBUTES );
/* ... */

/* Create the envelope and add the signing key and signature
   information */
cryptCreateEnvelope( &cryptEnvelope, CRYPT_FORMAT_CMS );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   signatureKeyContext );
```

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_EXTRADATA,
   cmsAttributes );
cryptDestroyCert( cmsAttributes );

/* Add the data size information and data, push a zero-byte data block
   to wrap up the enveloping, and pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptPushData( cryptEnvelope, NULL, 0, NULL );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Verifying a signature which includes this extra information works just like standard signature verification since cryptlib handles it all for you. Just as you can obtain a certificate chain from a signature, you can also obtain the extra signature information from the envelope:

```
CRYPT_CERTIFICATE cmsAttributes;

cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_EXTRADATA,
   &cmsAttributes );
```

You can now work with the signing attributes as usual, for example you may want to display any relevant information to the user. More details on working with signature attributes are given in “Further Certificate Objects” on page 124.

The example above created a CRYPT_FORMAT_CMS envelope which means that cryptlib will add certain default signing attributes to the signature when it creates it. If the envelope is created with CRYPT_FORMAT_SMIME instead of CRYPT_FORMAT_CMS, cryptlib will add an extra set of S/MIME-specific attributes which indicate the preferred encryption algorithms for use when an S/MIME enabled mailer is used to send mail to the signer. This information is used for backwards-compatibility reasons because most S/MIME mailers will quietly default to using very weak 40-bit keys if they’re not explicitly told to use proper encryption such as triple DES (cryptlib will never use weakened encryption since it doesn’t even provide this capability).

Because of this default-to-insecure encryption problem, cryptlib includes with a CRYPT_FORMAT_SMIME signature additional information to indicate that the sender should use a non-weakened algorithm such as triple DES, CAST-128, or IDEA. With a CRYPT_FORMAT_CMS signature this additional S/MIME-specific information isn’t needed so cryptlib doesn’t include it.

From Envelopes to S/MIME

The enveloping process produces binary data as output which then needs to be wrapped up in the appropriate MIME headers and formatting before it can really be called S/MIME. The exact mechanisms used depend on the mailer code or software interface to the mail system you’re using, general guidelines for the different enveloped data types are given below.

S/MIME Content Types

MIME is the Internet standard for communicating complex data types via email, and provides for tagging of message contents and safe encoding of data to allow it to pass over data paths which would otherwise damage or alter the message contents. Each MIME message has a top-level type, subtype, and optional parameters. The top-level types are `application`, `audio`, `image`, `message`, `multipart`, `text`, and `video`.

Most of the S/MIME secured types have a content type of `application/pkcs7-mime`, except for detached signatures which have a content type of `application/pkcs7-signature`. The content type usually also includes an additional `smime-type` parameter whose value depends on the S/MIME type and is

described in further detail below. In addition it's usual to include a content-disposition field whose value is also explained below.

Since MIME messages are commonly transferred via email and this doesn't handle the binary data produced by cryptlibs enveloping, MIME also defines a means of encoding binary data as text. This is known as content-transfer-encoding.

Data

The innermost, plain data content should be converted to canonical MIME format and have a standard MIME header which is appropriate to the data content, with optional encoding as required. For the most common type of content (plain text), the header would have a content-type of `text/plain`, and possibly optional extra information such as a content transfer encoding (in this case `quoted-printable`), content disposition, and whatever other MIME headers are appropriate. This formatting is normally handled for you by the mailer code or software interface to the mail system you're using.

Signed Data

For signed data the MIME type is `application/pkcs7-mime`, the smime-type parameter is `signed-data`, and the extensions for filenames specified as parameters is `.p7m`. A typical MIME header for signed data is therefore:

```
Content-Type: application/pkcs7-mime; smime-type=signed-data;
   name=smime.p7m
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename=smime.p7m

encoded signed data
```

Detached Signature

Detached signatures represent a special instance of signed data in which the data to be signed is carried as one MIME body part and the signature is carried as another body part. The message is encoded as a multipart MIME message with the overall message having a content type of `multipart/signed` and a protocol parameter of `application/pkcs7-signature`, and the signature part having a content type of `application/pkcs7-signature`.

Since the data precedes the signature, it's useful to include the hash algorithm used for the data as a parameter with the content type (cryptlib processes the signature before the data so it doesn't require it, but other implementations may not be able to do this). The hash algorithm parameter is given by `micalg=sha1` or `micalg=md5` as appropriate. When receiving S/MIME messages you can ignore this value since cryptlib will automatically use the correct type based on the signature.

A typical MIME header for a detached signature is therefore:

```
Content-Type: multipart/signed; protocol=application/pkcs7-signature;
   micalg=sha1; boundary=boundary

--boundary
Content-Type: text/plain Content-Transfer-Encoding: quoted-printable

signed text

--boundary
Content-Type: application/pkcs7-signature; name=smime.p7s
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename=smime.p7s

encoded signature

--boundary—
```

Encrypted Data

For encrypted data the MIME type is `application/pkcs7-mime`, the smime-type parameter is `enveloped-data`, and the extension for filenames specified as parameters is `.p7m`. A typical MIME header for encrypted data is therefore:

```
Content-Type: application/pkcs7-mime; smime-type=enveloped-data;
   name=smime.p7m
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename=smime.p7m

encoded encrypted data
```

Nested Content

Unlike straight CMS nested content, S/MIME nested content requires a new level of MIME encoding for each nesting level. For the minimum level of nesting (straight signed or encrypted data) you need to first MIME-encode the plain data, then envelope it to create CMS signed or encrypted data, and then MIME-encode it again. For the typical case of signed, encrypted data you need to MIME-encode, sign, MIME-encode again, encrypt, and then MIME-encode yet again (rumours that S/MIME was designed by a consortium of network bandwidth vendors and disk drive manufacturers are probably unfounded).

Since the nesting information is contained in the MIME headers, you don't have to specify the nested content type using CRYPT_ENVINO_CONTENTTYPE as you do with straight CMS enveloped data (this is one of the few actual differences between CRYPT_FORMAT_CMS and CRYPT_FORMAT_SMIME), cryptlib will automatically set the correct content type for you. Conversely, you need to use the MIME header information rather than CRYPT_ENVINFO_CONTENTTYPE when deenveloping data (this will normally handled for you by the mailer code or software interface to the mail system you're using).

Implementing S/MIME using cryptlib

Most of the MIME processing and encoding issues described above will be handled for you by the mail software which cryptlib is used with. To use cryptlib to handle S/MIME messages, you would typically register the various S/MIME types with the mail software and, when they are encountered, the mailer will hand the message content (the data which remains after the MIME wrapper has been removed) to cryptlib. cryptlib can then process the data and hand the processed result back to the mailer. The same applies for generating S/MIME messages.

Example: c-client/IMAP4

c-client is a portable Swiss army chainsaw interface to a wide variety of mail and news handling systems. One of the services it provides is full handling of MIME message parts which involves breaking a message down into a sequence of `BODY` structures each of which contains one MIME body part. The `type` member contains the content type (typically TYPEMULTIPART or TYPEAPPLICATION for the types used in S/MIME), the `subtype` member contains the MIME subtype, the `parameter` list contains any required parameters, and the `contents.binary` member contains outgoing binary data straight from the cryptlib envelope (c-client will perform any necessary encoding such as base64 if required). All of this information is converted into an appropriately-formatted MIME message by c-client before transmission.

Since IMAP supports the fetching of individual MIME body parts from a server, `contents.binary` can't be used to access incoming message data since only the header information may have been fetched, with the actual content still residing on the server. To fetch a particular body part, you need to use `mail_fetchbody`. If the body part is base64-encoded (denoted by the `encoding` member of the `BODY` having the value ENCBASE64) then you also need to call `rfc822_base64` to

decode the data so cryptlib can process it. In the unlikely event that the binary data is encoded as quoted-printable (denoted by ENCQUOTEDPRINTABLE, at least one broken mailer occasionally does this) you need to call `rfc822_qprint` instead. In either case the output can be pushed straight into a cryptlib envelope.

Example: Eudora

Eudora handles MIME content types through plug-in translators which are called through two functions, `ems_can_translate` and `ems_translate_file`. Eudora calls `ems_can_translate` with an `emsMIMEtype` parameter which contains information on the MIME type contained in the message. If this is an S/MIME type (for example `application/pkcs7-mime`) the function should return `EMSR_NOW` to indicate that it can process this MIME type, otherwise is returns `EMSR_CANT_TRANSLATE`.

Once the translator has indicated that it can process a message, Eudora calls `ems_translate_file` with input and output files to read the data from and write the processed result to. The translation is just the standard cryptlib enveloping or deenveloping process depending on whether the translator is an on-arrival or on-display one (used for deenveloping incoming messages) or a Q4-transmission or Q4-completion one (used for enveloping outgoing messages).

Example: MAPI

MAPI (Microsoft's mail API) defines two types of mailer extensions which allow cryptlib-based S/MIME functionality to be added to Windows mail applications. The first type is a spooler hook or hook provider, which can be called on delivery of incoming messages and on transmission of outgoing messages. The second type is a preprocessor, which is less useful and operates on outgoing messages only. The major difference between the two in terms of implementation complexity is that hook providers are full (although simple) MAPI service providers while preprocessors are extensions to transport providers (that is, if you've already written a transport provider you can add the preprocessor without too much effort; if you don't have a transport provider available, it's quite a bit more work). In general it's probably easiest to use a single spooler hook to handle inbound and outbound messages. You can do this by setting both the HOOK_INBOUND and HOOK_OUTBOUND flags in the hook's PR_RESOURCE_FLAGS value.

Messages are passed to hooks via `ISpoolerHook::OutboundMsgHook` (for outgoing messages) and `ISpoolerHook::InboundMsgHook` (for incoming messages). The hook implementation itself is contained in a DLL which contains the `HPProviderInit` entry point and optional further entry points used to configure it (for example a message service entry point for program-based configuration and a `WIZARDENTRY` for user-based configuration).

Example: Windows'95/98 and NT Shell

Windows allows a given MIME content type to be associated with an application to process it. You can set up this association by calling `MIMEAssociationDialog` and setting the MIMEASSOCDLG_FL_REGISTER_ASSOC flag in the `dwInFlags` parameter, which will (provided the user approves it) create an association between the content type you specify in the `pcszMIMEContentType` parameter and the application chosen by the user. This provides a somewhat crude but easy to set up mechanism for processing S/MIME data using a cryptlib-based application.

Encryption Devices and Modules

cryptlib's standard cryptographic functionality is provided through its built-in implementations of the required algorithms and mechanisms, however in some cases it may be desirable to use external implementations contained in cryptographic hardware or portable cryptographic devices like smart cards or PCMCIA cards. Examples of external implementations are:

- Cryptographic hardware accelerators
- PCMCIA crypto cards such as Fortezza cards
- Cryptographic smart cards
- Datakeys
- PKCS #11 crypto tokens
- Dallas iButtons
- Software encryption modules

The most common use for an external implementation is one where the hardware provides secure key storage and management functions, or where it provides specific algorithms or performance which may not be available in software.

Using an external implementation involves conceptually plugging in the external hardware or software alongside the built-in capabilties provided by cryptlib and then creating cryptlib objects (for example encryption contexts) via the device. The external cryptographic implementation is viewed as a logical device, although the "device" may be just another software implementation.

Creating/Destroying Device Objects

Devices are accessed as device objects which work in the same general manner as other cryptlib objects. You open a connection to a device using **cryptDeviceOpen**, specifying the type of device you want to use and the name of the particular device if required or null of there's only one device type possible. This opens a connection to the device. Once you've finished with the device, you use **cryptDeviceClose** to sever the connection and destroy the device object:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, deviceType, deviceName );

/* Use the services provided by the device */

cryptDeviceClose( cryptDevice );
```

The available device types are:

Device	Description
CRYPT_DEVICE_FORTEZZA	Fortezza PCMCIA card.
CRYPT_DEVICE_PKCS11	PKCS #11 crypto token. These devices are accessed via their names, see the section on PKCS #11 devices for more details.

Most of the devices are identified implicitly so there's no need to specify a device name and you can pass null as the name parameter (the exception is PKCS #11 devices, which are covered in more detail further on). Once you've finished with the device, you use **cryptDeviceClose** to deactivate it and destroy the device object. For example to work with a Fortezza card you would use:

```
CRYPT_DEVICE cryptDevice;
```

```
cryptDeviceOpen( &cryptDevice, CRYPT_DEVICE_FORTEZZA, NULL );

/* Use the services provided by the device */

cryptDeviceClose( cryptDevice );
```

If the device can't be accessed, cryptlib will return CRYPT_ERROR_OPEN to indicate that it couldn't establish a connection and activate the device. Note that the CRYPT_DEVICE is passed to **cryptDeviceOpen** by reference, as it modifies it when it activates the device. In all other routines in cryptlib, CRYPT_DEVICE is passed by value.

Activating and Controlling Cryptographic Devices

Once cryptlib has established a connection to the device, you may need to authenticate yourself to it or perform some other control function with it before it will allow itself to be used. You can do this by setting various device attributes, specifying the type of action you want to perform on the device and any additional information which may be required. In the case of user authentication, the additional information will consist of a PIN or password which enables access. Many devices recognise two types of access code, a user-level code which provides standard access (for example for encryption or signing) and a supervisor-level code which provides extended access to device control functions, for example key generation and loading. An example of someone who may require supervisor-level access is a site security officer (SSO) who can load new keys into a device or reenable its use after a user has been locked out.

The control functions you can perform are as follows.

Initialise Device

By setting the CRYPT_DEVINFO_INITIALISE attribute, you can initialise the device. This clears keys and other information in the device and prepares it for use. In devices which support supervisor access you need to supply the initial supervisor PIN when you call this function:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_INITIALISE,
   initialPin, initialPinLength );
```

User Authentication

Before you can use the device you generally need to authenticate yourself to it with a PIN or password. To authenticate yourself as supervisor, set the CRYPT_-DEVINFO_AUTHENT_SUPERVISOR attribute; to authenticate yourself as user, set the CRYPT_DEVINFO_AUTHENT_USER attribute. For example to authenticate yourself to the device using a PIN as a normal user you would use:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_AUTHENT_USER, pin,
   pinLength );
```

To authenticate yourself to the device using a PIN for supervisor-level access you would use:

```
cryptSetAttributeString( cryptDevice,
   CRYPT_DEVINFO_AUTHENT_SUPERVISOR, pin, pinLength );
```

If the PIN or password you've supplied is incorrect, cryptlib will return CRYPT_ERROR_WRONGKEY. If the device doesn't support this type of access, it will return CRYPT_ERROR_PARAM2. Note that, as is traditional for most PIN and password checking systems, some devices may only allow a limited number of access attempts before locking out the user, requiring CRYPT_DEVINFO_AUTHENT_-SUPERVISOR access to reenable user access.

Zeroise Device

The CRYPT_DEVINFO_ZEROISE attribute works much like CRYPT_DEVINFO_-INITIALISE except that its specific goal is to clear any sensitive information such as

encryption keys from the device (it's often the same as device initialisation, but sometimes will only specifically erase the keys and in some cases may even disable the device). In devices which support supervisor access you need to supply the initial supervisor PIN when you call this function, otherwise you should set the data value to null and the length to CRYPT_UNUSED:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_INITIALISE, NULL,
   CRYPT_UNUSED );
```

Extended Device Control Functions

Some device control functions may require more than a single parameter. You can perform these functions using **cryptDeviceControlEx**.

Setting/Changing User Authentication Values

You can set or change a user authentication value such as a password or PIN with CRYPT_DEVINFO_SET_AUTHENT_SUPERVISOR (for the supervisor authentication value) or CRYPT_DEVINFO_SET_AUTHENT_USER (for the user authentication value), specifying the current and new authentication values:

```
cryptDeviceControlEx( cryptDevice, CRYPT_DEVINFO_SET_AUTHENT_USER,
   currentPin, currentPinLength, newPin, newPinLength );
```

Working with Device Objects

With the device activated and the user authenticated, you can use its cryptographic capabilities in encryption contexts as if it were a standard part of cryptlib. In order to specify the use of the cryptographic device rather than cryptlibs built-in functionality, cryptlib provides the **cryptDeviceCreateContext** and **cryptDeviceQueryCapability** functions which are identical to **cryptCreateContext** and **cryptQueryCapability** but take as an additional argument the handle to the device. For example to create a standard RSA encryption context you would use:

```
cryptCreateContext( &cryptContext, CRYPT_ALGO_RSA );
```

To create an RSA encryption context using an external cryptographic device you would use:

```
cryptDeviceCreateContext( cryptDevice, &cryptContext, CRYPT_ALGO_RSA
   );
```

After this you can use the encryption context as usual, both will function in an identical manner with cryptlib keeping track of whether the implementation is via the built-in functionality or the external device. In this way the use of any form of external hardware for encryption is completely transparent after the initial step of activating and initialising the hardware.

For an example of how you might utilise external hardware, let's use a generic DES/triple DES hardware accelerator (identified by the label "DES/3DES accelerator") accessed as a PKS #11 device. To use the triple DES hardware instead of cryptlibs built-in triple DES implementation you would use:

```
CRYPT_DEVICE cryptDevice;
CRYPT_CONTEXT cryptContext;

/* Activate the DES hardware and create a context in it */
cryptDeviceOpen( &cryptDevice, CRYPT_DEVICE_PKCS11, "DES/3DES
   accelerator" );
cryptDeviceCreateContext( cryptDevice, &cryptContext, CRYPT_ALGO_3DES
   );

/* Generate a key in the DES hardware */
cryptGenerateKey( cryptContext );

/* Encrypt data using the hardware */
cryptEncrypt( cryptContext, data, dataLength );

/* Destroy the context and shut down the DES hardware */
```

```
cryptDestroyContext( cryptContext );
cryptDeviceClose( cryptDevice );
```

After the context has been created with **cryptDeviceCreateContext**, the use of the context is identical to a standard encryption context. There is no other (perceptual) difference between the use of a built-in implementation and an external implementation.

Key Storage in Crypto Devices

When you create a normal public-key context and load or generate a key into it, the context goes away when you destroy it or shut down cryptlib. If the context is created in a crypto device, the public and private keys from the context don't go away when the context is destroyed but are stored inside the device for later use. You can later recreate the context using the key stored in the device by treating the device as a keyset containing a stored key. For example to create an RSA key in a device you would use:

```
CRYPT_CONTEXT privKeyContext;

/* Create the RSA context, set a label for the key, and generate a key
   into it */
cryptCreateContext( &privKeyContext, CRYPT_ALGO_RSA );
cryptSetAttributeString( privKeyContext, CRYPT_CTXINFO_LABEL, label,
   labelLength );
cryptGenerateKey( privKeyContext );

/* Destroy the context */
cryptDestroyContext( privKeyContext );
```

Although the context has been destroyed, the key itself is still held inside the device. To recreate the context at a later date, you can treat the device as if it were a keyset, using the label as the key ID:

```
CRYPT_CONTEXT privKeyContext;

cryptGetPrivateKey( cryptDevice, &privKeyContext, CRYPT_KEYID_NAME,
   label, NULL );
```

Since you've already authenticated yourself to the device, you don't need to specify a password.

Considerations when Working with Devices

There are several considerations to be taken into account when using crypto devices, the major one being that requiring that crypto hardware be present in a system automatically limits the flexibility of your application. There are some cases where the use of certain types of hardware (for example Fortezza cards) may be required, but in many instances the reliance on specialised hardware can be a drawback.

The use of hardware crypto implementations can also complicate key management, since keys generated or loaded into the hardware usually can't be extracted again afterwards (this is a security feature of the hardware which makes external access to the key impossible, and works in the same way as cryptlib's own storing of keys inside it's security perimeter). This means that if you have a crypto device which supports (say) DES and RSA encryption, then to export an encrypted DES key from a context stored in the device, you need to use an RSA context also stored inside the device, since a context located outside the device won't have access to the DES context's key.

Another consideration which needs to be taken into account is the data processing speed of the device. In many cases it is preferable to use cryptlibs built-in implementation of an algorithm rather than the one provided by the device, especially where security isn't a major concern. For example when hashing data prior to signing it, cryptlibs built-in hashing capabilities should be used in preference to any provided by the device, since cryptlib can process data at the full memory bandwidth using a processor typically clocked at hundreds of megahertz while a crypto device has to move data over a slow I/O bus to be processed by a processor typically clocked at

tens of megahertz or even a few megahertz. In addition when encrypting or decrypting sizeable amounts of data using one-off session keys (as opposed to long-term keys used for multiple lots of data, which are usually stored securely inside the device) it may be preferable to use cryptlibs high-speed encryption capabilities, particularly with devices such as smart cards and to a lesser extent PCMCIA cards, which are severely limited by their slow I/O throughput.

A final consideration concerns the limitations of the encryption engine in the device itself. Although cryptlib provides a great deal of flexibility in its software crypto implementations, most hardware devices have only a single encryption engine (possibly augmented by the ability to store multiple encryption keys in the device) through which all data must pass. What this means is that each time a different key is used, it has to be loaded into the device's encryption engine before it can be used to encrypt or decrypt data, a potentially time-consuming process. For example if two encryption contexts are created via a device and both are used alternately to encrypt data, the key corresponding to each context has to be loaded by the device into its encryption engine before the encryption can begin (while most devices can store multiple keys, few can keep more than one at a time ready for use in their encryption engine).

What this means is that although cryptlib will allow you to create as many contexts via a device as the hardware allows, it's generally not a good idea to have more than a single context of each type in use at any one time. For example you could have a single conventional encryption context (using the devices crypto engine), a single digital signature context (using the devices public-key engine), and a single hash context (using the devices CPU, or preferably cryptlib itself) active, but not two conventional encryption contexts (which would have to share the encryption engine) or two digital signature contexts (which would have to share the public-key engine).

Fortezza Cards

cryptlib provides compete Fortezza card management capabilities, allowing you to initialise and program a card, generate or load keys into it, add certificates for the generated/loaded keys, update and change PINs, and perform other management functions. This provides full certificate authority workstation (CAW) capabilities.

The steps involved in programming a blank Fortezza card involve zeroising it (to erase any existing information, returning it to its factory-fresh status), initialising it with the default SSO PIN, changing the SSO PIN from the default setting to the actual SSO PIN, installing the CA root (PAA) certificate in the card, and finally setting the user PIN. This can be done using the following code:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, CRYPT_DEVICE_FORTEZZA, NULL );

/* Zeroise the card, erasing any existing information */
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_ZEROISE,
   zeroisePIN, strlen( zeroisePIN ) );

/* Initialise the card and set the SSO PIN*/
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_INITIALISE,
   defaultSSOPIN, strlen( defaultSSOPIN ) );
cryptDeviceControlEx( cryptDevice,
   CRYPT_DEVINFO_SET_AUTHENT_SUPERVISOR, defaultSSOPIN, strlen(
   defaultSSOPIN ), ssoPIN, strlen( ssoPIN ) );

/* Install the CA root (PAA) certificate */
cryptAddPublicKey( cryptDevice, cryptCert );

/* Set the user PIN */
cryptDeviceControlEx( cryptDevice, CRYPT_DEVINFO_SET_AUTHENT_USER,
   userPIN, strlen( userPIN ), userPIN, strlen( userPIN ) );

cryptDeviceClose( cryptDevice );
```

Note that the Fortezza control firmware requires that these steps be performed in a continuous sequence of operations, without removing the card or closing the device. If you interrupt the process halfway through, you need to start again.

After the above programming process has completed, you can generate further keys into the device, load certificates, and so on. This provides the same functionality as a CAW.

PKCS #11 Devices

Although most of the devices which cryptlib interfaces with have specialised, single-purpose interfaces, PKCS #11 provides a general-purpose interface which can be used with a wide selection of parameters and in a variety of ways. The following section covers the installation of PKCS #11 modules and documents the way in which cryptlib interfaces to PKCS #11 modules.

Installing New PKCS #11 Modules

You can install new PKCS #11 modules by setting the names of the drivers in cryptlibs configuration database. The module names are specified using the configuration options CRYPT_OPTION_DEVICE_PKCS11_DVR01 ... CRYPT_OPTION_DEVICE_PKCS11_DVR05, cryptlib will step through the list and load each module in turn. For example to use the Gemplus GemSAFE driver, you would use:

```
cryptSetAttributeString( CRYPT_UNUSED,
   CRYPT_OPTION_DEVICE_PKCS11_DVR01, "w32pk2ig.dll", 12 );
```

On startup, cryptlib will load the specified modules and make them available as CRYPT_DEVICE_PKCS11 devices. When the module is loaded, cryptlib will query each module for the device name, this is the name which you should use to access it using **cryptDeviceOpen**.

Since the drivers are dynamically loaded on startup by cryptlib, specifying a driver as a configuration option won't immediately make it available for use. To make the driver available, you have to restart cryptlib or the application using it so that cryptlib can load the driver on startup.

Accessing PKCS #11 Devices

PKCS #11 devices are identified by the device name, for example the Litronix PKCS #11 driver indentifies itself as "Litronix CryptOki Interface" so you would create a device object of this type with:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, CRYPT_DEVICE_PKCS11, "Litronix CryptOki
   Interface" );
```

Some PKCS #11 devices allow the use of multiple physical or logical crypto tokens as part of a single device (for example a smart card reader device might have two slots which can each contain a smart card, or the reader itself might function as a crypto token alongside the smart card which is inserted into it). To identify a particular token in a device, you can specify its name after the device name, separated with a double colon. For example if the Litronix reader given in the example above contained two smart cards, you would access the one called "Signing smart card" with:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, CRYPT_DEVICE_PKCS11, "Litronix CryptOki
   Interface::Signing smart card " );
```

PKCS #11 Functions used by cryptlib

When loading a PKCS #11 module, cryptlib calls `C_Initialize` with a null parameter and then calls `C_GetTokenInfo` to obtain information about the device and `C_GetSlotList` to obtain details on the modules' slots. Before it unloads the module it calls `C_Finalize`.

When creating a device object, cryptlib calls `C_OpenSession` followed by `C_GetTokenInfo` to obtain information on a device such as whether it has its own random number generation capabilities. After this it calls `C_GetMechanismInfo` for each encryption capability required by cryptlib to allow the later creation of encryption contexts. cryptlib always opens serial sessions with the device since it does its own thread locking, so there's no need to support any complex locking capabilities.

When destroying a device object, cryptlib calls `C_Logout` (if necessary) followed by `C_CloseSession` to shut down the session. To perform encryption and signing, cryptlib calls the various PKCS #11 functions which are required for this purpose. These are `C_SignInit`, `C_Sign`, `C_VerifyInit`, `C_Verify`, `C_EncryptInit`, `C_Encrypt`, `C_DecryptInit`, and `C_Decrypt` depending on the function which is being performed. In addition cryptlib calls `C_CreateObject` or `C_GenerateKeyPair` as appropriate to load or generate keys, `C_FindObjectsInit`, `C_FindObjects`, and `C_FindObjectsFinal` to find objects in the device, and `C_GetAttributeValue` and `C_GetAttributeValue` to read and write object attributes. cryptlib will also call `C_GenerateRanom` if the device provides this functionality to augment its own built-in random number generation routines.

Alongside the encryption and signing functions, cryptlib may call a number of additional functions related to device management to handle device control functions. For example it will call `C_Login` and `C_Logout` to handle CRYPT_DEVINFO_-AUTHENT_USER and CRYPT_DEVINFO_AUTHENT_SUPERVISOR, and `C_InitToken` and `C_InitToken` to handle CRYPT_DEVINFO_INITIALISE and CRYPT_DEVINFO_ZEROISE.

In general, cryptlib has been designed to require as few specialised support functions from the underlying hardware as possible. In the case of PKCS #11 this means it will perform its own thread locking, device capability management, and so on, which means that it should handle even minimal PKCS #11 implementations.

Miscellaneous Topics

This chapter covers various miscellaneous topics not covered in other chapters such as how to obtain information about the encryption capabilities provided by cryptlib, how to obtain information about a particular encryption context, and how to ensure that your code takes advantage of new encryption capabilities provided with future versions of cryptlib.

Querying cryptlib's Capabilities

cryptlib provides two functions to query encryption capabilities, one of which returns information about a given algorithm and mode and the other which returns information on the algorithm and mode used in an encryption context. In both cases the information returned is in the form of a CRYPT_QUERY_INFO structure, which is described in "Data Structures" on page 173.

You can interrogate cryptlib about the details of a particular encryption algorithm and mode using **cryptQueryCapability**:

```
CRYPT_QUERY_INFO cryptQueryInfo;

cryptQueryCapability( algorithm, mode, &cryptQueryInfo );
```

This function will return a status of CRYPT_OK if the algorithm and mode are available or CRYPT_ERROR_NOTAVAIL if they aren't. If you just want to check whether a particular algorithm and more are available (without obtaining further information on them), you can set the query information parameter to null:

```
cryptQueryCapability( algorithm, mode, NULL );
```

This will return CRYPT_OK or CRYPT_ERROR_NOTAVAIL without trying to return algorithm information.

If you just want to check whether an algorithm is available, you can set the mode parameter to CRYPT_UNUSED:

```
cryptQueryCapability( algorithm, CRYPT_UNUSED, NULL );
```

Working with Configuration Options

In order to allow extensive control over its security and operational parameters, cryptlib provides a configuration database which can be used to tune its operation for different environments using either the Windows registry or configuration files which function similarly to Unix .rc files. This allows cryptlib to be customised on a per-user basis (for example it can remember which key the user usually uses to sign messages and offer to use this key by default), allows a system administrator or manager to set a consistent security policy (for example mandating the use of 1024-or 2048 bit public keys on a company-wide basis instead of the unsafe 512-bit keys used in most US-sourced products), and provides information on the use of optional features such as smart card readers, encryption hardware, and cryptographically strong random number generators. The configuration options which affect encryption parameter settings are automatically applied by cryptlib to operations such as key generation and data encryption and signing when you supply the CRYPT_USE_-DEFAULT option to a cryptlib function. When you call a function with a default parameter, cryptlib will consult its configuration database to determine which value it should use. If there are no database entries present, it will fall back to using hard-coded defaults, generally the safest, most conservative values available which emphasize security over performance.

The configuration database can be used to tune the way cryptlib works, with options ranging from algorithms and key sizes through to locations of key collections, preferred public/private keys to use for signing and encryption, and what to do when certain unusual conditions are encountered. The available options are listed below,

with the data type associated with each value being either a boolean (B), numeric (N), or string (S) value:

Value	Type	Description
CRYPT_OPTION_CERT_-ENCODECRITICAL CRYPT_OPTION_CERT_-DECODECRITICAL	B	Whether to encode certificates with the critical flag set in extensions, and whether to process the critical flag when decoding certificate extensions.
CRYPT_OPTION_CERT_-CHECKENCODING	B	Whether to verify that imported certificates have a valid encoding.
CRYPT_OPTION_CERT_-CREATEV3CERT	B	Whether to create X.509v3 certificates when signing/ exporting certificates.
CRYPT_OPTION_CERT_-ENCODE_VALIDITYNESTING CRYPT_OPTION_CERT_-DECODE_VALIDITYNESTING	B	Whether to encode certificates with validity period nesting, and whether to enforce validity period nesting when decoding certificates.
CRYPT_OPTION_CERT_-FIXEMAILADDRESS	B	Whether to update an obsolete email address encoding format when reading certificate objects.
CRYPT_OPTION_CERT_-FIXSTRINGS	B	Whether to correct invalid string encodings when importing certificate objects.
CRYPT_OPTION_CERT_-ISSUERNAMEBLOB	B	Whether to treat the certificate issuer name as a blob when writing it to a certificate. If the original encoding of the issuer name is invalid, this will propagate the invalid encoding rather than correcting it (this is required by some software to perform correct certificate chaining).
CRYPT_OPTION_CERT_-KEYIDBLOB	B	Whether to treat the data in the authorityKeyIdentifer extension as an opaque blob. This is recommended in order to handle the large number of incompatible formats for this extension.
CRYPT_OPTION_CERT_-PKCS10ALT	B	Whether to use the alternative encoding for PKCS #10 certification requests.
CRYPT_OPTION_CERT_-SIGNUNRECOGNISED-ATTRIBUTES	B	Whether to sign a certificate containing unrecognised extensions. If this option is set to false, the extensions will be omitted from the certificate when it is signed.

Value	Type	Description
CRYPT_OPTION_CERT_- TRUSTCHAINROOT	B	Whether to explcitly trust the root CA certificate in a certificate chain. If the chain doesn't contain a trusted CA certificate, this option can be used to allow the chain to be validated.
CRYPT_OPTION_CERT_- UPDATEINTERVAL	N	The update interval in days for CRL's.
CRYPT_OPTION_CERT_- VALIDITY	N	The validity period in days for certificates.
CRYPT_OPTION_CMS_- DEFAULTATTRIBUTES CRYPT_OPTION_SMIME_- DEFAULTATTRIBUTES	B	Whether to add the default CMS / S/MIME attributes to signatures (these are alternative names for the same option, since S/MIME uses CMS as the underlying format).
CRYPT_OPTION_- CONFIGCHANGED	B	Whether any configuration options have been changed from their original settings.
CRYPT_OPTION_DEVICE_- PKCS11_DVR01 ... CRYPT_OPTION_DEVICE_- PKCS11_DVR05	S	The module names of any PKCS #11 drivers which cryptlib should load on startup.
CRYPT_OPTION_DEVICE_- SERIALRNG	S	The port which the serial-based hardware random number generator is connected to, for example "COM1".
CRYPT_OPTION_DEVICE_- SERIALRNG_PARAMS	S	The port parameters for the serial-based hardware random number generator, for example "9600,8,N,1".
CRYPT_OPTION_ENCR_ALGO	N	Encryption algorithm given as a conventional-encryption CRYPT_ALGO.
CRYPT_OPTION_ENCR_HASH	N	Hash algorithm given as a hash CRYPT_ALGO.
CRYPT_OPTION_INFO_- COPYRIGHT	S	cryptlib copyright notice.
CRYPT_OPTION_INFO_- DESCRIPTION	S	cryptlib description.
CRYPT_OPTION_INFO_- MAJORVERSION CRYPT_OPTION_INFO_- MINORVERSION CRYPT_OPTION_INFO_- STEPPING	N	cryptlib major and minor version numbers and stepping number.

Value	Type	Description
CRYPT_OPTION_KEYING_ALGO	N	Key processing algorithm given as a hash CRYPT_ALGO.
CRYPT_OPTION_KEYING_-ITERATIONS	N	Number of times to iterate the key-processing algorithm.
CRYPT_OPTION_KEYS_LDAP_-CACERTNAME CRYPT_OPTION_KEYS_LDAP_-CERTNAME CRYPT_OPTION_KEYS_LDAP_-CRLNAME CRYPT_OPTION_KEYS_LDAP_-EMAILNAME CRYPT_OPTION_KEYS_LDAP_-OBJECTCLASS	S	The names of various LDAP attributes and object classes used for certificate storage/retrieval.
CRYPT_OPTION_MISC_-ASYNCINIT	B	Whether to bind in various drivers asynchronously when cryptlib is initialised. This performs the initialisation in a background thread rather than blocking on startup until the initialisation has completed.
CRYPT_OPTION_MISC_-FORCELOCK	B	Whether to force memory locking. If this option is set and memory can't be page-locked, many functions will return with the error code CRYPT_NOSECURE. This option can be enabled when it is absolutely essential that memory be locked to make it non-pageable.
CRYPT_OPTION_PKC_ALGO	N	Public-key encryption algorithm given as a public-key CRYPT_ALGO.
CRYPT_OPTION_PKC_KEYSIZE	N	Public-key encryption key size in bytes.
CRYPT_OPTION_SIG_ALGO	N	Signature algorithm given as a public-key encryption CRYPT_ALGO.
CRYPT_OPTION_SIG_KEYSIZE	N	Signature keysize in bytes.

CRYPT_OPTION_CONFIGCHANGED has special significance in that it contains the current state of the configuration options. If this value is FALSE, the current in-memory configuration options are still set to the same value they had when cryptlib was started. If set to TRUE, one or more options have been changed and they no longer match the values saved in permanent storage such as a hard disk. Writing this value back to FALSE forces the current in-memory values to be committed to permanent storage, so the two match up again.

Querying/Setting Configuration Options

You can manipulate the configuration options by getting or setting the appropriate attribute values. Since these apply to all of cryptlib rather than to any specific object,

you should set the object handle to CRYPT_UNUSED. For example to query the current default encryption algorithm you would use:

```
CRYPT_ALGO cryptAlgo;

cryptGetAttribute( CRYPT_UNUSED, CRYPT_OPTION_ENCR_ALGO, &cryptAlgo );
```

To set the default encryption algorithm to CAST-128, you would use:

```
CRYPT_ALGO cryptAlgo;

cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_ENCR_ALGO,
   CRYPT_ALGO_CAST );
```

A few of the options are used internally by cryptlib and are read-only (this is indicated in the options' description). These will return CRYPT_ERROR_-PERMISSION if you try to modify them to indicate that you don't have permission to change this option.

Saving Configuration Options

The changes you make to the configuration options only last while your program is running or while cryptlib is loaded. In order to make the changes permanent, you can save them to a permanent storage medium such as a hard disk by setting the CRYPT_OPTION_CONFIGCHANGED option to FALSE, indicating that the in-memory settings will be synced to disk so that the two match up. cryptlib will automatically reload the saved options when it starts.

The location of the saved configuration options depend on the system type on which cryptlib is running:

System	Location
BeOS Unix	$(HOME)/cryptlib.p15
DOS OS/2	./cryptlib.p15
MVS VM/CMS	CRYPTLIB P15
Tandem	$system.system.cryptlib
Windows 3.x	Windows/cryptlib.p15
Windows 2000 Windows 95/98 Windows NT	Windows/Profiles/*user_name*/Application Data/cryptlib.p15 (determined by the Windows application data CSIDL)

Where the operating system supports it, cryptlib will set the security options on the configuration information so that only the person who created it (and, usually, the system administrator) can access it. For example under Unix the file access bits are set to allow only the file owner (and, by extension, the superuser) to access the file, and under Windows NT with NTFS the file ACL's are set so that only the user who owns it can access or change it.

Obtaining Information About Cryptlib

cryptlib provides a number of read-only configuration options which you can use to obtain information about the version of cryptlib which you're working with.

These options are:

Value	Type	Description

Value	Type	Description
CRYPT_OPTION_INFO_-MAJORVERSION CRYPT_OPTION_INFO_-MINORVERSION CRYPT_OPTION_INFO_-STEPPING	N	The cryptlib major and minor version numbers and release stepping. For cryptlib 3.0 the major version number is 3 and the minor version number is 0. For beta release 1 the stepping is 1
CRYPT_OPTION_INFO_-DESCRIPTION	S	A text string containing a description of cryptlib.
CRYPT_OPTION_INFO_-COPYRIGHT	S	The cryptlib copyright notice.

Random Numbers

Several cryptlib functions require access to a source of cryptographically strong random numbers. These numbers are obtained by taking system information and stirring it into an internal data pool using the Secure Hash Algorithm. The random-data-gathering operation is controlled with the **cryptAddRandom** function, which can be used to either inject your own random information into the internal pool or to tell cryptlib to poll the system for random information. To add your own random data (such as keystroke timings when the user enters a password) to the pool, use:

```
cryptAddRandom( buffer, bufferLength );
```

Gathering Random Information

cryptlib can also gather its own random data by polling the system for random information. There are two polling methods you can use, a fast poll which returns immediately and retrieves a moderate amount of random information, and a slow poll which may take some time but which retrieves much larger amounts of random information. A fast poll is performed with:

```
cryptAddRandom( NULL, CRYPT_RANDOM_FASTPOLL );
```

In general you should sprinkle these throughout your code to build up the amount of randomness in the pool.

A slow poll is performed with:

```
cryptAddRandom( NULL, CRYPT_RANDOM_SLOWPOLL );
```

The effect of this call varies depending on the operating system. Under DOS the call returns immediately (see below). Under Windows 3.x the call will get all the information it can in about a second, then return (there is usually more information present in the system than can be obtained in a second). Under BeOS, OS/2, and on the Macintosh, the call will get all the information it can and then return. Under Unix, Windows 95, and Windows NT the call will spawn one or more separate processes or threads to perform the polling and will return immediately while the poll continues in the background.

Before the first use of a high-level capability such as encryption envelopes or calling **cryptGenerateKey** or **cryptExportKey** you must perform at least one slow poll (or, in some cases, several fast polls — see below) in order to accumulate enough random information in the pool to safely generate a key into an encryption context or export a key. On most systems cryptlib will perform a nonblocking randomness poll, so you can usually do this by calling the slow poll routine when your program starts so that the random information will have accumulated by the time you envelope the data or call **cryptGenerateKey** or **cryptExportKey**:

```
/* Program startup */

cryptAddRandom( NULL, CRYPT_RANDOM_SLOWPOLL );

/* Other code, slow poll runs in the background */
```

```
cryptGenerateKey( cryptContext );
```

If you forget to perform a slow poll beforehand, the high-level function will block until the slow poll completes. The fact that the call is blocking is usually fairly obvious, because your program will stop for the duration of the randomness poll. If no reliable random data is available then the high-level function which requires it will return the error CRYPT_ERROR_RANDOM.

Obtaining Random Numbers

You can obtain random data from cryptlib by using an encryption context with an algorithm which produces byte-oriented output (for example a stream cipher or a block cipher employed in a stream mode like CFB or OFB). To obtain random data, create a context, generate a key into it, and use the context to generate the required quantity of output by encrypting the contents of a buffer. Since the encryption output is random, it doesn't matter whether you zero the buffer beforehand or not. For example you can use the RC4 algorithm (a stream cipher) to generate random data with:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, CRYPT_ALGO_RC4 );
cryptGenerateKey( cryptContext )
cryptEncrypt( cryptContext, randomDataBuffer, randomDataLength );
cryptDestroyContext( cryptContext );
```

This will fill the data buffer with the required number of random bytes.

Random Information Gathering Techniques

The information obtained by the slow and fast polls under various operating systems and DOS is as follows:

BeOS

A fast poll adds the state of the high-speed system timer, and miscellaneous system-related information.

If a `/dev/random` driver which continually accumulates random data from the system is available, cryptlib will try to use this. A slow poll adds information on all teams (BeOS applications), threads, memory areas, message ports, semaphores, and images (code chunks) currently present in the system.

At least one slow poll is required to accumulate enough randomness for use by cryptlib.

DOS

Since DOS is so simple, it provides very little random information. A fast poll simply adds the value of the system clock to a 1-second resolution (which is next to useless). A slow poll relies on the presence of a `/dev/random`-style driver such as noise.sys, which records the timing between keystrokes, variations in disk access times, clock drift while the system is idling at the command line, the variation in EGA/VGA retrace events, and mouse movements.

The addition of external random data via **cryptAddRandom**() is strongly recommended under DOS. Without either this or the presence of a `/dev/random`-style driver, the random numbers (and therefor the encryption keys) generated by cryptlib will be extremely easy to guess and will provide virtually no security. For this reason any functions which require random numbers will always return CRYPT_NORANDOM unless a `/dev/random` driver is present or external randomness is added via **cryptAddRandom**.

At least one `/dev/random` slow poll is required to accumulate enough randomness for use by cryptlib.

Macintosh

A fast poll adds the status of the last alert, how much battery time is remaining and the voltage from all batteries, the internal battery status, the current date and time and time since system startup in ticks, the application heap limit and current and heap zone, free memory in the current and system heap, microseconds since system startup, whether QuickDraw has finished drawing, modem status, SCSI status information, maximum block allocatable without compacting, available stack space, the last QuickDraw error code, the event code and message, time, and mouse location for the next event in the event queue and the OS event queue, information on the topmost window which includes device-specific info, grafport information, visible and clipping region, pattern, pen, text, and colour information, the window variant and the colour table record for the window if there is one, mouse-related such as the mouse button status and mouse position, information on the window underneath the mouse, the size, handle, and location of the desk scrap/clipboard, information on the current thread, the sound mananger status (the number of allocated sound channels and the current CPU load from these channels), the speech manager status, and the serial port status (which includes information on recent errors, read and write pending status, and flow control values).

A slow poll adds information about each graphics device, information about each process including the name and serial number of the process, file and resource information, memory usage information, the name of the launching process, launch time, and accumulated CPU time, and the command type, trap address, and parameters for all commands in the file I/O queue (the parameters are quite complex and are listed on page 117 of IM IV, and include reference numbers, attributes, time stamps, length and file allocation information, finder info, and large amounts of other volume and filesystem-related data).

In addition it adds more constant but still application- and system-specific information such as the current font family ID, node ID of the local AppleTalk router, caret blink delay, CPU speed, double-click delay, sound volume, application and system heap zone, the number of resource types in the application, the number of sounds voices available, the FRef of the current resource file, volume of the sysbeep, primary line direction, computer SCSI disk mode ID, timeout before the screen is dimmed and before the computer is put to sleep, number of available threads in the thread pool, whether hard drive spin-down is disabled, the handle to the i18n resources, timeout time for the internal hard drive, the number of documents/files which were selected when the app started and for each document get the vRefNum, name, type, and version, the applications name, resource file reference number, and handle to the finder information, all sorts of statistics such as physical information, disk and write-protect present status, error status, and handler queue information, on floppy drives attached to the system. Also get the volume name, volume reference number and number of bytes free, for the volume in the drive, information on the head and tail of the vertical retrace queue, the parameter RAM settings, information about the machines geographic location, information on current graphics devices including device information such as dimensions and cursor information, and a number of handles to device-related data blocks and functions, and information about the dimentions and contents of the devices pixel image as well as the images resolution, storage format, depth, and colour usage, the current system environment, including the machine and system software type, the keyboard type, where there’s a colour display attached, the AppleTalk driver version, and the VRefNum of the system folder, the AppleTalk node ID and network number for this machine, information on each device connected to the ADB including the device handler ID, the devices ADB address, and the address of the devices handler and storage area, the general device status information and (if possible) device-specific status for the most common device types (the general device information contains the device handle and flags, I/O queue information, event information, and other driver-related details), the name and volume reference number for the current volume, the time information, attributes, directory information and bitmap, volume allocation information, volume and drive information, pointers to various pieces of volume-related information, and details on path and directory caches, for each volume, global script manager variables

and vectors, including the globals changed count, font, script, and internationalisation flags, various script types, and cache information, the script code for the font script the internationalisation script, and for each one add the changed count, font, script, internationalisation, and display flags, resource ID's, and script file information, the device ID, partition, slot number, resource ID, and driver reference number for the default startup device, the slot number and resource ID for the default video device, the default OS type, and the AppleTalk command block and data size and number of sessions.

At least one slow poll is required to accumulate enough randomness for use by cryptlib.

OS/2

A fast poll adds various (fairly constant) pieces of machine information and timestamps, date and time, the thread information block and process information block, and the IRQ0 hi-res timer count.

A slow poll adds information on each window and attached process on the Presentation Manager desktop.

At least ten fast polls or one slow poll are required to accumulate enough randomness for use by cryptlib.

UNIX

A fast poll adds the current time to a reasonably high resolution (usually milliseconds or microseconds), the total user and system time, resident set size, page fault statistics, number of I/O operations, messages sent and received, signals received, number of context switches, and various other system statistics.

A slow poll varies with the Unix flavour being used. If a `/dev/random` driver which continually accumulates random data from the system is available, cryptlib will try to use this. If this is not present it will use a variety of information on disk I/O, network traffic, NFS traffic, packet filter statistics, multiprocessor statistics, process information, users, VM statistics, process statistics, open files, inodes, terminals, vector processors, streams, and loaded code. The exact data collected depends on the system, but generally includes quite detailed operating statistics and information.

At least one slow poll is required to accumulate enough randomness for use by cryptlib.

Windows 3.x

A fast poll adds the number of bytes free in the global heap, the cursor position when the last message was received, a 55ms time for the last message, whether the system queue has any events in it, the number of active tasks, the 55ms time since Windows started, the handle of the window with the mouse capture and input focus, the current mouse cursor and caret position, the largest free memory block, number of lockable pages, number of unlocked pages, number of free and used pages, number of swapped pages, 1ms execution time of the current task and virtual machine, and percentage free and memory segment of the user and GDI heaps.

A slow poll adds the linear address, size in bytes, handle, lock count, owner, object type, and segment type of every object in the global heap, the module name, handle, reference count, executable path, and next module link of every loaded module, the task handle, parent task handle, instance handle, stack segment and offset, stack size, number of pending events, task queue, code segment and instruction pointer, and the name of the module executing the task for every running task.

At least one slow poll or five fast polls are required to accumulate enough randomness for use by cryptlib.

Windows 95

A fast poll adds the handle of the active window, the handle of the window with mouse capture, the handle of the clipboard owner, the handle of the start of the clipboard viwer list, the pseudohandle of the current process, the current process ID, the pseudohandle of the current thread, the current thread ID, the number of milliseconds since Windows started, the handle of the desktop window, the handle of the window with the keyboard focus, whether the system queue has any events in it, the cursor position for the last message in the queue, the millisecond time for the last message in the queue, the handle of the window with the clipboard open, the handle of the process heap, the handle of the processes window station, a bitmap of the types of events in the input queue, the current caret and mouse cursor position, the percentage of memory in use, bytes of physical memory available, bytes of free physical memory, bytes in the paging file, free bytes in the paging file, user bytes of address space, and free user bytes of memory, the thread and process creation time, exit time, time in kernel mode, and time in user mode in 100ns intervals, the minimum and maximum working set size for the current process, the name of the desktop, console window title, position and size for new windows, window flags, and handles for stdin, stdout, and stderr.

A slow poll adds the process ID, heap ID, size in bytes, handle, linear address, type, and lock count of every object on the local heap, the module ID, process ID, global usage count, module usage count, base address, size in bytes, handle, and executable path of every module in the system, the usage count, process ID, heap ID, module ID, number of threads, parent process ID, base priority class, and executable path of every running process, and the thread ID, process ID, base priority, and priority level change of every executing thread in the system.

At least one slow poll or three fast polls are required to accumulate enough randomness for use by cryptlib.

Windows NT

A fast poll adds the handle of the active window, the handle of the window with mouse capture, the handle of the clipboard owner, the handle of the start of the clipboard viwer list, the pseudohandle of the current process, the current process ID, the pseudohandle of the current thread, the current thread ID, the number of milliseconds since Windows started, the handle of the desktop window, the handle of the window with the keyboard focus, whether the system queue has any events in it, the cursor position for the last message in the queue, the millisecond time for the last message in the queue, the handle of the window with the clipboard open, the handle of the process heap, the handle of the processes window station, a bitmap of the types of events in the input queue, the current caret and mouse cursor position, the percentage of memory in use, bytes of physical memory available, bytes of free physical memory, bytes in the paging file, free bytes in the paging file, user bytes of address space, and free user bytes of memory, the thread and process creation time, exit time, time in kernel mode, and time in user mode in 100ns intervals, the minimum and maximum working set size for the current process, the name of the desktop, console window title, position and size for new windows, window flags, and handles for stdin, stdout, and stderr.

A slow poll adds the number of bytes read from and written to disk, the disk read and write time, the number of read and write operations, the depth of the I/O queue, a large number of network statistics such as the number of bytes sent and received, the number of SMB's sent and received, the number of paging, nonpaging, and cached bytes sent and received, the number of failed initial and failed completion operations, the number of reads, random reads, large/small SMB reads, writes, random writes, large and small SMB writes, the number of reads and writes denied, the number of sessions, failed sessions, reconnects, and hung sessions, and various lan manager statistics, and an equally large number of system performance-related statistics such covering virtually every aspect of the operation of the system (the exact details vary from machine to machine, but cryptlib will query every available performance counter and system monitor).

At least one slow poll or three fast polls are required to accumulate enough randomness for use by cryptlib.

Hardware Random Number Generation

On some hardware platforms cryptlib can make use of external hardware random number generators which function alongside the built-in system polling generator. A number of these generators are serial-port based, with data being read from them over a standard serial-port interface. The parameters for the interface can be set using the cryptlib configuration options as explained in "Miscellaneous Topics" on page 146. The two parameters which are used to control serial-port based generators are the serial port which the generator is connected to, set using the CRYPT_OPTION_HARDWARE_SERIALRNG option, and the parameters for the serial port, set using the CRYPT_OPTION_HARDWARE_SERIALRNG_PARAMS option. The serial port is the standard serial port name (for example "COM1"), and the parameters are the standard serial port parameters which specify the interface speed, number of data bits, parity, and stop bits (for example "9600,8,N,1"). For example to configure cryptlib to use a serial-port based generator on COM2 running at 9600bps, you would use:

```
cryptSetAttributeString( CRYPT_UNUSED,
   CRYPT_OPTION_HARDWARE_SERIALRNG, "COM2", 4 );
cryptSetAttributeString(CRYPT_UNUSED,
   CRYPT_OPTION_HARDWARE_SERIALRNG_PARAMS, "9600,8,N,1", 10 );
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_CONFIGCHANGED, FALSE );
```

The next time cryptlib is started, it will try to connect to the hardware generator using these options.

In addition to serial-port based generators, cryptlib supports the use of hardware generators in crypto devices such as PKCS #11 crypto tokens and Fortezza cards, and can use these as additional sources of randomness.

Working with Newer Versions of cryptlib

Your software can automatically support new encryption algorithms and modes as they are added to cryptlib if you check for the range of supported algorithms and modes instead of hard-coding in the values which existed when you wrote the program. In order to support this, cryptlib predefines the values CRYPT_ALGO_FIRST_CONVENTIONAL and CRYPT_ALGO_LAST_CONVENTIONAL for the first and last possible conventional encryption algorithms, CRYPT_ALGO_FIRST_PKC and CRYPT_ALGO_LAST_PKC for the first and last possible public-key encryption algorithms, CRYPT_ALGO_FIRST_HASH and CRYPT_ALGO_LAST_HASH for the first and last possible hash algorithms, and CRYPT_ALGO_FIRST_MAC and CRYPT_ALGO_LAST_MAC for the first and last possible MAC algorithms. By checking each possible algorithm value within this range using **cryptQueryCapability**, your software can automatically incorporate any new algorithms as they are added. For example to scan for all available conventional encryption algorithms you would use:

```
CRYPT_ALGO cryptAlgo;

for( cryptAlgo = CRYPT_ALGO_FIRST_CONVENTIONAL;
     cryptAlgo <= CRYPT_ALGO_LAST_CONVENTIONAL;
     cryptAlgo++ )
   if( cryptStatusOK( cryptQueryCapability( cryptAlgo, CRYPT_UNUSED,
      NULL ) )
      /* Perform action using algorithm */;
```

The action you would perform would typically be building a list of available algorithms and allowing the user to choose the one they preferred. The same can be done for the public-key and hash algorithms.

cryptlib also predefines CRYPT_MODE_FIRST_CONVENTIONAL and CRYPT_MODE_LAST_CONVENTIONAL which cover the range of available

conventional encryption modes. Once you've determined which conventional algorithm to use you can use:

```
CRYPT_MODE cryptMode;

for( cryptMode = CRYPT_MODE_FIRST_CONVENTIONAL;
     cryptMode <= CRYPT_MODE_LAST_CONVENTIONAL;
     cryptMode++ )
   if( cryptStatusOK( cryptQueryCapability( cryptAlgo, cryptMode, NULL
      ) )
      /* Perform action with algorithm */;
```

If your code follows these guidelines, it will automatically handle any new encryption algorithms and modes which are added in newer versions of cryptlib. If you are using the shared library or DLL form of cryptlib, your softwares' encryption capabilities will be automatically upgraded every time cryptlib is upgraded.

Error Handling

Each function in cryptlib performs extensive parameter and error checking (although monitoring of error codes has been omitted in the code samples for readability). In addition, when cryptlib is initialised with **cryptInitEx** each of the built-in encryption algorithms goes through a self-test procedure which checks the implementation using standard test vectors and methods given with the algorithm specification (typically FIPS publications, ANSI or IETF standards, or standard reference implementations). This self-test is used to verify that each encryption algorithm is performing as required.

The macros `cryptStatusError()` and `cryptStatusOK()` can be used to determine whether a return value denotes an error condition, for example:

```
CRYPT_CONTEXT cryptContext;
int status;

status = cryptCreateContext( &cryptContext, CRYPT_ALGO_IDEA );
if( cryptStatusError( status ) )
   /* Perform error processing */
```

The error codes which can be returned are grouped into a number of classes which cover areas such as function parameter errors, resource errors, and data access errors.

The first group contains a single member, the "no error" value:

Error code	Description
CRYPT_OK	No error.

The next group contains parameter error codes which identify erroneous parameters passed to cryptlib functions:

Error code	Description
CRYPT_ERROR_-PARAM1… CRYPT_ERROR_-PARAM7	There is a problem with a parameter passed to a cryptlib function. The exact code depends on the parameter in error.

The next group contains resource-related errors such as a certain resource not being available or initialised:

Error code	Description
CRYPT_ERROR_-FAILED	The operation, for example a public-key encryption or decryption, failed.
CRYPT_ERROR_-INITED	The object or portion of an object which you have tried to initialise has already been initialised previously.
CRYPT_ERROR_-MEMORY	There is not enough memory available to perform this operation.
CRYPT_NOSECURE	cryptlib cannot perform an operation at the requested security level (for example allocated pages can't be locked into memory to prevent them from being swapped to disk, or an LDAP connection can't be established using SSL).
CRYPT_ERROR_-NOTINITED	The object or portion of an object which you have tried to use hasn't been initialised yet, or a resource which is required isn't available.
CRYPT_ERROR_-RANDOM	Not enough random data is available for cryptlib to perform the requested operation.

The next group contains cryptlib security violations such as an attempt to use the wrong object for an operation or to use an object for which you don't have access permission:

Error code	Description
CRYPT_ERROR_-BUSY	The object you are trying to use (usually a keyset object or encryption context) is currently in use by an asynchronous operation such as a key generation or key database lookup operation.
CRYPT_ERROR_-COMPLETE	An operation which consists of multiple steps (such as a message hash) is complete and cannot be continued.
CRYPT_ERROR_-INCOMPLETE	An operation which consists of multiple steps (such as a message hash) is still in progress and requires further steps before it can be regarded as having completed.
CRYPT_ERROR_-INVALID	The public/private key context or certificate object is invalid for this type of operation. You can obtain the exact nature of the problem by reading the CRYPT_ATTRIBUTE_ERRORLOCUS and CRYPT_ATTRIBUTE_ERRORTYPE attributes..
CRYPT_ERROR_-NOTAVAIL	The requested operation is not available for this object (for example an attempt to load an encryption key into a hash context, or to decrypt a Diffie-Hellman shared integer with an RSA key).
CRYPT_ERROR_-PERMISSION	You don't have the permission to perform this type of operation (for example an encrypt-only key being used for a decrypt operation, or an attempt to modify a read-only data object).
CRYPT_ERROR_-SIGNALLED	An external event such as a signal from a hardware device caused a change in the state of the object. For example if a smart card is removed from a card reader, all the objects which had been loaded or derived from the data on the smart card would return CRYPT_SIGNALLED if you tried to use them. Once an object has entered this state, the only available option is to destroy it, typically using **cryptDestroyObject**.
CRYPT_ERROR_-WRONGKEY	The key being used to decrypt a piece of data is incorrect.

The next group contains errors related to the higher-level encryption functions such as the key export/import and signature generation/checking functions:

Error code	Description
CRYPT_ERROR_-BADDATA	The data item (typically encrypted or signed data, or a key certificate) was corrupt, or not all of the data was present, and it can't be processed.
CRYPT_ERROR_-OVERFLOW	There is too much data for this function to work with. For a public-key encryption or signature function this means there is too much data for this public/private key to encrypt/sign. You should either use a larger public/private key (in general a 1024-bit or larger key should be sufficient for most purposes) or less data (for example by reducing the key size in the encryption context passed to **cryptExportKey**). For a key certificate function this means the amount of

Error code	Description
	data you have supplied is more than what is allowed for the field you are trying to store it in. For an enveloping function, you need to call **cryptPopData** before you can add any more data to the envelope.
CRYPT_ERROR_-SIGNATURE	The signature or integrity check value didn't match the data.
CRYPT_ERROR_-UNDERFLOW	There is too little data in the envelope for cryptlib to process (for example only a portion of a data item may be present, which isn't enough for cryptlib to work with).

The next group contains data/information access errors, usually arising from keyset, certificate, or device container object accesses:

Error code	Description
CRYPT_ERROR_-DUPLICATE	The given item is already present in the container object.
CRYPT_ERROR_-NOTFOUND	The requested item (for example a key being read from a key database or a certificate component being extracted from a certificate) isn't present in the container object.
CRYPT_ERROR_-OPEN	The container object (for example a keyset or configuration database) could not be opened, either because it was not found or because the open operation failed.
CRYPT_ERROR_-READ	The requested item could not be read from the container object.
CRYPT_ERROR_-WRITE	The item couldn't be written to the container object or the data object couldn't be updated (for example a key couldn't be written to a keyset, or couldn't be deleted from a keyset).

The next group contains errors related to data enveloping:

Error code	Description
CRYPT_ENVELOPE_RESOURCE	A resource such as an encryption key or password needs to be added to the envelope before cryptlib can continue processing the data in it.

Extended Error Reporting

Sometimes the standard cryptlib error codes aren't capable of returning full details on the large variety of possible error conditions which can be encountered. This is particularly true for complex objects such as certificates or ones which are tied to other software or hardware which is outside cryptlib's control, for example database or directory keyset objects or crypto devices. For example if there is a problem with a certificate, cryptlib will return a generic CRYPT_ERROR_INVALID code. In order to obtain more information on the problem you can read the CRYPT_ATTRIBUTE_ERRORLOCUS attribute to obtain the locus of the error (the certificate component which caused the problem) and the CRYPT_ATTRIBUTE_ERRORTYPE attribute to identify the type of problem which occurred. These error attributes are present in all objects and can often provide more extensive information on why an operation with the object failed, for example if a function returns CRYPT_ERROR_NOTINITED then the CRYPT_ATTRIBUTE_ERRORLOCUS attribute will tell you which object attribute hasn't been initialised.

The error types are:

Error Type	Description
CRYPT_ERRTYPE_- ATTR_ABSENT	The attribute is required but not present in the object.
CRYPT_ERRTYPE_- ATTR_PRESENT	The attribute is already present in the object, or present but not permitted for this type of object.
CRYPT_ERRTYPE_- ATTR_SIZE	The attribute is smaller than the minimum allowable or larger than the maximum allowable size.
CRYPT_ERRORTYPE_- ATTR_VALUE	The attribute is set to an invalid value.
CRYPT_ERRTYPE_- CONSTRAINT	The attribute violates some constraint for the object, or represents a constraint which is being violated, for example a validity period or key usage or certificate policy constraint.
CRYPT_ERRTYPE_- ISSUER_CONSTRAINT	The attribute violates a constraint set by an issuer certificate, for example the issuer may set a name constraint which is violated by the certificate object's subjectName or subject altName.

For example to obtain more information on why an attempt to sign a certificate failed you would use:

```
CRYPT_CERTINFO_TYPE errorLocus;
CRYPT_ERRTYPE_TYPE errorType;

status = cryptSignCert( cryptCert, cryptCAKey );
if( cryptStatusError( status ) )
   {
   cryptGetAttribute( cryptCert, CRYPT_ATTRIBUTE_ERRORLOCUS,
      &errorLocus );
   cryptGetAttribute( cryptCert, CRYPT_ATTRIBUTE_ERRORTYPE, &errorType
      );
   }
```

In addition to the error type and locus, keyset objects and objects tied to devices often provide internal error codes for the keyset or device. The device- or keyset-specific error code and message is accessible as the CRYPT_ATTRIBUTE_INT_-ERRORCODE and CRYPT_ATTRIBUTE_INT_ERRORMESSAGE attributes. For example to obtain more information on why an attempt to read a key from an SQL Server database failed you would use:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_HANDLE publicKey
int status;

status = cryptGetPublicKey( &cryptKeyset, &publicKey,
   CRYPT_KEYID_NAME, "John Doe" );
if( cryptStatusError( status ) )
   {
   int errorCode, errorStringLengh;
   char *errorString;

   errorString = malloc( ... );
   cryptGetAttribute( cryptKeyset, CRYPT_ATTRIBUTE_INT_ERRORCODE,
      &errorCode );
   cryptGetAttributeString( cryptKeyset,
      CRYPT_ATTRIBUTE_INT_ERRORMESSAGE, errorString,
      &errorStringLength );
   }
```

Note that the error information being returned is passed through by cryptlib from the underlying software or hardware, and will be specific to the implementation. For

example if the software which underlies a keyset database is SQL Server then the data returned will be the SQL Server error code and message.

Algorithms and Modes

Blowfish

Blowfish is a 64-bit block cipher with a 448-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_BLOWFISH.

CAST-128

CAST-128 is a 64-bit block cipher with a 128-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_CAST.

DES

DES is a 64-bit block cipher with a 56-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_DES.

Although cryptlib uses 64-bit DES keys, only 56 bits of the key are actually used. The least significant bit in each byte is used as a parity bit (cryptlib will set the correct parity values for you, so you don't have to worry about this). You can treat the algorithm as having a 64-bit key, but bear in mind that only the high 7 bits of each byte are actually used as keying material.

Loading a key will return a CRYPT_ERROR_PARAM2 error if the key is a weak key. **cryptExportKey** will export the correct parity-adjusted version of the key.

Triple DES

Triple DES is a 64-bit block cipher with a 112/168-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_3DES.

Although cryptlib uses 128, or 192-bit DES keys (depending on whether two- or three-key triple DES is being used), only 112 or 168 bits of the key are actually used. The least significant bit in each byte is used as a parity bit (cryptlib will set the correct parity values for you, so you don't have to worry about this). You can treat the algorithm as having a 128 or 192-bit key, but bear in mind that only the high 7 bits of each byte are actually used as keying material.

Loading a key will return a CRYPT_ERROR_PARAM2 error if the key is a weak key. **cryptExportKey** will export the correct parity-adjusted version of the key.

Diffie-Hellman

Diffie-Hellman is a key exchange algorithm with a key size of up to 4096 bits and has the cryptlib algorithm identifier CRYPT_ALGO_DH.

cryptEncrypt and **cryptDecrypt** will return a bad parameter code if the length parameter is not CRYPT_USE_DEFAULT (CRYPT_ERROR_PARAM3 in this case).

Diffie-Hellman was formerly covered by a patent in the US, this has now expired.

DSA

DSA is a digital signature algorithm with a key size of up to 1024 bits and has the cryptlib algorithm identifier CRYPT_ALGO_DSA.

cryptEncrypt and **cryptDecrypt** will return a bad parameter code if the length parameter is not CRYPT_USE_DEFAULT (CRYPT_ERROR_PARAM3 in this case).

DSA is covered by US patent 5,231,668, with the patent held by the US government. This patent has been made available royalty-free to all users worldwide. The

Department of Commerce is not aware of any other patents which would be infringed by the DSA. US patent 4,995,082, "Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system" ("the Schnorr patent") relates to the DSA algorithm but only applies to a very restricted set of smart-card based applications and does not affect the DSA implementation in cryptlib.

ElGamal

ElGamal is a public-key encryption/digital signature algorithm with a key size of up to 4096 bits and has the cryptlib algorithm identifier CRYPT_ALGO_ELGAMAL.

cryptEncrypt and **cryptDecrypt** will return a bad parameter code if the length parameter is not CRYPT_USE_DEFAULT (CRYPT_ERROR_PARAM3 in this case).

ElGamal was formerly covered by a patent in the US; this has now expired.

HMAC-MD5
HMAC-SHA1
HMAC-RIPEMD-160

HMAC-MD5, HMAC-SHA1, and HMAC-RIPEMD-160 are MAC algorithms with a key size of up to 1024 bits and have the cryptlib algorithm identifiers CRYPT_ALGO_HMAC_MD5, CRYPT_ALGO_HMAC_SHA, and CRYPT_ALGO_HMAC_RIPEMD160.

IDEA

IDEA is a 64-bit block cipher with a 128-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_IDEA.

IDEA is covered by patents in Austria, France, Germany, Italy, Japan, the Netherlands, Spain, Sweden, Switzerland, the UK, and the US. A statement from the patent owners is included below.

> The IDEA algorithm is patented by Ascom Systec Ltd. of CH-5506 Maegenwil, Switzerland, who allow it to be used on a royalty-free basis for certain non-profit applications. Commercial users must obtain a license from the company in order to use IDEA. IDEA may be used on a royalty-free basis under the following conditions:
>
> **Free use for private purposes**
>
> The free use of software containing the algorithm is strictly limited to non revenue generating data transfer between private individuals, ie not serving commercial purposes. Requests by freeware developers to obtain a royalty-free license to spread an application program containing the algorithm for non-commercial purposes must be directed to Ascom.
>
> **Special offer for shareware developers**
>
> There is a special waiver for shareware developers. Such waiver eliminates the upfront fees as well as royalties for the first US$10,000 gross sales of a product containing the algorithm if and only if:
>
> 1. The product is being sold for a minimum of US$10 and a maximum of US$50.
> 2. The source code for the shareware is available to the public.
>
> **Special conditions for research projects**
>
> The use of the algorithm in research projects is free provided that it serves the purpose of such project and within the project duration. Any use of the algorithm

after the termination of a project including activities resulting from a project and for purposes not directly related to the project requires a license.

Ascom Tech requires the following notice to be included for freeware products:

This software product contains the IDEA algorithm as described and claimed in US patent 5,214,703, EPO patent 0482154 (covering Austria, France, Germany, Italy, the Netherlands, Spain, Sweden, Switzerland, and the UK), and Japanese patent application 508119/1991, "Device for the conversion of a digital block and use of same" (hereinafter referred to as "the algorithm"). Any use of the algorithm for commercial purposes is thus subject to a license from Ascom Systec Ltd. of CH-5506 Maegenwil (Switzerland), being the patentee and sole owner of all rights, including the trademark IDEA.

Commercial purposes shall mean any revenue generating purpose including but not limited to:

i) Using the algorithm for company internal purposes (subject to a site license).

ii) Incorporating the algorithm into any software and distributing such software and/or providing services relating thereto to others (subject to a product license).

iii) Using a product containing the algorithm not covered by an IDEA license (subject to an end user license).

All such end user license agreements are available exclusively from Ascom Systec Ltd and may be requested via the WWW at `http://www.ascom.ch/systec` or by email to `idea@ascom.ch`.

Use other than for commercial purposes is strictly limited to non-revenue generating data transfer between private individuals. The use by government agencies, non-profit organizations, etc is considered as use for commercial purposes but may be subject to special conditions. Any misuse will be prosecuted.

MD2

MD2 is a message digest/hash algorithm with a digest/hash size of 128 bits and has the cryptlib algorithm identifier CRYPT_ALGO_MD2.

MD4

MD5 is a message digest/hash algorithm with a digest/hash size of 128 bits and has the cryptlib algorithm identifier CRYPT_ALGO_MD4.

MD5

MD5 is a message digest/hash algorithm with a digest/hash size of 128 bits and has the cryptlib algorithm identifier CRYPT_ALGO_MD5.

MDC2

MDC2 is a DES-based MAC algorithm with a key size of 56 bits and a MAC size of 128 bits and has the cryptlib algorithm identifier CRYPT_ALGO_MDC2.

MDC2 is patented by IBM can cannot be used in the US without a license.

RC2

RC2 is a 64-bit block cipher with a 1024-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_RC2.

RC2 is trademarked in the US, it may be necessary to refer to it as "an algorithm compatible with RC2" in products which use RC2 and are distributed in the US.

RC4

RC4 is an 8-bit stream cipher with a key of up to 1024 bits and has the cryptlib algorithm identifier CRYPT_ALGO_RC4.

RC4 is trademarked in the US, it may be necessary to refer to it as "an algorithm compatible with RC4" in products which use RC4 and are distributed in the US. A number of products refer to it as ArcFour.

RC5

RC5 is a 64-bit block cipher with an 832-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_RC5.

RC5 is covered by US patent 5,724,428, "Block Encryption Algorithm with Data-Dependent Rotation", issued 3 March 1998. The patent is held by RSA Data Security Inc. 100 Marine Parkway, Redwoord City, California 94065, ph.+1 415 595-8782, fax +1 415 595-1873, and the algorithm cannot be used commercially in the US without a license.

RIPEMD-160

RIPEMD-160 is a message digest/hash algorithm with a digest/hash size of 160 bits and has the cryptlib algorithm identifier CRYPT_ALGO_RIPEMD160.

RSA

RSA is a public-key encryption/digital signature algorithm with a key size of up to 4096 bits and has the cryptlib algorithm identifier CRYPT_ALGO_RSA.

cryptEncrypt and **cryptDecrypt** will return a bad parameter code if the length parameter is not CRYPT_USE_DEFAULT (CRYPT_ERROR_PARAM3 in this case).

RSA is patented in the US, with the patent held by RSA Data Security Inc. 100 Marine Parkway, Redwoord City, California 94065, ph.+1 415 595-8782, fax +1 415 595-1873, and cannot be used commercially in the US without a license. RSA licenses can most easily be obtained by waiting another two years until the patent expires. The patent licensing requirements for RSA seem to change quite a bit and are difficult to obtain, a small amount of relevant information can be found at `http://www.rsa.com`.

SAFER
SAFER-SK

Safer and Safer-SK are 64-bit block ciphers with a 64 or 128-bit key and have the cryptlib algorithm identifier CRYPT_ALGO_SAFER.

SHA

SHA is a message digest/hash algorithm with a digest/hash size of 160 bits and has the cryptlib algorithm identifier CRYPT_ALGO_SHA.

Skipjack

Skipjack is a 64-bit block cipher with an 80-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_SKIPJACK.

Data Types and Constants

This section describes the data types and constants used by cryptlib.

CRYPT_ALGO

The CRYPT_ALGO is used to identify a particular encryption algorithm. More information on the individual algorithm types can be found in "Algorithms and Modes" on page 158.

Value	Description
CRYPT_ALGO_BLOWFISH	Blowfish
CRYPT_ALGO_CAST	CAST-128
CRYPT_ALGO_DES	DES
CRYPT_ALGO_3DES	Triple DES
CRYPT_ALGO_IDEA	IDEA
CRYPT_ALGO_RC2	RC2
CRYPT_ALGO_RC4	RC4
CRYPT_ALGO_RC5	RC5
CRYPT_ALGO_SAFER	Safer/Safer-SK
CRYPT_ALGO_SKIPJACK	Skipjack
CRYPT_ALGO_DH	Diffie-Hellman
CRYPT_ALGO_DSA	DSA
CRYPT_ALGO_ELGAMAL	ElGamal
CRYPT_ALGO_RSA	RSA
CRYPT_ALGO_MD2	MD2
CRYPT_ALGO_MD4	MD4
CRYPT_ALGO_MD5	MD5
CRYPT_ALGO_MDC2	MDC-2
CRYPT_ALGO_RIPEMD160	RIPE-MD 160
CRYPT_ALGO_SHA	SHA/SHA-1
CRYPT_ALGO_HMAC_MD5	HMAC-MD5
CRYPT_ALGO_HMAC_RIPEMD160	HMAC-RIPEMD-160
CRYPT_ALGO_HMAC_SHA	HMAC-SHA
CRYPT_ALGO_VENDOR1 CRYPT_ALGO_VENDOR2 CRYPT_ALGO_VENDOR3	Optional vendor-defined algorithms.
CRYPT_ALGO_FIRST_- CONVENTIONAL CRYPT_ALGO_LAST_- CONVENTIONAL	First and last possible conventional encryption algorithm type.
CRYPT_ALGO_FIRST_PKC CRYPT_ALGO_LAST_PKC	First and last possible public-key algorithm type.

Value	Description
CRYPT_ALGO_FIRST_HASH CRYPT_ALGO_LAST_HASH	First and last possible hash algorithm type.
CRYPT_ALGO_FIRST_MAC CRYPT_ALGO_LAST_MAC	First and last possible MAC algorithm type.

CRYPT_ATTRIBUTE_TYPE

The CRYPT_ATTRIBUTE_TYPE is used to identify the attribute associated with a cryptlib object. Object attributes are introduced in "Working with Object Attributes" on page 17 and are used extensively throughout this manual.

CRYPT_CERTFORMAT_TYPE

The CRYPT_CERTFORMAT_TYPE is used to specify the format for exported certificate objects. More information on certificates and exporting certificate objects is given in "Certificate Management" on page 76.

Value	Description
CRYPT_CERTFORMAT_- CERTCHAIN	Certificate object encoded as a PKCS #7 certificate chain. This encoding is only possible for objects which are certificates or certificate chains.
CRYPT_CERTFORMAT_- CERTIFICATE	Certificate object encoded according to the ASN.1 distinguished encoding rules (DER).
CRYPT_CERTFORMAT_SMIME_- CERTIFICATE	S/MIME data format. The certificate object is encoded as for the basic CRYPT_CERTFORMAT_*type* format, and an extra layer of base64 encoding with MIME wrapping is added. This format is required by some web browsers and applications.
CRYPT_CERTFORMAT_TEXT_- CERTCHAIN CRYPT_CERTFORMAT_TEXT_- CERTIFICATE	Base64-encoded text format. The certificate object is encoded as for the basic CRYPT_CERTFORMAT_*type* format, and an extra layer of base64 encoding with BEGIN/END CERTIFICATE tags is added. This format is required by some web browsers and applications.

CRYPT_CERTTYPE_TYPE

The CRYPT_CERTTYPE_TYPE is used to specify the type of a certificate object, including certificate-like objects like PKCS #7/CMS signing attributes and OCSP messages. More information on certificates and certificate objects is given in "Certificate Management" on page 76.

Value	Description
CRYPT_CERTTYPE_- ATTRIBUTE_CERT	Attribute certificate.
CRYPT_CERTTYPE_CERTCHAIN	PKCS #7 certificate chain.
CRYPT_CERTTYPE_- CERTIFICATE	Certificate.
CRYPT_CERTTYPE_-	PKCS #10 certification request.

Value	Description
CERTREQUEST	
CRYPT_CERTTYPE_CMS_ATTRIBUTES	PKCS #7/CMS attributes.
CRYPT_CERTTYPE_CRL	CRL
CRYPT_CERTTYPE_OCSP_REQUEST	OCSP request and response (not currently available).
CRYPT_CERTTYPE_OCSP_RESPONSE	

CRYPT_DEVICE_TYPE

The CRYPT_DEVICE_TYPE is used to specify encryption hardware or an encryption device such as a PCMCIA or smart card. More information on encryption devices is given in “Encryption Devices and Modules” on page 140.

Value	Description
CRYPT_DEVICE_FORTEZZA	Fortezza card.
CRYPT_DEVICE_PKCS11	PKCS #11 crypto token.

CRYPT_FORMAT_TYPE

The CRYPT_FORMAT_TYPE is used to identify a data format type for exported keys, signatures, and encryption envelopes. Of the formats supported by cryptlib, the cryptlib native format is the most flexible and is the recommended format unless you require compatibility with a specific security standard. More information on the different formats is given in “Enveloping Concepts” on page 22, “Exchanging Keys” on page 66, and “Signing Data” on page 72.

Value	Description
CRYPT_FORMAT_CRYPTLIB	cryptlib native format.
CRYPT_FORMAT_PGP	PGP format (not currently available).
CRYPT_FORMAT_CMS CRYPT_FORMAT_PKCS7	PKCS #7/CMS format.
CRYPT_FORMAT_SMIME	As CMS but with S/MIME MSG-specific behaviour.

CRYPT_KEYID_TYPE

The CRYPT_KEYID_TYPE is used to identify the type of key identifier which is being passed to **cryptGetPublicKey** or **cryptGetPrivateKey**. More information on using these functions to read keys from keysets is given in “Key Databases” on page 40

Value	Description
CRYPT_KEYID_NAME	The name of the key owner.
CRYPT_KEYID_EMAIL	The email address of the key owner.

CRYPT_KEYOPT

The CRYPT_KEYOPT is used to contain keyset option flags passed to **cryptOpenKeyset** or **cryptOpenKeysetEx**. The keyset options may be used to optimise access to keysets by enabling cryptlib to perform enhanced transaction management in cases where, for example, read-only access to a database is desired.

Because this can improve performance when accessing the keyset, you should always specify whether you will be using the keyset in a restricted access mode when you call **cryptOpenKeyset** or **cryptOpenKeysetEx**.

More information on using these options when opening a connection to a keyset is given in "Key Databases" on page 40

Value	Description
CRYPT_KEYOPT_CREATE	Create a new keyset. This option is only valid for writeable keyset types, which includes keysets implemented as relational databases, cryptlib private key files, and smart cards.
CRYPT_KEYOPT_NONE	No special access options.
CRYPT_KEYOPT_READONLY	Read-only keyset access. This option is turned on by default for non-writeable keyset types, which includes X.509/SET flat files, PGP public and private keyrings, and other keyset types which have read-only restrictions enforced by the operating system or user access rights.

CRYPT_KEYSET_TYPE

The CRYPT_KEYSET_TYPE is used to identify a keyset type (or, more specifically, the format and access method used to access a keyset) when used with **cryptOpenKeyset** or **cryptOpenKeysetEx**. Some keyset types may be unavailable on some systems (for example CRYPT_KEYSET_ODBC is limited to Windows machines; CRYPT_KEYSET_POSTGRES is mostly limited to Unix machines). More information on keysets is given in "Key Databases" on page 40.

Value	Description
CRYPT_KEYSET_FILE	A flat-file keyset, either an individual X.509/SET key stored in a file, or a PGP public or private keyring or a cryptlib private key file.
CRYPT_KEYSET_LDAP	LDAP directory service.
CRYPT_KEYSET_SMARTCARD	Smart card key carrier.
CRYPT_KEYSET_MSQL	mSQL RDBMS.
CRYPT_KEYSET_MYSQL	MySQL RDBMS.
CRYPT_KEYSET_ODBC	Generic ODBC interface.
CRYPT_KEYSET_ORACLE	Oracle RDBMS.
CRYPT_KEYSET_POSTGRES	Postgres RDBMS.

CRYPT_MODE

The CRYPT_MODE is used to identify a particular encryption mode. More information on the individual modes can be found in "Algorithms and Modes" on page 158.

Value	Description
CRYPT_MODE_NONE	No encryption (hashes and MAC's).
CRYPT_MODE_STREAM	Stream cipher

Value	Description
CRYPT_MODE_ECB	ECB
CRYPT_MODE_CBC	CBC
CRYPT_MODE_CFB	CFB
CRYPT_MODE_OFB	OFB
CRYPT_MODE_PKC	Public-key encryption/digital signature.
CRYPT_MODE_FIRST_-CONVENTIONAL CRYPT_MODE_LAST_-CONVENTIONAL	First and last possible conventional encryption mode.

CRYPT_OBJECT_TYPE

The CRYPT_OBJECT_TYPE is used to identify the type of an exported key or signature object which has been created with **cryptExportKey** or **cryptCreateSignature**. More information on working with these objects is given in “Exchanging Keys” on page 66, and “Signing Data” on page 72.

Value	Description
CRYPT_OBJECT_ENCRYPTED_KEY	Conventionally exported key object..
CRYPT_OBJECT_KEYAGREEMENT	Key agreement object.
CRYPT_OBJECT_PKCENCRYPTED_-KEY	Public-key exported key object.
CRYPT_OBJECT_SIGNATURE	Signature object.

Data Size Constants

The following values define various maximum lengths for data objects which are used in cryptlib. These can be used for allocating memory to contain the objects, or as a check to ensure that an object isn’t larger than the maximum size allowed by cryptlib.

Constant	Description
CRYPT_MAX_HASHSIZE	Maximum hash size in bytes.
CRYPT_MAX_IVSIZE	Maximum initialisation vector size in bytes.
CRYPT_MAX_KEYSIZE	Maximum conventional-encryption key size in bytes.
CRYPT_MAX_PKCSIZE	Maximum public-key component size in bytes. This value specifies the maximum size of individual components, since public/private keys are usually composed of a number of components the overall size is larger than this.
CRYPT_MAX_TEXTSIZE	Maximum size of a text string (eg a public or private key owner name) in characters. This defines the string size in characters rather than bytes, so a Unicode string of size CRYPT_MAX_TEXTSIZE could be twice as long as an ASCII string of size CRYPT_MAX_TEXTSIZE. This value

Constant	Description
	does not include the terminating null character in C strings.

Miscellaneous Constants

The following values are used for various purposes by cryptlib, for example to specify that default parameter values are to be used, that the given parameter is unused and can be ignored, or that a special action should be taken in response to seeing this parameter.

Constant	Description
CRYPT_COMPONENTS_- BIGENDIAN CRYPT_COMPONENTS_- LITTLENDIAN	The endianness of the external components of a public/private key when passed to `cryptInitComponents()`/ `cryptSetComponent()`.
CRYPT_KEYTYPE_PRIVATE CRYPT_KEYTYPE_PUBLIC	Whether the key being passed to `cryptInitComponents()`/ `cryptSetComponent()` is a public or private key.
CRYPT_RANDOM_FASTPOLL CRYPT_RANDOM_SLOWPOLL	The type of polling to perform to update the internal random data pool.
CRYPT_UNUSED	A value indicating that this parameter is unused and can be ignored.
CRYPT_USE_DEFAULT	A value indicating that the default setting for this parameter should be used.

Data Structures

This chapter describes the data structures used by cryptlib.

CRYPT_OBJECT_INFO Structure

The CRYPT_OBJECT_INFO structure is used with **cryptQueryObject** to return information about an data object created with **cryptExportKey** or **cryptCreateSignature**. Some of the fields are only valid for certain algorithm and mode combinations, or for some types of data objects. If they don't apply to the given algorithm and mode or context, they will be set to CRYPT_ERROR, null, or filled with zeroes as appropriate.

Field	Description
CRYPT_OBJECT_TYPE objectType	Data object type.
CRYPT_ALGO cryptAlgo	Encryption/signature algorithm.
CRYPT_MODE cryptMode	Encryption/signature mode.
CRYPT_ALGO hashAlgo	The hash algorithm used to hash the data if the data object is a signature object.
unsigned char salt[CRYPT_MAX_HASHSIZE]	The salt used to derive the export/import key if the object is a conventionally encrypted key object
int saltSize	

CRYPT_PKCINFO Structures

The CRYPT_PKCINFO structures are used to load public and private keys (which contain multiple key components) into encryption contexts by setting them as the CRYPT_CTXINFO_KEY attribute. All fields are multiprecision integer values which are set using the `cryptSetComponent()` macro.

The CRYPT_PKCINFO_DLP structure is used to load keys for algorithms based on the discrete logarithm problem, which includes keys for Diffie-Hellman, DSA, and Elgamal. The structure contains the following fields:

Field	Description
p	Prime modulus.
q	Prime divisor.
g	Element of order q mod p
x	Private random integer.
y	Public random integer, g^x mod p.

The CRYPT_PKCINFO_RSA structure is used to load Rivest-Shamir-Adelman public-key encryption keys and contains the following fields:

Field	Description
n	Modulus.
e	Public exponent.
D	Private exponent.
P	Prime factor 1.
Q	Prime factor 2.
U	CRT coefficient q^{-1} mod p.
e1	Private exponent 1 (PKCS #1), d mod (p-1).

Field	Description
e2	Private exponent 2 (PKCS #1) , d mod (q-1).

The e1 and e2 components of CRYPT_PKCINFO_RSA may not be present in some keys. cryptlib will make use of them if they are present, but can also work without them. The loading of private keys is slightly slower if these values aren't present since cryptlib needs to generate them itself.

CRYPT_QUERY_INFO Structure

The CRYPT_QUERY_INFO structure is used with **cryptQueryCapability** to return information about an encryption algorithm or an encryption context or key-related certificate object (for example a public-key certificate or certification request). Some of the fields are only valid for certain algorithm types, or for some types of encryption contexts. If they don't apply to the given algorithm or context, they will be set to CRYPT_ERROR, null, or filled with zeroes as appropriate.

Field	Description
char algoName[CRYPT_MAX_TEXTSIZE]	Algorithm name.
int blockSize	Algorithm block size in bytes.
int minKeySize int keySize int maxKeySize	The minimum, recommended, and maximum key size in bytes (if the algorithm uses a key).

Function Reference

cryptAddCertExtension

The **cryptAddCertExtension** function is used to add a generic blob-type certificate extension to a certificate object.

int cryptAddCertExtension(const CRYPT_CERTIFICATE *certificate***, const char** ******oid***, const int** *criticalFlag***, const void** ******extension***, const int** *extensionLength* **);**

Parameters

certificate
The certificate object to which to add the extension.

oid
The object identifier value for the extension being added, specified as a sequence of integers.

criticalFlag
The critical flag for the extension being added.

extension
The address of the extension data.

extensionLength
The length in bytes of the extension data.

Remarks

cryptlib directly supports extensions from X.509, PKIX, SET, and various vendors itself, so you shouldn't use this function for anything other than unknown, proprietary extensions.

See also

cryptGetCertExtension, **cryptDeleteCertExtension**.

cryptAddPrivateKey

The **cryptAddPrivateKey** function is used to add a users private key to a keyset.

int cryptGetPrivateKey(const CRYPT_KEYSET *keyset***, const CRYPT_CONTEXT** *cryptContext***, const char** ******password* **);**

Parameters

keyset
The keyset object to which to write the key.

cryptContext
The private key context to write to the keyset.

password
The password used to encrypt the private key, or null if no encryption is required.

Remarks

The use of a password to encrypt the private key is strongly recommended, even for supposedly secure keyset types such as smart cards, since calling the function without a password would leave the private key in an unprotected state in the keyset.

See also

cryptAddPublicKey, **cryptDeleteKey**, **cryptGetPrivateKey**, **cryptGetPublicKey**.

cryptAddPublicKey

The **cryptAddPublicKey** function is used to add a users public key certificate to a keyset.

int cryptAddPublicKey(const CRYPT_KEYSET *keyset***, CRYPT_CERTIFICATE** *certificate* **);**

Parameters

keyset
The keyset object from which to read the key.

certificate
The certificate to add to the keyset.

Remarks This function requires a key certificate object rather than an encryption context, since the certificate contains additional identification information which is used when the certificate is written to the keyset.

See also **cryptAddPrivateKey**, **cryptDeleteKey**, **cryptGetPrivateKey**, **cryptGetPublicKey**.

cryptAddRandom

The **cryptAddRandom** function is used to add random data to the internal random data pool maintained by cryptlib, or to tell cryptlib to poll the system for random information. The random data pool is used to generate session keys and public/private keys, and by several of the high-level cryptlib functions.

int cryptAddRandom(const void **randomData***, const int** *randomDataLength* **);**

Parameters *randomData*
The address of the random data to be added, or null if cryptlib should poll the system for random information.

randomDataLength
The length of the random data being added, or CRYPT_RANDOM_SLOWPOLL to perform an in-depth, slow poll or CRYPT_RANDOM_FASTPOLL to perform a less thorough but faster poll for random information.

cryptAsyncCancel

The **cryptAsyncCancel** function is used to cancel an asynchronous operation on an object.

int cryptAsyncCancel(const CRYPT_HANDLE *cryptObject* **);**

Parameters *cryptObject*
The object on which an asynchronous operation is to be cancelled.

Remarks Because of the asynchronous nature of the operation being performed the cancel may not take effect immediately. In the worst case it may take a second or two for the cancel command to be processed by the object.

See also **cryptAsyncQuery**, **cryptGenerateKeyAsync**, **cryptGenerateKeyAsyncEx**.

cryptAsyncQuery

The **cryptAsyncQuery** function is used to obtain the status of an asynchronous operation on an object.

int cryptAsyncQuery(const CRYPT_HANDLE *cryptObject* **);**

Parameters *cryptObject*
The object to be queried.

Remarks **cryptAsyncQuery** will return CRYPT_ERROR_BUSY if an asynchronous operation is in progress and the object is unavailable for use until the operation completes.

See also **cryptAsyncCancel**, **cryptGenerateKeyAsync**, **cryptGenerateKeyAsyncEx**.

cryptCheckCert

The **cryptCheckCert** function is used to check the signature on a certificate object, or to verify a certificate object against a CRL or a keyset containing a CRL.

int cryptCheckCert(const CRYPT_CERTIFICATE *certificate***, const CRYPT_HANDLE** *sigCheckKey* **);**

Parameters

certificate
The certificate container object which contains the certificate item to check.

sigCheckKey
A public-key context or key certificate object containing the public key used to verify the signature, or alternatively CRYPT_UNUSED if the certificate item is self-signed. If the certificate is to be verified against a CRL, this should be a certificate object or keyset containing the CRL.

Remarks

If the signature data is invalid, the function will return CRYPT_ERROR_-BADDATA. If the signature itself is invalid, the function will return CRYPT_-ERROR_BADSIG. If the certificate is being checked against a CRL and has been revoked, the function will return CRYPT_ERROR_INVALID.

See also

cryptSignCert.

cryptCheckSignature

The **cryptCheckSignature** function is used to check the digital signature on a piece of data. Due to various speed and security requirements, what is actually checked is the signature on the hash of the data rather than the signature on the data itself.

int cryptCheckSignature(void **signature***,* **const CRYPT_HANDLE** *sigCheckKey***, const CRYPT_CONTEXT** *hashContext* **);**

Parameters

signature
The address of a buffer which contains the signature.

sigCheckKey
A public-key context or key certificate object containing the public key used to verify the signature.

hashContext
A hash context containing the hash of the data.

Remarks

If the signature data is invalid, the function will return CRYPT_ERROR_-BADDATA. If the signature itself is invalid, the function will return CRYPT_-ERROR_BADSIG.

See also

cryptCheckSignatureEx, **cryptCreateSignature**, **cryptCreateSignatureEx**, **cryptQueryObject**.

cryptCheckSignatureEx

The **cryptCheckSignatureEx** function is used to check the digital signature on a piece of data with extended control over the signature information. Due to various speed and security requirements, what is actually checked is the signature on the hash of the data rather than the signature on the data itself.

int cryptCheckSignatureEx(void **signature***,* **const CRYPT_HANDLE** *sigCheckKey***, const CRYPT_CONTEXT** *hashContext***, CRYPT_HANDLE** **extraData* **);**

Parameters

signature
The address of a buffer which contains the signature.

sigCheckKey
A public-key context or key certificate object containing the public key used to verify the signature.

hashContext
A hash context containing the hash of the data.

extraData
The address of a certificate object containing extra information which is included with the signature, or null if you don't require this information.

Remarks If the signature data is invalid, the function will return CRYPT_ERROR_-BADDATA. If the signature itself is invalid, the function will return CRYPT_-ERROR_BADSIG.

See also **cryptCheckSignature**, **cryptCreateSignature**, **cryptCreateSignatureEx**, **cryptQueryObject**.

cryptCreateCert

The **cryptCreateCert** function is used to create a certificate object which contains a certificate, certification request, certificate chain, CRL, or other certificate-like object.

int cryptCreateCert(CRYPT_CERTIFICATE **cryptCert***, const CRYPT_CERT_TYPE** *certType* **);**

Parameters *cryptCert*
The address of the certificate object to be created.

certType
The type of certificate item which will be created in the certificate object.

See also **cryptDestroyCert**.

cryptCreateContext

The **cryptCreateContext** function is used to create an encryption context for a given encryption algorithm.

int cryptCreateContext(CRYPT_CONTEXT **cryptContext***, const CRYPT_ALGO** *cryptAlgo* **);**

Parameters *cryptContext*
The address of the encryption context to be created.

cryptAlgo
The encryption algorithm to be used in the context.

See also **cryptDestroyContext**, **cryptDeviceCreateContext**.

cryptCreateEnvelope

The **cryptCreateEnvelope** function is used to create an envelope object for encrypting or decrypting, signing or signature checking, compressing or decompressing, or otherwise processing data.

int cryptCreateEnvelope(CRYPT_ENVELOPE **cryptEnvelope*, **const CRYPT_FORMAT_TYPE** *formatType* **);**

Parameters *cryptEnvelope*
The address of the envelope to be created.

formatType
The data format for the enveloped data.

See also **cryptDestroyEnvelope**.

cryptCreateSignature

The **cryptCreateSignature** function digitally signs a piece of data. Due to various speed and security requirements, what is actually signed is the hash of the data rather than the data itself. The signature is placed in a buffer in a portable format which allows it to be checked using **cryptCheckSignature**.

int cryptCreateSignature(void **signature***, int** **signatureLength*, **const CRYPT_CONTEXT** *signContext***, const CRYPT_CONTEXT** *hashContext* **);**

Parameters

signature
The address of a buffer to contain the signature. If you set this parameter to null, **cryptCreateSignature** will return the length of the signature in *signatureLength* without actually generating the signature.

signatureLength
The address of the signature length.

signContext
A public-key encryption or signature context containing the private key used to sign the data.

hashContext
A hash context containing the hash of the data to sign.

See also **cryptCheckSignature**, **cryptCheckSignatureEx**, **cryptCreateSignatureEx**, **cryptQueryObject**.

cryptCreateSignatureEx

The **cryptCreateSignatureEx** function digitally signs a piece of data with extended control over the signature format. Due to various speed and security requirements, what is actually signed is the hash of the data rather than the data itself. The signature is placed in a buffer in a portable format which allows it to be checked using **cryptCheckSignatureEx**.

int cryptCreateSignatureEx(void **signature***, int** **signatureLength*, **const CRYPT_FORMAT_TYPE** *formatType***, const CRYPT_CONTEXT** *signContext***, const CRYPT_CONTEXT** *hashContext***, const CRYPT_CERTIFICATE** *extraData* **);**

Parameters

signature
The address of a buffer to contain the signature. If you set this parameter to null, **cryptCreateSignature** will return the length of the signature in *signatureLength* without actually generating the signature.

signatureLength
The address of the signature length.

formatType
The format of the signature to create.

signContext
A public-key encryption or signature context containing the private key used to sign the data.

hashContext
A hash context containing the hash of the data to sign.

extraData
Extra information to include with the signature or CRYPT_UNUSED if the format is the default signature format (which doesn't use the extra data) or CRYPT_USE_DEFAULT if the signature isn't the default format and you want to use the default extra information.

See also **cryptCheckSignature**, **cryptCheckSignatureEx**, **cryptCreateSignature**, **cryptQueryObject**.

cryptDecrypt

The **cryptDecrypt** function is used to decrypt or hash data.

int cryptDecrypt(const CRYPT_CONTEXT *cryptContext***, const void** **buffer***, **const int** *length* **);**

Parameters

cryptContext
The encryption context to use to decrypt or hash the data.

buffer
The address of the data to be decrypted or hashed.

length
The length in bytes of the data to be decrypted or hashed. For public-key encryption and signature algorithms the data length is determined by the key size of the algorithm and this parameter should be set to CRYPT_UNUSED.

Remarks Public-key encryption and signature algorithms have special data formatting requirements which need to be taken into account when this function is called. You shouldn't use this function with these algorithm types, but instead should use the higher-level functions **cryptCreateSignature**, **cryptCheckSignature**, **cryptExportKey**, and **cryptImportKey**.

See also **cryptEncrypt**.

cryptDeleteAttribute

The **cryptDeleteAttribute** function is used to delete an attribute from an object.

int cryptDeleteAttribute(const CRYPT_HANDLE *cryptObject***, const CRYPT_ATTRIBUTE_TYPE** *attributeType* **);**

Parameters

certificate
The object from which to delete the attribute.

attributeType
The attribute to delete.

Remarks Most attributes are always present and can't be deleted, in general only certificate attributes are deletable.

See also **cryptGetAttribute**, **cryptGetAttributeString**, **cryptSetAttribute**, **cryptSetAttributeString**.

cryptDeleteCertExtension

The **cryptGetCertExtension** function is used to delete a generic blob-type certificate extension from a certificate object.

int cryptDeleteCertExtension(const CRYPT_CERTIFICATE *certificate***, const char** **oid* **);**

Parameters

certificate
The certificate object from which to delete the extension.

oid
The object identifier value for the extension being deleted, specified as a sequence of integers.

Remarks cryptlib directly supports extensions from X.509, PKIX, SET, and various vendors itself, so you shouldn't use this function for anything other than unknown, proprietary extensions.

See also **cryptAddCertExtension**, **cryptGetCertExtension**.

cryptDeleteKey

The **cryptDeleteKey** function is used to delete a key or certificate from a keyset or device. The key to delete is identified either through the key owners name or their email address.

int cryptDeleteKey(const CRYPT_HANDLE *cryptObject***, const CRYPT_KEYID_TYPE** *keyIDtype***, const void** **keyID* **);**

Parameters *cryptObject*
The keyset or device object from which to delete the key.

keyIDtype
The type of the key ID, either CRYPT_KEYID_NAME for the name or key label, or CRYPT_KEYID_EMAIL for the email address.

keyID
The key ID of the key to delete.

See also **cryptAddPrivateKey**, **cryptAddPublicKey**, **cryptGetPrivateKey**, **cryptGetPublicKey**.

cryptDestroyCert

The **cryptDestroyCert** function is used to destroy a certificate object after use. This erases all keying and security information used by the object and frees up any memory it uses.

int cryptDestroyCert(const CRYPT_CERTIFICATE *cryptCert* **);**

Parameters *cryptCert*
The certificate object to be destroyed.

See also **cryptCreateCert**.

cryptDestroyContext

The **cryptDestroyContext** function is used to destroy an encryption context after use. This erases all keying and security information used by the context and frees up any memory it uses.

int cryptDestroyContext(const CRYPT_CONTEXT *cryptContext* **);**

Parameters *cryptContext*
The encryption context to be destroyed.

See also **cryptCreateContext**, **cryptDeviceCreateContext**.

cryptDestroyEnvelope

The **cryptDestroyEnvelope** function is used to destroy an envelope after use. This erases all keying and security information used by the envelope and frees up any memory it uses.

int cryptDestroyEnvelope(const CRYPT_ENVELOPE *cryptEnvelope* **);**

Parameters *cryptEnvelope*
The envelope to be destroyed.

See also **cryptCreateEnvelope**.

cryptDestroyObject

The **cryptDestroyObject** function is used to destroy a cryptlib object after use. This erases all security information used by the object, closes any open data sources, and frees up any memory it uses.

int cryptDestroyObject(const CRYPT_HANDLE *cryptObject* **);**

Parameters

cryptObject
The object to be destroyed.

Remarks

This function is a generic form of the specialised functions which destroy/close specific cryptlib object types such as encryption contexts and certificate and keyset objects. In some cases it may not be possible to determine the exact type of an object (for example the keyset access functions may return a key certificate object or only an encryption context depending on the keyset type), **cryptDestroyObject** can be used to destroy an object of an unknown type.

See also

cryptCloseKeyset, **cryptDestroyContext**, **cryptDestroyCert, cryptDestroyEnvelope**.

cryptDeviceClose

The **cryptDeviceOpen** function is used to destroy a device object after use. This closes the connection to the device and frees up any memory it uses.

int cryptDeviceClose(const CRYPT_DEVICE *device* **);**

Parameters

device
The device object to be destroyed.

See also

cryptDeviceOpen, **cryptDeviceOpenEx**.

cryptDeviceControlEx

The **cryptDeviceControlEx** function is used to perform a control function such as user authentication on a crypto device with extended control over the device control parameters.

int cryptDeviceControlEx(const CRYPT_DEVICE *device***, const CRYPT_DEVICECONTROL_TYPE** *controlType***, const void** **data1***,* **const int** *data1Length***, const void** **data2***,* **const int** *data2Length* **);**

Parameters

device
The device object to perform the control function on.

controlType
The control function to perform on the device.

data1
The first data item to send to the device, or null if none is required.

data1Length
The length of the first data item to send to the device, or CRYPT_UNUSED if none is required.

data2
The second data item to send to the device, or null if none is required.

data2Length
The length of the second data item to send to the device, or CRYPT_UNUSED if none is required.

cryptDeviceCreateContext

The **cryptDeviceCreateContext** function is used to create an encryption context for a given encryption algorithm via an encryption device.

int cryptDeviceCreateContext(const CRYPT_DEVICE *cryptDevice***, CRYPT_CONTEXT** **cryptContext***, const CRYPT_ALGO** *cryptAlgo* **);**

Parameters

cryptDevice
The device object used to create the encryption context.

cryptContext
The address of the encryption context to be created.

cryptAlgo
The encryption algorithm to be used in the context.

See also **cryptCreateContext**, **cryptDestroyContext**.

cryptDeviceOpen

The **cryptDeviceOpen** function is used to establish a connection to a crypto device such as a crypto hardware accelerator or a PCMCIA card or smart card.

int cryptDeviceOpen(CRYPT_DEVICE **device***, const CRYPT_DEVICE_TYPE** *deviceType***, const char** **name* **);**

Parameters

device
The address of the device object to be created.

deviceType
The device type to be used.

name
The name of the device, or null if a name isn't required.

See also **cryptDeviceClose**.

cryptEncrypt

The **cryptEncrypt** function is used to encrypt or hash data.

int cryptEncrypt(const CRYPT_CONTEXT *cryptContext***, const void** **buffer***, const int** *length* **);**

Parameters

cryptContext
The encryption context to use to encrypt or hash the data.

buffer
The address of the data to be encrypted or hashed.

length
The length in bytes of the data to be encrypted or hashed. For public-key encryption and signature algorithms the data length is determined by the key size of the algorithm and this parameter should be set to CRYPT_UNUSED.

Remarks Public-key encryption and signature algorithms have special data formatting requirements which need to be taken into account when this function is called. You shouldn't use this function with these algorithm types, but instead should use the higher-level functions **cryptCreateSignature**, **cryptCheckSignature**, **cryptExportKey**, and **cryptImportKey**.

See also **cryptDecrypt**.

cryptEnd

The **cryptEnd** function is used to shut down cryptlib after use. This function should be called after you have finished using cryptlib.

int cryptEnd(*void* **);**

Parameters None

See also **cryptInit**, **cryptInitEx**.

cryptExportCert

The **cryptExportCert** function is used to export an encoded signed public key certificate, certification request, CRL, or other certificate-related item from a certificate container object.

int cryptExportCert(void **certObject***, int** **certObjectLength***, const CRYPT_CERTFORMAT_TYPE** *certFormatType***, const CRYPT_CERTIFICATE** *certificate* **);**

Parameters

certObject
The address of a buffer to contain the encoded certificate.

certObjectLength
The address of the exported certificate length.

certFormatType
The encoding format for the exported certificate object.

certificate
The address of the certificate object to be exported.

Remarks The certificate object needs to have all the required fields filled in and must then be signed using **cryptSignCert** before it can be exported.

See also **cryptImportCert**.

cryptExportKey

The **cryptExportKey** function is used to share a session key between two parties by either exporting a session key from a context in a secure manner or by establishing a new shared key. The exported/shared key is placed in a buffer in a portable format which allows it to be imported back into a context using **cryptImportKey**.

If an existing session key is to be shared, it can be exported using either a public key or key certificate or a conventional encryption key. If a new session key is to be established, it can be done using a Diffie-Hellman encryption context.

int cryptExportKey(void **encryptedKey***, int** **encryptedKeyLength*, **const CRYPT_HANDLE** *exportKey***, const CRYPT_CONTEXT** *sessionKeyContext* **);**

Parameters

encryptedKey
The address of a buffer to contain the exported key. If you set this parameter to null, **cryptExportKey** will return the length of the exported key in *exportedKeyLength* without actually exporting the key.

exportedKeyLength
The address of the exported key length.

exportKey
A public-key or conventional encryption context or key certificate object containing the public or conventional key used to export the session key.

sessionKeyContext
An encryption context containing the session key to export (if the key is to be shared) or an empty context with no key loaded (if the key is to be established).

Remarks A session key can be shared in one of two ways, either by one party exporting an existing key and the other party importing it, or by both parties agreeing on a key to use. The export/import process requires an existing session key and a public/private or conventional encryption context or key certificate object to export/import it with. The key agreement process requires a Diffie-Hellman context and an empty session key context (with no key loaded) which the new shared session key is generated into.

See also **cryptExportKeyEx**, **cryptImportKey**, **cryptImportKeyEx**, **cryptQueryObject**.

cryptExportKeyEx

The **cryptExportKey** function is used to share a session key between two parties by either exporting a session key from a context in a secure manner or by establishing a new shared key, with extented control over the exported key format. The exported/shared key is placed in a buffer in a portable format which allows it to be imported back into a context using **cryptImportKeyEx**.

If an existing session key is to be shared, it can be exported using either a public key or key certificate or a conventional encryption key. If a new session key is to be established, it can be done using a Diffie-Hellman encryption context.

int cryptExportKeyEx(void **encryptedKey***, int** **encryptedKeyLength*, **const CRYPT_FORMAT_TYPE** *formatType***, const CRYPT_HANDLE** *exportKey***, const CRYPT_CONTEXT** *sessionKeyContext* **);**

Parameters *encryptedKey*
The address of a buffer to contain the exported key. If you set this parameter to null, **cryptExportKeyEx** will return the length of the exported key in *exportedKeyLength* without actually exporting the key.

exportedKeyLength
The address of the exported key length.

formatType
The format for the exported key.

exportKey
A public-key or conventional encryption context or key certificate object containing the public or conventional key used to export the session key.

sessionKeyContext
An encryption context containing the session key to export (if the key is to be shared) or an empty context with no key loaded (if the key is to be established).

Remarks A session key can be shared in one of two ways, either by one party exporting an existing key and the other party importing it, or by both parties agreeing on a key to use. The export/import process requires an existing session key and a public/private or conventional encryption context or key certificate object to export/import it with. The key agreement process requires a Diffie-Hellman context and an empty session key context (with no key loaded) which the new shared session key is generated into.

See also **cryptExportKey**, **cryptImportKey**, **cryptImportKeyEx**, **cryptQueryObject**.

cryptGenerateKey

The **cryptGenerateKey** function is used to generate a new key into an encryption context.

int cryptGenerateKey(const CRYPT_CONTEXT *cryptContext* **);**

Parameters *cryptContext*
The encryption context into which the key is to be generated.

Remarks Hash contexts don't require keys, so an attempt to generate a key into a hash context will return CRYPT_ERROR_NOTAVAIL.

cryptGenerateKey will generate a key of a length appropriate for the algorithm being used into an encryption context. If you want to specify the generation of a key of a particular length, you should use **cryptGenerateKeyEx** instead of this function.

The generation of large public-key encryption or digital signature keys can take quite some time. If the environment you are working in supports background processing, you should use **cryptGenerateKeyAsync** to generate the key instead.

See also **cryptGenerateKeyAsync**, **cryptGenerateKeyAsyncEx**, **cryptGenerateKeyEx**.

cryptGenerateKeyAsync

The **cryptGenerateKey** function is used to asynchronously generate a new key into an encryption context.

int cryptGenerateKeyAsync(const CRYPT_CONTEXT *cryptContext* **);**

Parameters *cryptContext*
The encryption context into which the key is to be generated.

Remarks Hash contexts don't require keys, so an attempt to generate a key into a hash context will return CRYPT_ERROR_NOTAVAIL.

cryptGenerateKeyAsync will generate a key of a length appropriate for the algorithm being used into an encryption context. If you want to specify the generation of a key of a particular length, you should use **cryptGenerateKeyAsyncEx** instead of this function.

See also **cryptAsyncCancel**, **cryptAsyncQuery**, **cryptGenerateKeyAsyncEx**.

cryptGenerateKeyAsyncEx

The **cryptGenerateKeyAsyncEx** function is used to asynchronously generate a new key into an encryption context with extended control over the length of the key being generated.

int cryptGenerateKeyAsyncEx(const CRYPT_CONTEXT *cryptContext***, const int** *keyLength* **);**

Parameters *cryptContext*
The encryption context into which the key is to be generated.

keyLength
The length in bytes of the key to be generated.

Remarks Hash contexts don't require keys, so an attempt to generate a key into a hash context will return CRYPT_ERROR_NOTAVAIL.

cryptGenerateKeyAsyncEx will generate a key of a given length into an encryption context. If you just want to generate a key of a length appropriate for the algorithm being used, you should use **cryptGenerateKeyAsync** instead of this function.

See also **cryptAsyncCancel**, **cryptAsyncQuery**, **cryptGenerateKeyAsync**.

cryptGenerateKeyEx

The **cryptGenerateKeyEx** function is used to generate a new key into an encryption context with extended control over the length of the key being generated.

int cryptGenerateKeyEx(const CRYPT_CONTEXT *cryptContext***, const int** *keyLength* **);**

Parameters *cryptContext*
The encryption context into which the key is to be generated.

keyLength
The length in bytes of the key to be generated.

Remarks Hash contexts don't require keys, so an attempt to generate a key into a hash context will return CRYPT_ERROR_NOTAVAIL.

cryptGenerateKeyEx will generate a key of a given length into an encryption context. If you just want to generate a key of a length appropriate for the algorithm being used, you should use **cryptGenerateKey** instead of this function.

The generation of large public-key encryption or digital signature keys can take quite some time. If the environment you are working in supports background processing, you should use **cryptGenerateKeyAsync** to generate the key instead.

See also **cryptGenerateKey**, **cryptGenerateKeyAsync**, **cryptGenerateKeyAsyncEx**.

cryptGetAttribute

The **cryptGetAttribute** function is used to obtain a boolean or numeric value, status information, or object from a cryptlib object.

int cryptGetAttribute(const CRYPT_HANDLE *cryptObject***, const CRYPT_ATTRIBUTE_TYPE** *attributeType***, int** **value* **);**

Parameters *cryptObject*
The object from which to read the boolean or numeric value, status information, or object.

attributeType
The attribute which is being read.

value
The boolean or numeric value, status information, or object.

See also **cryptDeleteAttribute**, **cryptGetAttributeString**, **cryptSetAttribute**, **cryptSetAttributeString**.

cryptGetAttributeString

The **cryptGetAttribute** function is used to obtain attribute data from a cryptlib object.

int cryptGetAttributeString(const CRYPT_HANDLE *cryptObject***, const CRYPT_ATTRIBUTE_TYPE** *attributeType***, void** **value***, int** **valueLength* **);**

Parameters *cryptObject*
The object from which to read the boolean or numeric value, status information, or object.

attributeType
The attribute which is being read.

value
The address of a buffer to contain the data. If you set this parameter to null, **cryptGetAttributeString** will return the length of the data in *attributeLength* without returning the data itself.

valueLength
The length of the data in bytes.

See also **cryptDeleteAttribute**, **cryptGetAttribute**, **cryptSetAttribute**, **cryptSetAttributeString**.

cryptGetCertExtension

The **cryptGetCertExtension** function is used to obtain a generic blob-type certificate extension from a certificate object or public or private key with an attached certificate.

int cryptGetCertExtension(const CRYPT_HANDLE *cryptObject***, const char** **oid***, int** **criticalFlag***, void** **extension***, int** **extensionLength* **);**

Parameters

cryptObject
The certificate or public/private key object from which to read the boolean or numeric value.

oid
The object identifier value for the extension being queried, specified as a sequence of integers.

criticalFlag
The critical flag for the extension being read.

extension
The address of a buffer to contain the data. If you set this parameter to null, **cryptGetCertExtension** will return the length of the data in *extensionLength* without returning the data itself.

extensionLength
The length in bytes of the extension data.

Remarks

cryptlib directly supports extensions from X.509, PKIX, SET, and various vendors itself, so you shouldn't use this function for anything other than unknown, proprietary extensions.

See also

cryptAddCertExtension, **cryptDeleteCertExtension**.

cryptGetPrivateKey

The **cryptGetPrivateKey** function is used to create an encryption context from a private key in a keyset or crypto device. The private key is identified either through the key owners name or their email address.

int cryptGetPrivateKey(const CRYPT_HANDLE *cryptHandle***, CRYPT_CONTEXT** **cryptContext***, const CRYPT_KEYID_TYPE** *keyIDtype***, const void** **keyID***, const char** **password* **);**

Parameters

cryptHandle
The keyset or device from which to obtain the key.

cryptContext
The address of a context used to contain the private key.

keyIDtype
The type of the key ID, either CRYPT_KEYID_NAME for the name or key label, or CRYPT_KEYID_EMAIL for the email address.

keyID
The key ID of the key to read.

password
The password required to decrypt the private key, or null if no password is required.

Remarks

cryptGetPrivateKey will return CRYPT_ERROR_WRONGKEY if an incorrect password is supplied. This can be used to determine whether a password is necessary by first calling the function with a null password and then retrying the read with a user-supplied password if the first call returns with CRYPT_ERROR_WRONGKEY.

See also

cryptAddPrivateKey, **cryptAddPublicKey**, **cryptDeleteKey**, **cryptGetPublicKey**.

cryptGetPublicKey

The **cryptGetPublicKey** function is used to create an encryption context from a public key in a keyset or crypto device. The public key is identified either through the key owners name or their email address.

int cryptGetPublicKey(const CRYPT_HANDLE *cryptObject***, CRYPT_HANDLE** **publicKey***, const CRYPT_KEYID_TYPE** *keyIDtype***, const void** **keyID* **);**

Parameters

cryptObject
The keyset or device from which to obtain the key.

publicKey
The address of a context or object used to contain the public key or certificate.

keyIDtype
The type of the key ID, either CRYPT_KEYID_NAME for the name or key label, or CRYPT_KEYID_EMAIL for the email address.

keyID
The key ID of the key to read.

Remarks

The type of object in which the key is returned depends on the keyset or device from which it is being read. Most sources will provide a key certificate object, but some will return only an encryption context containing the key. Both types of object can be passed to **cryptCheckCert**, **cryptCreateSignature**, or **cryptExportKey**.

See also

cryptAddPrivateKey, **cryptAddPublicKey**, **cryptDeleteKey**, **cryptGetPrivateKey**.

cryptImportCert

The **cryptImportCert** function is used to import an encoded certificate, certification request, CRL, or other certificate-related item into a certificate container object.

int cryptImportCert(void **certObject***, CRYPT_CERTIFICATE** **certificate* **);**

Parameters

certObject
The address of a buffer which contains the encoded certificate.

certificate
The certificate object to be created using the imported certificate data.

See also

cryptExportCert.

cryptImportKey

The **cryptImportKey** function is used to share a session key between two parties by importing an encrypted session key which was previously exported with **cryptExportKey** into an encryption context.

If an existing session key being shared, it can be imported using either a private key or a conventional encryption key. If a new session key is being established, it can be done using a Diffie-Hellman encryption context.

int cryptImportKey(void **encryptedKey***, const CRYPT_CONTEXT** *importContext***, CRYPT_CONTEXT** *sessionKeyContext* **);**

Parameters

encryptedKey
The address of a buffer which contains the exported key created by **cryptExportKey**.

importContext
A public-key or conventional encryption context containing the private or conventional key required to import the session key.

sessionKeyContext
The context used to contain the imported session key.

Remarks A session key can be shared in one of two ways, either by one party exporting an existing key and the other party importing it, or by both parties agreeing on a key to use. The export/import process requires an existing session key and a public/private or conventional encryption context or key certificate object to export/import it with. The key agreement process requires a Diffie-Hellman context and an empty session key context (with no key loaded) which the new shared session key is generated into.

See also **cryptExportKey**, **cryptExportKeyEx**, **cryptImportKey**, **cryptQueryObject**.

cryptImportKeyEx

The **cryptImportKeyEx** function is used to share a session key between two parties by importing an encrypted session key which was previously exported with **cryptExportKeyEx** into an encryption context.

If an existing session key being shared, it can be imported using either a private key or a conventional encryption key. If a new session key is being established, it can be done using a Diffie-Hellman encryption context.

int cryptImportKeyEx(void **encryptedKey***, const CRYPT_CONTEXT** *importContext***,**
CRYPT_CONTEXT *sessionKeyContext* **);**

Parameters *encryptedKey*
The address of a buffer which contains the exported key created by **cryptExportKeyEx**.

importContext
A public-key or conventional encryption context containing the private or conventional key required to import the session key.

sessionKeyContext
The context used to contain the imported session key.

Remarks A session key can be shared in one of two ways, either by one party exporting an existing key and the other party importing it, or by both parties agreeing on a key to use. The export/import process requires an existing session key and a public/private or conventional encryption context or key certificate object to export/import it with. The key agreement process requires a Diffie-Hellman context and an empty session key context (with no key loaded) which the new shared session key is generated into.

See also **cryptExportKey**, **cryptExportKeyEx**, **cryptImportKey**, **cryptQueryObject**.

cryptInit

The **cryptInit** function is used to initialise cryptlib before use. Either this function or **cryptInitEx** should be called before any other cryptlib function is called.

int cryptInit(*void* **);**

Parameters None

See also **cryptInitEx**, **cryptEnd**.

cryptInitEx

The **cryptInitEx** function is used to initialise cryptlib before use. **cryptInitEx** is identical to **cryptInit**, but it also performs a self-test of all the encryption algorithms provided by cryptlib. Either this function or **cryptInitEx** should be called before any other cryptlib function is called.

int cryptInitEx(*void* **);**

Parameters None

Remarks This function performs exhaustive testing of all the encryption algorithms contained in cryptlib. Since this can take some time to complete, the **cryptInit** function should be used in preference to this one.

See also **cryptInit**, **cryptEnd**.

cryptKeysetClose

The **cryptKeysetOpen** function is used to destroy a keyset object after use. This closes the connection to the key collection or keyset and frees up any memory it uses.

int cryptKeysetClose(const CRYPT_KEYSET *keyset* **);**

Parameters *keyset*
The keyset object to be destroyed.

See also **cryptKeysetOpen**, **cryptKeysetOpenEx**.

cryptKeysetOpen

The **cryptKeysetOpen** function is used to establish a connection to a key collection or keyset.

int cryptKeysetOpen(CRYPT_KEYSET **keyset***, const CRYPT_KEYSET_TYPE** *keysetType***, const char** **name***, CRYPT_KEYOPT** *options* **);**

Parameters *keyset*
The address of the keyset object to be created.

keysetType
The keyset type to be used.

name
The name of the keyset.

options
Option flags to apply when opening or accessing the keyset.

See also **cryptKeysetClose**, **cryptKeysetOpenEx**.

cryptKeysetOpenEx

The **cryptKeysetOpenEx** function is used to establish a connection to a key collection or keyset with the ability to specify extra connection parameters. This functionality is required by some database servers which may require a database, server name, user name and password, and smart card readers which may require a reader type, card type, and communication port.

int cryptKeysetOpenEx(CRYPT_KEYSET **keyset***, const CRYPT_KEYSET_TYPE** *keysetType***, const char** **name***, const char** **param1***, const char** **param2***, const char** **param3***, CRYPT_KEYOPT** *options* **);**

Parameters *keyset*
The address of the keyset object to be created.

keysetType
The keyset type to be used.

name
The name of the keyset, database, or smart card reader, or null if this parameter isn't required.

param1
The name of the database server for database keysets, or the smart card type for smart card keysets, or null if this parameter isn't required.

param2
The name of the user who will be accessing the database for database keysets or the card reader communication port for smart card keysets, or null if this parameter isn't required.

param3
The password for the user who will be accessing the database for database keysets, or null if this parameter isn't required.

options
Option flags to apply when opening or accessing the keyset.

See also **cryptKeysetClose**, **cryptKeysetOpen**.

cryptPopData

The **cryptPopData** function is used to remove data from an envelope container object. Depending on the envelope type, the data will be enveloped or de-enveloped when it is inside the envelope

int cryptPopData(const CRYPT_ENVELOPE *envelope***, const void** **buffer***, const int* *length***, const int** **bytesCopied* **);**

Parameters *envelope*
The envelope object from which to remove the data.

buffer
The address of the data to remove.

length
The length of the data to remove.

bytesCopied
The address of the number of bytes copied from the envelope.

See also **cryptPushData**.

cryptPushData

The **cryptPushData** function is used to add data to an envelope container object. Depending on the envelope type, the data will be enveloped or de-enveloped when it is inside the envelope

int cryptPushData(const CRYPT_ENVELOPE *envelope***, const void** **buffer***, const int* *length***, const int** **bytesCopied* **);**

Parameters *envelope*
The envelope object to which to add the data.

buffer
The address of the data to add.

length
The length of the data to add.

bytesCopied
The address of the number of bytes copied into the envelope.

See also **cryptPopData**.

cryptQueryCapability

The **cryptQueryCapability** function is used to obtain information about the characteristics of a particular encryption algorithm. The information returned covers the algorithm's key size, data block size, and other algorithm-specific information.

int cryptQueryCapability(const CRYPT_ALGO *cryptAlgo***, CRYPT_QUERY_INFO** **cryptQueryInfo* **);**

Parameters

cryptAlgo
The encryption algorithm to be queried.

cryptQueryInfo
The address of a **CRYPT_QUERY_INFO** structure which is filled with the information on the requested algorithm and mode, or null if this information isn't required.

Remarks Any fields in the CRYPT_QUERY_INFO structure which don't apply to the algorithm being queried are set to null or zero as appropriate. To determine whether an algorithm is available (without returning information on it), set the query information pointer to null.

See also **cryptQueryDeviceCapability**.

cryptQueryDeviceCapability

The **cryptQueryDeviceCapability** function is used to obtain information about the characteristics of a particular encryption algorithm provided by an encryption device. The information returned covers the algorithm's key size, data block size, and other algorithm-specific information.

int cryptQueryDeviceCapability(const CRYPT_DEVICE *cryptDevice***, const CRYPT_ALGO** *cryptAlgo***, CRYPT_QUERY_INFO** **cryptQueryInfo* **);**

Parameters

cryptDevice
The encryption device to be queried.

cryptAlgo
The encryption algorithm to be queried.

cryptQueryInfo
The address of a **CRYPT_QUERY_INFO** structure which is filled with the information on the requested algorithm and mode, or null if this information isn't required.

Remarks Any fields in the CRYPT_QUERY_INFO structure which don't apply to the algorithm being queried are set to null or zero as appropriate. To determine whether an algorithm is available (without returning information on them), set the query information pointer to null.

See also **cryptQueryCapability**.

cryptQueryObject

The **cryptQueryObject** function is used to obtain information about an exported key object created with **cryptExportKey**, a signature object created with **cryptCreateSignature**, or a key certificate or certificate request. It returns information such as the type and algorithms used by the object.

int cryptQueryObject(const void **objectPtr***, CRYPT_OBJECT_INFO** *cryptObjectInfo* **);**

Parameters

objectPtr
The address of a buffer which contains the object created by **cryptExportKey** or **cryptCreateSignature**.

cryptObjectInfo
The address of a CRYPT_OBJECT_INFO structure which contains information on the exported key or signature.

Remarks Any fields in the CRYPT_OBJECT_INFO structure which don't apply to the object being queried are set to null or zero as appropriate.

See also **cryptCheckSignature**, **cryptCreateSignature**, **cryptExportKey**, **cryptImportKey**.

cryptSetAttribute

The **cryptSetAttribute** function is used to add boolean or numeric information, command codes, and objects to a cryptlib object.

int cryptSetAttribute(const CRYPT_HANDLE *cryptObject***, const CRYPT_ATTRIBUTE_TYPE** *attributeType***, const int** *value* **);**

Parameters *cryptObject*
The object to which to add the value.

attributeType
The attribute which is being added.

value
The boolean or numeric value, command code, or object which is being added.

See also **cryptDeleteAttribute**, **cryptGetAttribute**, **cryptGetAttributeString**, **cryptSetAttributeString**.

cryptSetAttributeString

The **cryptSetAttributeString** function is used to add attribute data to an object.

int cryptSetAttributeString(const CRYPT_HANDLE *cryptObject***, const CRYPT_ATTRIBUTE_TYPE** *attributeType***, const void** **value***, **const int** *valueLength* **);**

Parameters *cryptObject*
The object to which to add the value.

attributeType
The attribute which is being added.

value
The address of the data being added.

valueLength
The length in bytes of the data being added.

See also **cryptDeleteAttribute**, **cryptGetAttribute**, **cryptGetAttributeString**, **cryptSetAttribute**.

cryptSignCert

The **cryptSignCert** function is used to digitally sign a public key certificate, CA certificate, certification request, CRL, or other certificate-related item held in a certificate container object.

int cryptSignCert(const CRYPT_CERTIFICATE *certificate***, const CRYPT_CONTEXT** *signContext* **);**

Parameters *certificate*
The certificate container object which contains the certificate item to sign.

signContext
A public-key encryption or signature context containing the private key used to sign the certificate.

Remarks Once a certificate item has been signed, it can no longer be modified or updated using the usual certificate manipulation functions. If you want to add further data to the certificate item, you have to start again with a new certificate object.

See also **cryptCheckCert**.

Standards Conformance

All algorithms, security methods, and data encoding systems used in cryptlib either comply with one or more national and international banking and security standards, or are implemented and tested to conform to a reference implementation of a particular algorithm or security system. Compliance with national and international security standards is automatically provided when cryptlib is integrated into an application. The standards which cryptlib follows are listed below.

Blowfish

Blowfish has been implemented as per:

> "Description of a New Varible-Length Key, 64-bit Block Cipher (Blowfish)", Bruce Schneier, "Fast Software Encryption", *Lecture Notes in Computer Science No. 809*, Springer-Verlag 1994.

The Blowfish modes of operation are given in:

> ISO/IEC 8372:1987, "Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm".
>
> ISO/IEC 10116:1997, "Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm".

The Blowfish code has been validated against the Blowfish reference implementation test vectors.

CAST-128

CAST-128 has been implented as per:

> RFC 2144, "The CAST-128 Encryption Algorithm", Carlisle Adams, May 1997.

The CAST-128 modes of operation are given in:

> ISO/IEC 8372:1987, "Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm".
>
> ISO/IEC 10116:1997, "Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm".

The CAST-128 code has been validated against the RFC 2144 reference implementation test vectors.

DES

DES has been implemented as per:

> ANSI X3.92, "American National Standard, Data Encryption Algorithm", 1981.
>
> FIPS PUB 46-2, "Data Encryption Standard", 1994.
>
> FIPS PUB 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard", 1981.
>
> ISO/IEC 8731:1987, "Banking — Approved Algorithms for Message Authentication — Part 1: Data Encryption Algorithm (DEA)".

The DES modes of operation are given in:

> ANSI X3.106, "American National Standard, Information Systems — Data Encryption Algorithm — Modes of Operation", 1983.
>
> FIPS PUB 81, "DES Modes of Operation", 1980.
>
> ISO/IEC 8372:1987, "Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm".

ISO/IEC 10116:1997, “Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm”.

The DES MAC mode is given in:

ANSI X9.9, “Financial Institution Message Authentication (Wholesale)”, 1986.

FIPS PUB 113, “Computer Data Authentication”, 1984.

ISO/IEC 9797:1994, “Information technology — Security techniques — Data integrity mechanism using a cryptographic check function employing a block cipher algorithm”.

The DES code has been validated against the test vectors given in:

NIST Special Publication 500-20, “Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard”.

Triple DES

Triple DES has been implemented as per:

ANSI X9.17, “American National Standard, Financial Institution Key Management (Wholesale)”, 1985.

ANSI X9.52, “Triple Data Encryption Algorithm Modes of Operation”, 1999.

FIPS 46-3, “Data Encryption Standard (DES)”, 1999.

ISO/IEC 8732:1987, “Banking — Key Management (Wholesale)”.

The triple DES modes of operation are given in:

ISO/IEC 8372:1987, “Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm”.

ISO/IEC 10116:1997, “Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm”.

The DES code has been validated against the test vectors given in:

NIST Special Publication 800-20, “Modes of Operation Validation System for the Triple Data Encryption Algorithm”.

Diffie-Hellman

DH has been implemented as per:

PKCS #3, “Diffie-Hellman Key Agreement Standard”, 1991.

ANSI X9.42, “Public Key Cryptography for the Financial Services Industry — Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms”, 2000.

DSA

DSA has been implemented as per:

ANSI X9.30-1, “American National Standard, Public-Key Cryptography Using Irreversible Algorithms for the Financial Services Industry”, 1993.

FIPS PUB 186, “Digital Signature Standard”, 1994.

Elgamal

Elgamal has been implemented as per

“A public-key cryptosystem based on discrete logarithms”, Taher Elgamal, *IEEE Transactions on Information Theory*, **Vol.31**, **No.4** (1985), *p.469*.

HMAC-MD5

HMAC-MD5 has been implemented as per:

RFC 2104, “HMAC: Keyed-Hashing for Message Authentication”, Hugo Krawczyk, Mihir Bellare, and Ran Canetti, February 1997.

The HMAC-MD5 code has been validated against the test vectors given in:

“Test Cases for HMAC-MD5 and HMAC-SHA-1”, Pau-Chen Cheng and Robert Glenn, March 1997.

HMAC-SHA1

HMAC-SHA1 has been implemented as per:

RFC 2104, “HMAC: Keyed-Hashing for Message Authentication”, Hugo Krawczyk, Mihir Bellare, and Ran Canetti, February 1997.

The HMAC-SHA1 code has been validated against the test vectors given in:

“Test Cases for HMAC-MD5 and HMAC-SHA-1”, Pau-Chen Cheng and Robert Glenn, March 1997.

IDEA

IDEA has been implemented as per:

“Device for the Conversion of a Digital Block and the Use Thereof”, James Massey and Xuejia Lai, International Patent PCT/CH91/00117, 1991.

“Device for the Conversion of a Digital Block and Use of Same”, James Massey and Xuejia Lai, US Patent #5,214,703, 1993.

“On the Design and Security of Block Ciphers”, Xuejia Lai, ETH Series in Information Processing, Vol.1, Hartung-Gorre Verlag, 1992.

ISO/IEC 9979, “Data Cryptographic Tecniques — Procedures for the Registration of Cryptographic Algorithms”.

The IDEA modes of operation are given in:

ISO/IEC 8372:1987, “Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm”.

ISO/IEC 10116:1997, “Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm”.

The IDEA code has been validated against the ETH reference implementation test vectors.

MD2

MD2 has been implemented as per:

RFC 1319, “The MD2 Message Digest Algorithm”, Burt Kaliski, 1992.

The MD2 code has been validated against the RFC 1319 reference implementation test vectors.

MD4

MD4 has been implemented as per:

RFC 1320, “The MD4 Message Digest Algorithm”, Ronald Rivest, 1992.

The MD4 code has been validated against the RFC 1320 reference implementation test vectors.

MD5

MD5 has been implemented as per:

RFC 1321, “The MD5 Message Digest Algorithm”, Ronald Rivest, 1992.

The MD5 code has been validated against the RFC 1321 reference implementation test vectors.

MDC-2

MDC-2 has been implemented as per:

ISO/IEC 10118-2:1994, “Information Technology — Security Techniques — Hash functions, Part 2: Hash functions using an *n*-bit block cipher algorithm” 1994.

The MDC-2 code has been validated against the reference implementation test vectors.

RC2

The RC2 code is implemented as per:

“The RC2 Encryption Algorithm”, Ronald Rivest, RSA Data Security Inc, 1992.

RFC 2268, “A Description of the RC2 Encryption Algorithm”, Ronald Rivest, 1998.

The RC2 modes of operation are given in:

ISO/IEC 8372:1987, “Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm”.

ISO/IEC 10116:1997, “Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm”.

The RC2 code has been validated against RSADSI BSAFE test vectors.

RC4

The RC4 code is implemented as per:

“The RC4 Encryption Algorithm”, Ronald Rivest, RSA Data Security Inc, 1992.

The RC4 code has been validated against RSADSI BSAFE and US Department of Commerce test vectors.

RC5

The RC5 code is implemented as per:

“The RC5 Encrypion Algorithm”, Ronald Rivest, “Fast Software Encryption II”, Lecture Notes in Computer Science No.1008, Springer-Verlag 1995.

RFC 2040, “The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms”, Robert Baldwin and Ronald Rivest, October 1996.

The RC5 modes of operation are given in:

ISO/IEC 8372:1987, “Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm”.

ISO/IEC 10116:1997, “Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm”.

The RC5 code has been validated against the RC5 reference implementation test vectors.

RIPEMD-160

The RIPEMD-160 code has been implemented as per:

"RIPEMD-160: A strengthened version of RIPEMD", Hans Dobbertin, Antoon Bosselaers, and Bart Preneel, "Fast Software Encryption III", *Lecture Notes in Computer Science No.1008*, Springer-Verlag 1995.

ISO/IEC 10118-3:1997, "Information Technology — Security Techniques — Hash functions — Part 3: Dedicated hash functions".

The RIPEMD-160 code has been validated against the RIPEMD-160 reference implementation test vectors.

RSA

The RSA code is implemented as per:

ANSI X9.31-1, "American National Standard, Public-Key Cryptography Using Reversible Algorithms for the Financial Services Industry", 1993.

ISO IEC 9594-8/ITU-T X.509, "Information Technology — Open Systems Interconnection — The Directory: Authentication Framework".

PKCS #1, "RSA Encryption Standard", 1991.

RFC 2313, "PKCS #1: RSA Encryption Version 1.5", Burt Kaliski, 1998.

SHA/SHA1

The SHA code has been implemented as per:

ANSI X9.30-2, "American National Standard, Public-Key Cryptography Using Irreversible Algorithms for the Financial Services Industry", 1993.

FIPS PUB 180, "Secure Hash Standard", 1993.

FIPS PUB 180-1, "Secure Hash Standard", 1994.

ISO/IEC 10118-3:1997, "Information Technology — Security Techniques — Hash functions — Part 3: Dedicated hash functions".

The SHA code has been validated against the test vectors given in:

FIPS PUB 180, "Secure Hash Standard", 1993.

The SHA1 code has been validated against the test vectors given in:

FIPS PUB 180-1, "Secure Hash Standard", 1994.

Safer/Safer-SK

The Safer code has been implemented as per:

"SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm", James L.Massey, "Fast Software Encryption", *Lecture Notes in Computer Science No. 809*, Springer-Verlag 1994.

The Safer-SK code has been implemented as per:

"SAFER K-64: One Year Later", James L.Massey, "Fast Software Encryption II", *Lecture Notes in Computer Science No.1008*, Springer-Verlag 1995.

The Safer/Safer-SK modes of operation are given in:

ISO/IEC 8372:1987, "Information Technology — Modes of Operation for a 64-bit Block Cipher Algorithm".

ISO/IEC 10116:1997, "Information technology — Security techniques — Modes of operation for an n-bit block cipher algorithm".

The Safer/Safer-SK code has been validated against the ETH reference implementation test vectors.

Skipjack

The Skipjack code has been implemented as per:

"Skipjack and KEA Algorithm Specifications, Version 2.0", National Security Agency, 28 May 1998.

"Capstone (MYK-80) Specifications", R21 Informal Technical Report, R21-TECH-30-95, National Security Agency, 14 August 1995.

Certificates

Certificates are implemented as per:

ISO IEC 9594-8/ITU-T X.509, "Information Technology — Open Systems Interconnection — The Directory: Authentication Framework".

PKCS #9, "Selected Attribute Types", 1993.

PKCS #10, "Certification Request Syntax Standard", 1993

RFC 2312, "S/MIME Version 2 Certificate Handling", Dusse et al, 1998.

RFC 2314, "PKCS #10: Certification Request Syntax Version 1.5", Burt Kaliski, 1998.

RFC 2459, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", Russ Housley, Warwick Ford, Tim Polk, and David Solo, January 1999.

RFC 2528, "Representation of Key Exchange Algorithm (KEA) Keys in Internet X.509 Public Key Infrastructure Certificates", Russ Housley and Tim Polk, March 1999.

RFC 2632, "S/MIME Version 3 Certificate Handling", Blake Ramsdell, June 1999.

In addition to the above standards there are a large and ever-changing number of organisational, national, and international certificate profiles. cryptlib tries to remain compatible with the latest revisions of the various profiles, with configuration options available to select different behaviour if there are conflicts in the standards. The default profile is the IETF PKIX one. Further information about certification standards is given in the chapters on certificate handling.

Data Structures

All message exchange data structures are specified and encoded as per:

ISO/IEC 8824:1993/ITU-T X.680, "Information Technology — Open Systems Interconnection — Abstract Syntax Notation One (ASN.1)".

ISO/IEC 8825:1993/ITU-T X.692, "Information Technology — Open Systems Interconnection — Specification of ASN.1 Encoding Rules".

S/MIME

The cryptographic message syntax of cryptlib data is given in:

RFC 2315, "PKCS #7: Cryptographic Message Syntax", Burt Kaliski, 1998.

RFC 2630, "Cryptographic Message Syntax", Russ Housley, June 1999.

RFC 2634, "Enhanced Security Services for S/MIME", Paul Hoffman, June 1999.

The ASN.1 specifications for the message structures are given in the file cryptlib.asn.

Y2K Compliance

cryptlib's date management complies with the requirements in the British Standards Institutes Year 2000 Conformity standard:

DISC PD2000-1:1998, "A Definition of Year 2000 Conformity Requirements", 1998.

General

The encryption subsystem has been implemented at a level equivalent to level 1 of the standard given in:

FIPS PUB 140-1, "Security Requirements for Cryptographic Modules", 1993.

The random-data acquisition routines follow the guidelines laid out in:

"Randomness Recommendations for Security", RFC 1750, Donald Eastlake, Stephen Crocker, and Jeffrey Schiller, December 1994.

"Cryptographic Random Numbers", IEEE P1363 Appendix E, Draft version 1.0, 11 November 1995.

Acknowledgements

Chris Wedgwood and Paul Kendall helped write the Unix random data gathering routines.

Cylink Corporation very generously granted permission for the use of their patents for non-commercial purposes.

Eric Young wrote the Blowfish, CAST-128, DES, 3DES, MD5, and SHA code and bignum library.

Jean-Loup Gailly and Mark Adler wrote the zlib compression code.

Joerg Plate did the Amiga port.

Leonard Janke wrote the 80x86 RIPEMD-160 code.

Markus F.X.J. Oberhumer did the 32-bit DOS port.

Masayasu Kumagai wrote the 80x86 IDEA code.

Matt Thomlinson and Blake Coverett helped fix up and debug the Win32 random data gathering routines.

Matthijs van Duin did the Macintosh port.

Osma Ahvenlampi did the BeOS port.

Stuart Woolford and Mario Korva did the OS/2 port.

Wolfgang Gothier tracked down a number of *really* obscure probl^H^H^H^H^Hundocumented features.