cryptlib

Security Toolkit

Version 3.4.8

January 2026

Introduction

The information age has seen the development of electronic pathways that carry vast amounts of valuable commercial, scientific, and educational information between financial institutions, companies, individuals, and government organisations. Unfortunately the security systems required to protect data are often extremely difficult to implement, and even when available tend to require considerable understanding of the underlying principles in order to be used. The cryptlib security toolkit provides the answer to this problem. A complete description of the capabilities provided by cryptlib are given below.

cryptlib Overview

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

At the highest level, cryptlib provides implementations of complete security services such as S/MIME and PGP/OpenPGP secure enveloping, TLS and SSH secure sessions, CA services such as CMP, SCEP, RTCS, OCSP, and SCVP, and other security operations such as secure timestamping (TSP). Since cryptlib uses industry-standard X.509, S/MIME, PGP/OpenPGP, and SSH/TLS data formats, the resulting encrypted or signed data can be easily transported to other systems and processed there, and cryptlib itself runs on virtually any operating system — cryptlib doesn't tie you to a single platform. This allows email, files, and EDI transactions to be authenticated with digital signatures and encrypted in an industry-standard format.

cryptlib provides an extensive range of other capabilities including full X.509/PKIX certificate handling (all X.509 versions from X.509v1 to X.509v3) with additional support for Microsoft AuthentiCode, RPKI, SigG, S/MIME, TLS, and Qualified certificates, PKCS #7 certificate chains, handling of certification requests and CRLs including automated checking of certificates against CRLs and online checking using RTCS, OCSP, and SCVP, and issuing and revoking certificates using CMP and SCEP. In addition cryptlib implements a full range of certificate authority (CA) functions, as well as providing complete CMP, SCEP, RTCS, OCSP, and SCVP server implementations to handle online certificate enrolment/issue/revocation and certificate status checking. Alongside the certificate handling, cryptlib provides a sophisticated key storage interface that allows the use of a wide range of key database types ranging from PKCS #11 devices, PKCS #15 key files, and PGP/OpenPGP key rings through to commercial-grade RDBMS' and LDAP directories with optional TLS protection.

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, hardware security modules (HSMs), and crypto smart cards. For particularly demanding applications cryptlib can be used with a variety of crypto devices that have received appropriate FIPS 140 or ITSEC/Common Criteria certification. The crypto device interface also provides a convenient general-purpose plug-in capability for adding new functionality that will be automatically used by cryptlib. cryptlib also provides a general-purpose crypto HAL (hardware abstraction layer) interface that allows it to use the native crypto capabilities available in some ARM, MIPS, and PPC cores used in embedded systems and devices.

cryptlib is supplied as source code for AMX, Arinc653, BeOS, ChorusOS, CMSIS-RTOS, CMS, DOS, DOS32, eCOS, embOS, µC/OS, embedded Linux, FreeRTOS/OpenRTOS, IBM MVS, µITRON, Mongoose OS, MQX, Nucleus, OS/2, OS X, OSEK, Quadros, RiotOS, RTEMS, SMX, Tandem, Telit, ThreadX, T-Kernel, a variety of Unix versions (including AIX, Digital Unix, DGUX, FreeBSD/NetBSD/OpenBSD, HP-UX, IRIX, Linux, MP-RAS, OSF/1, QNX, SCO/UnixWare, Solaris, SunOS, Ultrix, and UTS4), uClinux, VM/CMS, VxWorks, Windows (32- and 64-bit versions), Windows CE/PocketPC/SmartPhone/Embedded, VDK, Xilinx XMK, and Zephyr. cryptlib's highly portable nature means that it is also being used in a variety of custom embedded system environments. cryptlib comes with language bindings for C / C++, C# / .NET, Delphi, Java, Perl, Python, and Visual Basic (VB).

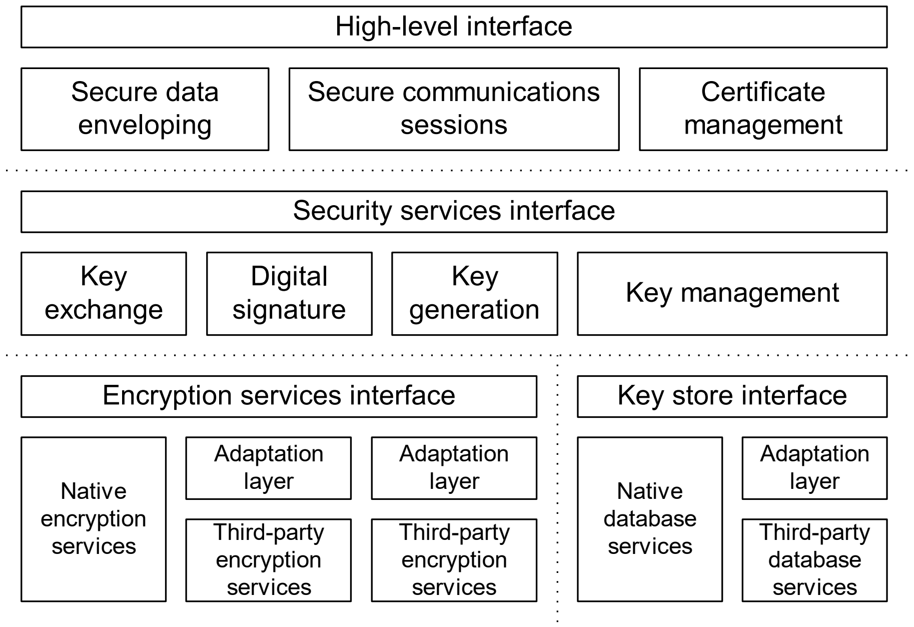
cryptlib features

cryptlib provides a standardised interface to a number of popular encryption algorithms, as well as providing a high-level interface that hides most of the implementation details and uses operating-system-independent encoding methods that make 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.

Architecture

cryptlib consists of a set of layered security services and associated programming interfaces that provide an integrated set of information and communications security capabilities. Much like the network reference model, cryptlib contains a series of layers that provide each level of abstraction, with higher layers building on the capabilities provided by the lower layers.

At the lowest level are basic components such as core encryption and authentication routines, which are usually implemented in software but may also be implemented in hardware (due to the speed of the software components used in cryptlib, the software is usually faster than dedicated hardware). At the next level are components that wrap up the specialised and often quite complex core components in a layer that provides abstract functionality and ensures complete cross-platform portability of data. These functions typically cover areas such as "create a digital signature" or "exchange an encryption key". At the highest level are extremely powerful and easy-to-use functions such as "encrypt a message", "sign a message", "open a secure link", and "create a digital certificate" that require no knowledge of encryption techniques, and that take care of complex issues such as key management, data encoding, en/decryption, and digital signature processing.



cryptlib's 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 that make the encryption or authentication work. The automatic object-management routines take care of encoding issues and cross-platform portability problems, so that a handful of function calls is all that's needed to wrap up data in signed or encrypted form with all of the associated information and parameters needed to recreate it on the other side of a communications channel. This provides a considerable advantage over other encryption toolkits that 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 medical messaging and the Session Initiation Protocol (SIP) for services like Internet telephony and instant messaging.

The S/MIME implementation uses cryptlib's enveloping interface which allows simple, rapid integration of strong encryption and authentication capabilities into existing email agents and messaging software. The resulting signed enveloped data format provides message integrity and origin authentication services, the encrypted enveloped data format provides confidentiality, and the authenticated encrypted data format provides all of these services at once. In addition cryptlib's S/MIME implementation allows external services such as trusted timestamping authorities (TSAs) to be used when a signed message is created, providing externally-certified proof of the time of message creation. The complexity of the S/MIME format means that the few other toolkits that 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.

PGP/OpenPGP

Alongside the PKCS #7/CMS/SMIME formats, cryptlib supports the PGP/OpenPGP message format, allowing it to be used to send and receive PGP-encrypted email and data. As with the S/MIME implementation, the PGP implementation uses cryptlib's enveloping interface to allow simple, rapid integration of strong encryption and authentication capabilities into existing email agents and messaging software. Since the enveloping interface is universal, the process involved in creating PGP and S/MIME messages is identical except for the envelope format specifier, allowing a one-off development effort to handle any secure message format.

SSH and TLS

cryptlib supports secure network sessions using the SSH and TLS security protocols. As with envelopes, cryptlib takes care of the session details for you so that all you need to do is provide basic communications information such as the name of the server or host to connect to and any other information required for the session such as a password or certificate. cryptlib takes care of establishing the session and managing the details of the communications channel and its security parameters, and provides both client and server implementations of all of these session types.

cryptlib also supports additional protocols in combination with the lower-layer TLS secure tunnel. These include the WebSockets protocol in client and server implementations and various EAP / RADIUS-based protocols including EAP-TLS, EAP-TTLS, and EAP-PEAP in client implementations.

PKI Services: CMP, SCEP, RTCS, OCSP, SCVP, TSP

In addition to SSH and TLS, cryptlib also implements a full range of PKI services in its secure session interface, again providing both client and server implementations of all protocols. These services include the certificate management protocol (CMP), simple certificate enrolment protocol (SCEP), real-time certificate status protocol (RTCS), online certificate status protocol (OCSP), server-based certificate validation protocol (SCVP), and timestamping (TSP).

By tying a key or certificate store to the session, you can let cryptlib take care of any key management issues for you. With a CMP or SCEP server session cryptlib will use the certificate store to handle the certificate management process. In this way a complete CMP-based CA that handles enrolment, certificate update and renewal, and certificate revocation, can be implemented with only a handful of function calls.

Plug-and-play PKI

Working with certificates can be complex and painful, requiring the use of a number of arcane and difficult-to-use mechanisms to perform even the simplest operations. To eliminate this problem cryptlib provides a plug-and-play PKI interface that manages all certificate processing and management operations for you, requiring no special knowledge of certificate formats, protocols, or operations. Using the plug-and-play PKI interface with an appropriately-configured CA means that cryptlib will automatically and transparently handle key generation, certificate enrolment, securely obtaining trusted CA certificates, and certifying the newly-generated keys for the user, all in a single operation. Similarly, certificate validity checking can be performed using an online real-time status check that avoids the complexity and delayed status information provided by mechanisms like CRLs. The plug-and-play PKI interface removes most of the complexity and difficulty involved in working with certificates, making it easier to use certificates than with any of the conventional certificate management mechanisms.

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, Identrus certificates, qualified certificates, resource PKI (RPKI) certificates, S/MIME and TLS client and server certificates, SigG extensions, 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 certification requests suitable for submission to certification authorities (CAs) 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 creating and handling of certificate chains required for S/MIME, TLS, and other applications, and the creation of certificate revocation lists (CRLs) with the capability to check certificates against existing or new CRLs either automatically or under programmer control. In addition to CRL-based revocation checking, cryptlib also supports online status protocols such as RTCS, OCSP, and SCVP. cryptlib also implements the CMP protocol which fully automates the management of certificates, allowing online certificate enrolment, issue, update/replacement, and revocation of certificates, and the SCEP protocol, which automates the certificate issue process. Using CMP removes from the user any need for technical knowledge of certificate management, since all details are managed by the CA.

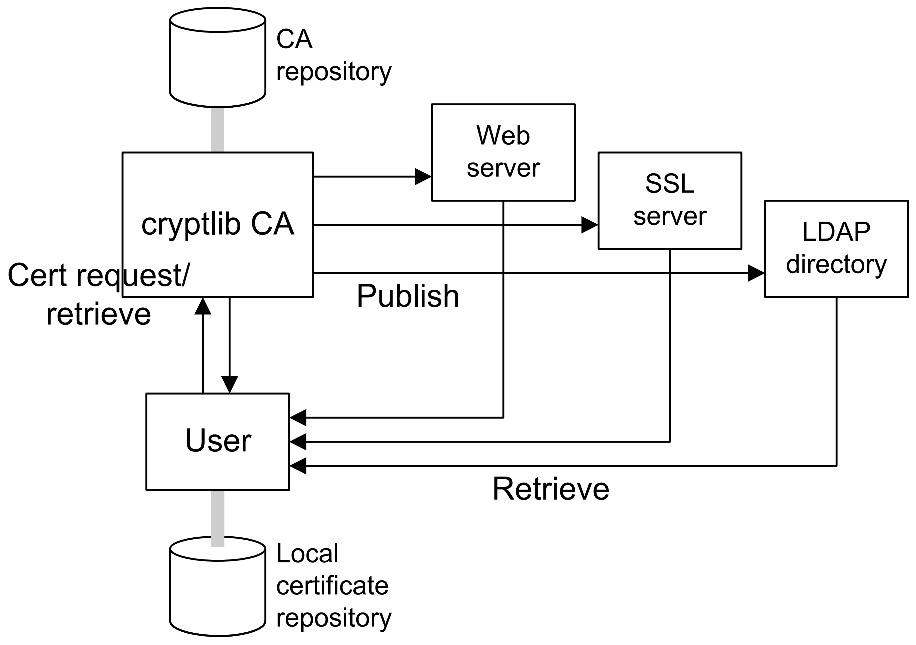
cryptlib can import and export certification requests, certificates, certificate chains, and CRLs, covering the majority of certificate transport formats used by a wide variety of software such as web browsers and servers. The certificate types that are supported include:

- Standard X.509 certificates
- AuthentiCode code signing certificates
- Certificates conformant to the IETF PKIX profile
- IPsec server, client, end-user, and tunnelling certificates

- Qualified certificates
- Resource PKI (RPKI) certificates
- SigG certificate extensions
- S/MIME email certificates
- TLS server and client certificates
- Timestamping certificates

In addition cryptlib supports X.509v3 IETF, S/MIME, SET, and SigG certificate extensions and 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.

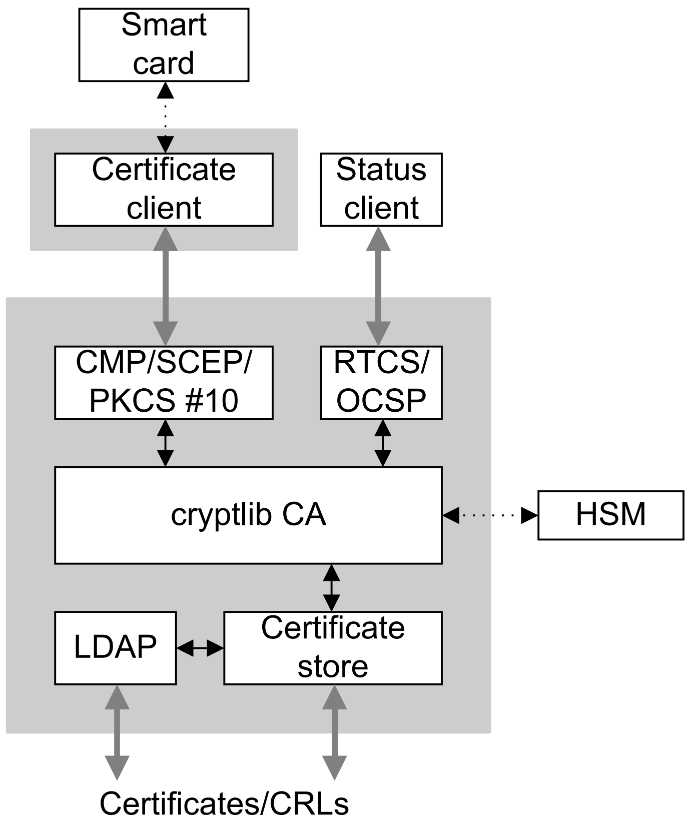
The diagram below shows a typical cryptlib application, in which it provides the full functionality of both a CA (processing certification requests, storing the issued certificates locally in a certificate database, and optionally publishing the certificates on the web or in an LDAP directory) and an end entity (generating certification requests, submitting them to a CA, and retrieving the result from the web or a directory service).



To handle certificate trust and revocation issues, cryptlib includes a certificate trust manager that can be used to automatically manage CA trust settings. For example a CA can be designated as a trusted issuer that will allow cryptlib to automatically evaluate trust along certificate chains. Similarly, cryptlib can automatically check certificates against RTCS, OCSP, and SCVP responders and CRLs published by CAs, removing from the user the need to perform complex manual checking.

CA Operations

cryptlib includes a scalable, flexible Certificate Authority (CA) engine built on the transaction-processing capabilities of a number of proven, industrial-strength relational databases running on a variety of hardware platforms. The CA facility provides an automated means of handling certificate issuance without dealing directly with the details of processing request, signing certificates, saving the resulting certificates in keys stores, and assembling CRLs. This constitutes a complete CA system for issuance and management of certificates and CRLs. A typical cryptlib CA configuration is shown below.



Available CA operations include:

- Certificate enrolment/initialisation operations
- Certificate issue
- Certificate update/key update
- Certificate expiry management
- Revocation request processing
- CRL issue

All CA operations are recorded to an event log using cryptlib's built-in CA logging/auditing facility, which provides a comprehensive audit facility via a full account of certificate requests, certificates issued or renewed, revocations requested and issued, certificates expired, and general CA management operations. The logs can be queried for information on all events or a specified subset of events, for example all certificates that were issued on a certain day.

cryptlib contains a full implementation of a CMP server (to handle online certificate management), and SCEP server (to handle online certificate issue), a RTCS server (to handle real-time certificate status checking), and a OCSP and SCVP servers (to handle revocation checking). All of these servers are fully automated, requiring little user intervention beyond the initial enrolment process in which user eligibility for a certificate is established. These services make it easier than ever to manage your own CA. Certificate expiration and revocation are handled automatically by the CA engine. Expired certificates are removed from the certificate store, and CRLs are assembled from previously processed certificate revocation requests. These operations are handled with a single function call, for example issuing a CRL is done with:

```
cryptCACertManagement( &cryptCRL, CRYPT_CERTACTION_ISSUE_CRL,
    cryptCertStore, CRYPT_UNUSED );
```

The CA keys can optionally be generated and held in tamper-resistant hardware security modules, with certificate signing being performed by the hardware module. Issued certificates can be stored on smart cards or similar crypto devices in addition to being managed using software-only implementations. The CA facility supports the simultaneous operation of multiple CAs, for example to manage users served through divisional CAs certified by a root CA. Each CA can issue multiple certificates to users, allowing the use of separate keys bound to signature and encryption certificates.

Crypto Devices, Hardware, and Smart Card Support

In addition to its built-in capabilities, cryptlib can make use of the crypto capabilities of external crypto devices and on-chip encryption accelerators, crypto cores, and other hardware. External device types supported include:

- Crypto smart cards
- Dallas iButtons
- Datakeys/iKeys
- Fortezza cards
- Hardware crypto accelerators
- Hardware security modules (HSMs)
- PCMCIA crypto tokens
- PCI crypto cards
- PKCS #11 devices
- USB tokens

Both these external devices and on-chip encryption hardware present in some embedded processors and devices can be used by cryptlib to handle functions such as key generation and storage, certificate creation, digital signatures, and message en- and decryption. Typical applications include:

- Running a certification authority inside tamper-resistant hardware
- Smart-card based digital signatures
- Message encryption/decryption in secure hardware

cryptlib manages any device-specific interfacing requirements so that the programming interface for any crypto device is identical to cryptlib's native interface, allowing existing applications that use cryptlib to be easily and transparently migrated to using crypto devices. The ability to mix and match crypto devices and the software-only implementation allows appropriate tradeoffs to be chosen between flexibility, cost, and security.

Certificate Store Interface

cryptlib utilizes commercial-strength RDBMS' to store keys in the internationally standardised X.509 format. The certificate store 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.

In addition to standard certificate stores, cryptlib supports the storage and retrieval of certificates in LDAP directories, HTTP access for keys accessible via the web, and external flat-file key collections such as PKCS #15 soft-tokens and PGP/OpenPGP key rings. The key collections can be freely mixed (so for example a private key could be stored in a PKCS #15 soft-token, a PGP/OpenPGP key ring or on a smart card with the corresponding X.509 certificate being stored in a certificate store, an LDAP directory, or on the web).

Private keys can be stored on disk encrypted with an algorithm such as AES (selectable by the user), with the password processed using several thousand iterations of a hashing algorithm like SHA-2 (also selectable by the user) and the key data protected from tampering with an algorithm like HMAC-SHA2 (again selectable by the user). Where the operating system supports it, cryptlib will apply system security features such as ACLs under Windows and file permissions under Unix to the private key file to further restrict access.

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 (inter-object 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 that 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 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. In general keying information isn't even visible to the user, since it's generated inside the object itself and never leaves the security perimeter. This design is ideally matched to hardware implementations that perform strict red/black separation, since sensitive information can never leave the hardware.

Associated with each object is a set of mandatory ACLs that determine who can access a particular object and under which conditions the access is allowed. If the operating system supports it, all sensitive information used will be page-locked to ensure that it's 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 sanitised 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 that it creates or manages. For example under Windows cryptlib private key files will be created with an access control list (ACL) that allows only the key owner access to the file; under Unix the file permissions will be set to achieve the same result.

Embedded Systems

cryptlib's high level of portability and configurability makes it ideal for use in embedded systems with limited resources or specialised requirements, including ones based on Altera NIOS, ARM7, ARM9, ARM TDMI, Coldfire, Fujitsu FR-V, Hitachi SuperH, MIPS IV, MIPS V, Motorola ColdFire, NEC V8xx series, NEC VRxxxx series, Panasonic/Matsushita AM33/AM34, PowerPC, PowerQUICC, Risc-V, Samsung CalmRISC, SH3, SH4, SPARC, SPARClite, StrongArm, TI OMAP, and Xilinx MicroBlaze processors, as well as a large range of licensed derivatives of these cores, too many and varied to enumerate here. cryptlib doesn't perform any floating-point operations and runs directly on processors without an FPU, and through its crypto HAL (hardware abstraction layer) capabilities can take advantage of on-chip or in-system cryptographic hardware capabilities and crypto cores where available, typically on some of the more advanced ARM, PPC, and Risc-V cores.

The code is fully independent of any underlying storage or I/O mechanisms, and works just as easily with abstractions like named memory segments in flash memory as it does with standard key files on disk. It has been deployed on embedded systems without any conventional I/O capabilities (stdio) or dynamic memory allocation facilities, with proprietary operating system architectures and services including ATMs, printers, web-enabled devices, POS systems, embedded device controllers, and similar environments, and even in devices with no operating system at all (cryptlib runs on the bare metal). It can also run independent of any form of operating system, and has been run on the bare metal in environments with minimal available resources, in effect functioning as a complete crypto operating system for the underlying hardware.

Because cryptlib functions identically across all supported environments, it's possible to perform application development in a full-featured development environment such as Windows or Unix and only when the application is complete and tested move it to the embedded system. This flexibility saves countless hours of development time, greatly reducing the amount of time that needs to be spent with embedded systems debuggers or in-circuit emulators since most of the development and code testing can be done on the host system of choice.

If required the cryptlib developers can provide assistance in moving the code to any new or unusual environments.

Performance

cryptlib is re-entrant and completely thread-safe, allowing it to be used with multithreaded applications under operating systems that 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.

Many of the core algorithms used in cryptlib have been implemented in assembly language in order to provide the maximum possible performance, and will take advantage of crypto hardware acceleration facilities present in some CPUs such as the Via CPU family. These routines provide an unprecedented level of performance, in most cases running faster than expensive, specialised encryption hardware designed to perform the same task. This means that 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, specialised encryption hardware.

Programming Interface

cryptlib's easy-to-use high-level routines allow for the exchange of encrypted or signed messages or the establishment of secure communications channels with a minimum of programming overhead. Language bindings are available for C / C++, C# / .NET, Delphi, Java, Perl, Python and Visual Basic (VB).

cryptlib has been written to be as foolproof as possible. On initialisation it performs extensive self-testing against test data from encryption standards documents, and the APIs 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.

Documentation

cryptlib comes with extensive documentation in the form of a 350-page user manual and a 320-page technical reference manual. The user manual is intended for everyday cryptlib use and contains detailed documentation on every aspect of cryptlib's functionality. In most cases the code needed to secure an application can be cut and pasted directly from the appropriate section of the manual, avoiding the need to learn yet another programming API.

The technical reference manual covers the design and internals of cryptlib itself, including the cryptlib security model and security mechanisms that protect every part of cryptlib's operation. In addition the technical manual provides a wealth of background information to help users understand the security foundations on which cryptlib is built.

Algorithm Support

Included as core cryptlib components are implementations of the most popular encryption and authentication algorithms, AES, CAST, ChaCha20, DES, triple DES, IDEA and RC4, conventional encryption, SHA-1, and SHA-2/SHA-256 hash algorithms, HMAC-SHA1, HMAC-SHA2, and Poly1305 MAC algorithms, and Diffie-Hellman, DSA, ECDSA, ECDH, Elgamal, and RSA public-key encryption algorithms. The algorithm parameters are summarised below:

Algorithm	Key size	Block size
AES	128/192/256	128
CAST-128	128	64
ChaCha20	256	8
DES	56	64
Triple DES	112 / 168	64
IDEA	128	64
RC4	2048	8
SHA-1	—	160
SHA-2 / SHA-256	—	256
HMAC-SHA1	160	160
HMAC-SHA2	256	256
Poly1305	256	128
Diffie-Hellman	4096	—
DSA	4096[1]	—
ECDSA	521	—
ECDH	521	—
Elgamal	4096	—
RSA	4096	—

Note that some of these algorithms are present only for backwards-compatibility purposes or have security issues and shouldn't be used unless you know what you're doing. In particular DES and SHA-1 are no longer regarded as secure, RC4 has serious security problems and shouldn't be used, IDEA is only present for PGP 2.x compatibility and is slated for removal in a future release, and Poly1305 and ChaCha20 are extremely difficult to use safely and are only present for internal use by cryptlib in protocols like TLS.

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 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, ANSI X9.55, ANSI X9.57, ANSI X9.62, ANSI X9.63, ANSI X9.73, ANSI X9.95, ETSI TS 101 733, ETSI TS 101 861, ETSI TS 101 862, ETSI TS 102, ETSI TS 133 310 (3GPP CMP), 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, FIPS PUB 198, 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, ISO/IEC 15782, ISO/IEC 18014, ITU-T X.842, ITU-T X.843, NSA Suite B, PKCS #1, PKCS #3, PKCS #5, PKCS #7, PKCS #9, PKCS #10, PKCS #11, PKCS #15, RFC 1319, RFC 1320, RFC 1321, RFC 1750, RFC 1991, RFC 2040, RFC 2058, RFC 2104, RFC 2138, RFC 2144, RFC 2202, RFC 2246, RFC 2268, RFC 2284, RFC 2311 (cryptography-related portions), RFC 2312, RFC 2313, RFC 2314, RFC 2315, RFC 2437, RFC 2440, RFC 2459, RFC 2510, RFC 2511, RFC 2528, RFC 2548, RFC 2560, RFC 2585, RFC 2630, RFC 2631, RFC 2632, RFC 2633 (cryptography-related portions), RFC 2634, RFC 2785, RFC 2865, RFC 2869, RFC 2876, RFC 2898, RFC

[1] The DSA standard only defines key sizes from 512 to 1024 bits, cryptlib supports longer keys.

2984, RFC 2985, RFC 2986, RFC 3039, RFC 3058, RFC 3114, RFC 3126, RFC 3161, RFC 3174, RFC 3183, RFC 3211, RFC 3218, RFC 3261 (cryptography-related portions), RFC 3268, RFC 3274, RFC 3278, RFC 3279, RFC 3280, RFC 3281, RFC 3369, RFC 3370, RFC 3447, RFC 3546, RFC 3526, RFC 3565, RFC 3579, RFC 3739, RFC 3748, RFC 3770, RFC 3779, RFC 3851, RFC 3852, RFC 4055, RFC 4086, RFC 4108, RFC 4134, RFC 4137, RFC 4210, RFC 4211, RFC 4226, RFC 4231, RFC 4250, RFC 4251, RFC 4252, RFC 4253, RFC 4254, RFC 4256, RFC 4262, RFC 4279, RFC 4325, RFC 4334, RFC 4346, RFC 4366, RFC 4387, RFC 4419, RFC 4476, RFC 4492, RFC 4590, RFC 4648, RFC 4680, RFC 4681, RFC 4853, RFC 4880, RFC 4945, RFC 5035, RFC 5055, RFC 5056, RFC 5083, RFC 5084, RFC 5090, RFC 5114, RFC 5116, RFC 5208, RFC 5216, RFC 5246, RFC 5280, RFC 5281, RFC 5288, RFC 5289, RFC 5430, RFC 5480, RFC 5487, RFC 5489, RFC 5639, RFC 5652, RFC 5656, RFC 5746, RFC 5750, RFC 5751, RFC 5753, RFC 5754, RFC 5755, RFC 5756, RFC 5758, RFC 5816, RFC 5869, RFC 5911, RFC 5912, RFC 5915, RFC 5924, RFC 6066, RFC 6070, RFC 6149, RFC 6150, RFC 6158, RFC 6151, RFC 6194, RFC 6211, RFC 6234, RFC 6238, RFC 6268, RFC 6455, RFC 6476, RFC 6484, RFC 6485, RFC 6487, RFC 6637, RFC 6668, RFC 6677, RFC 6712, RFC 6818, RFC 6929, RFC 6960, RFC 7027, RFC 7292, RFC 7366, RFC 7435, RFC 7465, RFC 7468, RFC 7507, RFC 7525, RFC 7539, RFC 7568, RFC 7627, RFC 7685, RFC 7696, RFC 7748, RFC 7905, RFC 7919, RFC 7935, RFC 8017, RFC 8018, RFC 8032, RFC 8044, RFC 8268, RFC 8270, RFC 8308, RFC 8332, RFC 8410, RFC 8422, RFC 8423, RFC 8439, RFC 8446, RFC 8448, RFC 8550, RFC 8551, RFC 8619, RFC 8701, RFC 8709, RFC 8731, RFC 8734, RFC 8758, RFC 8813, RFC 8894, RFC 8933 and RFC 8954. In addition cryptlib can be used as an add-on security module to provide security services as per ISO/IEC 62351 for SCADA protocols such as IEC 60870-5, DNP 3.0, IEC 60870-6 (TASE.2 or ICCP), IEC 61850, and IEC 61334 (DLMS), and for the Payment Card Industry (PCI) Data Security Standard (PCI-DSS). Because of the use of internationally recognised and standardised security algorithms, cryptlib users will avoid the problems caused by home-grown, proprietary algorithms and security techniques that often fail to provide any protection against attackers, resulting in embarrassing bad publicity and expensive product recalls.

Configuration Options

cryptlib works with a configuration database that can be used to tune its operation for different environments. This allows a system administrator to set a consistent security policy which is 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.

cryptlib Applications

The security services provided by cryptlib can be used in virtually any situation that requires the protection or authentication of sensitive data. Some areas in which cryptlib is currently used include:

- Protection of medical records transmitted over electronic links.
- Protection of financial information transmitted between branches of banks.
- Transparent disk encryption.
- Strong security services added to web browsers with weak, exportable security.
- Running a CA.
- Encrypted electronic mail.
- File encryption.
- Protecting content on Internet servers.
- Digitally signed electronic forms.

- S/MIME mail gateway.
- Secure database access.
- Protection of credit card information.

Encryption Code Example

The best way to illustrate what cryptlib can do is with an example. The following code encrypts a message using public-key encryption.

```
/* Create an envelope for the message */
cryptCreateEnvelope( &cryptEnvelope, cryptUser, CRYPT_FORMAT_SMIME );

/* Push in the message recipient's name */
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_RECIPIENT,
   recipientName, recipientNameLength );

/* Push in the message data and pop out the encrypted result */
cryptPushData( cryptEnvelope, message, messageSize, &bytesIn );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, encryptedMessage, encryptedSize,
   &bytesOut );

/* Clean up */
cryptDestroyEnvelope( cryptEnvelope );
```

This performs the same task as a program like PGP using just 6 function calls (to create a PGP/OpenPGP message, just change the CRYPT_FORMAT_SMIME to CRYPT_FORMAT_PGP). All data management is handled automatically by cryptlib, so there's no need to worry about encryption modes and algorithms and key lengths and key types and initialisation vectors and other details (although cryptlib provides the ability to specify all this if you feel the need). This is all that's required — just copy the above code into your application to S/MIME-enable it.

The code shown above results in cryptlib performing the following actions:

- Generate a random session key for the default encryption algorithm (usually AES).
- Look up the recipient's public key in a key database.
- Encrypt the session key using the recipient's public key.
- Encrypt the signed data with the session key.
- Pass the result back to the user.

However unless you want to call cryptlib using the low-level interface, you never need to know about any of this. cryptlib will automatically know what to do with the data based on the resources you add to the envelope — if you add a signature key it will sign the data, if you add an encryption key it will encrypt the data, and so on.

Secure Session Code Example

Establishing a secure session using TLS is similarly easy:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser, CRYPT_SESSION_TLS );

/* Add the server name and activate the session */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

If you prefer SSH to TLS, just change the CRYPT_SESSION_TLS to CRYPT_-SESSION_SSH and add a user name and password to log on. As with the encryption code example above, cryptlib provides a single unified interface to its secure session mechanisms, so you don't have to invest a lot of effort in adding special-case handling for different security protocols and mechanisms.

The corresponding TLS (or SSH if you prefer) server is:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser,
   CRYPT_SESSION_TLS_SERVER );

/* Add the server key/certificate and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

As with the secure enveloping example, cryptlib is performing a large amount of work in the background, but again there's no need to know about this since it's all taken care of automatically.

Certificate Management Code Example

The following code illustrates cryptlib's plug-and-play PKI interface:

```
CRYPT_SESSION cryptSession;

/* Create the CMP session and add the server name/address */
cryptCreateSession( &cryptSession, cryptUser, CRYPT_SESSION_CMP );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );

/* Add the username, password, and smart card */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_PRIVKEYSET,
   cryptDevice );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

This code takes a smart card and generates separate encryption and signing keys in it, requests a signature certificate from the CA for the signing key, uses that to obtain a certificate for the encryption key, obtains any further certificates that may be needed from the CA (for example for S/MIME signing or TLS server operation), and stores everything in the smart card. Compare this to the hundreds or even thousands of lines of code required to do the same thing using other toolkits.

Oh yes, and cryptlib provides the CA-side functionality as well — there's no need to pay an expensive commercial CA for your certificates, since cryptlib can perform the same function.

Document conventions

This manual uses the following document conventions:

Example	Description
cryptlib.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 that you need to supply.
`if( i == 0 )`	This font is used for sample code and operating system commands.

Recommended Reading

Cryptographic Security Architecture Design and Verification by Peter Gutmann is the technical documentation for cryptlib and complements the cryptlib user manual. It contains full details of the architectural and security features of cryptlib, as well as a wealth of background material to help you understand the security foundations on which cryptlib is built.

One of the best books to help you understand the crypto mechanisms used in 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. *Security Engineering: A Guide to Building Dependable Distributed Systems* by Ross Anderson also contains a large amount of useful information and advice on engineering secure systems. *Building Secure Software* by John Viega and Gary McGraw and *Writing Secure Software* by Michael Howard and David LeBlanc contain a wealth of information on safe programming techniques and related security issues.

In addition to this, there are a number of excellent books available that will help you in understanding the cryptography used in 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 that aren't covered by most books on computer security. *Computer Security: Art and Science* by Matt Bishop provides in-depth coverage of all aspects of computer security modelling and design, with a particular emphasis on access control and security models and high-assurance systems.

Installation

This chapter describes how to install cryptlib for a variety of operating systems. Note that embedded systems are usually built using cross-compilers which means that they require specialised handling of things like compiler options and include and library paths. Please refer to the documentation for your development environment for information on how to configure the build process. As a general rule for cross-compilation you can set any platform-specific compiler or build options that you may need via the XCFLAGS variable at the start of the makefile. For example if your build environment doesn't otherwise allow you to specify a non-standard include path and you need to include header files from `/opt/devkit/arm/include` you would modify the XCFLAGS variable to include `-I/opt/devkit/arm/include`.

ARINC 653

ARINC 653 is a safety-critical RTOS, or at least an RTOS kernel accessed via the APEX API. The APEX API provides a minimum set of system services that are then integrated into a higher-level RTOS. cryptlib uses the minimal APEX API to access ARINC 653-level functionality, but also requires additional functionality not provided in ARINC 653 for operations like memory management. Please contact the cryptlib developers before using the ARINC 653 functionality in cryptlib.

AMX

The AMX Multitasking Executive is an RTOS from KADAK Products with development hosted under Unix or Windows. You can build cryptlib for AMX using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make targets for AMX are `target-amx-arm`, `target-amx-mips`, `target-amx-ppc`, and `target-amx-arm`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for AMX, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

BeOS

The BeOS version of cryptlib can be built using a procedure which is identical to that given for Unix on page 20. Any current version of BeOS can build the code directly from the Unix makefile. Old versions of BeOS using the Be development environment will require that you edit the Unix makefile slightly by un-commenting the marked lines at the start of the file.

ChorusOS

ChorusOS is an embedded OS with development hosted under Unix. You can build cryptlib for ChorusOS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for ChorusOS is `target-chorus`, other targets can be added as variations on the existing rule. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for ChorusOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

CMSIS-RTOS/mbed-rtos

CMSIS-RTOS/mbed-rtos is a real-time kernel with development hosted under Windows. You can build cryptlib for CMSIS-RTOS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for CMSIS-RTOS is `target-cmsis`. Details on building and using cryptlib for CMSIS-RTOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

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. You can however treat DOS as an embedded systems environment and use the random seeding capability described in "Random Numbers" on page 297.

eCOS

eCOS is an embedded/real-time OS (RTOS) with development hosted under Unix or Windows. You can build cryptlib for eCOS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for eCOS is `target-ecos-arm`, `target-ecos-ppc`, `target-ecos-sh`, and `target-ecos-x86`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for eCOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Embedded Linux

The embedded Linux version of cryptlib can be built using the standard Linux development tools. Since this environment is identical to the generic Unix one, the installation instructions for Unix on page 20 apply here.

embOS

embOS is an embedded OS from Segger, with development hosted under Unix or Windows. You can build cryptlib for embOS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for embOS is `target-embos`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for embOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

FreeRTOS/OpenRTOS

FreeRTOS/OpenRTOS is an RTOS from Real Time Engineers with development hosted under Windows. You can build cryptlib for FreeRTOS/OpenRTOS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for FreeRTOS/OpenRTOS is `target-freertos-arm`, `target-freertos-mb`, and `target-freertos-ppc`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for FreeRTOS/OpenRTOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

µITRON/T-Kernel

µITRON and T-Kernel are embedded/real-time OSes (RTOS) with development usually hosted under Unix or a Unix-like OS. You can build cryptlib for µITRON or T-Kernel using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for µITRON is `target-itron` and for T-Kernel is `target-tkernel`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for µITRON and T-Kernel, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Mongoose OS

Mongoose OS is an IoT development framework from Cesanta Software with development hosted under Unix or Windows. You can build cryptlib for Mongoose OS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for Mongoose OS is `target-mgos`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for Mongoose OS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

MQX

MQX is an RTOS from Precise Software Technologies and others with development hosted under Unix or Windows. You can build cryptlib for MQX using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for MQX is `target-mqx`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for MQX, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

MVS

The MVS version of cryptlib can be built using the standard IBM C/C++ compiler and accompanying tools. Since this environment is very similar to the Unix one, the installation instructions for Unix on page 20 apply here also. Note that PTF UQ50384 (which fixes a bug in the macro version of the `strcat` function as described in APAR PQ43130) is required if you're using the V2R10 C/C++ compiler.

You can control the use of ddnames with the DDNAME_IO define. If DDNAME_IO is defined when building the code, cryptlib will use ddnames for all I/O, and user options will be saved in dynamically allocated datasets userid.CRYPTLIB.filename. If DDNAME_IO is not defined when building the code, cryptlib will use HFS for all I/O, and user options will be saved in $HOME/.cryptlib.

After you've built cryptlib, you should run the self-test program to make sure that everything is working OK. You can use the ussalloc USS shell script to allocate MVS data sets for testlib, and the usscopy shell script to copy the files in the test directory to the MVS data sets allocated with ussalloc. testlib.jcl is the JCL needed to execute testlib.

Nucleus

Nucleus is an RTOS from Mentor Graphics with development hosted under Unix or Windows. You can build cryptlib for Nucleus using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for Nucleus is `target-nucleus`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for Nucleus, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

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 that everything is working OK.

If you're using 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.

OS X and iOS

The standard Macintosh build environment uses Apple's Mac OS X command-line developer tools, driven by the standard makefile, for which the instructions in the section on building cryptlib for Unix on page 20 apply. Building for iOS works similarly to the OS X build, with the make target being given as `target-ios`. If necessary you can override auto-detection of the iOS SDK type by giving the make target type as `target-ios6`, `target-ios7`, `target-ios8`, or `target-ios9`.

OSEK/AUTOSAR

OSEK is an open-standard RTOS, sometimes called OSEK/VDX, also used by the AUTOSAR consortium, with development hosted under Unix or Windows. For simplicity it's referred to as OSEK in the documentation. You can build cryptlib for OSEK using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for OSEK is `target-osek`, other targets can be added as variations on the existing rules. Due to the wide variation of AUTOSAR configurations and environments it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for OSEK, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

PalmOS

PalmOS is the operating system for the Palm series of PDAs, with development hosted under Unix or Windows. You can build cryptlib for PalmOS using the PalmOS 6 SDK and the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for the PalmOS SDK is `target-palmos` and for the alternative PRC development tools is `target-palmos-prc`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for PalmOS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

QNX Neutrino

The QNX Neutrino version of cryptlib can be built using the standard QNX development tools. Since this environment is identical to the generic Unix one, the installation instructions for Unix on page 20 apply here.

Quadros

Quadros is an RTOS from Quadros Systems with development hosted under Unix or Windows. You can build cryptlib for Quadros using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for Quadros is `target-quadros`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for Quadros, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

RIOT-OS

RIOT is an IoT-targeted RTOS with development hosted under Unix or Windows. You can build cryptlib for RIOT using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for RIOT is `target-riot`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for RIOT, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

RTEMS

The Real-Time Operating System for Multiprocessor Systems (RTEMS) is an RTOS from OAR Corporation with development hosted under Unix or Windows. You can build cryptlib for RTEMS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for RTEMS is `target-rtems-arm`, `target-rtems-mips`, `target-rtems-ppc`, and `target-rtems-x86`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for RTEMS, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

SMX

The SMX RTOS is a real-time OS (RTOS) with development hosted under Unix or Windows. You can build cryptlib for SMX using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for SMX is `target-smx`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for SMX, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Tandem

The Tandem version of cryptlib can be built using the standard c89 compiler and accompanying tools under the OSS environment. Since this environment is very similar to the Unix one, the installation instructions for Unix on page 20 apply here also. The default target is Tandem OSS, you can re-target the built for NSK using the `-Wsystype=guardian` directive in the makefile.

The Guardian sockets implementation changed in newer releases of the OS. Older releases required the use of non-standard nowait sockets handled via AWAITIOX() instead of the standard BSD sockets interface. If you're running an older version of the OS and need to use any of the secure networking protocols such as TLS, SSH, CMP, SCEP, RTCS, OCSP, or SCVP, you'll need to use cryptlib's alternative network data-handling strategy described in "Network Issues" on page 130.

Telit

Telit is an IoT/M2M platform from Telit Communications with development hosted under Unix or Windows. You can build cryptlib for Telit using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for Telit is `target-telit`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for Telit, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

ThreadX

ThreadX (and optionally FileX) is an RTOS from Express Logic with development hosted under Unix or Windows. You can build cryptlib for RTEMS using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for ThreadX is `target-threadx-arm`, `target-threadx-mb`, `target-threadx-mips`, `target-threadx-ppc`, and `target-threadx-x86`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for ThreadX, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

µC/OS-II

µC/OS-II is an embedded/ RTOS from Micrium with development usually hosted under Windows. You can build cryptlib for µC/OS-II using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for µC/OS-II is `target-ucos-arm`, `target-ucos-ppc`, and `target-ucos-x86`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for µC/OS-II, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

uClinux

uClinux is a real-mode/embedded version of Linux with development hosted under Unix. You can build cryptlib for uClinux using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for uClinux is `target-uclinux`, so you'd build cryptlib with make target-uclinux. Details on building and using cryptlib for uClinux, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

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 testlib, or `make stestlib` to build the shared-library self-test program stestlib. This will run fairly extensive self-tests of cryptlib that you can run after you've built it to make sure that everything is working OK. testlib needs to be run from the cryptlib root directory (the one that the main data files are in) since it uses a large number of pre-generated data files that are located in a subdirectory below this one. 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 (or an OS-specific equivalent) to make sure that the shared library is used.

If you're using the statically-linked form of cryptlib in your application rather than the shared library, you'll probably need to link in additional (system-specific) static libraries to handle threads, network access, and system-specific odds and ends. The makefile contains a list of the needed additional libraries, ordered by system type and version. The shared-library version of cryptlib doesn't require these additional libraries to be linked in, since the references are automatically resolved by the OS.

By default the makefile will build a version of cryptlib that's optimised for the system that it's running on. If you're building cryptlib for use on a wide range of systems including ones with CPUs older than the one used by the current system, you should use `make generic` rather than `make`. This will fall back to lowest-common-denominator code generation that's portable across a wide range of older CPUs, at the cost of a slight loss in performance.

If your system doesn't come pre-configured with a `/dev/random`, EGD, or PRNGD-style style randomness driver (which continually accumulates random data from the system), you may want to download one and install it, since cryptlib will make use of it for gathering entropy. cryptlib has a built-in randomness polling subsystem so it will function without an external randomness driver, but it never hurts to have one present to supplement the internal entropy polling.

If you're using a key database or certificate store, you need to enable the use of the appropriate interface module for the database backend. Details are given in "Key Database Setup" on page 24. 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're using an LDAP directory, you need to install the required LDAP client library on your system, enable the use of LDAP using the USE_LDAP define before you build cryptlib, and link the LDAP client library into your executable (on most systems the cryptlib build scripts will take care of this automatically). If you don't enable the use of an LDAP directory interface, the self-test code will issue a warning that no LDAP directory interface is present and continue without testing the LDAP interface.

If you're using special encryption hardware or an external encryption device such as a 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.

If your application forks, you shouldn't need to take any special actions for cryptlib beyond the usual precautions with forking a process. In particular forking a process that contains multiple threads has system-specific semantics, with the behaviour depending on whether the system implements *fork1* or *forkall* behaviour. With *fork1* behaviour (the Posix default), only the thread that calls `fork()` is copied to the child. With *forkall*, all threads in the process are copied. The *fork1* behaviour can lead to deadlock if a thread other than the one that called `fork()` holds a lock, since the fact that it's not copied to the child means that it'll never be released. You can work around this with `pthread_atfork()` to handle lock management, but a better approach is to simply not mix threads and forking unless you follow the `fork()` with an `exec()`. Note that this isn't a cryptlib issue, it's specific to the interaction of `fork()` and threads.

For any common Unix system, cryptlib will build without any problems, but in some rare cases you may need to edit random/unix.c and possibly io/file.h and io/tcp.h if you're running an unusual Unix variant that puts include files in strange places or has broken Posix or sockets support.

VDK

The Visual DSP++ Kernel (VDK) from Analog Devices is a kernel for AD processors with development hosted under Windows. You can build cryptlib for VDK using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for VDK is `target-vdk`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for VDK, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

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.

VxWorks

VxWorks is an embedded/ RTOS from VxWorks with development hosted under Unix or Windows. You can build cryptlib for VxWorks using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for VxWorks is `target-vxworks-arm`, `target-vxworks-mb`, `target-vxworks-mips`, `target-vxworks-ppc`, and `target-vxworks-x86`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for VxWorks, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Windows

The cryptlib distribution ships with pre-built 32- and 64-bit cryptlib DLLs built using the default configuration (see "Configuration Issues" on page 25). You can use these directly in your application, or if you prefer to build your own copies from the source code or want to build a customised version then you can build the 32- and 64-bit cryptlib DLLs using the crypt32 project file for Visual Studio (there's also a legacy Visual C++ 6 .dsw project file present, but the supported configuration uses Visual Studio). Once the DLL has been built, either as cl32.dll or cl64.dll, the test32 project file will build the cryptlib self-test program test32 or test64, which is a console application. You can run this after you've built cryptlib to make sure that everything is working OK. The self-test program needs to be run from the cryptlib root directory (the one that the main data files are in) since it uses a large number of pre-generated data files that are located in a subdirectory below this one.

If you're using a key database or certificate store, you need to set up an ODBC data source for this. Details are given in "Key Database Setup" on page 24.

If you're using special encryption hardware or an external encryption device such as a PC card, USB token, 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 " on page 282. cryptlib will automatically detect and use any devices that it recognises and that 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.

Personal firewall products from some vendors can interfere with network operations for devices other than standard web browsers and mail clients. If you're experiencing odd behaviour when using cryptlib for network operations (for example you can connect but can't exchange data, or you get strange error messages when you connect), you can try temporarily disabling the personal firewall to see if this fixes the problem. If it does, you should contact the personal firewall vendor to fix their product, or switch to a different product.

If you're using Borland C++ rather than Visual Studio, 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.

To install the ActiveX control, put the cryptlib DLL and the ActiveX wrapper clcom.dll into the Windows system directory and register the ActiveX wrapper with:

```
regsvr32 clcom.dll
```

To use the ActiveX control with Visual Basic, use Project | Reference to add clcom.dll, after which VB will recognise the presence of the ActiveX wrapper.

Windows CE / Pocket PC / SmartPhone / Embedded

The cryptlib DLL for Windows CE/PocketPC/SmartPhone/Embedded can be built using the crypt32ce project file, which is for version 3 or 4 of the eMbedded Visual C++ compiler. Once the DLL has been built, the test32ce project file will build the cryptlib self-test program test32ce, which is a (pseudo-)console application that produces its output on the debug console. You can run this after you've built cryptlib to make sure that everything is working OK. test32ce needs to be run from the cryptlib root directory (the one that the main data files are in) since it uses a large number of pre-generated data files that are located in a subdirectory below this one.

The cryptlib Windows CE self-test uses the 'Storage Card' pseudo-folder to access the files needed for the self-test. Depending on the system setup, you need to either copy the files to the storage card or (the easier alternative) use folder sharing to access the directory containing the test files. From the Windows CE menu, select Folder Sharing and share the testdata subdirectory, which will appear as \\Storage Card\ on the Windows CE device.

Windows CE is a Unicode environment, which means that all text strings are passed to and from cryptlib as Unicode strings. For simplicity the examples in this manual are presented using the standard `char` data type used on most systems, however under Windows CE all character types and strings are Unicode in line with standard Windows CE practice. When you're using the examples, you should treat any occurrence of characters and strings as standard Unicode data types.

A few older versions of eVC++ for some platforms don't include the ANSI/ISO C standard time.h header, which is a required file for a conforming ANSI/ISO C compiler. If you have a version of eVC++ that doesn't include this standard header, you need to add it from another source, for example an eVC++ distribution that does include it or the standard (non-embedded) VC++ distribution.

When compiling cryptlib under eVC++ 4.0 for the Arm architecture with optimisation enabled, a compiler bug may prevent three files from compiling. If you get an internal compiler error trying to compile context/kg_rsa.c or misc/base64.c, you can work around the problem by disabling optimisation using `#pragma optimize( "g", off )` / `#pragma optimize( "g", on )` around the functions `initCheckRSAkey()` in kg_rsa.c and `adjustPKIUserValue()` in base64.c.

XMK

The Xilinx Microkernel (XMK) is an RTOS from Xilinx with development hosted under Unix or Windows. You can build cryptlib for XMK using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for XMK is `target-xmk-mb` for the MicroBlaze core and `target-xmk-ppc` for the PowerPC core, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for XMK, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Zephyr

Zephyr is an RTOS from Wind River Systems with development hosted under Unix or Windows. You can build cryptlib for Zephyr using the cross-compilation capabilities of the standard makefile, see the entry for Unix on page 20 for more details on working with the makefile. The make target for Zephyr is `target-zephyr`, other targets can be added as variations on the existing rules. Due to the wide variation of environments and toolchains it may be necessary to modify the configuration slightly to build the code. Details on building and using cryptlib for Zephyr, and on embedded cryptlib in general, are given in "Embedded Systems" on page 324.

Other Systems

cryptlib should be fairly portable to other systems, the only part that needs special attention is the randomness-gathering in random/*os_name*.c (cryptlib won't work without this, the code will produce a link 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"). In addition the filesystem I/O code in io/file.c may need system-specific code and definitions added to it if the system you're running on doesn't use a standard form of file I/O, for example a system that has its own file I/O layer that isn't compatible with standard models or one that doesn't have file I/O at all such as an embedded device that uses flash memory for storage.

To find out what to compile, look at the Unix makefile, which contains all of the necessary source files (the `group_name_OBJS` dependencies) and compiler options. Link all of these into a library (as the makefile does) and then compile and link the modules in the test subdirectory with the library to create the self-test program. There is additional assembly-language code included that 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 key database or certificate store, 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) that 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 uses two sources called testkeys and testcertstore during the self-test procedure, and will create these itself if possible). In addition you may need to set up other parameters like the server that the database is located on and other access information. Once the data source is set up, you can access it as a CRYPT_-KEYSET_DATABASE keyset using the name that you've assigned to it.

Under Unix or similar systems the best way to work with a key database or certificate store is to use the ODBC interface, either via a layered driver such as unixODBC or iODBC, or directly via interfaces such as MyODBC. Alternatively, you can use the cryptlib generic database interface to compile database-specific support code directly into cryptlib, or the database network plugin capability to make a network connection to a database server such as IBM DB2, Informix, Ingres, Oracle, Postgres, or Sybase.

The easiest interface to use is the ODBC one, which hides all of the low-level database interface details. The ODBC configuration process follows the same pattern as the one given above for ODBC under Windows, with OS-specific variations depending on the platform that you're running it under. You can enable the use of the ODBC interface using the USE_ODBC define before you build cryptlib, and if you're not using Windows (which uses dynamic binding to the ODBC interface) you need to link the ODBC client library into your executable (on most systems the cryptlib build scripts will take care of this automatically).

For Unix and Unix-like systems the two most common ODBC implementations are unixODBC and iODBC, although a variety of other products are also available, and some databases have native ODBC support, examples being MySQL (via MyODBC) and IBM DB2. These interfaces support a wide range of commercial database including AdabasD, IBM DB2, Informix, Ingres, Interbase, MySQL, Oracle, Postgres, and Sybase. In its simplest form, you can use the freely-available SQLite to set up a test database under Linux with:

```
sudo apt-get install sqlite3 libsqlite3-dev libsqliteodbc unixodbc

cat > ~/.odbc.ini

[testkeys]
Description = cryptlib test keyset
Driver = SQLite3
Database = /[...]/testkeys.sqlite
Timeout = 1000
^D
```

If you're going to use the test certificate store you can do the same thing with the database name "testcertstore". Then to verify that the setup worked:

```
isql -m20 testkeys
```

You can now run the cryptlib self-test code, and it will use the SQLite database for its certificates. After running the test code, you can display the stored certificates with:

```
isql -m20 testkeys
select CN, email from certificates;
```

unixODBC can also use the ODBCConfig GUI application to configure data sources and drivers in a manner identical to the standard Windows interface, and also provides the odbcinst CLI utility to configure data sources and drivers. odbcinst can be used to automatically install and configure database drivers for ODBC using template files that contain information about the driver such as the location of the driver binaries, usually somewhere under /usr/local. For example to configure the Oracle drivers for ODBC using a prepared template file you'd use:

```
odbcinst -i -d -f oracle.tmpl
```

iODBC provides drivers as platform-specific binaries that are installed using the iODBC installation shell scripts. See the documentation for the particular ODBC interface that you're using for more information on installation and configuration issues.

Configuration Issues

For compatibility with existing deployed code, cryptlib supports a wide variety of encryption, signature, and hash algorithms, key types, and security mechanisms. Some of these backwards-compatible items are obsolete, unsound, or even entirely broken. For this reason the encryption algorithm RC4 is disabled by default. If you want to enable these obsolete and insecure items, you can do so via the cryptlib configuration file misc/config.h. Note that by enabling these unsafe items, you are voiding cryptlib's security guarantees and agree to indemnify the cryptlib authors against any claims or losses from any problems that may arise. In other words you really, really shouldn't do this.

Optional cryptlib Components

cryptlib includes a number of optional facilities that provide extra functionality but that because of the high level of complexity of the underlying code that they use and their corresponding large attack surface, or because they are little-used in practice, are disabled by default. If you want to use any of these facilities then you'll need to explicitly enable them via the cryptlib configuration build file misc/config.h, by setting them in the makefile/project file in the standard CFLAGS variable defined at the start of the makefile, or by specifying them using the BUILDOPTS variable for make when you build cryptlib:

```
make BUILDOPTS="-DUSE_xxx -DUSE_yyy"
```

You should only enable these facilities if you absolutely need them and, in the case of high-risk protocols like LDAP, are certain that the underlying subsystems that they use are secure and robust.

Certificate Capabilities

Certificate standards bodies have, over the years, created a vast array of certificate capabilities, most of which are ignored by implementations, and some of which produce highly counterintuitive behaviour comprehensible only to the people who created the standards (and sometimes not even to them). In order to reduce code size, the complexity of the certificate-processing code, and the unexpected side-effects caused by some of the more obscure certificate-processing requirements, cryptlib by default provides a standard level of certificate processing that excludes the more obscure and peculiar portions of PKI standards. This corresponds to CRYPT_-COMPLIANCELEVEL_STANDARD as discussed in "Certificate Compliance Level Checking" on page 214.

If you want to enable processing of obscure certificate attributes and handling of peculiar processing requirements you can change the level of processing that cryptlib applies to certificates. Defining `USE_CERTLEVEL_PKIX_PARTIAL` either via the makefile/project file or in misc/config.h sets the certificate-processing level to CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL, and defining `USE_CERTLEVEL_PKIX_FULKL` sets the certificate-processing level to CRYPT_COMPLIANCELEVEL_PKIX_FULL. Note that these extended certificate-processing levels significantly increase the certificate-handling code size and complexity and processing requirements, with little obvious benefit since few implementations ever use this level of certificate detail.

DNS SRV

The DNS SRV facility is used to implement the automatic PKI server detection facility for cryptlib's plug-and-play PKI facility (this is an optional capability that can be used if it's not possible to configure a PKI server name as part of the standard initialisation process, and isn't otherwise required for plug-and-play PKI). The underlying DNS management (particularly under Unix, where the interface is quite primitive) is complex and has a potentially large attack surface, so it's disabled by default. If you're certain that you really need DNS SRV access then you can enable it by defining `USE_DNSSRV`.

EAP / RADIUS

EAP /RADIUS is an additional protocol that can be run with TLS. If you want to use EAP with cryptlib, you can enable its use by defining `USE_EAP`.

LDAP

LDAP is an extraordinarily complex protocol (the "L" in its name is deceptive advertising, it should be called HDAP) with a very large attack surface, so it's disabled by default. If you're certain that you really need LDAP access then you can enable it by defining `USE_LDAP`.

SSH Subsystems

The SSH protocol provides a Swiss-army-knife set of mechanisms covering a range of special-purpose facilities like port forwarding, connection multiplexing, custom subsystems, and even user-defined channel types. These facilities are extremely complex and open to abuse by attackers, and are disabled by default. The standard cryptlib configuration supports straightforward encrypted-tunnel SSH sessions, which is what most people use SSH for. If you're certain that you really need the various SSH extended facilities like port forwarding, connection multiplexing, and subsystems, then you can enable them by defining `USE_SSH_EXTENDED`.

WebSockets

WebSockets are an additional protocol that can be run with TLS. If you want to use WebSockets with cryptlib, you can enable their use by defining `USE_WEBSOCKETS`.

X.500 DNs in string form

cryptlib supports the ability to specify X.500 Distinguished Names for certificates as free-format strings, allowing the creation of arbitrarily complex and non-standard

DNs. This is a somewhat dangerous capability both because it allows the creation of arbitrarily awkward and broken DNs, and because it bypasses cryptlib's standard safety checks for DN validity, so it's disabled by default. If you're certain that you really need X.500 free-format string access then you can enable it by defining `USE_CERT_DNSTRING`.

Customised and Cut-down cryptlib Versions

In some cases you may want to customise the cryptlib build or create a cut-down version that omits certain capabilities in order to reduce code size for constrained environments. You can do this by editing the configuration build file misc/config.h, which allows almost every part of cryptlib's functionality to be selectively enabled or disabled (some functionality is used by all of cryptlib and can't be disabled). Each portion of functionality is controlled by a USE_*name* define, by undefining the value before you build cryptlib the named functionality will be removed. More details on tuning cryptlib's size and capabilities (particularly for use in embedded systems) is given in "Embedded Systems" on page 324.

If you only want to use cryptlib for one particular purpose and don't feel like manually enabling all of the necessary configuration options then you can make use of the following predefined profiles to enable only the functionality that you need:

Profile	Description
CONFIG_PROFILE_SMIME	Enable only the capabilities needed for S/MIME envelopes.
CONFIG_PROFILE_PGP	Enable only the capabilities needed for PGP envelopes.
CONFIG_PROFILE_SSH	Enable only the capabilities needed for SSH sessions.
CONFIG_PROFILE_TLS	Enable only the capabilities needed for TLS sessions.

Note that these predefined profiles will enable only the functionality that you need for the particular application and no more. For example SSH doesn't use X.509 certificates so CONFIG_PROFILE_SSH won't enable the use of certificates, if you're using a keyset to store an SSH server's private (and public) key then you need to store the public portion as a raw public key rather than a public-key certificate because the latter can't be processed by the SSH-only profile.

Custom Cryptographic HALs

Alongside the built-in software- and hardware-based cryptography, it's also possible to use a custom crypto HAL with cryptlib that substitutes non-public algorithms for the standard ones used by cryptlib. This is different to the use of cryptographic hardware devices described in "Encryption Devices and Encryption Hardware" on page 282 in that the crypto HAL takes over all cryptographic operations performed by cryptlib.

The use of custom crypto HALs is controlled by two defines, `CONFIG_CRYPTO_HW1` and `CONFIG_CRYPTO_HW2` that you can define either via the makefile/project file or in misc/config.h. Defining `CONFIG_CRYPTO_HW1` replaces all of the crypto algorithms with custom ones that you need to provide via the interface defined in device/hw_template.c, while defining `CONFIG_CRYPTO_HW2` replaces the algorithms and higher-level crypto mechanisms such as PKCS #1 signing and encryption with custom ones that you need to provide. Using a custom crypto HAL requires working with the internals of cryptlib, please contact the cryptlib developers if you intend to do this.

Note that using a custom crypto HAL can affect the operation of some public-key algorithms implemented in crypto devices like PKCS #11 tokens, since cryptlib relies on the presence of native public-key operations to supplement what's implemented by the PKCS #11 token. In particular the DH/DSA and ECDH/ECDSA algorithms as implemented in PKCS #11 don't provide full key generation/key load support, so cryptlib implements the missing portions in native code. If you disable the native code by switching to a custom crypto HAL and your HAL doesn't implement support for the DH/DSA or ECDH/ECDSA operations needed by PKCS #11 then attempting to use those algorithms in a PKCS #11 token will fail with a CRYPT_ERROR_-NOTAVAIL error even if the token otherwise appears to support the algorithm.

Debug vs. Release Versions of cryptlib

cryptlib can be built in one of two forms, a debug version and a release version. The main difference between the two is that the release version is built with the NDEBUG value defined, which disables the large number of internal consistency checks that are present in the debug build of cryptlib. These consistency checks are used to catch conditions such as inappropriate error codes being returned from internal functions, invalid data values being passed to functions inside cryptlib, configuration errors, and general sanity checks that ensure that everything is operating as it should. If one of these internal checks is triggered, cryptlib will throw an exception and display an error message indicating that an assertion in the code has failed. These assertions are useful for tracking down areas of code that may need revision in later releases.

If you don't want to see these diagnostic messages, you should build cryptlib with the NDEBUG value defined (this is the default under Unix and is done automatically under Windows when you build a release version of the code with Visual Studio). Building a version in this manner will disable the extra consistency checks that are present in the debug version so that, for example, error conditions will be indicated by cryptlib returning an error code for a function call rather than throwing an exception. This will have the slight downside that it'll make tracking the exact location of a problem a bit more complex, since the error code which is returned probably won't be checked until the flow of execution has progressed a long way from where the problem was detected. On the other hand the release version of the code is significantly smaller than the debug version.

As always, if you're working with a debug build of the code and perform an operation that triggers an internal consistency check you should report the details and the code necessary to recreate it to the cryptlib developers in order to allow the exception condition to be analysed and corrected.

cryptlib Version Information

cryptlib uses 3-digit version numbers, available at runtime through the configuration options CRYPT_OPTION_INFO_MAJORVERSION, CRYPT_OPTION_INFO_-MINORVERSION, and CRYPT_OPTION_INFO_STEPPING, and at compile time through the define CRYPTLIB_VERSION. CRYPTLIB_VERSION contains the current version as a 3-digit decimal value with the first digit being the major version number (currently 3), the second digit being the minor version number, and the third digit being the update or stepping number. For example, cryptlib version 3.2.1 would have a CRYPTLIB_VERSION value of 321.

All cryptlib releases with the same stepping version number are binary-compatible. This means that if you move from (for example) cryptlib version 3.2.1 to 3.2.2, all you need to do is replace the cryptlib DLL or shared library to take advantage of new cryptlib features and updates. All cryptlib releases with the same minor version number are source-compatible, so that if you move from (for example) 3.2.1 to 3.3.5, you need to recompile your application to match new features in cryptlib.

cryptlib and Cloud Computing

Cloud computing is fundamentally incompatible with secure crypto operations. If you're running your crypto on someone else's computer then you have no way of telling what they're doing with your crypto or your keys. By extension if you're running your crypto on a platform that's shared with hostile processes that can access information like CPU statistics and cache information then you can't guarantee the safety of your crypto keys. Although cryptlib employs countermeasures to try and reduce the threat, it's not possible to guarantee the safety of your crypto or keys when they're exposed on someone else's hardware or to someone else's software (the same applies to anyone else's crypto in such an environment, at best you can try and obscure or hide your crypto operations and keys and hope that no-one figures out how to get at them). In other words if you're running your crypto in a cloud or shared computing environment then your security guarantees are void.

Similarly, if you allow someone to probe the system that you're running cryptlib on with measuring equipment that can analyse internal electrical or similar signals then your security guarantees are void. As with the cloud/shared computing problem, this issue isn't specific to cryptlib but affects all crypto implementations, no matter what their salespeople may tell you.

The solution to these issues is simple, don't run your sensitive crypto operations on systems controlled by others or that allow hostile code to run alongside, and attack, your crypto code, and don't allow them to probe your systems with measuring equipment to capture its internal operations.

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 a call to initialise the algorithm capability information to device/system.c alongside the existing algorithm initialisation, and then provide your implementation of the algorithm to compile and link into cryptlib. When you rebuild cryptlib with the preprocessor define USE_VENDOR_ALGOS set, the new algorithm types will be included in cryptlib's 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 that 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 that 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.

Encryption contexts are the action objects used by cryptlib. 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 an envelope or session 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. Session objects work in a similar manner, but are used to encapsulate a secure session with a remote client or server rather than a local data transformation. 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. After the basic concepts behind enveloping have been explained, more advanced techniques such as public-key based enveloping and digital signature enveloping for S/MIME and PGP are covered.

Session objects are very similar to envelope objects except that they represent a communications session with a remote client or server. The next section covers the use of session objects for protocols such as TLS, and SSH to secure communications or work with protocols such as CMS, SCEP, RTCS, OCSP, SCVP, and TSP that handle functions such as certificate status information and timestamping.

The use of public keys for enveloping requires the use of key management functions, and the next section covers key generation and storing and retrieving keys from keyset objects and crypto devices. The public portions of public/private key pairs are typically managed using X.509 certificates and certificate revocation lists. The next sections cover the management of certificates including certificate issue, certificate status checking, and certificate revocation list (CRL) creation and checking, as well as the full CA management process. This covers the full key life cycle from creation through certification to revocation and/or destruction.

So far all of the objects that have been covered are high-level container objects. The next section covers the creation of low-level action objects that 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 sections explain how to apply the action objects to data and cover the process of encryption, key exchange, and signature generation and verification, working at a much lower level than the enveloping or session interface.

The next sections cover certificates and certificate-like objects in more detail than the earlier ones, covering such topics as DN structures, certificate chains, trust management, and certificate extensions. This deals with certificates at a very low level at which they're rather harder to manage than with the high-level certificate management functions.

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

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. Using envelopes and session objects 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 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:

High-level Interface

This interface requires no knowledge of encryption and digital signature techniques, and is easiest for use with languages like Visual Basic and Java that 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. Because of its simplicity and ease of use, it's strongly recommended that you use this interface if at all possible.

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's better suited for applications that require high-speed data encryption, or encryption of many small data packets (such as an encrypted terminal session). The mid-level interface provides services such as routines to export and import encrypted keys and to create and check digital signatures. The high-level interface is built on top of this interface.

Low-level 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 reason for using these low-level routines is if you need them as building blocks for your own custom encryption protocol. Note though that cryptlib is designed to benefit the application of encryption in standard protocols and not the raw use of crypto in home-made protocols. Getting such security protocols right is very difficult, with many "obvious" and "simple" approaches being quite vulnerable to attack. This is why cryptlib encourages the use of vetted security protocols, and does not encourage roll-your-own security mechanisms. In particular if you don't know what PKCS #1 padding is, what CBC does, or why you need an IV, you shouldn't be using this interface.

The low-level interface serves as an interface to a range of plug-in encryption modules that 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 interface between the underlying encryption routines and the user software. The mid-level interface is built on top of this interface.

Objects and Interfaces

The cryptlib object types are as follows:

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

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 that 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 that you can get, set, 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 as follows:

Type	Description
CRYPT_ATTRIBUTE_*name*	Generic attributes that apply to all objects.
CRYPT_CERTINFO_*name*	Certificate object attributes.
CRYPT_CTXINFO_*name*	Encryption context attributes.
CRYPT_DEVINFO_*name*	Crypto device attributes.
CRYPT_ENVINFO_*name*	Envelope attributes.
CRYPT_KEYINFO_*name*	Keyset attributes.
CRYPT_OPTION_*name*	cryptlib-wide configuration options.
CRYPT_PROPERTY_*name*	Object properties.

Type	Description
CRYPT_SESSINFO_*name*	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 using the ANSI/ISO standard seconds-since-1970 format.

Interfacing with cryptlib

All necessary constants, types, structures, and function prototypes are defined in a language-specific header file as described below. You need to use these files for each module that makes use of cryptlib. Although many of the examples given in this manual are for C/C++ (the more widely-used ones are given for other languages as well), they apply equally for the other languages.

All language bindings for cryptlib are provided in the bindings subdirectory. Before you can use a specific language interface, you may need to copy the file(s) for the language that you're using into the cryptlib main directory or the directory containing the application that you're building. Alternatively, you can refer to the file(s) in the bindings directory by the absolute pathname.

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 after you've finished using cryptlib. **cryptInit** initialises 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. However you should only call **cryptEnd** once when you've finished using cryptlib.

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 that it can erase any sensitive data that 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.

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

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

Randomness polls are described in more detail in "Random Numbers" on page 297. The randomness poll executes asynchronously, so it won't stall the rest of your code while it's running. The one possible exception to this polling on start-up is when you're using cryptlib as part of a larger application where you're not certain that cryptlib will actually be used. For example a PHP script that's run repeatedly from the command line may only use the encryption functionality on rare occasions (or not at all), so that it's better to perform the slow poll only when it's actually needed rather than unconditionally every time the script is invoked. This is a somewhat special case though, and normally it's better practice to always perform the slow poll on start-up.

As the text above mentioned, you should initialise cryptlib when your program first starts and shut it down when your program is about to exit, rather than repeatedly starting cryptlib up and shutting it down again each time you use it. Since cryptlib consists of a complete crypto operating system with extensive initialisation, internal security self-tests, and full resource management, repeatedly starting and stopping it will unnecessarily consume resources such as processor time during each initialisation and shutdown. It can also tie up host operating system resources if the host contains subsystems that leak memory or handles (under Windows, ODBC and LDAP are particularly bad, with ODBC leaking memory and LDAP leaking handles. DNS is also rather leaky — this is one of the reasons why programs like web browsers and FTP clients consume memory and handles without bounds). To avoid this problem, you should avoid repeatedly starting up and shutting down cryptlib:

Right **Wrong**

```
cryptInit();            serverLoop:
serverLoop:               cryptInit();
  process data;           process data;
cryptEnd();               cryptEnd();
```

C / C++

To use cryptlib from C or C++ you would use:

```
#include "cryptlib.h"

cryptInit();

/* Calls to cryptlib routines */

cryptEnd();
```

C# / .NET

To use cryptlib from C# / .NET, add cryptlib.cs to your .NET project and the cryptlib DLL to your path, and then use:

```
using cryptlib;

crypt.Init();

// Calls to cryptlib routines

crypt.End();
```

If you're using a .NET language other than C# (for example VB.NET), you'll need to build cryptlib.cs as a class library first. From Visual Studio, create a new C# project of type Class Library, add cryptlib.cs to it, and compile it to create a DLL. Now go to your VB project and add the DLL as a Reference. The cryptlib classes and methods will be available natively using VB (or whatever .NET language you're using).

All cryptlib functions are placed in the `crypt` class, so that standard cryptlib functions like:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_KEYSIZE, 1024 / 8 );
```

become:

```
crypt.SetAttribute( cryptContext, crypt.CTXINFO_KEYSIZE, 1024 / 8 );
```

In general when calling cryptlib functions you can use Strings wherever the cryptlib interface requires a null-terminated C string, and byte arrays wherever the cryptlib interface requires binary data.

Instead of returning a status value like the native C interface, the .NET version throws `CryptException` for error status returns, and returns integer or string data return values as the return value:

```
value = crypt.GetAttribute( cryptContext, crypt.CTXINFO_ALGO );
stringValue = crypt.GetAttributeString( cryptContext,
   crypt.CTXINFO_ALGO_NAME );
```

Delphi

To use cryptlib from Delphi, add the cryptlib DLL to your path and then use:

```
implementation
   uses cryptlib;

   cryptInit;

   { Calls to cryptlib routines }

   cryptEnd;
end;
```

The Delphi interface to cryptlib is otherwise mostly identical to the standard C/C++ one.

Java

To use cryptlib with Java, put cryptlib.jar on your classpath and use `System.-LoadLibrary()` to load the cryptlib shared library. You can then use:

```
import cryptlib.*;

class Cryptlib
   {
   public static void main( String[] args )
      {
      System.loadLibrary( "cl" );    // cryptlib library name

      try
         {
         crypt.Init();

         //Calls to cryptlib routines

         crypt.End();
         }
      catch( CryptException e )
         {
         // cryptlib returned an error
         e.printStackTrace();
         }
      }
   };
```

All cryptlib functions are placed in the `crypt` class, so that standard cryptlib functions like:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_KEYSIZE, 1024 / 8 );
```

become:

```
crypt.SetAttribute( cryptContext, crypt.CTXINFO_KEYSIZE, 1024 / 8 );
```

In general when calling cryptlib functions you can use Java strings wherever the cryptlib interface requires a null-terminated C string, and Java byte arrays wherever the cryptlib interface requires binary data. In addition as of JDK 1.4 there is a `nio.ByteBuffer` class that can be “direct”, which provides a more efficient alternative to standard byte arrays since there’s no need to perform any copying.

Instead of returning a status value like the native C interface, the JNI version throws `CryptException` for error status returns, and returns integer or string data return values as the return value:

```
value = crypt.GetAttribute( cryptContext, crypt.CTXINFO_ALGO );
stringValue = crypt.GetAttributeString( cryptContext,
   crypt.CTXINFO_ALGO_NAME );
```

Python

To build the Python interface to cryptlib, change to the bindings directory and run `python setup.py install`. This will build and install the python.c extension module. In order to do this you'll need to have the Python source code installed. If you get an error like `Python.h: No such file or directory` (followed by a cascade of other errors) then you need to install the Python source in order to build the cryptlib interface. On a Unix platform you may need to create a symlink from cl to the actual shared library before you do this. Once you've done this you can use:

```
from cryptlib_py import *

cryptInit()

# Calls to cryptlib routines

cryptEnd()
```

Starting with Python 3.10, integer handles to cryptlib objects are treated as different types when the object is created and when it's subsequently used. This means that after creating an object you need to cast its integer handle to an integer, so instead of:

```
hashContext =  cryptCreateContext( cryptUser, CRYPT_ALGO_SHA2 )
algoNameSize = cryptGetAttributeString( hashContext,
   CRYPT_CTXINFO_NAME_ALGO, algoName )
```

you need to use:

```
hashContext_object = cryptCreateContext( cryptUser, CRYPT_ALGO_SHA2 )
hashContext = int ( hashContext_object )
algoNameSize = cryptGetAttributeString( hashContext,
   CRYPT_CTXINFO_NAME_ALGO, algoName )
```

Tcl

To use cryptlib from Tcl, you use the Cryptkit extension. Cryptkit is a stubs-enabled extension that can be used with any modern Tcl interpreter (at least, Tcl 8.4 or later). To build Cryptkit you'll need a copy of Tcl that can interpret Starkits, either Tclkit, the single file Tcl/Tk executable available from `http://www.equi4.com/pub/tk`, or ActiveTcl from `http://www.activestate.com`. You'll also need to download the Critcl Starkit from `http://mini.net/sdarchive/critcl.kit`, and make sure that the current directory contains cryptlib.h and a copy of the cryptlib static library, named libcl_$platfom.a, where $platform is the current platform name as provided by the Critcl platform command. For example under x86 Linux the library would be called libcl_Linux-x86.a. Then run the following Critcl command:

```
critcl -pkg cryptkit
```

This will leave you with a lib directory containing the information ready for use in any Tcl application. Once you've done this you can use:

```
package require Cryptkit

cryptInit

# Calls to cryptlib routines

cryptEnd
```

Since Tcl objects already contain length information, there's no need to pass length parameters to cryptlib function calls. This applies for the AddCertExtension, CheckSignature, CheckSignatureEx, CreateSignature, CreateSignatureEx, Decrypt, Encrypt, ExportCert, ExportKey, ExportKeyEx, GetCertExtension, ImportKey, PushData, and SetAttributeString functions.

Visual Basic

To use cryptlib from Visual Basic you would use:

```
' Add cryptlib.bas to your project

cryptInit
```

```
' Calls to cryptlib routines

cryptEnd
```

The Visual Basic interface to cryptlib is otherwise mostly identical to the standard C/C++ one.

Return Codes

Every cryptlib function returns a status code to tell you whether it succeeded or failed. If a function executes successfully, it returns CRYPT_OK. If it fails, it returns one of the status values detailed in "Error Handling" on page 305. The sample code used in this manual omits the checking of status values for clarity, but when using cryptlib you should check return values, particularly for critical functions such as any that perform active crypto operations like processing data in envelopes, activating and using secure sessions, signing and checking certificates, and encryption and signing in general.

The previous initialisation code, rewritten to include checking for returned status values, is:

```
int status;

status = cryptInit();
if( status != CRYPT_OK )
   /* cryptlib initialisation failed */;

/* Calls to cryptlib routines */

status = cryptEnd();
if( status != CRYPT_OK )
   /* cryptlib shutdown failed */;
```

The C/C++ versions of cryptlib provide the `cryptStatusOK()` and `cryptStatusError()` macros to make checking of these status values easier. The C#, Java, and Python versions throw exceptions of type `CryptException` instead of returning error codes. These objects contain both the status code and an English error message. In C# the `CryptException` class has Status and Message properties:

```
try
   {
   crypt.Init();

   crypt.End();
   }
catch( CryptException e )
   {
   int status = e.Status;
   String message = e.Message;
   }
```

In Java the `CryptException` class has `getStatus()` and `getMessage()` accessors:

```
try
   {
   crypt.Init();

   crypt.End();
   }
catch( CryptException e )
   {
   int status = e.getStatus();
   String message = e.getMessage();
   }
```

In Python the exception value is a tuple containing the status code, then the message:

```
try:
   cryptInit()

   cryptEnd()
except CryptException, e:
   status, message = e
```

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 that 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 different attributes.

The Visual Basic version is:

```
Dim emailAddress as String
Dim emailAddressLength as Integer

cryptGetAttributeString cryptCertificate, CRYPT_CERTINFO_RFC822NAME, _
   0, emailAddressLength
emailAddress = String( emailAddressLength, vbNullChar )
cryptGetAttributeString cryptCertificate, CRYPT_CERTINFO_RFC822NAME, _
   emailAddress, emailAddressLength
```

In Python you can use **cryptGetAttributeString** and **cryptSetAttributeString** as usual, or use a shortcut syntax to make accessing attributes less verbose. The normal syntax follows the C form but migrates the integer output values (the length from **cryptGetAttributeString** or the output value from **cryptGetAttribute**) to return values, and doesn't require a length for **cryptSetAttributeString**:

```
from array import *

emailAddress = array( 'c', 'x' * 128 )

cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   "1234" )
emailAddressLength = cryptGetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_RFC822NAME, emailAddress )
```

The shortcut syntax allows you to get/set attributes as if they were integer and string members of the object (without the CRYPT_ prefix):

```
cryptEnvelope.ENVINFO_PASSWORD = "1234"
emailAddress = cryptCertificate.CERTINFO_RFC822NAME
```

Just as with Python, C# and Java also migrate returned data to return values. In the C# and Java cases the string functions take byte arrays or Strings. When passing a byte array, you can optionally specify an offset following it for **cryptGetAttributeString** and an offset and length following it for **cryptSetAttributeString**. There is also a special version of **cryptGetAttributeString** that returns Strings for convenience:

```
crypt.SetAttributeString( cryptEnvelope, crypt.ENVINFO_PASSWORD,
   "1234" );
String emailAddress = crypt.GetAttributeString( cryptCertificate,
   crypt.CERTINFO_RFC822NAME );
```

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 that 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 that 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 that 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 the functions that take a length parameter) cryptlib will return CRYPT_ERROR_PARAM4. If you try to perform an attribute access which is disallowed (reading an attribute that can't be read, writing to or deleting a read-only attribute, or something similar) cryptlib will return CRYPT_ERROR_PERMISSION.

Finally, if you try to access an attribute that hasn't been initialised or isn't present, cryptlib will return CRYPT_ERROR_NOTINITED or CRYPT_ERROR_-NOTFOUND, the only real distinction between the two is that the former is typically returned for fixed attributes that haven't had a value assigned to them yet while the latter is returned for optional attributes that aren't present in the object.

Attribute Types

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

Type	Description
Binary	A binary data string that can contain almost anything. Get and set these with **cryptGetAttributeString** and **cryptSetAttributeString**.
Boolean	Flags that can be set to 'true' (any nonzero value) or 'false' (a zero value), and control whether a certain option or operation is enabled or not. For example the CRYPT_CERTINFO_CA

Type	Description
	attribute in a certificate controls whether a certificate is marked as being a CA certificate or not. Note that cryptlib uses the value 1 to represent 'true', some languages may represent this by the value –1. Get and set these with **cryptGetAttribute** and **cryptSetAttribute**.
Numeric	A numeric constant such as an integer value or a bitflag. For example the CRYPT_CTXINFO_KEYSIZE attribute specifies the size of a key (in bytes) and the CRYPT_CERTINFO_-CRLREASON attribute specifies a numeric reason code that indicates why a CRL was issued. Get and set these with **cryptGetAttribute** and **cryptSetAttribute**.
Object	A handle to a cryptlib object. For example the CRYPT_-CERTINFO_SUBJECTPUBLICKEYINFO attribute specifies the public key to be added to a certificate. Get and set these with **cryptGetAttribute** and **cryptSetAttribute**.
String	A text string that contains information such as a name, message, email address, or URL. Strings are encoded using the standard system local character set, usually ASCII or latin-1 or UTF-8 (depending on the system), however under Windows CE, which is a Unicode environment, these are Unicode strings. In (very rare) cases where the standard system character set doesn't support the characters used in the string (for example when encoding Asian characters), the characters used will be Unicode or widechars. For all intents and purposes you can assume that all strings are encoded in the standard character set that you'd normally use, cryptlib will perform all conversions for you. An example string attribute is CRYPT_CTXINFO_LABEL, which contains a human-readable label used to identify private keys. The most frequently used text string components are those that make up a certificate's distinguished name, which identifies the certificate owner. Most of these components are limited to a maximum of 64 characters by the X.500 standard that covers certificates and their components, and cryptlib provides the CRYPT_MAX_TEXTSIZE constant for use with these components (this value is also used for most other strings such as key labels). Since this value is specified in characters rather than bytes, Unicode strings 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. Get and set these with **cryptGetAttributeString** and **cryptSetAttributeString**.
Time	The ANSI/ISO C standard time value containing the local time expressed as seconds since 1970. This is a binary (rather than numeric) field, with the data being the time value (in C and C++ this is a `time_t`, usually a signed long integer). Due to the vagaries of international time zones and daylight savings time adjustments, it isn't possible to accurately compare two local times from different time zones, or made across a DST switch (consider for example a country switching to DST, which has two 2am times while another country only has one). Because of this ambiguity, times read from objects such as certificates may appear to be out by an

Type	Description
	hour or two. Get and set these with **cryptGetAttributeString** and **cryptSetAttributeString**.

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( cryptCertificate, CRYPT_CERTINFO_COMMONNAME,
   commonName, &commonNameLength );
commonName[ commonNameLength ] = '\0';
```

to read the value, in this case the common name of a certificate owner.

Note the explicit addition of the terminating null character, since the strings returned aren't null-terminated. You also need to be careful about the content of the strings, in some cases they can contain binary data that can't be displayed as text.

In Visual Basic this is:

```
Dim commonName As String
Dim commonNameLength As Long

commonName = String( CRYPT_MAX_TEXTSIZE + 1 , vbNullChar )
cryptGetAttributeString cryptCertificate, CRYPT_CERTINFO_COMMONNAME, _
   commonName, commonNameLength
commonName = Left( commonName, InStr( commonName, vbNullChar ) - 1 )
```

The description above assumes that the common name is expressed in a single-byte character set. Since the values passed to **cryptGetAttributeString** and **cryptSetAttributeString** are untyped, their length is given in bytes and not in characters (which may not be byte-sized). For Unicode strings, you need to multiply the size of the buffer by the size of a Unicode character on your system to get the actual size to pass to the function, or divide by the size of a Unicode character to get the number of characters returned. For example to perform the same operation as above in a Unicode environment you'd use:

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

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

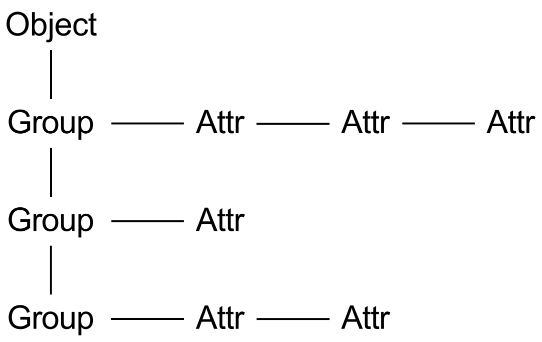
Attribute Lists and Attribute Groups

Several of the container object types (certificates, envelopes, and sessions) contain large collections of attributes that you can process as a list rather than having to access each attribute individually by type. The list of attributes is managed through the use of an attribute cursor that cryptlib maintains for each container object. You can set or move the cursor either to an absolute position in the list of attributes or relative to the current position.

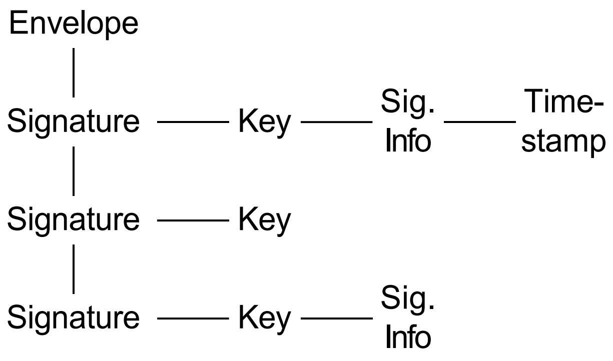
Object attributes are usually grouped into collections of related attributes. For example an envelope object may contain a group of attributes consisting of a signature, the key that generated the signature, associated signing attributes such as the time and data type being signed, and even a timestamp on the signature itself. Similarly, a session object may have a group of attributes consisting of a server name, server port, and server key. So instead of a straight linear list of attributes:

Object —— Attr —— Attr —— Attr —— Attr

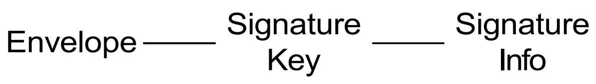
the attributes are arranged by group:



Some objects may contain multiple instances of attribute groups, each of which contains its own set of attributes. For example an envelope could contain several signature attribute groups, and each attribute group will contain its own signing keys, certificates, signature information such as the signing time, and so on. One particular instance of the abstract group/attribute view shown above would be:



In order to navigate across attribute groups, and across attributes within a group, cryptlib provides the attribute cursor functionality described in the section that follows. As well as moving the cursor back and forth across attribute groups and attributes within the group, you can also position it directly on a group or attribute. In the common case where only a single attribute group is present, for example an envelope object that contains a single signature or a session object that contains user information for a single user:



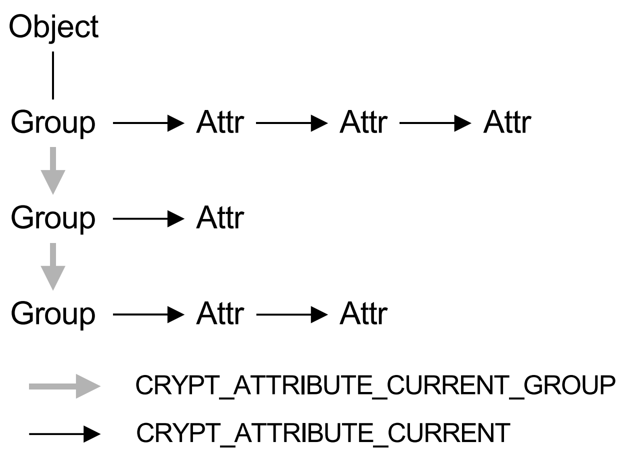
you can treat the attributes as a single flat list of attributes and not worry about the hierarchical arrangement into groups.

Attribute Cursor Management

You can move the attribute cursor by setting an attribute that tells cryptlib where to move it to. This attribute, either CRYPT_ATTRIBUTE_CURRENT_GROUP when moving by attribute group or CRYPT_ATTRIBUTE_CURRENT when moving by attribute within the current group, takes as value a cursor movement code that moves the cursor either to an absolute position (the first or last group or attribute in the list) or relative to its current position. The movement codes are:

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

Moving by attribute group or attribute then works as follows:



Note that CRYPT_ATTRIBUTE_CURRENT only moves the cursor within the current group. Once you get to the start or end of the group, you need to use CRYPT_ATTRIBUTE_CURRENT_GROUP to move on to the next one. Moving the cursor from one group to another will reset the cursor position to the first attribute within the group if it's been previously moved to some other attribute within the group. For example to move the cursor to the start of the first attribute group in a certificate you would use:

```
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT_GROUP,
   CRYPT_CURSOR_FIRST );
```

To advance the cursor to the start of the next group you would use:

```
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT_GROUP,
   CRYPT_CURSOR_NEXT );
```

To advance the cursor to the next attribute in the current group you would use:

```
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CURSOR_NEXT );
```

In some cases multiple instances of the same attribute can be present, in which case you can use a third level of cursor movement, handled via the CRYPT_-ATTRIBUTE_CURRENT_INSTANCE attribute, and relative cursor movement to step through the different instances of the attribute. Since the use of multi-valued attributes is rare, it's safe to assume one value per attribute in most cases, so that stepping through multiple attribute instances is usually unnecessary.

Once you've set the cursor position you can work with the attribute group or attribute at that position in the usual manner. To obtain the group or attribute type at the current position you would use:

```
CRYPT_ATTRIBUTE_TYPE groupID;

cryptGetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT_GROUP,
   &groupID );
```

This example obtains the attribute group type, to obtain the attribute type you would substitute CRYPT_ATTRIBUTE_CURRENT in place of CRYPT_ATTRIBUTE_-CURRENT_GROUP. Attribute accesses are relative to the currently selected group, so for example if you move the cursor in an envelope to a signature attribute group and then read the signature key/certificate or signing time, it'll be the one for the currently-selected signature attribute group. Since there can be multiple signatures present in an envelope, you can use this mechanism to read the signing information for each of the ones that are present.

To delete the attribute group at the current cursor position you would use:

```
cryptDeleteAttribute( cryptCertificate,
   CRYPT_ATTRIBUTE_CURRENT_GROUP );
```

Deleting the attribute group at the cursor position will move the cursor to the start of the group that follows the deleted one, or to the start of the previous group if the one being deleted was the last one present. This means that you can delete every attribute group simply by repeatedly deleting the one under the cursor.

The attribute cursor provides a convenient mechanism for stepping through every attribute group and attribute which is present in an object. For example to iterate through every attribute group you would use:

```
if( cryptSetAttribute( cryptCertificate,
   CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      CRYPT_ATTRIBUTE_TYPE groupID;

      /* Get the ID of the attribute group under the cursor */
      cryptGetAttribute( cryptCertificate,
         CRYPT_ATTRIBUTE_CURRENT_GROUP, &groupID );

      /* Handle the attribute if required */
      /* ... */
      }
   while( cryptSetAttribute( cryptCertificate,
      CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_NEXT ) ==
      CRYPT_OK );
```

The Visual Basic equivalent is:

```
Dim groupID As CRYPT_ATTRIBUTE_TYPE

If cryptSetAttribute( cryptCertificate, _
   CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_FIRST ) == CRYPT_OK
   Then
   Do
      ' Get the type of the attribute group under the cursor
      cryptGetAttribute cryptCertificate, CRYPT_ATTRIBUTE_CURRENT, _
         groupID

      ' Handle the attribute if required
      ' ...
   Loop While cryptSetAttribute( cryptCertificate, _
      CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_NEXT ) == CRYPT_OK
End If
```

To extend this a stage further and iterate through every attribute in every group in the object you would use:

```
if( cryptSetAttribute( cryptCertificate,
   CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      do
         {
         CRYPT_ATTRIBUTE_TYPE attributeID;

         /* Get the ID of the attribute under the cursor */
         cryptGetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
            &attributeID );

         /* Handle the attribute if required */
         /* ... */
         }
      while( cryptSetAttribute( cryptCertificate,
         CRYPT_ATTRIBUTE_CURRENT, CRYPT_CURSOR_NEXT ) == CRYPT_OK );
      }
   while( cryptSetAttribute( cryptCertificate,
      CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CURSOR_NEXT ) ==
      CRYPT_OK );
```

Note that iterating attribute by attribute works within the current attribute group, but as mentioned earlier won’t jump from one group to the next — to do that, you need to iterate by group.

You can also position the attribute cursor directly by telling cryptlib which attribute you want to move the cursor to. For example to move the cursor in a certificate object to the extended key usage attribute group you would use:

```
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT_GROUP,
   CRYPT_CERTINFO_EXTKEYUSAGE );
```

For certificate objects you can also use this cursor-positioning mechanism to select the certificate's subject and issuer names, which aren't part of the standard attributes but are complex enough that they're processed like attributes. For example to select the certificate's issuer name you would use:

```
cryptSetAttribute( cryptCertificate,
   CRYPT_ATTRIBUTE_CURRENT_ATTRIBUTE, CRYPT_CERTINFO_SUBJECTNAME );
```

Usually the absolute cursor-positioning capability is only useful for certificate objects where you know that certain attributes will be present and that only one instance of the attribute will be present. For envelope and session objects you generally can't tell in advance which attributes will be present and it's quite possible that multiple attribute instances (such as multiple signatures on a envelope) will be present. In this case selecting an attribute will only select the first one that's present, so it's better to use the attribute cursor to walk the list to see what's there.

Using this absolute cursor positioning in a variation of the attribute enumeration operation given earlier, you can enumerate only the attributes of a single attribute group (rather than all groups) by first selecting the group and then stepping through the attributes in it. For example to read all of a certificate's extended key usage types you would use:

```
if( cryptSetAttribute( cryptCertificate,
   CRYPT_ATTRIBUTE_CURRENT_GROUP, CRYPT_CERTINFO_EXTKEYUSAGE ) ==
   CRYPT_OK )
   do
      {
      CRYPT_ATTRIBUTE_TYPE attributeID;

      /* Get the ID of the attribute under the cursor */
      cryptGetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
         &attributeID );
      }
   while( cryptSetAttribute( cryptCertificate,
      CRYPT_ATTRIBUTE_CURRENT, CRYPT_CURSOR_NEXT ) == CRYPT_OK );
```

Object Security

Each cryptlib object has its own security settings that affect the way that 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's used. For example on a system that supports threads you can bind an object to an individual thread within a process so that only the thread that owns the object can see it. For any other thread in the process, the object handle is invalid.

You can get and set an object's 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 attribute's value. The object property attributes that you can get or set are:

Property/Description	Type
CRYPT_PROPERTY_FORWARDCOUNT	Numeric
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 object's 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 attribute's value to 1 then you can forward the object to another thread, but that thread can't forward it further.	

Property/Description	Type
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.	
CRYPT_PROPERTY_HIGHSECURITY	Boolean
This is a composite value that 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 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 non-exportable 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 that owns the object. The thread's identity is specified using a value that depends on the operating system, but is usually a thread handle or thread ID. For example under Windows the thread ID is the value returned by the `GetCurrentThreadID` function, which returns a system-wide 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 an AES 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, cryptUser, CRYPT_ALGO_AES );
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 since the Windows 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 that can themselves contain objects (for example keysets), if the container is bound to a thread then any objects that 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 that 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.

Role-based Access Control

cryptlib implements a form of access control called role-based access control or RBAC in which operations specific to a certain user role can't be performed by a user acting in a different role. For example in many organisations a cheque can only be issued by an accountant and can only be signed by a manager, which prevents a dishonest accountant or manager from both issuing a cheque to themselves and then signing it as well. This security measure is referred to as separation of duty, in which it takes at least two people to perform a critical operation. Similarly, cryptlib uses RBAC to enforce a strong separation of duty between various roles, providing the same effect as the corporate accounting controls that prevent an individual from writing themselves cheques.

cryptlib recognises a variety of user types or roles. The default user type has access to most of the standard functions in cryptlib but can't perform CA management operations or specialised administrative functions that are used to manage certain aspects of cryptlib's operation. When you use cryptlib in the role of a standard user, it functions as a normal crypto/security toolkit.

In addition to the standard user role, it's also possible to use cryptlib in the role of a security officer (SO), a special administrative user who can create new users and perform other administrative functions but can't perform general crypto operations like a normal user. This provides a clear separation of duty between administrative and end-user functionality.

Another role recognised by cryptlib is that of a certification authority that can make use of cryptlib's certificate management functionality but can't perform general administrative functions or non-CA-related crypto operations. Again, this provides a clear separation of duty between the role of the CA and the role of a general user or SO.

Managing User Roles

When a cryptlib object is created, it is associated with a user role which is specified at creation time and can't be accessed by any other user. For example if a private key is created by a CA for signing certificates, it can't be accessed by a normal user because it's only visible to the user acting in the role of the CA. Similarly, although a normal user may be able to see a certificate store, only a CA user can use it for the purpose of issuing certificates. The use of RBAC therefore protects objects from misuse by unauthorised users.

The identity of the user who owns the object is specified as a parameter for the object creation function. Throughout the rest of the cryptlib documentation this parameter is denoted through the use of the `cryptUser` variable. Usually this parameter is set to CRYPT_UNUSED to indicate that the user is acting in the role of a normal user and doesn't care about role-based controls. This is typically used in cases where there's only one cryptlib user, for example where cryptlib is running on an end-user PC (e.g. Windows, Macintosh) or a multi-user system that provides each user with the illusion of being on a single-user machine (e.g. Unix). In almost all cases therefore you'd pass in CRYPT_UNUSED as the user identity parameter.

In a few specialised cases where the user is acting in a role other than that of a normal user the default user role isn't enough. For example when you want to access a CA certificate store you can't use the role of a normal user to perform the access because only a CA can manipulate a certificate store. This prevents a normal user from issuing themselves certificates in the name of the CA and assorted other mischief such as revoking another user's certificate.

When acting in a different role than that of the default, normal user, you specify the identity of the user whose role you're acting in as a parameter of the object creation function as before, this time passing in the handle of the user identity instead of CRYPT_UNUSED. When the object is created, it is associated with the given user and role instead of the default user. The creation and use of user objects is covered in the next section.

Creating and Destroying Users and Roles

The following section is provided purely for forwards compatibility with functionality included in future versions of cryptlib. For the current version of cryptlib the user identity parameter should always be CRYPT_UNUSED since user object management isn't enabled in this version.

User objects can only be created and destroyed by an SO user, this being one of the special administrative functions mentioned earlier that can only be performed by an SO. You create a user object with **cryptCreateUser**, specifying the identity of the SO who is creating the user object, type of user role that the object is associated with, the name of the user, and a password that protects access to the user object:

```
CRYPT_USER cryptUser;

cryptCreateUser( &cryptUser, cryptSO, type, name, password );
```

The available user types or roles are:

Role	Description
CRYPT_USER_CA	A certification authority who can perform CA management functions but can't perform general-purpose crypto operations.
CRYPT_USER_NORMAL	A standard cryptlib user.
CRYPT_USER_SO	A security officer who can perform administrative functions such as creating or deleting users but who can't perform any other type of operation.

For example to create a normal user object for "John Doe" with the password "password" and a CA user object for "John's Certificate Authority" with the password "CA password" you would use:

```
CRYPT_USER cryptUser, cryptCAUser;

cryptCreateUser( &cryptUser, cryptSO, CRYPT_USER_NORMAL, "John Doe",
   "password" );
cryptCreateUser( &cryptUser, cryptSO, CRYPT_USER_CA, "John's
   Certification Authority", "CA password" );
```

Once a user object is created it can't be used immediately because it's still under the nominal control of the SO who created it rather than the user it's intended for. Before it can be used, control over the object needs to be handed over to the user that it's intended for. After the object is created by the SO, it is said to be in the SO initialised state. Any attempt to use an object when it's in the SO initialised state will result in cryptlib returning CRYPT_ERROR_NOTINITED.

To move the newly-created object into a usable state, it's necessary to change its password from the initial one set by the SO to one chosen by the user. Once this change occurs, the object is moved into the user initialised state and is ready for use. You can change the password from the initial one set by the SO to a user-chosen one with **cryptChangePassword**:

```
cryptChangePassword( cryptUser, oldPassword, newPassword );
```

When the password has been changed from the one set by the SO to the one chosen by the user, the user object is ready for use.

User objects can also be destroyed by the SO who created them:

```
cryptDeleteUser( cryptUser, "John Doe" );
```

Miscellaneous Issues

This section contains additional information that may be useful when working with cryptlib.

Multi-threaded cryptlib Operation

cryptlib is re-entrant and completely thread-safe (the threading model used is sometimes known as the free-threading model), allowing it to be used with multithreaded applications in systems that support threading. When you use cryptlib in a multithreaded application, you should take standard precautions to ensure that a single resource shared across multiple threads doesn't become a bottleneck, with all threads being forced to wait on a single shared object. For example if you're timestamping large numbers of messages then creating a single timestamping session object (see "Secure Sessions" on page 102) and using that for all timestamping services will result in all of the operations waiting on a single session object, which can often take several seconds to turn around a transaction with a remote server. A better option in this case would be to create a pool of timestamping session objects and use the next free one when required.

A similar situation occurs with other objects such as crypto devices and keysets that may be shared across multiple threads. For example cryptlib provides a facility for automatically fetching a decryption key from a keyset in order to decrypt data (see

"Public-Key Encrypted Enveloping" on page 71). This is convenient when handling one or two messages since cryptlib will automatically take care of all of the processing for you, however if you're processing large numbers of messages then the need to read and decrypt the same private key for each message is very inefficient, not only in terms of CPU overhead but also because every message must wait for each of the previous messages to be processed before it gets its turn at the keyset.

A better alternative in this case is to read the private key from the keyset just once and then use it with each envelope, rather than having each envelope read and decrypt the key itself. Extending this even further, if you're using a very large private key, running on a slower processor, or processing large numbers of transactions, you may want to instantiate multiple copies of the private-key object to avoid the single private key again becoming a bottleneck.

In general most private-key operations, when performed on modern processors, are fairly quick, so there's no need to create large numbers of private-key objects for fear of them becoming a bottleneck. In this case the primary bottleneck is the need to read and decrypt the key for each message processed. However, when run on a multiple-CPU system, you should make some attempt to match objects to CPUs — creating a single private-key object on a four-CPU system guarantees that the overall performance will be no better than that of a single-CPU system, since the single object instance acts as a mutex that can only be acquired by one CPU at a time. Standard programming practice for efficient utilisation of resources on multiprocessor systems applies to cryptlib just as it does for other applications. Creating a pool of objects that can be picked up and used as required would be one standard approach to this problem. Some operating systems provide special support for this with functions for thread pooling management. For example, Windows 2000 and XP provide the `QueueUserWorkItem` function, which submits a work item to a thread pool for execution when the next thread becomes available. Windows Vista and newer include an enhanced version of the thread-pool API that replaces the basic `QueueUserWorkItem` with a more conventional `CreateThreadpoolWork`/-`SubmitThreadpoolWork`/`CloseThreadpoolWork` combination that provides better control over thread pools, pool management, and pool cleanup.

In order to protect against potential deadlocks when multiple threads are waiting on a single object, cryptlib includes a watchdog timer that triggers after a certain amount of time has been spent waiting on the object. Once this occurs, cryptlib will return CRYPT_ERROR_TIMEOUT to indicate that an object is still in use after waiting for it to become available. If you experience timeouts of this kind, you should check your code to see if there are any bottlenecks due to a single object with a long response time being shared by several fast-response-time objects. Note that timeouts are also possible with normal cryptlib object use, for example when communicating data over a slow or stalled network link, so a CRYPT_ERROR_TIMEOUT status doesn't automatically mean that the watchdog timer signalled a problem.

To help diagnose situations of this kind, the debug build of cryptlib will display on the console output an indication that it waited on a particular object, along with the object type that it waited on. You can use this information to identify potential bottlenecks in your application.

Linux has a somewhat unusual threading implementation built around the `clone()` system call that can lead to unexpected behaviour with some kernel and/or glibc versions. Two common symptoms of glibc/kernel threading problems are phantom processes (which are actually glibc-internal threads created via `clone()`) being left behind when you application exits, and cryptlib's internal consistency-checking throwing an exception in the debug build when it detects an problem with threading. If you run into either of these situations, you can try different glibc and/or kernel versions to find a combination that works. Searching Internet newsgroups will provide a wealth of information and advice on problems with glibc and Linux threads.

Safeguarding Cryptographic Operations

Running cryptographic operations on general-purpose CPUs shared with other (often untrusted) programs can expose them to risk if the other programs can closely observe or even influence the behaviour of the crypto code. In addition it's possible for a remote system with the ability to precisely time network packet flows to deduce information about crypto operations like a TLS handshake that result in network traffic as the output of the crypto operation. The timing attacks only affect RSA private-key operations, and only those operations that are directly observable by another party, for example as a result of a network data exchange involving RSA decryption or signing.

There are several countermeasures that you can take to avoid this problem. The simplest approach is to use a different crypto mechanism that isn't vulnerable to this problem. By default cryptlib will try and use Diffie-Hellman or ECDH key exchange in TLS, which isn't vulnerable to this type of attack because it uses a new random value each time, making it impossible to get repeatable timing measurements. In addition the cryptlib security kernel provides a good degree of protection since it isolates the RSA crypto operations from external observation, making it quite difficult to obtain timing information.

If you must use RSA in a manner in which its operation is visible to an external observer, you can enable randomisation of the RSA operations (known as "blinding") in order to provide the same protection that comes built into Diffie-Hellman and ECDH. Enabling blinding adds a performance overhead of between two and five percent to each RSA operation. You can enable blinding for RSA operations (and a few other protection measures) by setting the CRYPT_OPTION_MISC_-SIDECHANNELPROTECTION configuration option as described in "Working with Configuration Options" on page 292.

Many modern CPUs include sophisticated diagnostic and monitoring facilities that provide extensive insight into both the operation of the CPU and the data that it processes. If untrusted processes are running on a CPU alongside ones performing crypto operations, it may be possible for the untrusted processes to recover sensitive data or even crypto keys using built-in CPU monitoring facilities. This can occur even through indirect means such as observing memory access latencies for cached vs. un-cached data, or branch times for cached vs. un-cached branch target information. Since the level of access provided by may of these diagnostic facilities is almost at the level of an in-circuit emulator (ICE), there are no truly effective defences against this level of threat.

If you're using a CPU that provides this detailed monitoring capability and you're also working with sensitive data or crypto keys, and in particular the private keys used in public-key encryption operations, you need to take precautions to ensure that other code can't misuse these monitoring capabilities to compromise your keys or sensitive data. The simplest and most effective defence is "don't do that, then": Don't allow untrusted code to run alongside your crypto code (or any other code processing sensitive information for that matter).

If you really need to run arbitrary untrusted code at the same time as code that's processing sensitive information, you'll need to use OS-level scheduling and CPU-control facilities to ensure that another process or thread can't run alongside your one and monitor its operation, and that out-of-band channels like CPU caches are flushed after your crypto operations have completed.

As an extension of this, it's almost impossible to defend against hardware-level attacks using purely software methods. If your threat model involves an attacker who can walk up to your hardware and attach sensors to it or point an antenna at it then you need to deal with it via hardware measures such as shielding, signal and power supply isolation and decoupling, filtering, and so on. Although software developers like to think that any hardware problem should be fixable via software, in this case it can't. If you need resistance to these types of attacks then you need to design or adapt the hardware to deal with them.

Data Enveloping

Encryption envelopes are the easiest way to use cryptlib. An envelope is a container object whose behaviour is modified by the data and resources that you push into it. To use an envelope, you add to it other container and action objects and resources such as passwords that control the actions performed by the envelope, and then push in and pop out data that's processed according to the resources that 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 that 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 functionality 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 that cryptlib supports, you can't perform more than one step 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, or some other conflicting combination, cryptlib will return a parameter error to indicate that the attribute type is invalid for this envelope since it's already being used for a different purpose.

Creating/Destroying Envelopes

Envelopes are accessed through envelope objects that 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 user who is to own the device object or CRYPT_UNUSED for the default, normal user, and the format for the enveloped data (for now you should use CRYPT_FORMAT_CRYPTLIB, the default format):

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

/* Perform enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

The Visual Basic version is:

```
Dim cryptEnvelope As Long

cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_CRYPTLIB

' Perform enveloping

cryptDestroyEnvelope cryptEvelope
```

The C#, Java, and Python versions (here as elsewhere) migrate the output value to the return value, and return errors by throwing exceptions. The Python version is:

```
cryptEnvelope = cryptCreateEnvelope( cryptUser,
   CRYPT_FORMAT_CRYPTLIB )
```

The C# and Java version is:

```
int cryptEnvelope;

cryptEnvelope = crypt.CreateEnvelope( cryptUser /* crypt.UNUSED */,
   crypt.FORMAT_CRYPTLIB );
```

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

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Perform de-enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

Note that the CRYPT_ENVELOPE is passed to the **cryptCreateEnvelope** by reference as the function modifies it when it creates the envelope. In all other routines in cryptlib, CRYPT_ENVELOPE is passed by value.

Sometimes when you're processing data in an envelope, you may not be able to add all of the data in an envelope, for example when you're trying to de-envelope a message that's been truncated due to a transmission error, or when you don't retrieve all of the processed data in the envelope before destroying it. When you destroy the envelope cryptlib will return CRYPT_ERROR_INCOMPLETE as a warning that not all of the data has been processed. The envelope will be destroyed as usual, but the warning is returned to indicate that you should have added further data or retrieved processed data before destroying the envelope.

The Data Enveloping Process

Although this section only covers basic data and password-based enveloping, the concepts that 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 signed 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 that 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 same operation in C# and Java is:

```
int bytesCopied = crypt.PushData( envelope, "Hello world" );
```

In Python this is:

```
bytesCopied = cryptPushData( envelope,  "Hello world" )
```

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.

When you push in data, cryptlib may return an advisory CRYPT_-ENVELOPE_RESOURCE status, which indicates that additional information such as a password or decryption key is required in order to continue. Until you supply the necessary resource, cryptlib can't process the data that you've given it, and any further attempts to push or pop data will fail with a CRYPT_ENVELOPE_-RESOURCE. The handling of de-encryption resources is covered in more detail in the following sections.

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 that you want to process into an envelope, you need to use **cryptFlushData** 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.

Checking the status of **cryptFlushData** when you're de-enveloping data is particularly important because, if there's an integrity check such as a digital signature applied to the envelope then any problem with it will be reported at this point (it may also be reported on the last **cryptPushData**, depending on how far cryptlib gets in processing the integrity-check information at the end of the envelope). Because of this it's important to always finish processing with a **cryptFlushData**, and to check the return values of the **cryptPushData**/**cryptFlushData** in case cryptlib indicates that the data that you've de-enveloped has been tampered with.

You can add enveloping and de-enveloping attributes to an envelope in the usual manner with **cryptSetAttribute** and **cryptSetAttributeString**. For example to add the password “password” to an envelope, you would set the CRYPT_ENVINFO_-PASSWORD attribute:

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

The same operation in Visual Basic is:

```
cryptSetAttributeString cryptEnvelope, CRYPT_ENVINFO_PASSWORD, _
   "password", 8
```

The various types of attributes that 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 that 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_ENVINFO_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 processing with **cryptFlushData**, cryptlib will return CRYPT_ERROR_-OVERFLOW; if you push in less, it will return CRYPT_ERROR_UNDERFLOW.

There is one exception to this rule, which occurs when you’re using the PGP/OpenPGP data format. PGP requires that the length be indicated at the start of every message, so you always have to set the CRYPT_ENVINFO_DATASIZE attribute when you perform PGP enveloping. If you try and push data into a PGP envelope without setting the data size, cryptlib will return CRYPT_ERROR_-NOTINITED to tell you that it can’t envelope the data without knowing its overall size in advance. PGP/OpenPGP enveloping is explained in more detail in “PGP” on page 92.

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.

Pushing 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 minimum block size of 32 bytes, so it won’t return an individual block containing less than 32 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.

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 that 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, cryptUser /* CRYPT_UNUSED */,
   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.

Basic Data Enveloping

In the simplest case the entire message you want to process will fit into the envelope's 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

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

The Visual Basic equivalent is:

```
Dim cryptEnvelope As Long
Dim bytesCopied As Long

' Create the envelope
cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_CRYPTLIB
```

```
' Add the data size information and data, wrap up the processing, and
' pop out the processed data
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_DATASIZE, messageLength
cryptPushData cryptEnvelope, message, messageLength, bytesCopied
cryptFlushData cryptEnvelope
cryptPopData cryptEnvelope, envelopedData, envelopedDataBufferSize, _
   bytesCopied

' Destroy the envelope
cryptDestroyEnvelope cryptEnvelope
```

The Python version is:

```
# Create the envelope
cryptEnvelope = cryptCreateEnvelope( cryptUser,
   CRYPT_FORMAT_CRYPTLIB )

# Add the data size information and data, wrap up the processing, and
# pop out the processed data
cryptEnvelope.ENVINFO_DATASIZE = len( message )
bytesCopied = cryptPushData( cryptEnvelope, message )
cryptFlushData( cryptEnvelope )
bytesCopied = cryptPopData( cryptEnvelope, envelopedData,
   envelopedDataBufferSize )

# Destroy the envelope
cryptDestroyEnvelope( cryptEnvelope )
```

The C# or Java version is:

```
int bytesCopied;

// Create the envelope
cryptEnvelope = crypt.CreateEnvelope( cryptUser /* crypt.UNUSED */,
   crypt.FORMAT_CRYPTLIB );

// Add the data size information and data, wrap up the processing, and
// pop out the processed data
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_DATASIZE,
   message.Length );
bytesCopied = crypt.PushData( cryptEnvelope, message );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, envelopedData,
   envelopedDataBufferSize );

// Destroy the envelope
crypt.DestroyEnvelope( cryptEnvelope );
```

(the above code is for C#, the Java version is virtually identical except that the `message.Length` of a C# byte array is `message.length` in Java).

To de-envelope the resulting data you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

/* Create the envelope */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* crypt.UNUSED */,
   CRYPT_FORMAT_AUTO );

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

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

The Visual Basic form is:

```
Dim cryptEnvelope As Long
Dim bytesCopied As Long

' Create the envelope
cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_AUTO
```

```
' Push in the enveloped data and pop out the recovered message
cryptPushData cryptEnvelope, envelopedData, envelopedDataSize, _
   bytesCopied
cryptFlushData cryptEnvelope
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 that 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

De-enveloping 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 decompressed data. This applies particularly to the final stages of enveloping when you flush through 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's more data to be flushed, requiring multiple iterations of flushing and copying out data:

```
/* ... */

/* Flush out any remaining data */
do
   {
```

```
   cryptFlushData( cryptEnvelope );
   cryptPopData( cryptEnvelope, outBuffer, BUFFER_SIZE, &bytesCopied
      );
   }
while( bytesCopied > 0 );
```

To handle this in a more general manner, you should use the processing techniques described in "Enveloping Large Data Quantities" on page 66.

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

The same operation in Java (for C# replace the `.length` with `.Length`) is:

```
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* crypt.UNUSED */, crypt.FORMAT_CRYPTLIB );

/* Add the password */
crypt.SetAttributeString( cryptEnvelope, crypt.ENVINFO_PASSWORD,
   password );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_DATASIZE,
   message.length );
int bytesCopied = crypt.PushData( cryptEnvelope, message );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, envelopedData,
   envelopedData.length );

crypt.DestroyEnvelope( cryptEnvelope );
```

To de-envelope the resulting data you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

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

cryptDestroyEnvelope( cryptEnvelope );
```

The de-enveloping process in Java is:

```
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* CRYPT_UNUSED */, crypt.FORMAT_AUTO );
int bytesCopied;

// Push in the enveloped data and the password required to
// de-envelope it, and pop out the recovered message
try {
   bytesCopied = crypt.PushData( cryptEnvelope, envelopedData );
}
catch ( CryptException e ) {
   if( e.getStatus() != crypt.ENVELOPE_RESOURCE )
      throw e;
}
crypt.SetAttributeString( cryptEnvelope, crypt.ENVINFO_PASSWORD,
   password );
crypt.FlushData( cryptEnvelope );
crypt.PopData( cryptEnvelope, messageBuffer, messageBuffer.length );

// Destroy the envelope
crypt.DestroyEnvelope( cryptEnvelope );
```

The Visual Basic equivalent is:

```
Dim cryptEnvelope As Long
Dim bytesCopied As Long

cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_AUTO

' Push in the enveloped data and the password required to
' de-envelope it, and pop out the recovered message
cryptPushData cryptEnvelope, envelopedData, envelopedDataSize, _
   bytesCopied
cryptSetAttributeString cryptEnvelope, CRYPT_ENVINFO_PASSWORD, _
   password, Len( password )
cryptFlushData cryptEnvelope
cryptPopData cryptEnvelope, message, messageBufferSize, bytesCopied

' Destroy the envelope
cryptDestroyEnvelope cryptEnvelope
```

When you push in the password-protected data, **cryptPushData** will return CRYPT_ENVELOPE_RESOURCE to indicate that an additional resource (in this case the password) is required in order to continue. This is an advisory status that isn't needed in this case but can be useful for advanced de-envelope processing as described in "De-enveloping Mixed Data" on page 65.

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_ENVINFO_PASSWORD,
      password, passwordLength ) == CRYPT_OK )
      break;
   }
```

cryptlib uses a form of password-based encryption called authenticated encryption that provides both encryption and integrity protection, protecting the enveloped data from being tampered with by an attacker alongside the confidentiality protection provided by the encryption. Without this extra protection an attacker can freely modify the encrypted data, resulting in a corresponding (and undetectable) modification in the decrypted data (this is a standard property of encryption algorithms, they offer confidentiality protection but no integrity protection). In extreme cases an attacker could, for example, modify the payment amount in an encrypted payment instruction by doing as little as flipping a single bit of encrypted data, turning a $100 payment into a $100,000 payment or vice versa.

In order to protect against this type of attack, cryptlib uses authenticated encryption that both encrypts the envelope's contents and protects it against tampering by attackers. If even a single bit of the encrypted data is changed, cryptlib will return a CRYPT_ERROR_SIGNATURE when it's finished processing the data in the same way that it would for a digitally signed envelope as described in "Digitally Signed Enveloping" on page 75, providing the benefits of a digitally signed envelope without any need to explicitly manage a digital signature.

As with standard digitally-signed envelopes, cryptlib can't perform the integrity-verification until it's seen all of the envelope data. If you're using cryptlib in a streaming implementation in which messages aren't processed in one go you should take care not to hand the decrypted data over to the user until the integrity check bas completed. This could become an issue if, for example, you hand them the first half of a decrypted message while waiting for the second half to arrive, allowing them to act on a (partial) message whose integrity hasn't yet been verified.

A less obvious way in which the integrity-verification can be defeated is if an attacker truncates the encrypted data, leaving it (and the integrity check) incomplete. In general cryptlib will detect most types of truncation attack and report them as a CRYPT_ERROR_SIGNATURE, but in some cases it's not possible to tell whether information is missing because of an attack or because the caller hasn't pushed it into the envelope yet. If your application ignores some envelope error codes like CRYPT_ERROR_UNDERFLOW or treats them as non-serious then it may miss this truncation. As a general rule if you complete the de-enveloping and get a CRYPT_ERROR_UNDERFLOW error (that data required for the integrity check wasn't available) then you should treat it as a CRYPT_ERROR_SIGNATURE-type authentication failure.

Alternatively, you can make the authentication check an explicit part of the de-enveloping process by reading the envelope's CRYPT_ENVINFO_SIGNATURE_-RESULT attribute as described in "Digitally Signed Enveloping" on page 75:

```
int signatureResult;

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

Finally, some non-cryptlib implementations don't support authenticated encryption, creating encrypted data that isn't protected from tampering. If you suspect that you've been sent data from one of these implementations, you can check whether the envelope data is authenticated by reading the CRYPT_ENVINFO_INTEGRITY attribute:

```
int authenticationLevel;

status = cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_INTEGRITY,
    &authenticationLevel );
if( cryptStatusError( status ) || \
     authenticationLevel != CRYPT_INTEGRITY_FULL )
    ; /* No authentication present */
```

If the CRYPT_ENVINFO_INTEGRITY attribute isn't available (indicating that no authentication is in effect) or the attribute value isn't CRYPT_INTEGRITY_FULL then the enveloped data has no integrity protection and may be undetectably modified by an attacker.

Conventional Encryption Enveloping

In addition to encrypting enveloped data with a password, it's possible to bypass the password step and encrypt the data directly using an encryption context. This context can either be used to encrypt the data directly (CRYPT_ENVINFO_SESSIONKEY) or indirectly by wrapping up a session key (CRYPT_ENVINFO_KEY). For example to encrypt data directly using IDEA with a raw session key you would do the following:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_CONTEXT cryptContext;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

/* Create the session key context and add it */
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_IDEA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEY,
   "0123456789ABCDEF", 16 );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SESSIONKEY,
   cryptContext );
cryptDestroyContext( cryptContext );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

To de-envelope the resulting data you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
CRYPT_CONTEXT cryptContext;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and the session key context required to
   de-envelope it, and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_IDEA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEY,
   "0123456789ABCDEF", 16 );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SESSIONKEY,
   cryptContext );
cryptDestroyContext( cryptContext );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, message, messageBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Encrypting the data directly by using the context to wrap up the session key and then encrypting the data with that functions identically, except that the context is added as CRYPT_ENVINFO_KEY rather than CRYPT_ENVINFO_SESSIONKEY. The only real difference between the two is the underlying data format that cryptlib generates. As with the password-based de-enveloping, cryptlib will return an advisory CRYPT_ENVELOPE_RESOURCE status when you push in the data to let you know that you need to provide a decryption key in order to continue, and will use authenticated encryption to protect the encrypted data from tampering by attackers.

Raw session-key based enveloping isn't recommended since it bypasses much of the automated key management which is performed by the enveloping code, and requires the direct use of low-level encryption contexts. If all you want to do is change the underlying encryption algorithm used from the default AES, it's easier to do this by setting the CRYPT_OPTION_ENCR_ALGO attribute for the envelope as described in "Working with Configuration Options" on page 292. In addition it's not possible to use authenticated encryption in combination with raw session-key based enveloping because the use of the low-level encryption context precludes the use of authenticated encryption modes.

Authenticated Enveloping

Sometimes you may want to protect data against modification without encrypting it, a variation on the use of authenticated encryption described above. To provide integrity protection for the contents of an envelope, you need to use an authenticated envelope. You can tell cryptlib to add authentication to an envelope by setting the CRYPT_ENVINFO_INTEGRITY attribute before you push data into the envelope. By default this has a setting of CRYPT_INTEGRITY_NONE, which means that the contents are protected by encryption only. If you want to provide authentication (without encryption) for the data, you can set the CRYPT_ENVINFO_INTEGRITY to CRYPT_INTEGRITY_MACONLY (a MAC is a cryptographic message authentication code used to protect the contents of the envelope). The data processing works just like a standard encrypted envelope:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

/* Tell cryptlib that we want integrity-protection instead of
   encryption and add the password */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_INTEGRITY,
   CRYPT_INTEGRITY_MACONLY );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Note that you have to set the CRYPT_ENVINFO_INTEGRITY attribute before you add an encryption key or password, otherwise cryptlib will assume that you want to use the key to encrypt rather than authenticate the envelope contents.

When you de-envelope the data, cryptlib will use the MAC to check the integrity of the envelope contents. If the data has been modified, cryptlib will return CRYPT_ERROR_SIGNATURE once you've pushed the final portion of the enveloped data into the envelope:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and the password required to de-envelope
   it, checking whether the data has been altered */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
status = cryptFlushData( cryptEnvelope );
if( status == CRYPT_ERROR_SIGNATURE )
   /* Data has been altered */;

/* The data is un-altered, pop out the recovered message */
cryptPopData( cryptEnvelope, message, messageBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

You can also read the result of the integrity check by reading the CRYPT_-ENVINFO_SIGNATURE_RESULT attribute, which works in much the same way as it does for signed envelopes, which are discussed in “Digitally Signed Enveloping” on page 75.

De-enveloping Mixed Data

Sometimes you won’t know exactly what type of processing has been applied to the data that 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 that you’ve pushed into an envelope, it will return a CRYPT_-ENVELOPE_RESOURCE status if you try and push in any more data or pop out the processed data. This status code is returned as soon as cryptlib knows enough about the data that 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 that you were processing was either plain, unencrypted data, compressed data, or password-encrypted data created using the code shown earlier, then you could de-envelope it with:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, status;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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 );
cryptFlushData( cryptEnvelope );
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 “Enveloping with Multiple Attributes” on page 78).

De-enveloping with a Large Envelope Buffer

If you’ve increased the envelope buffer size to allow the processing of large data quantities, the de-enveloping process may be slightly different. When de-enveloping data, cryptlib only reads an initial fixed amount of data before stopping and asking for user input such as the password or private key which is required to process the data. This is to avoid the situation where an envelope absorbs megabytes or even gigabytes of data only to report that it can’t even begin to process it for lack of a decryption key. In this case the envelope will return CRYPT_ERROR_RESOURCE to indicate that it requires further information in order to continue. Once you’ve added the necessary de-enveloping attribute(s), you can either pop what’s already been processed and continue as normal (see “Enveloping Large Data Quantities” on page 66) or, for a sufficiently large envelope buffer, push in the remaining data before popping it all at once:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, status;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and see if we need any special handling
   */
status = cryptPushData( cryptEnvelope, envelopedData,
   envelopedDataLength, &bytesCopied );
if( status == CRYPT_ENVELOPE_RESOURCE )
   {
   /* Add the necessary de-enveloping attributes */
   /* ... */

   /* If only some of the data was accepted because the envelope
      stopped to request further instructions, push in the rest now */
   if( bytesCopied < envelopedDataLength )
      {
      int remainingBytesCopied;

      status = cryptPushData( cryptEnvelope, envelopedData + bytesIn,
         envelopedDataLength - bytesIn, &remainingBytesCopied );
      bytesIn += remainingBytesCopied;
      }
   }
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, message, messageBufferSize, &bytesCopied
   );

cryptDestroyEnvelope( cryptEnvelope );
```

This code checks whether the envelope has absorbed all of the enveloped data and, if not, pushes the remainder after adding the attribute(s) necessary for processing it. Once all of the data has been pushed, it pops the result as usual.

Obtaining Envelope Security Parameters

If you want to know the details of the encryption mechanism that's being used to protect the enveloped data, you can read various CRYPT_CTXINFO_*xxx* attributes from the envelope object which will return information from the encryption context(s) that are being used to secure the data. For example if you're encrypting or decrypting data you can get the encryption algorithm and mode and the key size being used with:

```
CRYPT_ALGO_TYPE cryptAlgo;
CRYPT_MODE_TYPE cryptMode;
int keySize;

cryptGetAttribute( cryptEnvelope, CRYPT_CTXINFO_ALGO, &cryptAlgo );
cryptGetAttribute( cryptEnvelope, CRYPT_CTXINFO_MODE, &cryptMode );
cryptGetAttribute( cryptEnvelope, CRYPT_CTXINFO_KEYSIZE, &keySize );
```

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 that 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, cryptUser /* CRYPT_UNUSED */,
   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 */
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
if( bytesCopied > 0 )
   writeFile( outputFile, buffer, bytesCopied );
free( buffer );

cryptDestroyEnvelope( cryptEnvelope );
```

The Visual Basic version is:

```
Dim cryptEnvelope As Long
Dim buffer() As Byte
Dim bufferCount As Integer
Dim bytesCopied As Long

' Create the envelope with a buffer of size BUFFER_SIZE and add the
' password attribute
cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_CRYPTLIB
cryptSetAttribute cryptEnvelope, CRYPT_ATTRIBUTE_BUFFERSIZE, _
   BUFFER_SIZE
cryptSetAttributeString cryptEnvelope, CRYPT_ENVINFO_PASSWORD, _
   password, Len( password )

' Allocate a buffer for file I/O
buffer = String( BUFFER_SIZE, vbNullChar )

Do While Not EndOfFile( inputFile )
   ' Read a (BUFFER_SIZE - 4K) block from the input file, envelope
   ' it, and write the result to an output file
   bufferCount = ReadFile inputFile, buffer, BUFFERSIZE - 4096
   cryptPushData cryptEnvelope, buffer, bufferCount, bytesCopied
   cryptPopData cryptEnvelope, buffer, BUFFER_SIZE, bytesCopied
   WriteFile outputFile, buffer, bytesCopied
Loop
cryptFlushData cryptEnvelope, buffer, BUFFER_SIZE, bytesCopied
If bytesCopied > 0 Then WriteFile outputFile, buffer, bytesCopied

cryptDestroyEnvelope cryptEnvelope
```

The code allocates a BUFFER_SIZE byte I/O buffer, reads up to BUFFER_SIZE – 4K bytes 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 that 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 that 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 envelope's 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, cryptUser /* CRYPT_UNUSED */,
   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 > 0 )
      {
      /* 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 > 0 )
         {
         cryptPopData( cryptEnvelope, outBuffer, BUFFER_SIZE,
            &bytesCopiedOut );
         writeFile( outputFile, outBuffer, bytesCopiedOut );
         }
      }
   }

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

cryptDestroyEnvelope( cryptEnvelope );
```

Running the code to fill/empty the envelope in a loop is useful when you're applying a transformation such as data compression, which dramatically changes the length of the enveloped/de-enveloped data. In this case it's not possible to tell how much data you can push into or pop out of the envelope because the length is transformed by the compression operation. It's also generally good practice to not write code that 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 that occurs if you've 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 envelope's 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, cryptUser /* CRYPT_UNUSED */,
   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 */
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, buffer, BUFFER_SIZE, &bytesCopied );
if( bytesCopied > 0 )
   writeFile( outputFile, buffer, bytesCopied );
free( buffer );

cryptDestroyEnvelope( cryptEnvelope );
```

In the most extreme case (hundreds or thousands of passwords, encryption, or keying attributes 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.

De-enveloping data that 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 that are explained in various other chapters that 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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 145, or by setting the CRYPT_ENVINFO_RECIPIENT attribute as described in "S/MIME Enveloping" on page 82. Using the CRYPT_ENVINFO_-

RECIPIENT attribute is the preferred option since it lets cryptlib handle a number of the complications that arise from reading keys for you.

When cryptlib encrypts the data in the envelope, it will use the algorithm specified with the CRYPT_OPTION_ENCR_ALGO option. If you want to change the encryption algorithm which is used, you can set the CRYPT_OPTION_ENCR_-ALGO attribute for the envelope (or as a global configuration option) to the algorithm type you want, as described in "Working with Configuration Options" on page 292. Alternatively, you can push a raw session-key context into the envelope before you push in a public key, in which case cryptlib will use the context to encrypt the data rather than generating one itself.

The same operation in Java (for C# replace the `.length` with `.Length`) is:

```
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* crypt.UNUSED */, crypt.FORMAT_CRYPTLIB );

/* Add the public key */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_PUBLICKEY,
   pubKeyContext );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_DATASIZE,
   message.length );
int bytesCopied = crypt.PushData( cryptEnvelope, message );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, envelopedData,
   envelopedData.length );

crypt.DestroyEnvelope( cryptEnvelope );
```

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and the private decryption key required
   to de-envelope it, and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PRIVATEKEY,
   privKeyContext );
cryptFlushData( cryptEnvelope );
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, cryptUser /* CRYPT_UNUSED */,
   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_ATTRIBUTE_CURRENT,
   &requiredAttribute );
if( requiredAttribute != CRYPT_ENVINFO_PRIVATEKEY )
   /* Error */;
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
cryptFlushData( cryptEnvelope );

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

The Visual Basic equivalent is:

```
Dim cryptEnvelope As Long
Dim requiredAttribute As CRYPT_ATTRIBUTE_TYPE
Dim bytesCopied As Long
Dim status As Long

' Create the envelope and add the private key and data
cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_AUTO
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_KEYSET_DECRYPT, _
   privateKeyset
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_ATTRIBUTE_CURRENT, _
   requiredAttribute
If ( requredAttribute <> CRYPT_ENVINFO_PRIVATEKEY ) Then
   ' Error
End If
cryptSetAttributeString cryptEnvelope, CRYPT_ENVINFO_PASSWORD, _
   password, len( password )
cryptFlushData cryptEnvelope

' 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.

Note that there is one situation where having cryptlib retrieve the private key from a keyset isn't a good idea and that's when you're processing a high volume of messages with a single key or one of a small number of keys. Retrieving and unwrapping a private key has a considerable CPU overhead since it's deliberately made to be a slow operation in order to defeat brute-force guessing attacks on the key. This means that quite a bit of CPU time is required each time a private key is read and unwrapped. If you're using the key repeatedly to process many messages then it makes more sense to read it into memory once with **cryptGetPrivateKey** and then add it on each decrypt as a CRYPT_ENVINFO_PRIVATEKEY, rather than having cryptlib read and unwrap it from a CRYPT_ENVINFO_KEYSET_DECRYPT each time you need it.

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, cryptUser /* CRYPT_UNUSED */,
   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_ATTRIBUTE_CURRENT,
   &requiredAttribute );
if( requiredAttribute != CRYPT_ATTRIBUTE_NONE )
   /* Error */;
cryptFlushData( cryptEnvelope );

/* 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 that 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, cryptUser /* CRYPT_UNUSED */,
   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_ATTRIBUTE_CURRENT,
   &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 );
   }
cryptFlushData( cryptEnvelope );

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

As with password-based enveloping, cryptlib can use authenticated encryption that protects the envelope contents from tampering as well as just encrypting them if you've configured this by setting the CRYPT_ENVINFO_INTEGRITY protection level for the envelope (for password-based enveloping it's enabled by default). If the encrypted data has been tampered with in any way, cryptlib will return a CRYPT_-ERROR_SIGNATURE when it finishes processing the encrypted data, with a full discussion of authenticated encryption given in "Password-based Encryption Enveloping" on page 60.

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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.

The Java version of the signed enveloping process (for C# replace the `.length` with `.Length`) is:

```
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* crypt.UNUSED */, crypt.FORMAT_CRYPTLIB );

/* Add the public key */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_SIGNATURE,
   sigKeyContext );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_DATASIZE,
   message.length );
int bytesCopied = crypt.PushData( cryptEnvelope, message );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, envelopedData,
   envelopedData.length );

crypt.DestroyEnvelope( cryptEnvelope );
```

The Visual Basic equivalent is:

```
cryptCreateEnvelope cryptEnvelope, cryptUser, CRYPT_FORMAT_CRYPTLIB

' Add the signing key
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_SIGNATURE, _
   sigKeyContext

' Add the data size information and data, wrap up the processing,
' and pop out the processed data
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_DATASIZE, messageLength
cryptPushData cryptEnvelope, message, messageLength, bytesCopied
cryptFlushData cryptEnvelope
cryptPopData cryptEnvelope, envelopedData, envelopedDataBufferSize, _
   bytesCopied

cryptDestroyEnvelope cryptEnvelope
```

When cryptlib signs the data in the envelope, it will hash it with the algorithm specified with the CRYPT_OPTION_ENCR_HASH option, optionally modified by the CRYPT_OPTION_ENCR_HASHPARAM option. If you want to change the hashing algorithm which is used, you can set the CRYPT_OPTION_ENCR_HASH attribute and optionally the CRYPT_OPTION_ENCR_HASHPARAM for the envelope (or as a global configuration option) to the algorithm type you want, as described in "Working with Configuration Options" on page 292. Alternatively, you can push a hash context into the envelope before you push in a signature key, in which case cryptlib will associate the signature key with the last hash context you pushed in.

If you're worried about some obscure (and rather unlikely) attacks on private keys, you can enable the CRYPT_OPTION_MISC_SIDECHANNELPROTECTION option as explained in "Working with Configuration Options" on page 292.

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, cryptUser /* CRYPT_UNUSED */,
   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 );
cryptFlushData( cryptEnvelope );
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;
int bytesCopied, signatureResult, status;

/* Create the envelope and add the signature check keyset */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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 );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, message, messageBufferSize,
   &bytesCopied );

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

The same process in Java (for C# replace the `.length` with `.Length`) is:

```
/* Create the envelope and add the signature check keyset */
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* crypt.UNUSED */, crypt.FORMAT_AUTO );
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_KEYSET_SIGCHECK,
   sigCheckKeyset );

/* Push in the signed data and pop out the recovered message */
int bytesCopied = crypt.PushData( cryptEnvelope, envelopedData );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, message, message.length );

/* Determine the result of the signature check */
int signatureResult = crypt.GetAttribute( cryptEnvelope,
   crypt.ENVINFO_SIGNATURE_RESULT );
```

The Visual Basic version is:

```
Dim signatureResult As Long

' Create the envelope and add the signature check keyset
cryptCreateEnvelope cryptEnvelope, cryptUser, 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
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).

Most signed data in use today uses a format popularised in S/MIME that includes the signature verification key with the data being signed as a certificate chain. For this type of data you don't need to provide a signature verification key, since it's already included with the signed data. Details on creating and processing data in this format is given in "S/MIME Enveloping" on page 82.

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, cryptUser /* CRYPT_UNUSED */,
   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, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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 multi-user 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) either a private decryption key or a password could be used to decrypt data.

Similarly, an envelope can have multiple signatures applied to it:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CRYPTLIB );

/* Add two different signing keys to the envelope */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   cryptSigKey1 );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   cryptSigKey2 );
```

```
/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

In this case the envelope will be signed by both keys. As with password-based enveloping, this can also be extended indefinitely to allow additional signatures on the data, although it would be somewhat unusual to place more than one or two signatures on a piece of data.

When de-enveloping data that has been enveloped with a choice of multiple attributes, cryptlib builds a list of the attributes required to decrypt or verify the signature on the data, and allows you to query the required attribute information and choose the one you want to work with.

Processing Multiple De-enveloping Attributes

The attributes required for de-enveloping are managed through the use of an attribute cursor as described in "Attribute Lists and Attribute Groups" on page 41. You can use the attribute cursor to determine which attribute is required for the de-enveloping process. Once you're iterating through the attributes, all that's left to do is to plug in the appropriate handler routines to manage each attribute requirement that could be encountered. 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 processing loop. For example to try a password against all of the possible passwords that might decrypt the message that was enveloped above you would use:

```
int status

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

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

      /* Get the type of the required attribute at the cursor position
         */
      cryptGetAttribute( envelope, CRYPT_ATTRIBUTE_CURRENT,
         &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_ATTRIBUTE_CURRENT_GROUP,
             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.

De-enveloping a public-key encrypted envelope works identically, except that you'd use the CRYPT_ENVINFO_PRIVATEKEY attribute at each step instead of the CRYPT_ENVINFO_PASSWORD attribute (as discussed in "Public-Key Encrypted Enveloping" on page 71 though, it's much easier to push in a decryption keyset and have cryptlib perform the key management for you than to explicitly add the private key yourself).

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_ATTRIBUTE_CURRENT,
   &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.

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, many envelopes carry their own signature verification keys with them) and display the resulting signature information. Unlike encryption de-enveloping attributes, cryptlib won’t delete the signature information once it has been processed, so you can re-read the information multiple times:

```
int status

if( cryptSetAttribute( envelope, CRYPT_ATTRIBUTE_CURRENT_GROUP,
   CRYPT_CURSOR_FIRST ) == CRYPT_OK )
   do
      {
      CRYPT_ATTRIBUTE_TYPE requiredAttribute;
      int sigResult;

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

      /* Make sure we really do have signature */
      if( requiredAttribute != CRYPT_ENVINFO_SIGNATURE )
         /* Error */;

      /* Get the signature result */
      status = cryptSetAttribute( envelope,
         CRYPT_ENVINFO_SIGNATURE_RESULT, & sigResult );
      }
   while( cryptStatusOK( status ) && \
          cryptSetAttribute( envelope, CRYPT_ATTRIBUTE_CURRENT_GROUP,
             CRYPT_CURSOR_NEXT ) == CRYPT_OK );
```

This steps through each signature in turn and reads the result of the signature verification for that signature, stopping when an invalid signature is found or when all signatures are processed.

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_CONTENT_-*type*. For example to specify that the data being enveloped is signed data you would use:

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_CONTENTTYPE,
   CRYPT_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. The other content types are described in "Other Certificate Object Extensions" on page 271.

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_CONTENT_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.

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 is 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 that doesn't tie it to the use of certificates (so it'll work with PGP/OpenPGP 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 that were added in CMS.

You can specify the use of the restricted CMS/SMIME format when you create an envelope with the formatting specifier CRYPT_FORMAT_CMS or CRYPT_-FORMAT_SMIME (they're almost identical, the few minor differences are explained in "Extra Signature Information" on page 89), which tells cryptlib to use the restricted CMS/SMIME rather than the (default) unrestricted CMS format. You can also use the format specifiers with **cryptExportKeyEx** and **cryptCreateSignatureEx** (which take as their third argument the format specifier) as explained in "Exchanging Keys" on page 203, and "Signing Data" on page 209.

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 that 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, cryptUser /* CRYPT_UNUSED */,
   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. In addition for backwards-compatibility with existing implementations cryptlib doesn't use authenticated encryption as it would for standard cryptlib envelopes. This means that when you envelope data with

CRYPT_FORMAT_CMS or CRYPT_FORMAT_SMIME the data is encrypted but has no protection against tampering. If an attacker modifies some of the encrypted data this will result in a corresponding (and undetectable) modification to the decrypted data (this is a standard property of encryption algorithms, they offer confidentiality protection but no integrity protection). In extreme cases an attacker could, for example, modify the payment amount in an encrypted payment instruction by doing as little as flipping a single bit of encrypted data, turning a $100 payment into a $100,000 payment or vice versa.

To protect against this you can enable the use of authenticated encryption as for CRYPT_FORMAT_CRYPTLIB envelopes, which both encrypts the envelope's contents and protects it against tampering by attackers. To enable the use of authenticated encryption, set the CRYPT_ENVINFO_INTEGRITY attribute to CRYPT_INTEGRITY_FULL:

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_INTEGRITY,
   CRYPT_INTEGRITY_FULL );
```

Authenticated encryption is described in more detail in "Data Enveloping" on page 53.

Encrypted Enveloping

S/MIME supports password-based enveloping in the same way as ordinary cryptlib envelopes (in fact the two formats are identical apart from the fact that cryptlib envelopes default to using authenticated encryption while S/MIME envelopes don't for compatibility with older S/MIME implementations). 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 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_SMIME );

/* Add the certificate */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_PUBLICKEY,
   certificate );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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, cryptUser /* CRYPT_UNUSED */,
   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, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

The same thing in Java (for C# replace the `.length` with `.Length`) is:

```
int cryptEnvelope = crypt.CreateEnvelope( cryptUser
   /* crypt.UNUSED */, crypt.FORMAT_SMIME );

/* Add the encryption keyset and recipient email address */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_KEYSET_ENCRYPT,
   cryptKeyset );
crypt.SetAttributeString( cryptEnvelope, crypt.ENVINFO_RECIPIENT,
   "person@company.com" );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
crypt.SetAttribute( cryptEnvelope, crypt.ENVINFO_DATASIZE,
   message.length );
int bytesCopied = crypt.PushData( cryptEnvelope, message );
crypt.FlushData( cryptEnvelope );
bytesCopied = crypt.PopData( cryptEnvelope, envelopedData,
   envelopedData.length );

crypt.DestroyEnvelope( cryptEnvelope );
```

The Visual Basic equivalent is:

```
cryptCreateEnvelope cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_SMIME

' Add the encryption keyset and recipient email address
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_KEYSET_ENCRYPT, _
   cryptKeyset
cryptSetAttributeString cryptEnvelope, CRYPT_ENVINFOR_RECIPIENT, _
   "person@company.com", 18

' Add the data size information and data, wrap up the processing,
' and pop out the processed data
cryptSetAttribute cryptEnvelope, CRYPT_ENVINFO_DATASIZE, messageLength
cryptPushData cryptEnvelope, message, messageLength, bytesCopied
cryptFlushData cryptEnvelope
cryptPopData cryptEnvelope, envelopedData, envelopedDataBufferSize, _
   bytesCopied

cryptDestroyEnvelope cryptEnvelope
```

For each message recipient that you add, cryptlib will look up the 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 want to handle this case yourself, you have to use a keyset query to search the duplicate certificates and select the appropriate one as described in “Handling Multiple Certificates with the Same Name” on page 149.

The encryption keyset doesn’t have to be local. If you use an HTTP keyset as described in “HTTP Keysets” on page 140, cryptlib will fetch the required certificate directly from the remote CA, saving you the effort of having to maintain and update a local set of certificates. This use of HTTP keysets makes it very easy to distribute certificates over the Internet.

De-enveloping works as for standard enveloping:

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

/* Create the envelope and add the private key keyset and data */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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_ATTRIBUTE_CURRENT,
   &requiredAttribute );
if( requiredAttribute != CRYPT_ENVINFO_PRIVATEKEY )
   /* Error */;
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
cryptFlushData( cryptEnvelope );

/* 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 "Public-Key Encrypted Enveloping" on page 71.

Digitally Signed Enveloping

S/MIME digitally signed enveloping works just like standard enveloping except that the signing key is restricted to one that 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_SMIME );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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 that 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 that 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 for verification. This means that you don't have to push a signature verification keyset or key into the envelope because the required certificate is already included with the data:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the enveloped data and pop out the recovered message */
cryptPushData( cryptEnvelope, envelopedData, envelopedDataLength,
   &bytesCopied );
cryptFlushData( cryptEnvelope );
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_ENVINFO_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**. You may also want to perform a validity check using RTCS, revocation checking using CRLs, OCSP, or SCVP, and any other certificate checks that you consider necessary. More details on working with certificate chains are given in "Certificate Chains" on page 236, details on basic signed enveloping (including its use with crypto devices like smart cards and hardware crypto accelerators) are given in "Digitally Signed Enveloping" on page 75, details on validity checking with RTCS are given in "Certificate Status Checking using RTCS" on page 168, details on revocation checking with OCSP are given in "Certificate Revocation Checking using OCSP" on page 174, and details on validity checking with SCVP are given in "Certificate Status Checking using SCVP" on page 177. Remember to destroy the certificate chain object with **cryptDestroyCert** once you're finished with it.

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 set the envelope's CRYPT_ENVINFO_DETACHED-SIGNATURE attribute to 'true' (any nonzero value) before you push in any data

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_SMIME );
```

```
/* Add the signing key and specify that we're using a detached
   signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   1 );

/* Add the data size information and data, wrap up the processing, and
   pop out the detached signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
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 **cryptFlushData** to wrap up the processing, you need to push in the data that was signed by the detached signature as the second processing step:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );

/* Push in the detached signature */
cryptPushData( cryptEnvelope, detachedSignature, detachedSigLength,
   &bytesCopied );
cryptFlushData( cryptEnvelope );

/* Push in the data */
cryptPushData( cryptEnvelope, data, dataLength, NULL );
cryptFlushData( cryptEnvelope );

/* 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 de-envelope, which means there's nothing to pop out of the envelope apart from the signing certificate chain that 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_DETACHEDSIGNATURE 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_DETACHEDSIGNATURE,
   &isDetachedSignature );
if( isDetachedSignature )
   /* Detached signature */;
else
   /* Signed data + signature */;
```

Alternative Detached Signature Processing

Besides the method described above there is a second way to verify a detached signature which involves hashing the data yourself and then adding the hash to the envelope rather than pushing the data into the envelope and having it hashed for you.

This is useful in situations where the signed data is present separate from the signature, or is in a non-standard format (for example an AuthentiCode signed file) that can't be recognised by the enveloping code.

Verifying a detached signature in this manner is a slight variation of the standard detached signature verification process in which you first add to the envelope the hash value for the signed data and then push in the detached signature:

```
CRYPT_CONTEXT hashContext;
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;

/* Create the hash context and hash the signed data */
cryptCreateContext( &hashContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_SHA2 );
cryptEncrypt( hashContext, signedData, dataLength );
cryptEncrypt( hashContext, signedData, 0 );

/* Create the envelope, tell it that the hash is for a detached
   signature, and add the hash */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   TRUE );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_HASH, hashContext );
cryptDestroyContext( hashContext );

/* Add the detached signature */
cryptPushData( cryptEnvelope, signatureData, signatureDataLength,
   &bytesCopied );
cryptFlushData( cryptEnvelope );

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

cryptDestroyEnvelope( cryptEnvelope );
```

When you push in the detached signature cryptlib will verify that the hash information in the signature matches the hash that you've supplied. If the two don't match, cryptlib will return CRYPT_ERROR_SIGNATURE to indicate that the signature can't be verified using the given values. Because of this check, you must add the hash before you push in the detached signature.

Finally, you can also create a detached signature in this manner, although it's probably not terribly useful to do things this way since you can just have the enveloping code do it for you automatically:

```
CRYPT_CONTEXT hashContext;
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

/* Create the hash context and hash the data to be signed */
cryptCreateContext( &hashContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_SHA2 );
cryptEncrypt( hashContext, data, dataLength );
cryptEncrypt( hashContext, data, 0 );

/* Create the envelope, add hash and signing key, and specify that
   we're using a detached signature */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_AUTO );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_HASH, hashContext );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   1 );
cryptDestroyContext( hashContext );

/* Wrap up the processing and pop out the detached signature */
   cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, detachedSignature,
   detachedSignatureBufferSize, &bytesCopied );
cryptDestroyEnvelope( cryptEnvelope );
```

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, with the details of what's added based on the setting of the CRYPT_OPTION_CMS_DEFAULTATTRIBUTES option as described in "Working with Configuration Options" on page 292. If this option is set to false (zero), cryptlib won't add any additional signature information, which minimises the size of the resulting signature. If this option is set to true (any nonzero value), cryptlib will add default signing attributes such as the signing time for you.

You can also handle the extra signing information 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. 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CMS_ATTRIBUTES );
/* ... */

/* Create the envelope and add the signing key and signature
   information */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_CMS );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE_EXTRADATA,
   cmsAttributes );
cryptDestroyCert( cmsAttributes );

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

You can also use this facility to extend or overwrite the attributes added by cryptlib. For example if you wanted to add a security label to the data being signed, you would add it to the CMS attribute object and add that to the envelope. cryptlib will then add any additional required information (for example the signing time) and finally generate the signature using the combined collection of attributes. This means that you can fill in whatever attributes you want, and cryptlib till take care of the rest for you.

Verifying a signature that 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 in the same manner as standard certificate attributes, for example you may want to display any relevant information to the user. More details on working with these attributes are given in "Certificate Extensions" on page 246, and the attributes themselves are covered in "Other Certificate Object Extensions" on page 271.

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 that 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 some 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 AES (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 AES. With a CRYPT_FORMAT_CMS signature this additional S/MIME-specific information isn't needed so cryptlib doesn't include it.

Timestamping

In addition to the standard signature information which is provided by the signer, cryptlib also supports the use of a message timestamp which is provided by an external timestamp authority (TSA). Timestamping signed data in an envelope is very simple and requires only the addition of a CRYPT_ENVINFO_TIMESTAMP attribute to tell cryptlib which TSA to obtain the timestamp from. The TSA is specified as a TSP session object as described in "Secure Sessions" on page 102. For example to specify a TSA located at `http://www.timestamp.com/-tsa/request.cgi`, you would create the TSP session with:

```
CRYPT_SESSION cryptSession;

/* Create the TSP session and add the server name */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TSP );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "http://www.timestamp.com/tsa/request.cgi", 40 );
```

You can also specify additional session information in the usual manner for cryptlib sessions, after which you add the session to the envelope. Once you've added it, you can destroy it since it's now managed by the envelope:

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_TIMESTAMP,
   cryptSession );
cryptDestroySession( cryptSession );
```

When cryptlib signs the data in the envelope, it will communicate with the TSA to obtain a timestamp on the signature, which is then included with the other signed data. This timestamp can be verified at a later date to prove that the envelope was indeed signed at the indicated time.

Since communicating with a TSA over a network can be a slow process, the signature generation may take somewhat longer than usual. When the timestamp is created cryptlib doesn't communicate any part of the message or any indication of its contents to the TSA, it merely sends it the message signature information which is then countersigned by the TSA. In this way no confidential or sensitive information is leaked to the outside world through the timestamping process.

A time-stamped message appears the same as a standard signed message, with the exception that the timestamp data is present as additional signature information of type CRYPT_ENVINFO_TIMESTAMP. You can read the timestamp data in the same way that you read other extra signature information:

```
CRYPT_ENVELOPE timeStamp;

cryptGetAttribute( cryptEnvelope, CRYPT_ENVINFO_TIMESTAMP,
   &timestamp );
```

The returned timestamp is a standard signed envelope object that you can check in the usual manner, for example by verifying the signature on the timestamp data and checking the certificates used for the timestamp signature.

PGP

PGP is a standard format for encrypting, signing, and compressing data. The original format, PGP 2.x or PGP classic, has since been superseded by OpenPGP, partially implemented in PGP 5.0 and later fully in NAI PGP, GPG, and various variations such as the ckt builds. cryptlib can read both the PGP 2.x and OpenPGP formats, including handling for assorted variations and peculiarities of different implementations. As output cryptlib produces data in the OpenPGP format, which can be read by any recent PGP implementation. Note that PGP 2.x used the patented IDEA encryption algorithm (see "Algorithms" on page 341 for details), if you're using the code for commercial purposes you need to either obtain a license for IDEA or use only the OpenPGP format (which cryptlib does by default anyway, so this usually isn't a concern).

You can specify the use of the PGP format when you create an envelope with the formatting specifier CRYPT_FORMAT_PGP, which tells cryptlib to use the PGP format rather than the (default) CMS format. cryptlib doesn't restrict the use of PGP envelopes to PGP keys. Any type of keys, including standard cryptlib keys and X.509 certificates, can be used with PGP envelopes. By extension it's also possible to use smart cards, crypto accelerators, and Fortezza cards with PGP envelopes (as an extreme example, it's possible to use a Fortezza card to create a PGP envelope).

Note that the PGP format places a number of restrictions on data types, algorithms, and keys that aren't imposed by the PKCS #7/CMS/SMIME format. In particular you need to be aware of the fact that some operations that are possible in CMS won't be possible when you're using the same code with the PGP format. If you want the greatest flexibility then you should use the CMS format instead of the PGP one.

PGP Enveloping

To create an envelope that uses the PGP format, call **cryptCreateEnvelope** as usual but specify a format type of CRYPT_FORMAT_PGP instead of the usual CRYPT_FORMAT_CRYPTLIB:

```
CRYPT_ENVELOPE cryptEnvelope;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_PGP );

/* Perform enveloping */

cryptDestroyEnvelope( cryptEnvelope );
```

Creating the envelope in this way restricts cryptlib to using the PGP data format instead of the more flexible data format which is used for envelopes by default. This imposes a number of restrictions on the use of envelopes that are described in more detail in the sections that cover individual PGP enveloping types. One restriction that applies to all enveloping types is that PGP requires the presence of the CRYPT_ENVINFO_DATASIZE attribute before data can be enveloped. This attribute is described in more detail in "Data Size Considerations" on page 56. If you try to push data into an envelope without setting the CRYPT_ENVINFO_-DATASIZE attribute, cryptlib will return CRYPT_ERROR_NOTINITED to indicate that you haven't provided the information which is needed for the enveloping to proceed.

Another peculiarity of the PGP data format is that messages created by PGP implementations tend to nest multiple layers of processing within each other, so that decrypting a PGP-encrypted message won't necessarily produce the original data but a further PGP message containing compressed data, and that in turn will contain the original data. Alternatively, there could be a layer of PGP-signed data in there as well. In order to figure out what you've got to, once you've de-enveloped PGP data you need to read the CRYPT_ENVINFO_CONTENTTYPE attribute as described in "Nested Envelopes" on page 81 to see what the inner content type is and if necessary process that further. Once you get to CRYPT_CONTENT_DATA, you've reached the original data.

Encrypted Enveloping

PGP supports password-based enveloping in the same general way as ordinary cryptlib envelopes. However, due to constraints imposed by the PGP format, it's not possible to mix password- and public-key-based key exchange actions in the same envelope. In addition it's not possible to specify more than one password for an envelope. If you try to add more than one password, or try to add a password when you've already added a public key or vice versa, cryptlib will return CRYPT_ERROR_INITED to indicate that the key exchange action has already been set.

Public-key based PGP enveloping works the same way as standard cryptlib enveloping. For example to encrypt data using the a public key you would use:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_PGP );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Since the key will originally have come from a keyset, a simpler alternative to reading the key 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_PGP );

/* 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, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

For each message recipient that you add, cryptlib will look up the 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 key details for you and makes it possible to manage problem areas such as cases where the same email address is present for multiple keys of which only one is valid for message encryption.

De-enveloping works as for standard enveloping:

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

/* Create the envelope and add the private key keyset and data */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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_ATTRIBUTE_CURRENT,
   &requiredAttribute );
if( requiredAttribute != CRYPT_ENVINFO_PRIVATEKEY )
   /* Error */;
cryptSetAttributeString( cryptEnvelope, CRYPT_ENVINFO_PASSWORD,
   password, passwordLength );
cryptFlushData( cryptEnvelope );

/* 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, is given in "Public-Key Encrypted Enveloping" on page 71.

Digitally Signed Enveloping

PGP digitally signed enveloping works just like standard enveloping. 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_PGP );

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

/* Add the data size information and data, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Verifying the signature works in the usual way:

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

/* Create the envelope and add the signature check keyset */
cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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 );
cryptFlushData( cryptEnvelope );
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).

When you sign data in the PGP format, the nested content type is always set to plain data. This is a limitation of the PGP format that always signs data as the innermost step, so that what's signed is always plain data. In addition to this restriction, it's not possible to have more than one signer per envelope. Multiple signers requires the use of nested envelopes, however it's necessary to intersperse a layer of encryption or compression between each signature pass since PGP can't easily distinguish which signature belongs to which signature pass. In general it's best not to try to apply multiple signatures to a piece of data.

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 set the envelope's CRYPT_ENVINFO_DETACHED-SIGNATURE attribute to 'true' (any nonzero value) before you push in any data

```
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_FORMAT_PGP );

/* Add the signing key and specify that we're using a detached
   signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_SIGNATURE,
   sigKeyContext );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   1 );

/* Add the data size information and data, wrap up the processing, and
   pop out the detached signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, detachedSignature,
   detachedSignatureBufferSize, &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Verifying a detached signature works somewhat differently from standard cryptlib detached signature processing since the PGP format doesn't differentiate between standard and detached signatures. Because of this lack of differentiation, it's not possible for cryptlib to automatically determine whether a signature should have data associated with it or not. Normally, cryptlib assumes that a signature is associated with the data being signed, which is the most common case. When verifying a detached signature, you need to use the alternative signature processing technique that involves hashing the data yourself and then adding the hash to the envelope rather than pushing the data into the envelope and having it hashed for you. Since PGP hashes further information after hashing the data to be signed, you shouldn't complete the hashing before you push the hash context into the envelope. This is in contrast to standard cryptlib detached signature processing which requires that you complete the hashing before pushing the context into the envelope:

```
CRYPT_CONTEXT hashContext;
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied, sigCheckStatus;

/* Create the hash context and hash the signed data without completing
   the hashing */
cryptCreateContext( &hashContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_SHA2 );
cryptEncrypt( hashContext, data, dataLength );

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

/* Tell the envelope that the hash is for a detached signature, then
   add the hash and follow it with the detached signature */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DETACHEDSIGNATURE,
   1 );
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_HASH, hashContext );
cryptPushData( cryptEnvelope, detachedSignature,
   detachedSignatureBufferSize, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptDestroyContext( hashContext );

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

cryptDestroyEnvelope( cryptEnvelope );
```

When you push in the detached signature cryptlib will verify that the hash information in the signature matches the hash that you've supplied. If the two don't match, cryptlib will return CRYPT_ERROR_SIGNATURE to indicate that the signature can't be verified using the given values. Because of this check, you must add the hash before you push in the detached signature.

From Envelopes to email

The enveloping process produces binary data as output that then needs to be wrapped up in the appropriate MIME headers and formatting before it can really be called S/MIME or PGP mail. 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.

Note that cryptlib is a security toolkit and not a mail client or server. Although cryptlib provides all the crypto functionality needed to implement S/MIME and PGP, it cannot send or receive email, process MIME message parts or base64 or PGP ASCII encoding, or otherwise act as a mail agent. These functions are performed y mail-handling software. For mail-processing operations you need to combine it with mail-handling software of the kind described further on.

S/MIME email

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 that 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 that 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 cryptlib's 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=sha256` 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_ENVINFO_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 de-enveloping data (this will normally be handled for you by the mailer code or software interface to the mail system you're using).

PGP email

Traditionally, PGP has employed its own email encapsulation format that predates MIME and isn't directly compatible with it. A PGP message is delimited with the string `-----BEGIN PGP MESSAGE-----` and `-----END PGP MESSAGE-----`, with the (binary) message body present in base64-encoded format between the delimiters. The body is followed by a base64-encoded CRC24 checksum calculated on the message body before base64-encoding. In addition the body may be preceded by one or more lines of type-and-value pairs containing additional information such as software version information, and separated from the body by a blank line. More details on the format are given in the PGP standards documents.

An example of a PGP email message is:

```
-----BEGIN PGP MESSAGE-----
Version: cryptlib 3.1

base64-encoded message body
base64-encoded CRC24 checksum
-----END PGP MESSAGE-----
```

Signed data with a detached signature is delimited with `-----BEGIN PGP SIGNED MESSAGE-----` at the start of the message, followed by `-----BEGIN PGP SIGNATURE-----` and `-----END PGP SIGNATURE-----` around the signature that follows. The signature follows the standard PGP message-encoding rules given above:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

message body
-----BEGIN PGP SIGNATURE-----
Version: cryptlib 3.1

base64-encoded signature
base64-encoded CRC24 checksum
-----END PGP SIGNATURE-----
```

The example above shows another use for the type-and-value lines, in this case to indicate the hashing algorithm used in the signature to allow one-pass processing of the message.

In addition to the traditional PGP format, there exists a mechanism for encapsulating the traditional PGP format in an additional layer of MIME wrapping. This isn't true MIME message handling since it merely wraps MIME headers around the existing PGP email encapsulation rather than using the full MIME capabilities directly as does S/MIME. This format is almost never used, with software expected to use the traditional PGP format instead. If you need more information about PGP/MIME, you can find it in the PGP standards documentation.

Implementing S/MIME and PGP email using cryptlib

Most of the MIME processing and encoding issues described above will be handled for you by the mail software that cryptlib is used with. To use cryptlib to handle S/MIME and PGP email messages, you would typically register the various MIME types with the mail software and, when they are encountered, the mailer will hand the message content (the data that 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 and PGP email messages.

Note that cryptlib is a security toolkit and not a mail client or server. Although cryptlib provides all the crypto functionality needed to implement S/MIME and PGP, it cannot send or receive email, process MIME message parts, or otherwise act as a mail agent. For mail-processing operations you need to combine it with mail-handling software of the kind described below.

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 or PGP), 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.

Eudora

Eudora handles MIME content types through plug-in translators that are called through two functions, `ems_can_translate` and `ems_translate_file`. Eudora calls `ems_can_translate` with an `emsMIMEtype` parameter that contains information on the MIME type contained in the message. If this is an S/MIME or PGP 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 de-enveloping process depending on whether the translator is an on-arrival or on-display one (used for de-enveloping incoming messages) or a Q4-transmission or Q4-completion one (used for enveloping outgoing messages).

MAPI

MAPI (Microsoft's mail API) defines two types of mailer extensions that allow cryptlib-based S/MIME and PGP 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 pre-processors 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 that 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.

Windows 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 and PGP data using a cryptlib-based application.

Secure Sessions

cryptlib's secure session interface provides a session-oriented equivalent to envelope objects that can be used to secure a communications link with a host or server or otherwise communicate with another system over a network. Secure sessions can include SSH and TLS sessions, general request/response-style communications sessions can include protocols such as the certificate management protocol (CMP), simple certificate enrolment protocol (SCEP), real-time certificate status protocol (RTCS), online certificate status protocol (OCSP), server-based certificate validation protocol (SCVP), and timestamping protocol (TSP). As with envelopes, cryptlib takes care of all of the session details for you so that all you need to do is provide basic communications information such as the name of the server or host to connect to and any other information required for the session such as a password or certificate. cryptlib takes care of establishing the session and managing the details of the communications channel and its security parameters.

For a secure session like SSH and TLS, the only thing that you'll need to do yourself is provide information to let cryptlib verify the identity of the server that you're connecting to or the client that's connecting to you, see "Verifying Session Security Information" on page 126 for details on how to do this.

Secure sessions are very similar to envelopes, with the main difference being that while an envelope is a pure data object into which you can push data and pop the processed form of the same data, a session is a communications object into which you push data and then pop data that constitutes a response from a remote server or client. This means that a session object can be viewed as a bottomless envelope through which you can push or pop as much data as the other side can accept or provide.

As with an envelope, you use a session object by adding to it action objects and resources such as user names and passwords that control the interaction with the remote server or client and then push in data intended for the remote system and pop out data coming from the remote system. For example to connect to a server using SSH and obtain a directory of files using the ls command you would do the following:

```
create the session;
add the server name, user name, and password;
(add server verification information if available);
activate the session;
push data "ls";
pop the result of the ls command;
destroy the session
```

That's all that's necessary. Since you've added a user name and password, cryptlib knows that it should establish an encrypted session with the remote server and log on using the given user name and password. From then on all data which is exchanged with the server is encrypted and authenticated using the SSH protocol.

Creating an SSH server session is equally simple. In this case all you need is the server key:

```
create the session;
add the server key;
activate the session;
pop client data;
push server response;
destroy the session
```

When you activate the session, cryptlib will listen for an incoming connection from a client and return once a secure connection has been negotiated, at which point communication proceeds as before.

Creating/Destroying Session Objects

Secure sessions are accessed as session objects that work in the same general manner as other cryptlib objects. You create a session using **cryptCreateSession**, specifying the user who is to own the session object or CRYPT_UNUSED for the default, normal user, and the type of session that you want to create. This creates a session object ready for use in securing a communications link or otherwise communicating with a remote server or client. Once you've finished with the session, you use **cryptDestroySession** to end the session and destroy the session object:

```
CRYPT_SESSION cryptSession;

cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   sessionType );

/* Communicate with the remote server or client */

cryptDestroySession( cryptSession );
```

The available session types are:

Session	Description
CRYPT_SESSION_CMP	Certificate management protocol (CMP).
CRYPT_SESSION_OCSP	Online certificate status protocol (OCSP).
CRYPT_SESSION_RTCS	Real-time certificate status protocol (RTCS).
CRYPT_SESSION_SCEP	Simple certificate enrolment protocol (SCEP).
CRYPT_SESSION_SCVP	Server-based certificate validation protocol (SCVP).
CRYPT_SESSION_SSH	Secure shell (SSH).
CRYPT_SESSION_TLS	Transport layer security (TLS), formerly SSL.
CRYPT_SESSION_TSP	Timestamping protocol (TSP).

This section will mainly cover the secure communications session types such as SSH and TLS. CMP, SCEP, RTCS, OCSP and SCVP client sessions are certificate management services that are covered in "Obtaining Certificates using CMP", "Obtaining Certificates using SCEP", "Certificate Status Checking using RTCS", "Certificate Revocation Checking using OCSP", and "Certificate Status Checking using SCVP" on pages 174, 168, 168, 175, and 177, and a TSP client session is an S/MIME service which is covered in "Timestamping" on page 90. RTCS, OCSP, SCVP, and TSP server sessions are standard session types and are also covered here. CMP and SCEP server sessions are somewhat more complex and are covered in "Managing a CA using CMP or SCEP" on page 186. The general principles covering sessions apply to all of these session types, so you should familiarise yourself with the operation of session objects and associated issues such as network proxies and timeouts before trying to work with these other session types.

By default the secure communications session object which is created will have an internal buffer whose size is appropriate for the type of security protocol which is being employed. The size of the buffer may affect the amount of extra processing that cryptlib needs to perform, so that a large buffer can reduce the amount of copying to and from the buffer, but will consume more memory. If want to use a buffer for a secure communications session which is larger than the default size, you can specify its size using the CRYPT_ATTRIBUTE_BUFFERSIZE attribute after you've created the session. For example if you wanted to set the buffer for an SSH session to 64 kB you would use:

```
CRYPT_SESSION cryptSession;

cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );
cryptSetAttribute( cryptSession, CRYPT_ATTRIBUTE_BUFFERSIZE, 65536L );

/* Communicate with the remote server or client */

cryptDestroySession( cryptSession );
```

Since cryptlib streams data through the session object, the internal buffer size doesn't limit how much data you can push and pop (for example you could push 1 MB of data into a session object with a 32 kB internal buffer), the only reason you'd want to change the size is to provide tighter control over memory usage by session objects. Unless you're absolutely certain that the other side will only send very small data quantities, you shouldn't shrink the buffer below the default size set by cryptlib since the protocols that cryptlib implements have certain fixed bounds on packet sizes that need to be met, making the buffer too small would make it impossible to process data being sent by the other side.

Note that the CRYPT_SESSION is passed to **cryptCreateSession** by reference as the function modifies it when it creates the session. In all other routines in cryptlib, CRYPT_SESSION is passed by value.

Client vs. Server Sessions

cryptlib distinguishes between two types of session objects, client sessions and server sessions. Client sessions establish a connection to a remote server while server sessions wait for incoming communications from a remote client. To distinguish between client and server objects, you use a session type ending in _SERVER when you create the session object. For example to create an TLS server object instead of an TLS client you would specify its type on creation as CRYPT_SESSION_TLS_-SERVER instead of CRYPT_SESSION_TLS.

Because server sessions wait for an incoming connection request to arrive, you need to run each one in its own thread if you want to handle multiple connections simultaneously (cryptlib is fully thread-safe so there's no problem with having multiple threads processing incoming connections). For example to handle up to 10 connections at once you would do the following:

```
for i = 1 to 10 do
   start_thread( server_thread );
```

where the `server_thread` is:

```
loop
   create the session;
   add required information to the session;
   activate the session;
   process client request(s);
   destroy the session;
```

More information on using cryptlib with multiple threads is given in "Multi-threaded cryptlib Operation" on page 49.

Binding to the default ports used by the various session protocols may require special privileges on some systems that don't allow normal users to bind to ports below 1024. If you need to bind to a reserved port you should consult your operating system's documentation for details on any restrictions that may apply, and may need to take special precautions if binding to one of these ports requires the use of elevated security privileges. Alternatively, you can bind to a non-default port outside the reserved range by specifying the port using the CRYPT_SESSINFO_SERVER_-PORT attribute. You can also specify which interface you want to bind to if the system has more than one by using the CRYPT_SESSINFO_SERVER_NAME attribute. If you're testing code before deploying it, it's a good idea to specify that you want to bind to localhost to avoid listening on arbitrary externally-visible interfaces. For example to listen on local port 2000 you would use:

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "localhost", 9 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SERVER_PORT, 2000 );
```

Server Names/URLs

Server names can be given using IP addresses (in dotted-decimal form for IPv4 or colon-delimited form for IPv6), DNS names, or full URLs, with optional ports and other information provided in the usual manner. You can specify the server name or URL using the CRYPT_SESSINFO_SERVER_NAME attribute and the port (if you're not using the default port fro the protocol and it isn't already specified in the

URL) using the CRYPT_SESSINFO_SERVER_PORT attribute. For example to specify a connection to the server www.server.com on port 80 you would use:

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "www.server.com", 14 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SERVER_PORT, 80 );
```

Alternatively, you could specify both in the same name:

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "www.server.com:80", 17 );
```

Since this is a web server for which port 80 is the default port, you could also use the more common:

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "http://www.server.com", 20 );
```

TLS uses a predefined port and are often used in conjunction with HTTP, so you can specify these URLs with or without the `http://` or `https://` schema prefixes. SSH similarly uses a predefined port and can be used with or without the `ssh://`, `scp://`, or `sftp://` schema prefixes. All of these protocols allow you to specify user information before the host name, separated with an '@' sign. For example to connect as "user" to the SSH server ssh.server.com you could use:

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   "ssh://user@ssh.server.com", 25 );
```

which saves having to explicitly specify the user name with the CRYPT_-SESSINFO_USERNAME attribute.

All of the PKI protocols use HTTP as their transport mechanism, so cryptlib will automatically default to using HTTP transport whether you include the `http://` schema specifier or not. The CMP and TSP protocols also have alternative, deprecated transport mechanisms identified by `cmp://`... (for CMP) and `tcp://`... (for TSP) instead of `http://`.... These are occasionally used by CAs or timestamp servers, you may need to use these instead of the HTTP default.

Server Private Keys

Most server sessions require the use of a private key in one form or another to decrypt data from the client or sign responses returned to the client. The server key is typically stored in a private key file, but for extra security may be held in a crypto device such as a crypto coprocessor or accelerator. In addition, for most session types the server key needs to be associated with a certificate or certificate chain leading up to a trusted root certificate, so that you can't use just a raw private key as the server key. You can obtain the required certificate or certificate chain by creating it yourself using cryptlib or by obtaining it from a commercial CA (it's generally much cheaper and easier to create it yourself than to obtain one from a third-party CA).

When you create or obtain the certificate for your server, you may need to specify the server name in the common name field of the certificate (how to create your own certificate is explained in "Certificates and Certificate Management" on page 154). For example if your server was www.companyname.com then the certificate for the server would contain this as its common name component (you can actually put in anything you like as the common name component, but this will result in some web browsers that use your server displaying a warning message when they connect).

SSH server sessions require a raw RSA (or optionally DSA for SSHv2) key, although you can also use one with a certificate or certificate chain attached. All other session types require one with certificate(s) attached. You add the server key as the CRYPT_SESSINFO_PRIVATEKEY attribute, for example to use a private key held in a crypto device as the server key you would use:

```
CRYPT_CONTEXT privateKey;

cryptGetPrivateKey( cryptDevice, &privateKey, CRYPT_KEYID_NAME,
   serverKeyName, NULL );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptDestroyContext( privateKey );
```

Note that, as with envelopes, the private key object can be destroyed as soon as it's added to the session, since the session maintains its own copy of the object internally.

If you're worried about some obscure (and rather unlikely) attacks on private keys, you can enable the CRYPT_OPTION_MISC_SIDECHANNELPROTECTION option as explained in "Working with Configuration Options" on page 292.

Establishing a Session

Much of the secure session process is identical to the enveloping process, so you should familiarise yourself with the general concept of enveloping as described in "Data Enveloping" on page 53 if you haven't already done so. The secure session establishment process involves adding the information which is required to connect to the remote server as a client or to establish a server, and then activating the session to establish the secure session or wait for incoming connections. This process of activating the session has no real equivalent for envelopes because envelopes are activated automatically the first time data is pushed into them.

Client sessions can also be activated automatically, however the initial handshake process which is required to activate a session with a remote server is usually lengthy and complex so it's generally better to explicitly activate the session under controlled conditions and have the ability to react to errors in an appropriate manner rather than to have the session auto-activate itself the first time that data is pushed. Server sessions that wait for an incoming connection must be explicitly activated, which causes them to wait for a client connection.

You can activate a session by setting its CRYPT_SESSINFO_ACTIVE attribute to true (any nonzero value). You can also determine the activation state of a session by reading this attribute, if it's set to true then the session is active, otherwise it's not active.

Persistent Connections

Some cryptlib session types such as CMP, SCEP, RTCS, OCSP, SCVP, and TSP provide request/response protocols rather than continuous secure sessions like SSH and TLS. In many cases it's possible to perform more than one request/response transaction per session, avoiding the overhead of creating a new connection for each transaction. To handle persistent connections for client sessions, cryptlib uses the CRYPT_SESSINFO_CONNECTIONACTIVE attribute to indicate that the connection is still active and is ready to accept further transactions. Transactions after the initial one are handled in exactly the same way as the first one, except that there's no need to create a new session object for them:

```
CRYPT_SESSION cryptSession;
int connectionActive;

/* Create the session and add the server name */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_xxx );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );

/* Perform the first transaction */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRequest1 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptResponse1 );
```

```
/* Check whether the session connection is still open */
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_CONNECTIONACTIVE,
   &connectionActive );
if( !connectionActive )
   /* The other side has closed the connection, exit */;

/* Perform the second transaction */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRequest2 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptResponse2 );
```

Note the check of the CRYPT_SESSINFO_CONNECTIONACTIVE attribute. Since not all servers support persistent connections or may time out and close the connection after a period of inactivity, it's a good idea to check that the connection is still open before trying to submit further transactions. Note also that there's no need to explicitly delete the request from the first activation of the session, cryptlib automatically does this for you once the session activation has completed. This does mean, however, that if you want to repeat the session transaction using the same data as before (which would be somewhat unusual), you need to re-add the request to the session, since the previous activation will have cleared it in preparation for the next activation.

The process on the server side is similar, after a successfully-completed client transaction you can either destroy the session or, if you want to support persistent connections, recycle the connection as for the client-side example above.

SSH Sessions

SSH is a secure data transfer protocol that provides confidentiality, integrity-protection, protection against replay attacks, and a variety of other services. The SSH server is authenticated via the server's public key and the client is authenticated either via a user name and password or (less frequently) a public key-based digital signature. cryptlib supports both versions 1 and 2 of the SSH protocol, although the obsolete version 1 is disabled by default.

The SSH protocol exhibits a design flaw (informally known as the SSH performance handbrake) that can lead to poor performance when transferring data, which is particularly noticeable with applications such as SFTP. Although cryptlib avoids the handbrake, many other implementations don't, restricting data transfer rates to as little as one tenth of the network link speed (the actual slowdown depends on the link characteristics and varies from one situation to another). In order to obtain the maximum performance from SSH, you need to either use cryptlib at both ends of the link (that is, both the client and server must be ones that avoid the performance handbrake), or use another protocol like TLS that doesn't have the handbrake.

SSH Client Sessions

Establishing a session with an SSH server requires adding the server name or IP address, an optional port number if it isn't using the standard SSH port, and the user name and password which is needed to log on to the server via the CRYPT_-SESSINFO_USERNAME and CRYPT_SESSINFO_PASSWORD attributes (occasionally the server will use public-key based authentication instead of a password, which is covered later). Once you've added this information, you can activate the session and establish the connection:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );
```

```
/* Add the server name, user name, and password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The equivalent operation in Java or C# is:

```
/* Create the session */
int cryptSession = crypt.CreateSession( cryptUser /* crypt.UNUSED */,
   crypt.SESSION_SSH );

/* Add the server name, user name, and password */
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_SERVER_NAME,
   serverName );
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_USERNAME,
   username );
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_PASSWORD,
   password );

/* Activate the session */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_ACTIVE, 1 );
```

In Visual Basic this is:

```
Dim cryptSession As Long

' Create the session
cryptCreateSession cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH

' Add the server name, user name, and password
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_SERVER_NAME, _
   serverName, Len( serverName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_USERNAME, _
   userName, Len( userName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_PASSWORD, _
   password, Len( password )

' Activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Activating a session results in cryptlib performing a lot of work in the background. For example when activating the SSH session shown above cryptlib will connect to the remote host, read the host and server keys used for authentication and encryption, generate a secret data value to exchange with the host using its host and server keys, create the appropriate encryption contexts and load keys based on the secret data value into them, negotiate general session parameters, and log on over the encrypted link using the given user name and password.

When you connect to a server, it's very important that you verify the server's identity using one of the mechanisms covered in "Verifying Session Security Information" on page 126. If you don't do this then an attacker who sets up a fake server for you to connect to can intercept and decrypt your (apparently) secure session.

If the server that you're connecting to requires public-key authentication instead of password authentication, you need to provide a private key via the CRYPT_-SESSINFO_PRIVATEKEY attribute to authenticate yourself to the server before you activate the session. The private 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:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );
```

```
/* Add the server name, user name, and client key and activate the
   session */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   cryptPrivateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

When cryptlib connects to the server, it will use the provided private key as part of the SSH handshake to authenticate the client to the server, with the private key taking the place of the more usual password. If you're not sure which of the two options you need, you can provide both and cryptlib will use the appropriate one when it connects to the server.

SSH Server Sessions

Establishing an SSH server session requires specifying that the session is a server session and adding the SSH server key. Once you've added this information you can activate the session and wait for incoming connections:

```
CRYPT_SESSION cryptSession;
int bytesCopied;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH_SERVER );

/* Add the server key and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );

/* Process any remaining control messages from the client */
cryptPopData( cryptSession, buffer, bufferSize, &bytesCopied );
```

The Visual Basic form is:

```
Dim cryptSession As Long
Dim bytesCopied as Long

' Create the session
cryptCreateSession cryptSession, cryptUser, _
   CRYPT_SESSION_SSH_SERVER

' Add the server key and activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_PRIVATEKEY, privateKey
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1

' Process any remaining control messages from the client
cryptPopData cryptSession, buffer, bufferSize, bytesCopied
```

Note the use of the data pop call after the activation has been completed. SSH clients often send additional session control information such as channel requests or port forwarding information after the session has been activated. Telling cryptlib to try and read any additional messages that may have arrived from the client allows it to process these requests and respond to them as appropriate. In particular, your server shouldn't send data to the client immediately after the session has been established without first performing a data pop to respond to client requests, since the client may interpret the data that you send as an (incorrect) response to its request.

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will negotiate a secure connection with the client. When the client connects, cryptlib will ask for a user name and password before it allows the connection to proceed. The handling of the user authentication process is controlled by the CRYPT_SESSINFO_AUTHRESPONSE attribute, by default cryptlib will return a CRYPT_ENVELOPE_RESOURCE status when it receives the user name and password, allowing you to verify the information before continuing. If it's valid, you should set the CRYPT_SESSINFO_AUTHRESPONSE attribute to true and resume the session activation by setting the CRYPT_SESSINFO_ACTIVE response to true again. If not, you can either set the CRYPT_SESSINFO_AUTHRESPONSE attribute to false and resume the session activation (which will give the user another chance to authenticate themselves), or close the session:

```
int status;

status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
if( status == CRYPT_ENVELOPE_RESOURCE )
   {
   char username[ CRYPT_MAX_TEXTSIZE + 1 ];
   char password[ CRYPT_MAX_TEXTSIZE + 1 ];
   int usernameLength, passwordLength

   /* Get the user name and password */
   cryptGetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
      username, &usernameLength );
   cryptGetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
      password, &passwordLength );
   username[ usernameLength ] = '\0';
   password[ passwordLength ] = '\0';

   /* Check the user details and allow or deny the response as
      appropriate */
   if( checkUser( username, password ) )
      cryptSetAttribute( cryptSession, CRYPT_SESSINFO_AUTHRESPONSE,
         1 );
   else
      cryptSetAttribute( cryptSession, CRYPT_SESSINFO_AUTHRESPONSE,
         0 );

   /* Resume the session activation, sending the authentication
      response to the client and completing the handshake */
   cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
   }
```

To give the user the traditional three attempts at getting their name and password right, you would run the session activation code in a loop:

```
int status;

for( i = 0; i < 3; i++ )
   {
   status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
      1 );
   if( cryptStatusOK( status ) )
      break;          /* User authenticated, exit */
   if( status == CRYPT_ENVELOPE_RESOURCE )
      /* Perform password check as before */;
   else
      break;          /* Some other type of error, exit */
   }
```

Alternatively, you can set the CRYPT_SESSINFO_AUTHRESPONSE attribute to true before you activate the session and cryptlib will automatically allow the access and complete the activation, so you'll never need to handle the CRYPT_-ENVELOPE_RESOURCE response. In this case you need to check the user details after the session has been activated and shut it down if the authorisation check fails.

Alongside static passwords, cryptlib also supports one-time passwords, sometimes referred to as two-factor authentication (2FA) values. The 2FA value is a time-based authenticator, in technical terms a time-based one-time password (TOTP) that's generated by a range of applications including 1Password, Aegis Authenticator, Authy, Google Authenticator, IBM Verify, LastPass Authenticator, Microsoft Authenticator, Red Hat FreeOTP, TOTP Authenticator, and many others. To use 2FA on the server you need to provide the authentication seed value, a base32-encoded text string, instead of the user password for the server to authenticate the client. This value is given as the CRYPT_SESSINFO_AUTHTOKEN attribute, which is provided in place of the usual CRYPT_SESSINFO_PASSWORD attribute:

```
CRYPT_SESSION cryptSession;

/* Create the session and add the server key */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH_SERVER );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );

/* Add the username and 2FA authenticator seed */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_AUTHTOKEN, seed,
   seedLength );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

When the client connects, cryptlib will calculate the current one-time password value that the client needs to provide in order to connect and verify it against the value provided by the client in much the same way that password-based authentication works.

On the client side things are identical to standard password-based authentication except that the value provided as the password is the 2FA authentication value generated by the authenticator application rather than a fixed password.

SSH Pre-Authentication

cryptlib supports SSH pre-authentication which blocks any attempts at SSH scanning attacks before the SSH handshake even begins. This leads to both a large attack surface reduction on the SSH protocol and prevents potential attackers from finding out anything about the server beyond its SSH ID string.

Pre-authentication works by using a shared secret between the client and server which is used to authenticate the start of the SSH handshake in a challenge-response protocol. Since no user name or other information is present at this point, there’s a single shared secret per server rather than a unique value per user. Think of it like a WiFi password where the pre-authentication value allows access to the WiFi network and per-user passwords allow access to things on the network.

You can set the pre-authentication value on the client or server by setting the CRYPT_SESSINFO_SSH_PREAUTH attribute. If you set it on the client then it’ll use it if required by the server (if the server doesn’t use pre-authentication then the connection will proceed as normal), if you set it on the server then it’ll reject any attempts to connect unless the client provides the correct authentication value with its first SSH message:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );

/* Add the server name, user name, password, and pre-authentication
   value */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SSH_PREAUTH,
   preAuth, preAuthLength );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

You can test this by trying to connect to a server that uses pre-authentication with an SSH client that doesn’t, it won’t even be able to begin the SSH handshake because the connection will be rejected as soon as the client attempts it.

SSH Channels

By default, cryptlib provides the most frequently-used SSH service, a direct encrypted connection from client to server. When you establish the SSH connection, cryptlib creates an SSH communications channel that's used to exchange data. This process is entirely transparent, and you don't have to worry about it if you don't want to — just treat the SSH session as a secure data pipe from one system to another.

There are however cases where you may need to explicitly deal with SSH channels, and that's when you're using special-purpose SSH facilities such as port forwarding, subsystems, or even user-defined channel types. In this case you need to explicitly create the special-purpose channel and add information describing its use before the channel can be activated. This process consists of three steps, creating the channel using the CRYPT_SESSINFO_SSH_CHANNEL attribute, specifying its type using the CRYPT_SESSINFO_SSH_CHANNEL_TYPE attribute, and finally specifying any optional channel arguments using the CRYPT_SESSINFO_SSH_CHANNEL_-ARG1 and CRYPT_SESSINFO_SSH_CHANNEL_ARG2 attributes. For example to create a channel of the default type (which is normally done automatically by cryptlib, and that has no optional arguments) you would use:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   CRYPT_UNUSED );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_TYPE, "session", 7 );
```

Setting the CRYPT_SESSINFO_SSH_CHANNEL attribute to CRYPT_UNUSED tells cryptlib to create a new channel (rather than trying to select an existing one, which is what the attribute is normally used for), and the CRYPT_SESSINFO_SSH_-CHANNEL_TYPE attribute specifies its type. Once you've created a new channel in this manner you can read back the CRYPT_SESSINFO_SSH_CHANNEL attribute to get the channel ID that was assigned for the newly-created channel:

```
int channelID;

cryptGetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   &channelID );
```

This value is used to uniquely identify a particular channel, but it's only needed in the presence of multiple channels, which are described in "SSH Multiple Channels" on page 114.

On the server side, reading the details of a channel that's been opened by the client works similarly:

```
char channelType[ CRYPT_MAX_TEXTSIZE + 1 ];
char channelArg1[ CRYPT_MAX_TEXTSIZE + 1 ];
int channelID, channelTypeLength, channelArg1Length, status;

/* Get the channel ID and type */
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   &channelID );
cryptGetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_TYPE, channelType, &channelTypeLength );
channelType[ channelTypeLength ] = '\0';

/* Get the optional channel argument */
status = cryptGetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_ARG1, channelArg1, &channelArg1Length );
if( cryptStatusOK( status ) )
   channelArg1[ channelArg1Length ] = '\0';
```

If you don't specify otherwise, cryptlib will open a channel of the default type when it connects. If you want to instead use a special-purpose SSH facility, you should provide the information necessary for creating it before you activate the connection. You can also open further channels after the connection has been completed, the process is described in "SSH Multiple Channels" on page 114. If you try to specify the use of more than one channel before the session has been activated, cryptlib will return CRYPT_ERROR_INITED when you try to create any channel after the first one, since it’s only possible to request further channels once the initial channel has been successfully established.

SSH Subsystems

Alongside the default encrypted link service, SSH provides additional services such as SFTP, an application-level file transfer protocol tunnelled over the SSH link via a subsystem channel. If you plan to use SFTP, note the comment about the SSH performance handbrake at the start of this section. Although cryptlib avoids this problem, non-cryptlib implementations frequently don’t, so that the performance of SFTP can be quite poor (as much as ten times slower than the network link speed) in some cases.

You can specify the use of a subsystem by setting the channel type to “subsystem” and the first channel argument to the subsystem name, in this case “sftp”:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );

/* Add the server name, user name, and password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );

/* Request the creation of the subsystem channel */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   CRYPT_UNUSED );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_TYPE, "subsystem", 9 );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_ARG1, "sftp", 4 );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Note that SFTP is not a part of the SSH protocol but constitutes an RPC mechanism for the Posix filesystem API. Despite its name, SFTP has nothing to do with FTP but is more an analogue of NFSv3. Implementing an SFTP client or server is equivalent to implementing NFS, not implementing FTP.

The handling of this NFS-equivalent mechanism, and support for features such as marshalling and unmarshalling of SFTP messages, translation of filenames, types, and attributes, and handling of operations, is an application-specific issue outside the scope of cryptlib. For example the SFTP `open` operation requires the use of a Unix-style filename, NFS access flags, and an extended form of the Unix/Posix file flags, which must be translated to and from the form used by the local OS or filesystem as required.

SSH Port Forwarding

Alongside standard SSH connections and SSH subsystems, it’s also possible to perform port-forwarding using SSH channels. Port forwarding allows you to tunnel an arbitrary network connection over SSH to avoid having the data being sent over the network in the clear. For example you could use this to tunnel mail (SMTP to send, POP3 or IMAP to receive) over SSH to and from a remote host. SSH provides two types of port forwarding, forwarding from the client to the server, identified by a channel type of “direct-tcpip”, and forwarding from the server to the client, identified by a channel type of “tcpip-forward”. The only one that’s normally used is client-to-server forwarding.

For client-to-server forwarding with a channel type of “direct-tcpip”, the first channel argument is the remote host and port that you want to forward to. For example if you wanted to tunnel SMTP mail traffic to mailserver.com with SMTP being on port 25 (so the forwarding string would be mailserver.com:25) you would use:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SSH );

/* Add the server name, user name, and password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );

/* Request the creation of the port-forwarding channel */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   CRYPT_UNUSED );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_TYPE, "direct-tcpip", 12 );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_ARG1, "mailserver.com:25", 17 );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

When cryptlib activates the connection, it will indicate to the remote SSH server that it should forward data sent over the SSH link to port 25 on mailserver.com. You can now either push data directly into the session to tunnel it to the remote server, or create a socket to listen on port 25 on the local machine and push data received on it into the session, creating a local to remote system port forwarding over the SSH channel.

When cryptlib activates the connection, it will indicate to the remote SSH server that it should forward data sent over the SSH link to port 25 on mailserver.com. You can now either push data directly into the session to tunnel it to the remote server or create a socket to listen on port 25 on the local machine and push data received on it into the session, creating a local to remote system port forwarding over the SSH channel. Note that since cryptlib isn't a standalone SSH server it won't open arbitrary ports on the client and forward data over them, it's up to the application to obtain the data locally and push/pop it to/from the cryptlib SSH session.

Before you forward the data on the server as requested by the client, you should check to make sure that the requested forwarding is in fact permitted. For example a malicious user could use port forwarding to attack a machine inside your firewall by forwarding connections through the firewall over an SSH tunnel. Because of this, cryptlib will never open a forwarded connection by itself, but requires that you explicitly forward the data. In other words it will indicate that port forwarding has been requested, but will never of its own volition open and/or forward arbitrary ports just because a client has requested it.

If you don't want to allow the port forwarding, all you need to do is ignore the port-forwarding channel. cryptlib's default action is to not allow forwarded connections, making it impossible for a client to remotely access internal machines or ports unless you explicitly allow it.

SSH Multiple Channels

Although SSH is usually used to provide a straightforward secure link from one system to another, it's also possible to use it to multiplex multiple virtual sessions across a single logical session. This is done by tunnelling multiple data channels across the SSH link.

SSH implements this using in-band signalling, which means that control information and data share the same link. With a single data channel (the standard case) this isn't a problem, but with multiple data channels control information for one channel can be impeded by data being sent or received on another channel. For example if you need to send or receive control information (channel close/channel open/status information) and there's a data transfer in progress on another channel, the control information can't be sent or received until the data transfer has been completed. This is why virtually all networking protocols use out-of-band signalling, with a separate mechanism for control signalling that can't be impeded by data transfers on the link.

Because of the in-band signalling problem, there are a number of special-case considerations that you need to take into account when using multiple SSH data channels. The primary one is: Don't do it. Unless you really have a strong need to run with multiple channels, just stick to a single channel and everything will be OK.

If you really need to use multiple channels, your code will need to take some extra steps to handle the problems caused by SSH's in-band signalling. The standard approach to this problem is to run the SSH implementation as a standalone service or daemon, with a full-time thread or task dedicated to nothing but handling any control messages that may arrive. These standalone applications are capable of opening ports to local and remote systems, logging on users, initiating data transfers, and so on. Since it's probably not a good idea for cryptlib to open arbitrary ports or transfer files without any additional checking, your application needs to explicitly manage these control messages. This requires doing the following:

- Try and open all channels and send all control messages right after the connect, before any data transfers are initiated. This means that the control signalling won't be stalled behind data signalling.
- Avoid using the session in non-blocking mode or with a very small timeout. Using a very short timeout increases the chances of some data remaining unwritten or unread, which will cause control information to become stalled behind it.
- Periodically try and pop data to handle any new control messages that may have arrived on other channels. In standalone SSH implementations that run as services or daemons, this is handled by having a full-time thread or task dedicated to this function. If you want to take this approach in your application, you can use a user-supplied socket that you can wait on in your application as described in "Network Issues" on page 130.
- Trying to perform channel control actions can result in a CRYPT_ERROR_-INCOMPLETE status if there's data still waiting to be read or written. This occurs because it's not possible to send or receive control information for another channel until the data for the current channel has been cleared. Since new data can arrive after you've cleared the existing data but before you can send the control message, you may need to run this portion of your code in a loop to ensure that the channel is clear so that you can send the control information. Note that both the send and receive sides of the channel have to be cleared to allow the control message to be sent and a response received.

If you've decided that you really do need to use multiple SSH channels, you can manage them using the CRYPT_SESSINFO_SSH_CHANNEL attribute, which contains an integer value that uniquely identifies each channel. You can select the channel to send on by setting this attribute before you push data, and determine the channel that data is being received on by reading it before you pop data:

```
int receiveChannelID, bytesCopied;

/* Send data over a given channel */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   sendChannelID );
cryptPushData( cryptSession, data, dataSize, &bytesCopied );

/* Receive data sent over a channel */
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   &receiveChannelID );
cryptPopData( cryptSession, buffer, bufferSize, &bytesCopied );
```

Read and write channels are distinct, so setting the write channel doesn't change the read channel, which is specified in incoming data messages that arrive.

If you're opening additional channels after the session handshake has completed, you need to tell cryptlib when to activate the newly-created channel. To do this, you set its CRYPT_SESSINFO_SSH_CHANNEL_ACTIVE attribute to true, which activates the channel by sending the details to the remote system. Using the previous example of a port-forwarding channel, if you wanted to open this additional channel after the session had already been established you would use:

```
/* Request the creation of the port-forwarding channel */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   CRYPT_UNUSED );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_TYPE, "direct-tcpip", 12 );
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SSH_CHANNEL_ARG1, "mailserver.com:25", 17 );

/* Activate the newly-created channel */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL_ACTIVE,
   1 );
```

If you want to close one of the additional channels, you can select it in the usual manner and then deactivate it by setting its CRYPT_SESSINFO_SSH_CHANNEL_-ACTIVE attribute to false:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL,
   channelID );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_SSH_CHANNEL_ACTIVE,
   0 );
```

If you try to deactivate the last remaining channel, which corresponds to the session itself, cryptlib will return a CRYPT_ERROR_PERMISSION status. To close the final channel, you need to close the overall session.

TLS Sessions

TLS is a secure data transfer protocol that provides confidentiality, integrity-protection, protection against replay attacks, and a variety of other services, and was formerly known as SSL. The TLS server is authenticated via a certificate, and the client isn't authenticated (in rare circumstances client certificates may be used, but these are usually avoided due to the high degree of difficulty involved in working with them). Alternatively, the client and server may be mutually authenticated via a secret-key mechanism such as a user name and password, which avoids the need for certificates altogether.

cryptlib will automatically negotiate the highest protocol version supported by the other side and use that to secure the session. You can determine which version is in use once the session has been established by reading the CRYPT_SESSINFO_-VERSION attribute, a value of 1 indicates TLS 1.0, a value of 2 indicates TLS 1.1, a value of 3 indicates TLS 1.2, and so on (the reason for the offset of 1 is that the original SSL protocol used a value of 0, so TLS 1.0 had to start at 1). You can also force the use of a particular version by setting the protocol version attribute before you activate the connection, for example you can have cryptlib function as a TLS-1.1 server by setting the CRYPT_SESSINFO_VERSION to 2 to indicate the use of TLS version 1.1 rather than whatever version would otherwise be negotiated. A (fortunately) tiny and ever-decreasing number of buggy servers will fail the handshake if the protocol version is advertised as a much higher version of TLS than they support, if you receive a handshake failure alert when you try to activate the session (as indicated by the CRYPT_ATTRIBUTE_ERRORMESSAGE attribute) you can try forcing the use of lower protocol versions to see if the server can handle a connect using only the older protocol version.

TLS v1.3 is relatively new and incompatible with all other versions of TLS (it is in fact a completely new protocol that requires its own separate TLS 1.3 stack), which means that some clients and servers may break if they encounter a server or client that advertises this protocol version. Using TLS 1.3 instead of cryptlib's TLS 1.2 offers no security benefits (TLS 1.3 was designed to make content delivery from large hosting providers like Google and Facebook more efficient), while nearly doubling the TLS code size because of the need to implement a separate TLS 1.3 stack, as well as increasing the attack surface due to the added complexity of TLS 1.3. Because of this it's currently disabled by default, and will need to be enabled on build with `-DUSE_TLS13`.

TLS Client Sessions

Establishing a session with an TLS server requires adding the server name or IP address and an optional port number if it isn't using the standard TLS port. Once you've added this information, you can activate the session and establish the connection:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS );

/* Add the server name and activate the session */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The same operation in Java or C# is:

```
/* Create the session */
int cryptSession = crypt.CreateSession( cryptUser /* crypt.UNUSED */,
   crypt.SESSION_TLS );

/* Add the server name and activate the session */
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_SERVER_NAME,
   serverName );
crypt.SetAttribute( cryptSession, crypt.SESSINFO_ACTIVE, 1 );
```

The Visual Basic form of the code is:

```
Dim cryptSession As Long

' Create the session
cryptCreateSession cryptSession, cryptUser, CRYPT_SESSION_TLS

' Add the server name and activate the session
cryptSetAttributeString cryptSession, CRYPT_SESSIONFO_SERVER_NAME, _
   serverName, Len( serverName )
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Activating a session results in cryptlib performing a lot of work in the background. For example when activating the TLS session shown above cryptlib will connect to the remote host, read the server's certificate, generate a secret data value to exchange with the server using the key contained in the certificate, create the appropriate encryption contexts and load keys based on the secret data value into them, negotiate general session parameters, and complete negotiating the encrypted link with the server.

When you connect to a server, it's very important that you verify the server's identity using one of the mechanisms covered in "Verifying Session Security Information" on page 126. If you don't do this then an attacker who sets up a fake server for you to connect to can intercept and decrypt your (apparently) secure session.

TLS with Shared Keys

If the server that you're connecting to uses shared keys (for example a user name and password), you need to provide this information via the CRYPT_SESSINFO_-USERNAME and CRYPT_SESSINFO_PASSWORD attributes to authenticate yourself to the server before you activate the connection:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS );

/* Add the server name, user name, and password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, usernameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The equivalent operation in Java or C# is:

```
/* Create the session */
int cryptSession = crypt.CreateSession( cryptUser /* crypt.UNUSED */,
   crypt.SESSION_TLS );

/* Add the server name, user name, and password */
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_SERVER_NAME,
   serverName );
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_USERNAME,
   username );
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_PASSWORD,
   password );

/* Activate the session */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_ACTIVE, 1 );
```

In Visual Basic this is:

```
Dim cryptSession As Long

' Create the session
cryptCreateSession cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS

' Add the server name, user name, and password
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_SERVER_NAME, _
   serverName, Len( serverName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_USERNAME, _
   userName, Len( userName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_PASSWORD, _
   password, Len( password )

' Activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Authenticating yourself using shared keys avoids the need for both server and client certificates, providing strong mutual authentication for both client and server (conventional TLS only authenticates the server using a server certificate), and therefore a much higher level of security and protection against problems like phishing attacks, since the server has to prove possession of the client's credentials before the connection can proceed.

TLS with Client Certificates

If the server that you're connecting to requires a client certificate, you need to provide a private key with an attached signing certificate via the CRYPT_SESSINFO_-PRIVATEKEY attribute to authenticate yourself to the server before you activate the session. The private key and accompanying certificate 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 client authentication:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS );
```

```
/* Add the server name and client key/certificate and activate the
   session */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   cryptPrivateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

When cryptlib connects to the server, it will use the provided private key and signing certificate as part of the TLS handshake to authenticate the client to the server. If the server doesn't require the use of a client certificate, cryptlib won't do anything with the private key, so it's OK to add this even if you're not sure whether it'll be needed or not.

Note that client certificates are very rarely used in practice because of the high level of difficulty involved in working with them. If you require client authentication, a far better solution is to use TLS with shared keys as described in the previous section, which provides mutual cryptographic authentication of both client and server.

TLS Server Sessions

Establishing an TLS server session requires adding the server key with its accompanying certificate, activating the session, and waiting for incoming connections:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS_SERVER );

/* Add the server key/certificate and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The same procedure in Visual Basic is:

```
Dim cryptSession As Long

' Create the session
cryptCreateSession cryptSession, cryptUser, _
   CRYPT_SESSION_TLS_SERVER

' Add the server key/certificate and activate the session
cryptSetAttribute cryptSession, CRYPT_SESSIONFO_PRIVATEKEY, privateKey
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will negotiate a secure connection with the client.

TLS Servers with Shared Keys

If you're using shared keys (for example a user name and password) to provide security, you need to provide this information via the CRYPT_SESSINFO_-USERNAME and CRYPT_SESSINFO_PASSWORD attributes. For example if you have a server that allows one of three users/clients to connect to it you would use:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS_SERVER );

/* Add the user names and passwords */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username1, username1Length );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password1, password1Length );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username2, username2Length );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password2, password2Length );
```

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username3, username3Length );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password3, password3Length );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Using shared keys in this manner avoids the need for both server and client certificates, providing mutual cryptographic authentication for both client and server (conventional TLS only authenticates the server via a server certificate). This type of authentication also detects attacks like phishing attacks, since the server has to prove knowledge of the client's credentials before the connection can be established.

If you have clients who need to connect without providing a user name and password, you can still provide a server certificate in the usual manner using the CRYPT_-SESSINFO_PRIVATEKEY attribute, and clients who don't provide a user name and password will connect using public-key encryption. Note though that a client that uses the server certificate rather than a user name and password loses the benefits of mutual client/server authentication.

Once a client has authenticated themselves using a shared key, you can determine their identity by reading back the CRYPT_SESSINFO_USERNAME attribute:

```
char username[ CRYPT_MAX_TEXTSIZE + 1 ];
int usernameLength

/* Get the user name */
cryptGetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   username, &usernameLength );
username[ usernameLength ] = '\0';
```

If the attempt by the client to connect fails (typically due to the use of an incorrect password), the password information for that user will be reset to prevent password-guessing attacks in which an attacker repeatedly reconnects using every possible password until they succeed. If the password is reset, you need to re-add the user and password to the session before that particular user can connect again. In order to protect against password-guessing attacks you should employ standard precautions such as allowing a maximum of three incorrect attempts or inserting a time delay before another connect attempt is allowed.

TLS Servers with Client Certificates

There are two ways to use client certificates to authenticate incoming connections, you can either provide a public-key keyset like a database keyset for cryptlib to use to check certificates provided by client connections or you can manually check the client certificate yourself. When you use a public-key keyset and a client tries to establish a connection, cryptlib will check that their certificate is present in the keyset. If it isn't present, the connection isn't permitted. If you use manual checking, cryptlib will interrupt the session activation and wait for you to check that the client's certificate is OK before continuing.

This provides a very fine-grained level of access control through which individual end users can be permitted or denied access to the host. Since the keyset form of the certificate checking uses the presence of a certificate in the keyset to verify incoming connections, you can control who is allowed in by adding or removing their certificate to or from the keyset. Note that you must provide a public-key keyset that stores certificates (not a private-key keyset) to the session since TLS uses certificates for the access control functionality.

Using manual certificate checking provides an alternative means of access control, but requires more effort than the simple whitelist-based check that's provided through public-key keysets. You can use manual checking if you need to verify that a certificate is issued by a particular CA or has particular name components or other attributes in it.

If you're verifying certificates using a keyset-based whitelist, you can specify the public-key keyset to use for checking incoming connections with the CRYPT_SESSINFO_KEYSET attribute:

```
CRYPT_SESSION cryptSession;
CRYPT_KEYSET cryptKeyset;

/* Open the keyset containing the certificate whitelist */
cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,  CRYPT_KEYSET_DATABASE,
   "PublicKeys", CRYPT_KEYOPT_READONLY )

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS_SERVER );

/* Add the server key and public-key keyset and activate the
   session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET, cryptKeyset );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

When you set this attribute for a server session, cryptlib will require the use of client certificates for connections to the server, and won't allow connections from clients that aren't able to authenticate themselves using a certificate that was previously added to the keyset. Once you've opened the keyset containing the client certificate whitelist, you can re-use it for subsequent sessions, there's no need to re-open it for each new session.

If you're verifying certificates manually, you need to tell cryptlib that you're doing this by setting the CRYPT_SESSINFO_TLS_OPTIONS option CRYPT_TLSOPTION_MANUAL_CERTCHECK:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_OPTIONS,
   CRYPT_TLSOPTION_MANUAL_CERTCHECK );
```

This tells cryptlib to request a client certificate for authentication in the same way that setting the CRYPT_SESSINFO_KEYSET would. When the client connects, cryptlib will request a certificate before it allows the connection to proceed. The handling of the client authentication process is controlled by the CRYPT_SESSINFO_-AUTHRESPONSE attribute, by default cryptlib will return a CRYPT_ENVELOPE_-RESOURCE status when it receives the client's certificate, allowing you to verify the certificate before continuing. If it's valid, you should set the CRYPT_SESSINFO_-AUTHRESPONSE attribute to true and resume the session activation by setting the CRYPT_SESSINFO_ACTIVE response to true again. If not, you can either set the CRYPT_SESSINFO_AUTHRESPONSE attribute to false and resume the session activation (which will deny the connect request), or close the session:

```
int status;

status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
if( status == CRYPT_ENVELOPE_RESOURCE )
   {
   CRYPT_CERTIFICATE cryptCertChain;

   /* Get the client's certificate or certificate chain */
   cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
      &cryptCertChain );

   /* Check the certificate/chain details and allow or deny it as
      appropriate */
   if( checkCertChain( cryptCertChain )  )
      cryptSetAttribute( cryptSession, CRYPT_SESSINFO_AUTHRESPONSE,
         1 );
   else
      cryptSetAttribute( cryptSession, CRYPT_SESSINFO_AUTHRESPONSE,
         0 );
   cryptDestroyCert( cryptCertChain );

   /* Resume the session activation, completing the handshake */
   cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
   }
```

TLS Servers with Virtual Hosts

cryptlib supports the use of TLS virtual hosting in which a single server can be associated with multiple identities or DNS names. This is handled through a TLS mechanism called server name indication or SNI in which the client tells the server which virtual host it wants to connect to. On the client side this is handled automatically by cryptlib, on the server side you need to configure the server with one key/certificate for each virtual host that it supports and cryptlib will switch between TLS virtual hosts based on what the client requests.

To enable the virtual-hosting capability through SNI, you need to set the TLS option CRYPT_TLSOPTION_SERVER_SNI before you add the server key(s) to the session:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_OPTIONS,
   CRYPT_TLSOPTION_SERVER_SNI );
```

After this you can add server keys as normal as CRYPT_SESSINFO_PRIVATEKEY attributes, with the first one added being the default server key and any subsequent ones being alternative virtual server keys that are selected through the client's SNI:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   primaryServerKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   alternateServerKey1 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   alternateServerKey2 );
```

When a client connects, cryptlib will match the virtual host's key to the one indicated by the client's SNI and use that one for the TLS session. If no match for the SNI is found, it will use the default/primary server key.

TLS with WebSockets

Alongside the use of TLS as a direct tunnel, cryptlib also implements WebSockets with TLS. The use of WebSockets is specified using the CRYPT_SESSINFO_TLS_-SUBPROTOCOL attribute, giving the value CRYPT_SUBPROTOCOL_-WEBSOCKETS:

```
/* Create the session and set the WebSockets server to connect to */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLen );

/* Select WebSockets */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_SUBPROTOCOL,
   CRYPT_SUBPROTOCOL_WEBSOCKETS );
```

If there's a specific protocol being run over WebSockets, for example MQTT, you can specify the protocol name using the CRYPT_SESSINFO_TLS_WSPROTOCOL attribute (this usually isn't needed since the other side of the tunnel typically only implements the specific subprotocol that it's been set up for):

```
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_TLS_WSPROTOCOL,
   "mqtt", 4 );
```

Once you've enabled the use of WebSockets in this manner, all data sent over the TLS connection will use the WebSockets protocol for encapsulation.

TLS with EAP / RADIUS

cryptlib implements various EAP / RADIUS protocols combined with TLS, specifically EAP-TLS, EAP-TTLS, and EAP-PEAP. EAP is used in conjunction with RADIUS, so what's actually being run is EAP over RADIUS, or to give the full protocol stack, EAP-TTLS over EAP over RADIUS-EAP over RADIUS over UDP, with further protocol layers such as PAP, CHAP, or MSCHAP layered over the top of that, encapsulated inside the DIAMETER protocol, for a total of seven levels of protocol encapsulation. cryptlib itself provides the bottom five layers up to EAP-TLS/TTLS/PEAP, with the tunnelled authentication being provided by the application that uses cryptlib. This allows the tunnelling of arbitrary higher-level protocols over the secure EAP TLS tunnel provided by cryptlib.

The use of a particular TLS-based EAP protocol is specified using the CRYPT_SESSINFO_TLS_SUBPROTOCOL attribute, giving the value CRYPT_SUBPROTOCOL_EAPTLS, CRYPT_SUBPROTOCOL_EAPTTLS, or CRYPT_SUBPROTOCOL_PEAP as required:

```
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_SUBPROTOCOL,
   CRYPT_SUBPROTOCOL_EAPTTLS );
```

A typical flow for setting up a TLS session for EAP use is:

```
/* Create the session and set the RADIUS server to connect to */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TLS );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLen );

/* Select EAP-TTLS */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_SUBPROTOCOL,
   CRYPT_SUBPROTOCOL_EAPTTLS );

/* Set the RADIUS user name and password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLen );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLen );

/* Activate the EAP session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

In addition to the standard settings you may need to restrict the TLS version to TLS 1.0 since many servers won't understand TLS 1.2 or even 1.1 and return odd errors if newer versions of TLS are used. You may also need to disable certificate name verification if the server is using a generic self-signed certificate or on an RFC 1918 address with a certificate that doesn't correspond to its name or IP address:

```
/* Set the TLS version to TLS 1.0 and disable certificate name
   verification */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_VERSION, 1 );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_TLS_OPTIONS,
   CRYPT_TLSOPTION_DISABLE_NAMEVERIFY );
```

Once the EAP session is established, you can run whichever higher-level authentication mechanism you're working with over cryptlib's EAP TLS tunnel.

As the earlier description of protocol stacking indicates, running an authentication mechanism that consists of between five and seven protocol layers stacked on top of each other, with several of the layers never having been designed for this, can lead to all sorts of problems. For this reason cryptlib provides extensive diagnostics when built and run in debug mode, printing a complete trace of messages sent and received. You can use this debug trace, alongside debugging facilities on the server (for example running FreeRADIUS with the `-X` option), to track down problems in communicating with a RADIUS server.

Request/Response Protocol Sessions

cryptlib supports a variety of request/response protocols including protocols such as the certificate management protocol (CMP), simple certificate enrolment protocol (SCEP), real-time certificate status protocol (RTCS), online certificate status protocol (OCSP), server-based certificate validation protocol (SCVP), and timestamping protocol (TSP). CMP, SCEP, RTCS, OCSP, and SCVP client sessions are certificate management services that are covered in "Obtaining Certificates using CMP", "Obtaining Certificates using SCEP", "Certificate Status Checking using RTCS", and "Certificate Revocation Checking using OCSP" on pages 174, 168, 168, and 175, and a TSP client session is an S/MIME service which is covered in "Timestamping" on page 90. RTCS, OCSP, SCVP, and TSP server sessions are standard session types and are also covered here, CMP and SCEP server sessions are somewhat more complex and are covered in "Managing a CA using CMP or SCEP" on page 186.

RTCS Server Sessions

An RTCS server session is a protocol-specific session type that returns a real-time certificate status to a client. RTCS client sessions are used for certificate status checks and are described in "Certificate Status Checking using RTCS" on page 168.

Establishing an RTCS server session requires adding a database keyset that cryptlib can query for certificate status information, specified as the CRYPT_SESSINFO_-KEYSET attribute, and an optional RTCS responder key/certificate if you want cryptlib to sign the responses that it provides. The database keyset would typically be a CA certificate store as described in "Managing a Certification Authority" on page 180, but opened as a CRYPT_KEYSET_DATABASE rather than a CRYPT_-KEYSET_DATABASE_STORE since RTCS uses the keyset purely for certificate status information and not as a full-blown certificate store.

Once you've added this information you can activate the session and wait for incoming connections:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_RTCS_SERVER );

/* Add the database keyset and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
   cryptCertStore );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will read the RTCS request from the client and return a response optionally signed with the RTCS responder key.

OCSP Server Sessions

An OCSP server session is a protocol-specific session type that returns certificate revocation information to a client. OCSP client sessions are used for certificate revocation checks and are described in "Certificate Revocation Checking using OCSP" on page 168.

The difference between RTCS and OCSP is that RTCS provides real-time, live certificate status information while OCSP provides delayed revocation information, usually based on CRLs. In other words RTCS answers the question "Is this certificate OK to use right now?" while OCSP answers the question "Was this certificate revoked at some point in the past?". OCSP can't return true validity information, so that if fed a freshly-issued certificate and asked "Is this a valid certificate", it can't say "Yes" (a CRL can only answer "revoked"), and if fed a forged certificate it can't say "No" (it won't be present in any CRL). In addition OCSP will often return a status result drawn from stale information hours or even days old, while RTCS (as the name implies) will always return real-time information. Finally, OCSP uses a peculiar means of identifying certificates that some implementations disagree over, with the result that a certificate may be regarded as valid even if it isn't because client and server are talking about different things. In contrast RTCS returns an unambiguous yes-or-no response drawn from live certificate data. For these reasons RTCS is the cryptlib preferred certificate status protocol.

Establishing an OCSP server session requires adding the OCSP responder key/certificate and a database keyset that cryptlib can query for certificate status information, specified as the CRYPT_SESSINFO_KEYSET attribute. The database keyset would typically be a CA certificate store as described in "Managing a Certification Authority" on page 180, but opened as a CRYPT_KEYSET_-DATABASE rather than a CRYPT_KEYSET_DATABASE_STORE since OCSP uses the keyset purely for certificate status information and not as a full-blown certificate store.

Once you've added this information you can activate the session and wait for incoming connections:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_OCSP_SERVER );

/* Add the OCSP responder key/certificate and database keyset and
   activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
   cryptCertStore );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will read the OCSP request from the client and return a response signed with the OCSP responder key.

SCVP Server Sessions

An SCVP server session is a protocol-specific session type that returns certificate validity information to a client. SCVP client sessions are used for certificate validity checks and are described in “Certificate Status Checking using SCVP” on page 177.

The difference between RTCS and SCVP is that, while they perform the same function, RTCS is a very lightweight protocol that provides fast certificate status information while SCVP is an extraordinarily complex and heavyweight mechanism for doing more or less the same thing. In particular RTCS sends a single status value, “valid” or “not valid” while SCVP exchanges tens of kilobytes of complex data that needs to be parsed, analysed, and processed by both client and server in order to obtain the same result that RTCS conveys in a single value. As is standard for very complex protocols like this, the very few implementations of SCVP that exist can never quite agree on how some things are to be interpreted, so protocols like RTCS or HTTP status checks are to be preferred over any use of SCVP.

Establishing an SCVP server session requires adding the SCVP CA key/certificate and a database keyset that cryptlib can query for certificate status information, specified as the CRYPT_SESSINFO_KEYSET attribute. The database keyset would typically be a CA certificate store as described in “Managing a Certification Authority” on page 180, but opened as a CRYPT_KEYSET_DATABASE rather than a CRYPT_KEYSET_DATABASE_STORE since SCVP uses the keyset purely for certificate status information and not as a full-blown certificate store.

Once you’ve added this information you can activate the session and wait for incoming connections:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_OCSP_SERVER );

/* Add the SCVP CA key/certificate and database keyset and activate
   the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
   cryptCertStore );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will read the SCVP request from the client and return a response signed with the SCVP CA key.

TSP Server Sessions

A TSP server session is a protocol-specific session type that returns timestamp information to a client. TSP client sessions are used with S/MIME and are described in “Timestamping” on page 90. Establishing a TSP server session requires adding the timestamping authority (TSA) key/certificate, activating the session, and waiting for incoming connections:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_TSP_SERVER );

/* Add the TSA key/certificate and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The TSA certificate must be one that has the CRYPT_CERTINFO_EXTKEY_-TIMESTAMPING extended key usage attribute set to indicate that it can be used for generating timestamps. Extended key usage attributes are described in "Key Usage, Extended Key Usage, and Netscape certificate type" on page 252. If you add a key/certificate without this attribute, cryptlib will return CRYPT_ERROR_PARAM3 to indicate that the key parameter is invalid.

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will read the timestamp request from the client and return a timestamp signed with the TSA key.

Verifying Session Security Information

When a session is established a lot of state information is exchanged between the client and server and status information is generated by both sides. After the session has been activated it's important that you verify the session security information to make sure that you're talking to the correct server or client. If you don't do this then an attacker can slip in a fake server or client and you'd have a carefully encrypted, secure connection to the attacker rather than the real server or client.

The most secure way to verify that you're talking to the correct server is through key fingerprints, covered in "Authenticating the Host with Key Fingerprints" on page 126. If at all possible, you should use this means of authentication to check that you're communicating with the appropriate server.

A less secure way to authenticate the server is through the use of certificates. Anyone can create their own certificate claiming to be anyone they like, and CAs are often very lax in checking identities before they sell a certificate, so unless you perform very careful checking of the certificate and certificate chain, an attacker will still be able to substitute a fake server for the real one. Authenticating the server using certificates is covered in "Authenticating the Host or Client using Certificates" on page 128.

The least secure way to authenticate a server or client is through its IP address, covered in "Authenticating the Client via Port and Address" on page 129. Still, if you have the capability to do this then it's worth performing the check since it'll stop non-targeted attacks like general network scanning, and it costs almost nothing to do.

In addition to authenticating the client or server, you can query the session object for information such as the session status, which security parameters are being used, the identity of the remote client that connected to your server (the identity of the remote server is already known if you're the client), and authentication and identification information that was obtained from the client or server during the session establishment process.

Authenticating the Host with Key Fingerprints

The most secure way to use key fingerprints for authentication is to set the CRYPT_SESSINFO_SERVER_FINGERPRINT_SHA1 attribute before you connect to the server. cryptlib will then verify the fingerprint against the server key when it connects and break off the connection attempt with a CRYPT_ERROR_-WRONGKEY status if the server's certificate or key doesn't match the fingerprint that you've specified. This allows you to filter out fake servers and/or keys before you try to send any sensitive information to them.

For example if you have a known-good certificate for the server then you can make sure that you're connecting to the actual server that controls the certificate with:

```
unsigned char fingerprint[ CRYPT_MAX_HASHSIZE ];
int fingerprintSize;

/* Get the fingerprint from the local copy of the server
   certificate */
cryptGetAttributeString( serverCertificate,
   CRYPT_CERTINFO_FINGERPRINT_SHA1, &fingerprint, &fingerprintSize );

/* Set the fingerprint for the client session */
cryptSetAttributeString( cryptSession,
   CRYPT_SESSINFO_SERVER_FINGERPRINT_SHA1, fingerprint,
   fingerprintSize );
```

Alternatively, once you've connected to the server you can verify its certificate or key fingerprint by reading the CRYPT_SESSINFO_SERVER_FINGERPRINT_SHA1 attribute, which contains a fingerprint value that uniquely identifies the server's certificate or key. You can compare this to a stored copy of the fingerprint, or format it for display to the user.

In the case of SSH where you can't connect without providing a user name and password, if you want to check the fingerprint before providing a password then just set a dummy user name and password which will cause the connect to fail with a CRYPT_ERROR_WRONGKEY but will still provide a SESSINFO_SERVER_-FINGERPRINT_SHA1 to read. You can then decide whether you want to connect with your actual password based on the fingerprint value, just remember to set the fingerprint value you've read from the dummy connect as the CRYPT_SESSINFO_-SERVER_FINGERPRINT_SHA1 for the actual connect to make sure that you're still connecting to the same server.

Note that there is no standard for SSH fingerprints so everyone invents their own one. Fingerprints can be calculated using MD5, SHA-1, or SHA-256, and represented as ASCII hex data, ASCII hex data with colons between byte values, or base64-encoded binary data. cryptlib provides the binary SHA-1 hash fingerprint which you can then encode as required for comparison with whatever format the other side is using.

To get the server's key fingerprint from the server side you can read the CRYPT_-SESSINFO_SERVER_FINGERPRINT_SHA1 attribute from the SSH server session after you've added the server's private key, and the CRYPT_CERTINFO_-FINGERPRINT_SHA1 attribute from the TLS server certificate.

You can also get cryptlib to perform the key checking for you automatically by providing a whitelist of acceptable server keys stored in a database keyset. This works for TLS certificates where you can obtain a copy of each server's certificate, avoiding the need to deal with CAs, certificate chain verification, trust anchors, CRLs/OCSP, and all of the other PKI paraphernalia since only the certificates that you explicitly allow will be permitted for TLS connections.

To use a certificate whitelist, you add all of the permitted certificates to a database keyset and then add the keyset to the session as a CRYPT_SESSINFO_KEYSET attribute. When cryptlib connects to the server it will compare the certificate that the server uses to authenticate itself to the ones in the whitelist and only allow a connection to a server whose certificate is present in the whitelist:

```
CRYPT_KEYSET cryptKeyset;

/* Open the keyset containing the whitelist of permitted certificates
   and add it to the session */
cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_DATABASE, "PublicKeys", CRYPT_KEYOPT_READONLY );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET, cryptKeyset );
cryptKeysetClose( cryptKeyset );
```

When you use whitelist-based checking, run a safety check to ensure that you've configured your whitelist correctly by trying to connect to a server that's not on the whitelist and verifying that cryptlib rejects the connect attempt.

Using fingerprints for authentication is by far the most reliable of the methods covered here, since it provides a guaranteed match to a known key that can't be spoofed or forged.

Authenticating the Host or Client using Certificates

In addition to providing integrity and privacy protection for a communications session, some session protocols also provide a means of verifying that the host or client that you're connecting to really is who they claim to be. For everything but the SSH protocol this authentication is performed by having the host supply a certificate or certificate chain signed by a CA (or sometimes a self-signed certificate) which is used during the protocol initialisation phase to establish the session. The general idea is that the certificate contains the name of the host that you're connecting to or the name of the entity which is providing a particular service (for example an RTCS responder), so you can use the returned certificate to verify that you really are communicating with this host and not a machine that's been set up by an attacker to masquerade as the host.

In addition if you're using TLS with client certificates, you can use the certificate provided by the client when they connect to verify their identity. If you're using TLS with shared keys then you already have mutual authentication of client and server and don't need to bother with certificates.

When connecting to a server that uses certificates, cryptlib will automatically check the host name (or names) given in the certificate against the host that you're connecting to. If the two don't match, meaning that the certificate isn't valid for the given host then the activation will return CRYPT_ERROR_INVALID to indicate that the certificate isn't valid for this host. This situation can occur when an attacker has substituted a fake certificate (and host) for the real one (or sometimes due to a configuration error in which the wrong certificate is used). If you get a CRYPT_-ERROR_INVALID status you can then read the CRYPT_ATTRIBUTE_-ERRORMESSAGE attribute to get an explanation of what the problem was.

Once the session has been successfully activated, you can read the host or client's certificate chain as the CRYPT_SESSINFO_RESPONSE attribute:

```
CRYPT_CERTIFICATE cryptCertificate;

cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptCertificate );
```

You can then work with the certificate chain as usual, for example you can verify it using **cryptCheckCert** or fetch the subject name information as explained in "Certificate Identification Information" on page 225. It's important that you verify the chain, otherwise an attacker can create (or buy from a CA) a certificate that matches the host name that you're connecting to, again allowing them to substitute a fake host for the real one.

Note that in the case of a server what's returned is a full certificate chain and not just a single certificate. This means that the chain will contain not just an end entity certificate but also one or more CA certificates typically leading up to a trusted root CA certificate. You can find out more about how to work with these in "Certificate Chains" on page 236, and in particular how to verify them in "Checking Certificate Chains" on page 238.

In addition there is considerable confusion among CAs and implementations as to how identity information is stored in a certificate. Because of the infinitely flexible nature of certificates it's possible to stash multiple conflicting identities in a certificate, some of which may not be verified by CAs or other implementations, and different implementations will often use different identities in their verification. This is why cryptlib uses a keyset-based whitelist for access control with client certificates, because then only users with pre-approved certificates are allowed access no matter what identities the certificate contains or which CA signed them.

Authenticating the Client via Port and Address

In addition to the stronger fingerprint and certificate authentication mechanisms, you can also determine the IP address and port that a client is connecting from if you're running as a server (if you're the client, you already known which server and port you're connecting to). You can obtain this information by reading the CRYPT_-SESSINFO_CLIENT_NAME and CRYPT_SESSINFO_CLIENT_PORT attributes, which work in a similar manner to the CRYPT_SESSINFO_SERVER_NAME and CRYPT_SESSINFO_SERVER_PORT attributes:

```
char name[ CRYPT_MAX_TEXTSIZE + 1 ];
int nameLength, port

cryptGetAttributeString( cryptSession, CRYPT_SESSINFO_CLIENT_NAME,
   name, &nameLength );
name[ nameLength ] = '\0';
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_CLIENT_PORT, &port );
```

The same operation in Visual Basic is:

```
Dim name as String
Dim nameLength as Long
Dim port as Long

name = String( CRYPT_MAX_TEXTSIZE, vbNullChar );
cryptGetAttributeString cryptSession, CRYPT_SESSINFO_CLIENT_NAME, _
   name, nameLength
name = Left( name, nameLength )
cryptGetAttribute cryptSession, CRYPT_SESSINFO_CLIENT_PORT, port
```

Note that cryptlib returns the client's IP address in dotted-decimal form (for IPv4) or colon-delimited form (for IPv6) rather than its full name, since a single IP address can be aliased to multiple names and may require complex name resolution strategies. If you require a full name rather than an IP address you'll need to resolve it yourself, taking into account the multiple hostname issue, the fact that the client may be using NAT, and the possibility of DNS spoofing.

Obtaining Session Security Parameters

If you want to know the details of the encryption mechanism that's being used to protect the session, you can read various CRYPT_CTXINFO_*xxx* attributes from the session object, which will return information from the encryption context(s) that are being used to secure the session. For example once you've activated the session you can get the encryption algorithm, mode, and the key size being used with:

```
CRYPT_ALGO_TYPE cryptAlgo;
CRYPT_MODE_TYPE cryptMode;
int keySize;

cryptGetAttribute( cryptSession, CRYPT_CTXINFO_ALGO, &cryptAlgo );
cryptGetAttribute( cryptSession, CRYPT_CTXINFO_MODE, &cryptMode );
cryptGetAttribute( cryptSession, CRYPT_CTXINFO_KEYSIZE, &keySize );
```

Exchanging Data

Once a general-purpose secure communications session has been established, you can exchange data with the remote client or server over the encrypted, authenticated link that it provides. This works exactly like pushing and popping data to and from an envelope, except that the session is effectively a bottomless envelope that can accept or return (depending on the remote system) an endless stream of data. In many cases the overhead involved in wrapping up a block of data and exchanging it with a remote client or server can be noticeable, so you should always push and pop as much data at once into and out of a session as you can. For example if you have a 100-byte message and communicate it to the remote host as 10 lots of 10 bytes, this is much slower than sending a single lot of 100 bytes. This behaviour is identical to the behaviour in applications like disk I/O, where writing a single big file to disk is a lot more efficient than writing 10 smaller files.

cryptlib helps to eliminate this problem as much as possible by not wrapping up and dispatching session data until you explicitly tell it to by flushing the data through just as you would with an envelope:

```
cryptPushData( cryptSession, data, dataSize, &bytesCopied );
cryptFlushData( cryptSession );
```

In Visual Basic this is:

```
cryptPushData cryptSession, data, dataSize, bytesCopied
cryptFlushData cryptSession
```

This means that cryptlib will accumulate as much data as possible in the session's internal buffer before encrypting and integrity-protecting it and sending it through to the remote system, avoiding the inefficiency of processing and sending many small blocks of data. Note that you only need to flush data through in this manner when you explicitly want to force all of the data in the session buffer to be sent to the remote system. If you don't force a flush cryptlib handles this automatically in the most efficient manner possible using its built-in buffering mechanisms.

When you close a session, cryptlib will immediately shut down the session as is, without flushing data in internal session buffers. This is done to handle cases where a session is aborted (for example because the user cancels the transaction or because of a network error), and it becomes necessary to exit without sending further data. If you want to send any remaining data before destroying the session, you need to explicitly flush the data through before you destroy the session object (remember to check the return status of the final flush to make sure that all of your data was indeed sent).

Reading the response from the remote client or server works in a similar manner:

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

Unless you specify otherwise, all of cryptlib's network operations are non-blocking and near-asynchronous, waiting only the minimum amount of time for data to be sent or received before returning to the caller (you can make this fully asynchronous if you want, see the next section). cryptlib will provide whatever data is available from the remote client or server or write whatever is possible to the remote client or server and then return, which is particularly important for interactive sessions where small amounts of data are flowing back and forth simultaneously. The amount of data which is returned is indicated by the `bytesCopied` parameter. If this value is zero then no data is available or was written at the current time. Since the interface is non-blocking, your application can perform other processing while it waits for data to arrive.

If you'd prefer to have cryptlib not block at all or block for a longer amount of time when waiting for data to arrive or be sent, you can enable this behaviour as described in "Network Issues" on page 130. With blocking network operations enabled, cryptlib will wait for a user-defined amount of time for data to arrive before returning to the caller. If data arrives or is sent during the timeout interval, cryptlib returns immediately. With non-blocking behaviour it will return immediately without waiting for data to become available.

Since cryptlib reads and writes are asynchronous, you shouldn't assume that all the data you've requested has been transferred when the push or pop returns. cryptlib will only transfer as much data as it can before the timeout, which may be less than the total amount. In particular if data is flowing in both directions at once (that is, both sides are writing data and not reading it), the network buffers will eventually fill up, resulting in no more data being written. If this happens you need to occasionally interleave a read with the writes to drain the buffers and allow further data to be transferred.

Network Issues

Sometimes a machine won't connect directly to the Internet but has to access it through a proxy. cryptlib supports common proxy servers such as socks, and also supports HTTP proxies if the protocol being used is HTTP-based. In addition it may be necessary to adjust other network-related parameters such as timeout values in order to accommodate slow or congested network links or slow clients or servers. The following sections explain how to work with the various network-related options to handle these situations.

Secure Sessions with Proxies

Using a socks proxy requires that you tell cryptlib the name of the socks server and an optional user name (most servers don't bother with the user name). You can set the socks server name with the CRYPT_OPTION_NET_SOCKS_SERVER attribute and the optional socks user name with the CRYPT_OPTION_NET_SOCKS_-USERNAME attribute. For example to set the socks proxy server to the host called socks (which is the name traditionally given to socks servers) you would use:

```
cryptSetAttributeString( CRYPT_UNUSED, CRYPT_OPTION_NET_SOCKS_SERVER,
    "socks", 5 );
```

before activating the session. When you activate the session, cryptlib will communicate with the proxy using the socks protocol.

Using an HTTP proxy works in a similar manner, with the name of the proxy being specified with the CRYPT_OPTION_NET_HTTP_PROXY attribute. Note that HTTP proxies require that the protocol being used employs HTTP as its transport mechanism, so they're not used with any other protocol type.

However, it's also possible to move TLS traffic through most types of HTTP proxies, since TLS is frequently used to carry HTTP data. If you enable the use of an HTTP proxy, cryptlib will also use it for TLS sessions.

Under Windows, cryptlib provides automatic proxy discovery and support. You can enable this by setting the proxy server to [Autodetect]:

```
cryptSetAttributeString( CRYPT_UNUSED, CRYPT_OPTION_HTTP_PROXY,
    "[Autodetect]", 12 );
```

which instructs cryptlib to automatically detect and use whatever proxy is being employed. Since the proxy-discovery process can take a few seconds, you should only enable autodetection if you're sure that a proxy is actually present. Enabling it unconditionally will result in cryptlib spending a lot of time trying to find a proxy that may not exist, which slows down the network connection setup process.

Network Timeouts

When connecting to a server and carrying out other portions of a protocol such as security parameter and session key negotiation (for which cryptlib knows that a response must arrive within a certain time) cryptlib sets an interval timer and reports a connect or read error if no response arrives within that time interval. This means that if there's a network problem such as a host being down or a network outage, cryptlib won't hang forever but will give up after a certain amount of time, defaulting to 30 seconds. You can change the connect timeout value using the CRYPT_-OPTION_NET_CONNECTTIMEOUT attribute, which specifies the connect timeout delay in seconds. For example to set a longer timeout for a remote host or client which is slow in responding you would use:

```
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_NET_CONNECTTIMEOUT,
    60 );
```

to set a one minute timeout when activating the session. If you want to set the connect timeout for a specific session rather than system-wide for all sessions, you can set the attribute only for the session object in question:

```
cryptSetAttribute( cryptSession, CRYPT_OPTION_NET_CONNECTTIMEOUT,
    60 );
```

In addition to the connect timeout cryptlib has a separate timeout setting for network communications, specified using the CRYPT_OPTION_NET_READTIMEOUT and CRYPT_OPTION_NET_WRITETIMEOUT attributes. Since cryptlib session objects normally use non-blocking I/O once the session has been established and data is being exchanged, the read and write timeouts are set to minimal values during any general data exchanges that occur after the connection negotiation process has completed. This means that all communications after that point are near-asynchronous and non-blocking, however by changing the read/write timeout settings you can make cryptlib wait for a certain amount of time for data to arrive or be written before returning. For example to wait up to 30 seconds for data to arrive you would use:

```
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_NET_READTIMEOUT, 30 );
```

If data arrives during the wait interval, cryptlib will return as soon as the data becomes available, otherwise it'll wait for up to 30 seconds for data to arrive.

As with the connect timeout, you can also apply these options directly to session objects, which means that they'll only apply to that particular session rather than being a system-wide setting for all session objects:

```
cryptSetAttribute( cryptSession, CRYPT_OPTION_NET_READTIMEOUT, 30 );
```

If you need the quickest possible response (usually only interactive sessions need this), you can set network read/write timeouts of zero, which will result in cryptlib returning immediately if no data can be read or written. The downside to using a zero timeout is that it reduces data transfers to polled I/O, requiring repeated read or write attempts to transfer data. For write timeouts it's better to set at least a small non-zero timeout rather than a zero timeout to ensure that the data is successfully written. In almost all cases the write will complete immediately even with a non-zero timeout, only in very rare cases such as when network congestion occurs will it be necessary to wait for data to be sent. In other words during a read wait the session is frequently just idling waiting for something to happen, but during a write wait it's actively trying to move the data out, so setting a non-zero timeout will increase the chances of the network layer moving the data out during the current write attempt rather than having to retry the write later.

A second problem with very short timeouts occurs when you close a session. Since writes are fully asynchronous, the network session can be closed before all of the data is written. Although the network stack tries to flush the data through, if there's an error during transmission there's no way to indicate this since the session has already been closed. cryptlib tries to mitigate this by setting a minimum (non-zero) network timeout when it closes a session, but there's no way to guarantee that everything will be sent during the timeout interval (in general this is an unsolvable problem, for example if an intermediate router crashes and is rebooted or the routing protocols hunt around for an alternative route, the transfer will eventually complete, but it could take several minutes for this to happen, which would require an excessively long timeout setting).

To avoid this issue, you should avoid writing a large amount of data with a very small network timeout setting and then immediately closing the session. You can do this by writing data at a measured pace (with a non-zero timeout) during the session or by setting a reasonable write timeout before you flush the last lot of data through and close the session.

Managing your Own Network Connections and I/O

Instead of having cryptlib automatically manage network connections, it's possible for you to manage them yourself. This can be useful when you want to customise session establishment and connection management, for example to handle a STARTTLS upgrade during an SMTP or IMAP session, an STLS upgrade during a POP session, or an AUTH TLS upgrade during an FTP session. You can also use this facility if you want to use any high-performance I/O capabilities provided by your system, for example asynchronous I/O or hardware-accelerated I/O in which a dedicated subsystem manages all network transfers and posts a completion notification to your application when the transfer is complete. This allows you to use your own connection-management/socket-multiplexing/read-write code rather than using the facilities provided by cryptlib.

The following discussion refers to network sockets because this is the most common abstraction that's used for network I/O, however cryptlib will work with any network I/O identifier that can be represented by an integer value or handle. If your network abstraction requires more than a straightforward handle, you can pass in a reference or index to an array of whatever data structures your system requires to handle network I/O.

You can handle your own network connections by adding them to a cryptlib session as the CRYPT_SESSINFO_NETWORKSOCKET attribute before you activate the session. When you activate the session, cryptlib will use the socket that you've supplied rather than opening its own connection. Once you shut down the session, you can continue to use the socket or close it as required:

```
int socket;

/* Connect the network socket */
socket = ...;

/* Add the socket to the session and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_NETWORKSOCKET,
   socket );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

Before you hand the socket over to cryptlib, you should disable Nagle's algorithm, since cryptlib provides its own optimised packet management. cryptlib leaves this task to the caller to ensure that it doesn't have to make any changes to the socket settings itself. In other words, it will leave the socket exactly as it found it. In addition you need to use a blocking socket, since cryptlib implements its own non-blocking I/O layer for portability across different operating systems. This is particularly important for Windows, where the socket must be non-blocking to avoid false reports of the other side closing the connection due to bugs in some versions of Winsock. Note that if you use certain Winsock functions such as `WSAAsyncSelect` and `WSAEventSelect` on the socket, Windows will quietly switch the socket back to non-blocking mode, so you need to be careful about inadvertently changing the state of a socket behind cryptlib's back.

In addition to managing the connection process, you can also use externally-supplied sockets to handle network reads and writes. There are two general mechanisms used for external network I/O, the Berkeley sockets `select()`-style mechanism:

```
select( ... );
…
read( ... );
```

and the posted-read/posted-write mechanism used by high-performance and hardware-accelerated I/O subsystems:

```
read_async( ... );
…
wait_completion( ... );
```

An example of the latter is Windows' I/O completion ports, which allow a central dispatcher to initiate I/Os and a pool of client threads to manage them as required whenever a request completes. The equivalent under Unix (although it's less attuned towards high-performance server operation, being targeted mainly at file I/O) is Posix asynchronous I/O. Other operating systems provide similar facilities, for example Tandem NSK has the RECV_NW and AWAITIOX calls to perform posted reads and writes.

The more widely-used I/O model, using the `select()`-style mechanism, would wait until data is available to be read on the socket and then call **cryptPopData**:

```
/* Wait for data to become available */
select( ... );

/* Read data from the session */
cryptPopData( cryptSession, ... );
```

The posted-read/posted-write mechanism would have a read or write initiated by a central dispatcher (in the example below this is illustrated with Windows-style I/O handling):

```
/* Create an I/O completion port associated with the socket */
hCompletionPort = CreateIOCompletionPort( hSocket, ... );

/* Initiate the read request */
ReadFile( hSocket, ... );
```

Once the read request has been completed by the underlying I/O system, a thread from the thread pool that's waiting on the completion port is woken up and handles the result:

```
/* Wait for data to arrive */
GetQueuedCompletionStatus( hCompletionPort, ... );

/* Read data from the session */
cryptPopData( cryptSession, ... );
```

The Windows kernel contains a number of special optimisations to provide the best possible performance for this type of I/O. If you're running a high-performance server, you should consider using this style of I/O instead of the standard sockets interface for better performance. In fact this style of I/O is the one that's used by servers like IIS to maximise performance.

The Unix equivalent would be:

```
/* Initiate the read request */
aio_read( &aiocb );

/* Wait for data to arrive */
aio_suspend( &aiocb, ... );

/* Read data from the session */
cryptPopData( cryptSession, ... );
```

Unix asynchronous I/O is often used for high-performance I/O when the overhead of the standard BSD `select()` is unacceptable. A typical select implementation, for example, has to first copy and validate the socket descriptor masks for read, write, and exception conditions, then call the underlying device's poll routine for each socket descriptor in each mask to let the device know that an I/O operation is being requested for that descriptor, and finally wait for a notification on any of the descriptors from the lower-level device drivers. There's additional overhead created by the fact that the kernel can't afford to lock out I/O while all of this polling is taking place, so the select code has to be able to handle the case of I/O occurring during the polling process, usually by restarting the poll.

Asynchronous I/O, on the other hand, avoids all of this overhead by simply posting a read or write and then waiting for the kernel to notify it that the operation has completed. It therefore provides much better performance than an equivalent select-based implementation.

If you supply the network socket yourself and the socket is a server socket, you can no longer read the CRYPT_SESSINFO_CLIENT_NAME and CRYPT_SESSINFO_-CLIENT_PORT attributes, since these are recorded when the incoming client connection is established, and won't be present with a user-supplied socket.

Some protocols like SSH have a considerable amount of protocol overhead and require the sending and acknowledgement of messages that don't perform any useful data-exchange function. When these messages are received, the external `select()` will report that there's data available on the socket but cryptlib's **cryptPopData** won't return any data. This is a normal condition for this protocol, since the message is being processed and discarded internally (these message aren't visible if cryptlib is handling the socket since they're processed internally, but do become visible if you're handling socket data yourself). If you're implementing SSH and handling the socket data yourself, you should expect to see the periodic arrival of data on the socket with no data returned from **cryptPopData**.

Key Generation and Storage

The previous sections on enveloping and secure sessions mentioned the use of encryption contexts containing public and private keys. The creation and generation of public/private keys in encryption contexts and the long-term storage of key data is covered in this section. Keys are stored in keysets, an abstract container that can hold one or more keys and that can be a cryptlib key file, a PGP/OpenPGP key ring, a database, an LDAP directory, or a URL accessed via HTTP. cryptlib accesses all of these keyset types using a uniform interface that hides all of the background details of the underlying keyset implementations.

In addition you can generate and store keys in crypto devices such as smart cards, crypto accelerators, and Fortezza cards. Crypto devices are explained in more detail in "Encryption Devices and " on page 282. The direct loading of key data into a context is covered in "Encryption and Decryption" on page 193.

If you're working with very constrained devices then it may not be possible to run a full-featured public-key store like a database. In situations like this you can store public-key certificates as flat files, reading the file into memory and using **cryptImportCert** to turn it into a certificate object. Note that this trades off ease of use for code/memory space, if you have hundreds or thousands of certificates then to manage them they need to be indexed on names, email addresses, expiry dates, and so on, which doesn't work with flat files. The implicit assumption is that if you're working with a small number of certificates then flat files are OK, and if you're working with hundreds or thousands of certificates then whatever you're using should have enough capability to run some form of database, because managing thousands of individual files by hand doesn't work too well.

Key Generation

To create an encryption context, you must specify the user who is to own the object or CRYPT_UNUSED for the default, normal user, and the algorithm you want to use for that context. The available public-key algorithms are given in "Algorithms" on page 335. For example, to create and destroy an RSA context you would use:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_RSA );

/* Load key, perform en/decryption */

cryptDestroyContext( cryptContext );
```

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 that 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 **cryptDeviceCreateContext**, which tells cryptlib to create a context in a crypto device. The use of crypto devices is explained in "Encryption Devices and " on page 282.

Generating a Key Pair into an Encryption Context

Once you've created an encryption context the next step is to generate a public/private key pair into it. Before you can generate the key pair 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 using functions like **cryptAddPrivateKey** and **cryptGetPrivateKey**. If you try to generate a key pair 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_RSA );
cryptSetAttributeString( privKeyContext, CRYPT_CTXINFO_LABEL, label,
   labelLength );
cryptGenerateKey( privKeyContext );
```

To do this in Java or C# you would use:

```
int privKeyContext = crypt.CreateContext( cryptUser
   /* crypt.UNUSED */, crypt.ALGO_RSA );
crypt.SetAttributeString( privKeyContext, crypt.CTXINFO_LABEL,
   label );
crypt.GenerateKey( privKeyContext );
```

The Visual Basic equivalent is:

```
Dim privKeyContext As Long

cryptCreateContext privKeyContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_RSA
cryptSetAttributeString privKeyContext, CRYPT_CTXINFO_LABEL, label, _
   Len( label )
cryptGenerateKey privKeyContext
```

If you want to generate a key of a particular length, you can set the CRYPT_-CTXINFO_KEYSIZE attribute before calling **cryptGenerateKey**. For example to generate a 1536-bit (192-byte) key you would use:

```
CRYPT_CONTEXT privKeyContext;

cryptCreateContext( &privKeyContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_RSA );
cryptSetAttributeString( privKeyContext, CRYPT_CTXINFO_LABEL, label,
   labelLength );
cryptSetAttribute( privKeyContext, CRYPT_CTXINFO_KEYSIZE, 1536 / 8 );
cryptGenerateKey( privKeyContext );
```

You can also change the default encryption and signature key sizes using the cryptlib configuration options CRYPT_OPTION_PKC_KEYSIZE and CRYPT_OPTION_-SIG_KEYSIZE as explained in "Working with Configuration Options" on page 292. Once a key is generated 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.

Although cryptlib can work directly with public/private key data held in an encryption context, you can't communicate this key data to anyone else without first turning it into an encoded key object like a certificate. This is because the key consists of a (potentially large) number of abstract components that need to be encoded into a standard format in order to communicate them to someone else, with a certificate (or some equivalent object like a certificate request) being the standard way to do this.

Because of this key-encoding requirement, you can't immediately use a newly-generated private key for anything other than signing a certification request or a self-signed certificate (although you can store a raw key in a file keyset for later use, see the next section for more details). To use the key for any other purpose, you need to convert it into a certificate and then store the certificate alongside the private key in a cryptlib private key file or crypto device. The process of obtaining a certificate and updating a keyset or device with it is covered in more detail in "Certificates and Certificate Management" on page 154. Once you've obtained the certificate, you can add it to the keyset or device in which the key is stored, and cryptlib will automatically associate it with the key when you read it.

Keyset Types

cryptlib supports a wide variety of keyset types. Most of these are public-key keysets, which means that you can only store X.509 certificates (and by extension the public keys associated with them) in them, but not private keys. These keyset types include database keysets (the cryptlib native format for storing certificates), LDAP directories, and web pages accessed via HTTP.

In addition to the public-key keysets, cryptlib also supports the storage of private keys in cryptlib private key files (which use the PKCS #15 crypto token format) and crypto devices such as smart cards, Fortezza cards, and hardware crypto accelerators. cryptlib keysets can also be used to store certificates, but only those that already have a corresponding private key stored in the keyset. cryptlib private key keysets can't be used as general-purpose public-key or certificate stores, they can only store certificates associated with an existing private key.

The following table summarises the different keyset types and the operations that are possible with each one. Unless you have a strong reason not to do so, it's recommended that you use cryptlib private key files to store private keys and their associated certificates and database keysets to store standalone certificates.

Type	Access Allowed
cryptlib	Read/write access to public/private keys and any associated certificates stored in a file using the PKCS #15 crypto token format, 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 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. More information on crypto devices and on generating private keys in them is given in "Encryption Devices and " on page 282.
Database	Read/write access to X.509 certificates stored in a database. This is the cryptlib native keyset format for public keys and certificates 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 Informix, Ingres, MySQL, Oracle, Postgres, SQLite and Sybase databases via ODBC under other platforms.
HTTP	Read access to X.509 certificates and CRLs accessed via URLs or read access to a general-purpose HTTP certificate storage mechanism.
LDAP	Read/write access to X.509 certificates and CRLs stored in an LDAP directory.
PGP	Read/write access to PGP/OpenPGP key rings.

The recommended method for certificate storage is to use a 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.

Creating/Destroying Keyset Objects

Keysets are accessed as keyset objects that work in the same general manner as the other container objects used by cryptlib. You create a keyset object with **cryptKeysetOpen**, specifying the user who is to own the device object or CRYPT_UNUSED for the default, normal user, 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, cryptUser /* CRYPT_UNUSED */,
   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 private key file or a PGP/-OpenPGP key ring.
CRYPT_KEYSET_HTTP	URL specifying the location of a certificate or CRL or URL specifying the location of an HTTP certificate storage mechanism.
CRYPT_KEYSET_LDAP	LDAP directory.
CRYPT_KEYSET_DATABASE	Generic database interface.
CRYPT_KEYSET_DATABASE_-STORE	As for the basic keyset types, but representing a certificate store for use by a CA rather than a simple keyset. The user who creates and updates these keyset types must be a CA user.

These keyset types and any special conditions and restrictions on their use are covered in more detail below.

The keyset location 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.

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 databases and cryptlib key files.
CRYPT_KEYOPT_NONE	No special access options (this option implies read/write access).
CRYPT_KEYOPT_-READONLY	Read-only keyset access. This option is automatically enabled by cryptlib for keyset types that 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.

File Keysets

For cryptlib private key files and PGP/OpenPGP key rings, the keyset location is the path to the disk file. For example to open a connection to a cryptlib key file key.p15 located in /users/dave/ you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, "/users/dave/keys.p15", CRYPT_KEYOPT_READONLY );
```

cryptlib will automatically determine the file type and access it in the appropriate manner. Since cryptlib uses the PKCS #15 crypto token format to store private keys, the files are given a .p15 extension or an appropriate equivalent as dictated by the operating system being used. As another example, to open a connection to a cryptlib private key file located in the Keys share on the Windows server FileServer you would use:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, "\\FileServer\Keys\key.p15",
   CRYPT_KEYOPT_READONLY );
```

The same operation in Visual Basic is:

```
Dim cryptKeyset As Long

cryptKeysetOpen cryptKeyset, cryptUser, CRYPT_KEYSET_FILE, _
   "\\FileServer\Keys\key.p15", CRYPT_KEYOPT_READONLY
```

A cryptlib private key file contains one or more encrypted private keys and any associated certificates and other information needed to process them. Although cryptlib can write keys to non-cryptlib formats like PGP/OpenPGP, you shouldn't use these for general key storage since they don't contain the information needed for many of the high-level operations that the native format does.

To create a new cryptlib keyset you would use:

```
CRYPT_KEYSET cryptKeyset;

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

The equivalent in Java or C# is:

```
int cryptKeyset = crypt.KeysetOpen( cryptUser /* crypt.UNUSED */,
   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.

Note that cryptlib treats non-native key file formats as transport formats for interoperability and doesn't perform the extensive verification that's required for keys stored in these formats, which for cases like PGP/OpenPGP keyrings would require a complete key-management application with access to PGP key databases. While keys stored in cryptlib's native format are verified when they're loaded, it's up to the user to verify the integrity of keys imported from non-native formats. In addition you shouldn't rely on the non-native formats for key storage but instead rewrite the key to a cryptlib native keyset which provides integrity protection for public and private key components in a manner that the non-native formats often don't.

For security reasons, cryptlib won't write to a 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 the file's 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, you may need to manually enable access for others to allow the file to be backed up or copied.

If your application is running as another user (for example if it's running as a dæmon under Unix or a service under Windows), the keyset will be owned by the dæmon or service that creates it, following standard security practice. If you want to make the keyset accessible to standard users, you need to either change the security options to allow the required user access (for example by changing the file access permissions or running in the context of the intended user when you create it), or provide an interface to your dæmon/service to allow access to the keyset. The latter is generally the preferred option, since it allows your dæmon/service to control exactly what the user can do with the keyset.

In addition if you're installing or configuring cryptlib as one user for use by another user, you'll need to adjust the access for any files that are created during the install or configuration process to allow access by the target user. For example if you install and configure cryptlib as a Windows administrator to run as a system service, you'll need to change the ownership of any key and configuration files to the system account:

```
cacls filename /e /g system:f
```

If you don't do this then the service (running under the system account) can't access the key/configuration files created under the administrator account.

When you open a keyset that contains private keys, you should bind 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, "Private key file.p15", CRYPT_KEYOPT_READONLY );
cryptSetAttribute( cryptKeyset, CRYPT_PROPERTY_OWNER, threadID );
```

You can find out more about binding objects to threads in "Object Security" on page 45.

HTTP Keysets

HTTP keysets can use two ways of accessing keys, one is via an absolute URL that specifies the path to a single key, and the other is via a URL giving the location of an HTTP certificate storage mechanism. In either case the format is:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_HTTP, url, CRYPT_KEYOPT_READONLY );
```

HTTP keysets normally behave just like any other keysets, however if you're reading a key from a fixed URL (with no per-key ID) you need to use the special ID `[none]` to indicate that the keyset URL points directly at the certificate. For example to read a certificate from the static URL `http://www.server.com/cert.der` you would use:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_HANDLE cryptCertificate;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_HTTP, "http://www.server.com/cert.der",
   CRYPT_KEYOPT_READONLY );
cryptGetPublicKey( cryptKeyset, &cryptCertificate, CRYPT_KEYID_NAME,
   "[none]" );
```

The CRLs provided by some CAs 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.

If you're using a HTTP certificate storage mechanism located at `http://www.server.com/certstore/` then the same code would become:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_CERTIFICATE cryptCertificate;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_HTTP, "http://www.server.com/certstore/ ",
   CRYPT_KEYOPT_READONLY );
cryptGetPublicKey( cryptKeyset, &cryptCertificate, CRYPT_KEYID_NAME,
   "User Name" );
```

More details on reading keys are given in "Reading a Key from a Keyset" on page 145.

If you want to publish certificates online, the best way to do this is with an HTTP keyset. The server side of HTTP certificate access is handled as a standard cryptlib session, and is covered in "Making Certificates Available Online" on page 187.

Database Keysets

For keys (strictly speaking, X.509 certificates) that are stored in a database, the keyset name is the ODBC data source name (DSN) for the database. The nature of the name depends on the database type and ranges from a straight ODBC DSN through to a complex combination of the name or address of the server that contains the database, the name of the database on the server, and the user name and password required to access the database.

The database keyset has a type of CRYPT_KEYSET_DATABASE. With some systems that don't support any type of database access (for example some embedded systems have no database capability), cryptlib can't be used with a database keyset and is restricted to the simpler keyset types such as cryptlib private key files.

The simplest type of keyset to access is a local database that requires no extra parameters such as a user name or password. An example of this is an ODBC data source on the local machine identified by a straightforward DSN. For example if the keyset is stored in a database such as DB2, Ingres, MariaDB, MySQL, Oracle, SQL Server, SQLite, Sybase, or PostgreSQL, which is accessed through the "PublicKeys" ODBC DSN, you would access it with:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_DATABASE, "PublicKeys", CRYPT_KEYOPT_READONLY );
```

The same operation in Visual Basic is:

```
Dim cryptKeyset As Long

cryptKeysetOpen cryptKeyset, cryptUser, CRYPT_KEYSET_DATABASE,
   "PublicKeys", CRYPT_KEYOPT_READONLY
```

The database name parameter used above was a simple ODBC DSN, but this can also contain a user name, password, and server name (or at least server DSN), in the format `user:pass@server`. For example, you can specify a combination of database user name and password as `user:pass`, and a user name and server as `user@server`. Other, database-specific combinations and parameters may also be possible, depending on the database backend you're using.

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

To create a new certificate database, you can use the CRYPT_KEYOPT_CREATE option. If a keyset of the given name already exists, cryptlib will return CRYPT_-ERROR_DUPLICATE, otherwise it will create a new certificate database ready to have certificates added to it.

Database keysets can also be used as certificate stores, an extended type of keyset which is required in order to perform CA operations such as issuing certificates and CRLs. In order to create this type of keyset instead of a conventional one you must be a CA user and you need to specify its type as CRYPT_KEYSET_DATABASE_-STORE instead of the basic database keyset type. Because this is a CA-specific certificate store, certificates and CRLs can't be directly added to or deleted from it but have to be processed using cryptlib's certificate management functionality. For example you can't add a certificate directly to a certificate store using **cryptAddPublicKey** but have to create a certificate from a certificate request using **cryptCACertManagement**. More information on certificate stores and their use is given in "Managing a Certification Authority" on page 180.

In order to create or open a certificate store, you must be a CA user. If you try to access a certificate store and aren't a CA user, cryptlib will return CRYPT_ERROR_-PARAM2 to indicate that the user type isn't valid for accessing this type of keyset. Normal users can't update a certificate store in any way, however they can access them in read-only mode as normal database keysets. For example while a CA could open a certificate store as:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_DATABASE_STORE, "certstore.company.com",
   CRYPT_KEYOPT_NONE );
```

and perform updates on the store, a non-CA user could only access it in read-only mode as a standard database keyset:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_DATABASE, "certstore.company.com",
   CRYPT_KEYOPT_READONLY );
```

When opened in this manner the certificate store appears as a standard database keyset rather than as a full certificate store.

To provide additional security alongside the precautions taken by cryptlib, you should apply standard database security measures to ensure that all database keyset accesses are carried out with least privileges. For example if your application only needs read access to a keyset, you can use the SQL GRANT/REVOKE mechanism to allow read-only access of the appropriate kind for the application. An SQL statement like `REVOKE ALL ON certificates FROM user; GRANT SELECT ON certificates TO user` would allow only read accesses to the certificate keyset. You can also use server-specific security measures such as accessing the keyset through SQL Server's built-in db_datareader account, which only allows read access to tables, and the ability to run the application under a dedicated low-privilege account (a standard feature of Unix systems).

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 non-standard port. For example if the LDAP server was called `directory.ldapserver.com`, you would access the keyset with:

```
CRYPT_KEYSET cryptKeyset;

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_LDAP, "directory.ldapserver.com",
   CRYPT_KEYOPT_READONLY );
```

If the server is configured to allow access on a non-standard 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_LDAP, "directory.ldapserver.com:8389",
   CRYPT_KEYOPT_READONLY );
```

You can also optionally include the `ldap://` protocol specifier in the URL, this is ignored by cryptlib.

The storage of certificates in LDAP directories is haphazard and vendor-dependent, and you may need to adjust cryptlib's LDAP configuration options to work with a particular vendor's idea of how certificates and CRLs should be stored on a server. In order to make it easier to adapt cryptlib to work with different vendor's ways of storing information in a directory, cryptlib provides various LDAP-related configuration options that 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 that certificates are stored as. For some reason certificates belonging to certification authorities (CAs) 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). In addition a number of other attribute types have been invented to hide certificates under, it may require a bit of experimentation to determine how the server you're using stores things. The default settings for these options are `userCertificate;binary` and `cACertificate;binary`, a variety of other choices also exist. Note the use of the `binary` qualifier, this is required for a number of directories that 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 that certificate revocation lists (CRLs) are stored as, defaulting to `certificateRevocationList;binary`.
CRYPT_OPTION_KEYS_-LDAP_EMAILNAME	The X.500 attribute that email addresses are stored as, defaulting to `mail`. Since X.500 never defined an email address attribute, various groups defined their own ones, `mail` is the most common one but there are a number of other alternatives around, including `emailAddress`, `rfc822Name`, `rfc822MailBox`, and `email`. As usual, some experimentation will be necessary to find out what works.

Configuration Option	Description
CRYPT_OPTION_KEYS_-LDAP_FILTER	The filter used to selected returned LDAP attributes during a query, defaulting to `(objectclass=*)`. Experimentation will be necessary to determine what's required for this value.
CRYPT_OPTION_KEYS_-LDAP_OBJECTCLASS	The X.500 object class, defaulting to `inetOrgPerson`. Again, there is no consistency among servers, the usual amount of guesswork will be required to find out what works.
CRYPT_OPTION_KEYS_-LDAP_OBJECTTYPE	The object type to fetch, defaulting to CRYPT_CERTTYPE_NONE to fetch all object types. Setting this to CRYPT_-CERTTYPE_CERTIFICATE or CRYPT_-CERTTYPE_CRL will fetch only certificates or CRLs.

These configuration options apply to all LDAP keysets, you can also apply them to an individual keyset object rather than as a general configuration option, which means that they'll affect only the one LDAP keyset object.

There is no consistency in the configuration of LDAP directories, and since the query used to retrieve a certificate depends on how the directory is configured, it's often impossible to tell what to submit without asking the directory administrators for the correct formula. Since the actual values depend on the server configuration, there is no way that cryptlib can determine which ones to use for a given server.

Two examples of magic formulae that are required by different CAs running LDAP directories are "searchDN = CN=Norway Post Organizational CA, O=CA, C=NO, filter = (&(objectclass=*)(pssSubjectDNString=CN=RTV EDI-server 2, O=RTV, C=NO)), attributes = certificateRevocationList;binary" and "(&(|(&(objectclass=-inetorgperson)(objectclass=organizationalperson)) (objectClass=Strong-AuthenticationUser))(usercertificate;binary=*)(|(commonname=*name*)(rfc822-mailbox=*email address*)))". In order to handle some of these combinations you will have to set a selection of the CRYPT_OPTION_KEYSET_LDAP_*xxx* attributes as well as modifying the key ID you use when you actually read a key.

To allow even more flexibility in specifying LDAP access parameters, cryptlib will also accept RFC 1959 LDAP URLs as key IDs (see "Obtaining a Key for a User" on page 145). These have the general form `ldap://host:port/dn?-attributes?scope?filter`, and can be used to specify arbitrarily complex combinations of DN components (see RFC 1485), search scope, and filter (see RFC 1558). For example to specify the Norway post magic formula above as a key ID the LDAP URL would be `ldap:///CN=Norway%20Post%20Organizational%20CA,-%20O=CA,%20C=NO?certificateRevocationList;binary??(&(objectclass=*)-(pssSubjectDNString=%20CN=RTV%20EDI-server%202,%20O=RTV,%20C=NO))`. Note that the ability to use an LDAP URL for lookup in this manner may not be available in some LDAP client implementations.

The default settings used by cryptlib have been chosen to provide the best chance of working, however given that everyone who stores certificates in an LDAP server configures it differently it's almost guaranteed that trying to use LDAP to store certificates will require reconfiguration of the client, the server, the certificates being stored, or several of the above in order to function. In effect the LDAP configuration acts as a form of access control mechanism that makes it impossible to access certificates or CRLs until the CA reveals the correct magic formula. For this reason the use of LDAP is not recommended for storing certificates.

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 URLs can contain only one key, whereas cryptlib private key files, PGP/OpenPGP key rings, databases, and LDAP keysets may contain multiple keys.

You can also use a crypto device such as a smart card, Fortezza card, or crypto hardware accelerator 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 that 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 key's owner, specified as CRYPT_KEYID_NAME and CRYPT_KEYID_EMAIL, or the label assigned to the key as the CRYPT_CTXINFO_LABEL attribute when it's generated or loaded into a context, also specified as CRYPT_KEYID_NAME.

cryptGetPublicKey returns a generic CRYPT_HANDLE that can be either a CRYPT_CONTEXT or a CRYPT_CERTIFICATE depending on the keyset type. Most public-key keysets will return an X.509 certificate, but some keysets (like PGP/OpenPGP key rings) don't store the full certificate information and will return only an encryption context rather than a 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 Crypto device	The key ID is a label attached to the key via the CRYPT_-CTXINFO_LABEL attribute when it's generated or loaded into the context, and is specified using CRYPT_KEYID_-NAME. Alternatively, if a certificate is associated with the key, the key ID can also be the name or email address indicated in the certificate. The key ID is matched in full in a case-insensitive manner.
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 ID is either the name or the email address of the key owner, and is matched in full in a case-sensitive manner. The one exception is when the location is specified by a static URL, in which case the key ID has the special value `[none]`.
LDAP	The key ID is an X.500 distinguished name (DN), which is neither a name nor an email address but a peculiar construction that (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). You can also specify an LDAP URL as the key ID as described in "LDAP Keysets" on page 142.
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 owner's 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 that are present in the key ring. 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 that 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" );
```

In Java or C# this is:

```
int publicKey = crypt.GetPublicKey( cryptKeyset, crypt.KEYID_NAME,
   "Noki S.Crow" );
```

In Visual Basic the operation is:

```
Dim publicKey As Long

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. 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 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 privKeyContext;

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

The same operation in Visual Basic is:

```
Dim privKeyContext As Long

cryptGetPrivateKey cryptKeyset, privKeyContext, 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 that 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 privKeyContext;

/* Try to read the key without a password */
if( cryptGetPrivateKey( cryptKeyset, &privKeyContext,
   CRYPT_KEYID_NAME, name, NULL ) == CRYPT_ERROR_WRONGKEY )
   {
   /* Ask the user for the key's password and retry the read */
   password = ...;
   cryptGetPrivateKey( cryptKeyset, &privKeyContext, 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 that fit a given match criterion. For example you could retrieve a list of all the keys that are set to expire within the next fortnight (to warn their owners that they need to renew them), or that 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 that are set to expire within the next fortnight. cryptlib will return all certificates that 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_KEYINFO_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_KEYINFO_QUERY,
   "$email='noki@crow.com'", 22 );
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. Note that the key ID is unused because the keys that 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 that 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, $OU, $CN	Certificate country, state or province, locality, 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_KEYINFO_QUERY, "$C='US'", 7 );
```

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

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "$C='US' AND
    $SP='CA'", 20 );
```

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

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "$C='US' AND
    $SP='CA' AND $L='San Francisco'", 43 );
```

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_KEYINFO_QUERY, "$C='US' AND
    $SP='CA' AND $L='San Francisco' AND $CN LIKE 'A%'", 61 );
```

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_KEYINFO_QUERY, "$C='US' AND
    $SP='CA' AND $L='San Francisco' ORDER BY $CN", 56 );
```

To return them in reverse order you would use:

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "$C='US' AND
    $SP='CA' AND $L='San Francisco' ORDER BY $CN DESCENDING", 67 );
```

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

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "$date < today +
    1 week", length );
```

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

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

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

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

As these examples show, cryptlib’s 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_KEYINFO_QUERY attribute to "cancel":

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_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.

Unlike all other cryptlib keyset operations, the keyset query capability provides a powerful general-purpose access mechanism that allows you to send arbitrary command strings in the form of SQL queries to the keyset database. Because of this capability you should never allow user data to be used directly in a keyset query since (with the appropriate use of backend-specific escape sequences) it's possible for a malicious user to submit arbitrary SQL command strings to the database (SQL injection). If you must use the general database query facility, use it only with fixed strings containing no externally-supplied data whose format you can't control, and as an extra precaution open the database in read-only mode and access it as a limited-access user with no write or update privileges for the database.

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 CAs 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 perform a keyset query to read each in turn and try and find one that 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 is to issue a query based on the name and then read each certificate that matches the query until you find an appropriate one:

```
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "…", … );
while( cryptGetPublicKey( &certificate, keyset, … ) == CRYPT_OK && \
       certificate doesn't match required usage )
      /* Continue */;
cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "cancel", 6 );
```

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

Key Group Management

Sometimes it may be desirable to treat a group of keys in the same way. For example if a collection of servers use keys to protect their communications with each other then compromise of one key may require the revocation of all keys in the group and the issuance of a new group of keys. The easiest way to handle key groups is by assigning a common identifier to all the keys in the group when you issue certificates for them, and then replacing all keys with that identifier when it comes time to update the key group.

The first part of the process involves assigning a key group identifier to certificates. The easiest way to do this is to specify it as part of the PKI user information that's used with the CMP and SCEP protocols. For example to specify that a PKI user belongs to the remote access users key group using the organisational unit portion of the user DN you would use:

```
/* ... */

/* Add PKI user identification information */
cryptSetAttributeString( cryptPKIUser, CRYPT_CERTINFO_COUNTRYNAME,
   countryName, 2 );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_ORGANIZATIONNAME, organizationName,
   organizationNameLength );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, "Remote access key group",
   23 );
cryptSetAttributeString( cryptPKIUser, CRYPT_CERTINFO_COMMONNAME,
   commonName, commonNameLength );

/* ... */
```

When the user requests their certificate, the key group will be given as the organisational unit component (alongside the other components such as the organisation name and country) in their DN. More information on working with PKI users is given in "Initialising PKI User Information" on page 182. Alternatively, you can manually set the key group identifier when you issue a certificate to someone in the key group if you're manually issuing certificates rather than using an automated mechanism like CMP or SCEP.

The second part of the process involves identifying all of the certificates in a key group that need to be revoked or replaced. This is handled through cryptlib's keyset query capability, retrieving each certificate in the group in turn:

```
CRYPT_CERTIFICATE certificate;

cryptSetAttributeString( keyset, CRYPT_KEYINFO_QUERY, "$OU='Remote
   access key group'", 30 );
do
   status = cryptGetPublicKey( keyset, &certificate, CRYPT_KEYID_NONE,
      NULL );
while( cryptStatusOK( status ) );
```

Once the certificate has been fetched, you can revoke it or notify the owner that they need to replace it as required. More information on keyset queries is given in "General Keyset Queries" on page 147.

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 certificates, so the object that you write to these keysets must be a CRYPT_CERTIFICATE and not just a CRYPT_CONTEXT. In contrast, keysets such as cryptlib private key files primarily store public/private key pairs but can also store the certificate or certificates that are associated with the private key. If you try to write the incorrect type of object to a keyset (for example a private key to a certificate 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.

If you try to write a key to a read-only keyset, cryptlib will return CRYPT_ERROR_-PERMISSION to indicate that you can't write to the keyset. If you try to write a certificate to a cryptlib private key file or a crypto device that doesn't already have a corresponding private key present, cryptlib will return CRYPT_ERROR_PARAM2 to indicate that you can't add this type of object if there isn't already a matching private key present. If you just want to write a certificate to a file, you can use **cryptExportCert** to obtain the certificate and then write that to a file.

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

```
cryptAddPublicKey( cryptKeyset, cryptCertificate );
```

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

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 AES) to encrypt the key with the given password. If you're writing the private key to a crypto device, the password parameter should be set to NULL since the device provides its own protection for the key (not all devices support direct key loading, some require the key to be generated inside the device).

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

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

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 or crypto device, this would delete both the certificate and the key it corresponds to. Finally, certificate stores can't be directly manipulated by adding or deleting certificates and CRLs but must be managed using cryptlib's certificate management functionality. If you try to directly insert or delete a certificate or CRL, cryptlib will return CRYPT_ERROR_-PERMISSION to indicate that this operation isn't allowed.

There is one instance in which it's possible to add a new certificate to a cryptlib private key file when there's already an existing certificate present, and that's when the new certificate updates the existing one. For example some CAs will re-issue a certificate with a newer expiry date (rather than using a new key and certificate), if you add this new certificate to the keyset cryptlib will replace the existing, older certificate with the newer one and use the newer one in all future operations.

Writing to a PGP public keyring works somewhat differently from writing to a standard cryptlib key file because of various constraints imposed by the PGP format. Unlike standard cryptlib keysets, a PGP public keyring is just a collection of individual keys concatenated together, where an individual key consists of an interlinked collection of subkeys, identity packets, binding signatures, and assorted other paraphernalia. Because of the need to bind multiple keys and identities together into something that appears as a single key when accessed, cryptlib will allow a single key or linked set of keys to be written to a PGP public keyring but won't allow it to be updated later with further keys because the necessary key bindings won't be available any more. If you need to merge individual PGP public key files into a large keyring, you can just concatenate the files together.

Another restriction imposed by the PGP format is that even when writing to a public keyring you need to provide a private key when you call **cryptAddPublicKey**. This is necessary because the key needs to have a binding signature applied to it when it's written, and for that a private key is required.

A final complication is created by the fact that for some encryption algorithms PGP actually uses two different keys that appear as the same key, one that only does encryption and another that only does digital signatures. When you add one of these paired keys to a PGP keyring you first need to add the encryption key and then the signature private key, which is used to sign the encryption key.

Taking all of these restrictions into account, the process for saving a set of PGP encryption and signature keys to a PGP public keyring is:

```
cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED, CRYPT_KEYSET_FILE,
   "pubring.pgp" );
cryptAddPublicKey( cryptKeyset, encryptionPublicKey );
cryptAddPublicKey( cryptKeyset, signaturePrivateKey );
cryptKeysetClose( cryptKeyset );
```

If you're using a single key that can perform both encryption and signatures then you only need to add the single (dual-purpose) private key.

When you're working with devices, 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 that might be present in your system.

Although cryptlib can work directly with private keys, other formats like X.509 certificates, S/MIME messages, and TLS 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 that identifies the key and then store the certificate alongside the private key in a cryptlib private key file or crypto device. The process of obtaining a certificate and updating a keyset or device with it is covered in more detail in “Certificates and Certificate Management” on page 154. Once you’ve obtained the certificate, you can add it to the keyset or device and cryptlib will automatically associate it with the key when you 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. CRLs and their uses are explained in more detail in “Certificate Revocation using CRLs” on page 240.

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 crash or power outage in the middle of a long sequence of update operations).

To update a private key with a new password you would use:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_CONTEXT cryptKey;

/* Read the key from the keyset using the old password */
cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   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 same operation in Visual Basic is:

```
Dim cryptKeyset As Long
Dim cryptKey As Long

' Read the key from the keyset using the old password
cryptKeysetOpen cryptKeyset, cryptUser, 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
```

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 private key file or crypto device, this will delete the key and any certificates that 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.

Certificates and Certificate Management

Although cryptlib can work directly with private keys, other formats like X.509 certificates, S/MIME messages, and TLS 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 that identifies the key. Once you've obtained the certificate, you can update the keyset or device that contains the basic public/private key data with additional certificate information. This additional information can be a standalone certificate or a full certificate chain from a trusted root CA down to the end user certificate. This chapter covers the details of obtaining a certificate or certificate chain and attaching it to a private key.

The certificate management message exchange is usually carried out via HTTP or email or through some other unspecified mechanism, however cryptlib also supports the Certificate Management Protocol (CMP) and Simple Certificate Enrolment Protocol (SCEP), which define a mechanism for communicating with a CA to obtain certificates and request the revocation of existing certificates. This chapter explains how to use CMP and SCEP to obtain a certificate or request a revocation from a CA. In order to check a certificate's status, you can use the real-time certificate status protocol (RTCS) or the server-based certificate validation protocol (SCVP) to perform a certificate status check, or the online certificate status protocol (OCSP) to perform a certificate revocation check only. The RTCS, OCSP, and SCVP checking processes are also covered in this chapter.

High-level vs. Low-level Certificate Operations

As with the general cryptlib programming interface, cryptlib supports certificate management operations at three levels:

Plug-and-play PKI

The highest level is the plug-and-play PKI level, which is the easiest one to use and therefore the recommended one. At this level, cryptlib handles all certificate processing and management operations for you, requiring no special knowledge of certificate formats, protocols, or operations. Because of its simplicity and ease of use, it's strongly recommended that you use this interface if at all possible.

Mid-level Certificate Management

The intermediate level requires some knowledge of key generation procedures and certificate management operations. This level involves the use of CMP and SCEP to obtain certificates and manage a CA, and RTCS, OCSP, and SCVP for certificate status checking. Most of the details of certificate management are taken care of for you by cryptlib, but you'll need to perform some manual handling of certificate management operations.

Low-level Certificate Management

The lowest level involves manually managing certificates and certificate revocations, and requires dealing with an entire range of arcane, difficult-to-use, and largely dysfunctional mechanisms such as Distinguished Names, X.500 directories, certificate revocation lists, and assorted other paraphernalia. Working with certificates at this level is extraordinarily difficult, and you should be absolutely certain that you're prepared for the large amount of effort that will be required to make anything work. At a minimum, you should read through and understand the certificate tutorials mentioned in "Recommended Reading" on page 13 before trying to do anything with low-level certificate operations.

If you're absolutely certain that you must work with certificates at a low level, and that you understand just how much effort will be involved, you can find out more about low-level certificate operations in "Certificates in Detail" on page 213 and "Certificate Extensions" on page 246.

Certificates and Keys

Once a public/private key pair is saved to a private key keyset, cryptlib allows extra certificate information to be 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 to CA;
receive certificate from CA;
update keyset to include certificate;
```

If the certificate is a self-signed CA certificate, there's no need to obtain the certificate from an external CA and you can add it directly to the keyset after you create it. 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 to CA;
receive certificate from CA;
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.

Once you've updated the private key with a certificate (which is the only time you can write a public key certificate to a private key keyset), cryptlib will automatically associate the certificate with the private key so that when you read it with **cryptGetPrivateKey** cryptlib will recreate the certificate alongside the key and attach it to the key. You can then use the combined certificate and key to perform operations that require the use of certificates such as certificate signing, S/MIME email decryption and signing, and user authentication. If you update the private key with a complete certificate chain instead of just a single certificate, cryptlib will attach the full certificate chain to the key when you read it with **cryptGetPrivateKey**.

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 an existing private key, cryptlib will return a CRYPT_ERROR_PARAM2 error to indicate that the information in the certificate object being added is incorrect. If there's already a certificate for this key present, cryptlib will return a CRYPT_ERROR_DUPLICATE error to indicate that one key can't have two different certificates associated with it. See "Writing a Key to a Keyset" on page 150 for more on writing keys to keysets.

Using Separate Signature and Encryption Certificates

It's good security practice to use different keys for signing and encryption, and most digital signature laws contain some requirement that the two capabilities be implemented with separate keys. cryptlib supports the use of two (or more) keys belonging to a single user, the only issue to be aware of is that you should give each key a distinct label to allow it to be selected with **cryptGetPrivateKey**. For example the process of creating a keyset containing separate signature and encryption keys with the signature key labelled "My signature key" and the encryption key labelled "My encryption key" would be:

```
set key label to "Signature key";
generate public/private signature key pair;
set key label to "Encryption key";
generate public/private encryption key pair;
write key pairs to keyset;
submit certification requests to CA;
receive signature certificate from CA;
receive encryption certificate from CA;
update keyset to include certificates;
```

When you want to sign data, you would call **cryptGetPrivateKey** specifying the use of "Signature key"; when you want to decrypt data you would call **cryptGetPrivateKey** specifying the use of "Encryption key" (or cryptlib's automatic key management will find it for you if you're using it with a cryptlib envelope).

Plug-and-play PKI

The easiest way to set up keys and certificates is through cryptlib's plug-and-play PKI facility, which performs the operations described above for you. To set up keys and certificates in this manner, cryptlib requires a private-key keyset or crypto token such as a smart card or Fortezza card to store keys and certificates in, the URL of a plug-and-play PKI-capable CA, and a user name and password to authorise the issuing of the certificates. The session type for the plug-and-play PKI is CRYPT_-SESSION_CMP, the same type as a standard CMP session except that cryptlib manages everything for you. The private-key keyset or crypto token is specified as CRYPT_SESSINFO_CMP_PRIVKEYSET, and the user name and password to authorise the operation are provided as the CRYPT_SESSINFO_USERNAME and CRYPT_SESSINFO_PASSWORD:

```
CRYPT_SESSION cryptSession;
CRYPT_KEYSET cryptKeyset;

/* Create the CMP session and private-key keyset */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_CMP );
cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, keysetName, CRYPT_KEYOPT_CREATE );

/* Add the server name/address */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );

/* Add the username, password, and private-key keyset */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_PRIVKEYSET,
   cryptKeyset );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

The same operation in Visual Basic is:

```
Dim cryptSession As Long
Dim cryptKeyset As Long

' Create the CMP session and private-key keyset
cryptCreateSession cryptSession, cryptUser, CRYPT_SESSION_CMP
cryptKeysetOpen cryptKeyset, cryptUser, CRYPT_KEYSET_FILE, _
   keysetName, CRYPT_KEYOPT_CREATE


' Add the server name/address
cryptSetAttributeString cryptSession CRYPT_SESSINFO_SERVER, _
   server, Len( server )
```

```
' Add the username, password, and private-key keyset
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_USERNAME, _
   userName, Len( userName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_PASSWORD, _
   password, Len( password )
cryptSetAttribute cryptSession, CRYPT_SESSINFO_CMP_PRIVKEYSET, _
   cryptKeyset

' Activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Once this process has been completed, the private-key keyset or crypto token that you provided will contain a signature key identified by the label "Signature key", and an encryption key identified by the label "Encryption key" if the public-key algorithm being used is capable of encryption, along with any additional certificates and CA certificates that are required to use the keys. Both keys will be protected using the password that you provided to authenticate the certification process. If you want to change the password for either of the keys you can do so as described in "Changing a Private Key Password" on page 152 before you close the keyset and commit the data to disk. Alternatively, if you want to retain the password that you used for the certificate issue to protect the keys and certificates, you can close the keyset immediately after you add it to the session and cryptlib will manage it for you.

If the CA is issuing you a CA certificate of your own, the keyset or crypto token will contain a single CA signing key identified by the label "Signature key". Since CA keys can't be used for encryption or general-purpose signing but only for signing other certificates, only the single CA signing key is created.

In addition to returning your own certificates, the plug-and-play PKI mechanism also performs a PKIBoot certificate bootstrap operation that downloads an initial trusted certificate set for you to use. This trusted certificate set only contains a small number of known-good certificates trusted by the CA that provided you with your own certificates, rather than the 100+ certificates that you'd be forced to automatically trust when you use a web browser (some of these browser certificates have weak 512-bit keys, or are owned by CAs that have gone out of business, or whose private keys have been on-sold to third parties when the original owner went bankrupt, sometimes passing through multiple owners). The PKIBoot operation allows an end user, starting with nothing more than the user name and password used for the plug-and-play PKI operation to acquire all of the information necessary to use the PKI, without having to manually download and install certificates, or being forced to trust a large collection of certificates from unknown CAs.

Once the PKIBoot process has completed, the trusted certificates will be present in memory as standard cryptlib trusted certificates (see "Certificate Trust Management" on page 243). To commit them to permanent storage and make them available for future cryptlib sessions, you need to save the cryptlib configuration data as explained in "Working with Trust Settings" on page 243:

```
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_CONFIGCHANGED, FALSE );
```

If you don't want to rely on the PKIBoot trusted certificates, don't commit the configuration data to permanent storage and they'll be deleted from memory the next time cryptlib is restarted.

At this point the keys are ready for use for encryption, signing, email protection, authentication, and so on. Because of the ease of use provided by the plug-and-play PKI facility, it's strongly recommended that you use this in place of any other certificate management process, since the alternatives require significantly larger amounts of effort in order to do more or less the same thing.

The Certification Process

Creating a private key and an 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 CA;
receive certificate from CA;
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/private key context, set a label for it, and
   generate a key into it */
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, fileName, CRYPT_KEYOPT_CREATE );
cryptAddPrivateKey( cryptKeyset, cryptContext, password );
cryptKeysetClose( cryptKeyset );

/* Clean up */
cryptDestroyContext( cryptContext );
```

The same operation in Java or C# is:

```
/* Create an RSA public/private key context, set a label for it, and
   generate a key into it */
int cryptContext = crypt.CreateContext( cryptUser /* crypt.UNUSED */,
   crypt.ALGO_RSA );
crypt.SetAttributeString( cryptContext, crypt.CTXINFO_LABEL, "Private
   key" );
crypt.GenerateKey( cryptContext );

/* Save the generated public/private key pair to a keyset */
int cryptKeyset = crypt.KeysetOpen( cryptUser /* CRYPT_UNUSED */,
   crypt.KEYSET_FILE, fileName, crypt.KEYOPT_CREATE );
crypt.AddPrivateKey( cryptKeyset, cryptContext, password );
crypt.KeysetClose( cryptKeyset );

/* Clean up */
crypt.DestroyContext( cryptContext );
```

The Visual Basic equivalent is:

```
Dim cryptContext As Long
Dim cryptKeyset As Long

' Create an RSA public/private key context, set a label for it,
' and generate a key into it
cryptCreateContext cryptContext, cryptUser, 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, cryptUser, CRYPT_KEYSET_FILE, filename, _
   CRYPT_KEYOPT_CREATE
cryptAddPrivateKey cryptKeyset, cryptContext, password

' Clean up
cryptKeysetClose cryptKeyset
cryptDestroyContext cryptContext
```

The process for a crypto device is identical except that the keyset write is omitted, since the key is already held inside the device.

At the same time that you create and save the public/private key pair, you would create a certification request:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create a certification request and add the public key to it */
cryptCreateCert( &cryptCertRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTREQUEST );
cryptSetAttribute( cryptCertRequest,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext );

/* Fill in the certification request details */
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_COUNTRYNAME,
   countryName, countryNameLength );
cryptSetAttributeString( cryptCertRequest,
   CRYPT_CERTINFO_ORGANISATIONNAME, organisationName,
   organisationNameLength );
cryptSetAttributeString( cryptCertRequest,
   CRYPT_CERTINFO_ORGANISATIONALUNITNAME, organisationalUnitName,
   organisationalUnitNameLength );
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_COMMONNAME,
   commonName, commonNameLength );
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_EMAIL,
   emailAddress, emailAddressLength );

/* Sign the request */
cryptSignCert( cryptCertRequest, cryptContext );
```

The equivalent in Visual Basic is:

```
Dim cryptCertRequest As Long

' Create a certification request
cryptCreateCert cryptCertRequest, cryptUser, _
   CRYPT_CERTTYPE_CERTREQUEST
cryptSetAttribute cryptCertRequest,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext

' Fill in the certification request details
cryptSetAttributeString cryptCertRequest, _
   CRYPT_CERTINFO_COUNTRYNAME, countryName, countryNameLength
cryptSetAttributeString cryptCertRequest, _
   CRYPT_CERTINFO_ORGANISATIONNAME, organisationName, _
   organisationNameLength
cryptSetAttributeString cryptCertRequest, _
   CRYPT_CERTINFO_ORGANISATIONALUNITNAME, _
   organisationalUnitName, organisationalUnitNameLength
cryptSetAttributeString cryptCertRequest, _
   CRYPT_CERTINFO_COMMONNAME, commonName, _
   commonNameLength
cryptSetAttributeString cryptCertRequest, _
   CRYPT_CERTINFO_EMAIL, emailAddress, emailAddressLength

' Sign the request
cryptSignCert cryptCertRequest, cryptContext
```

The certificate request details vary depending on what you'll want in the certificate that you're requesting, the values given above, consisting of a country name, organisation details, and a name and email address, are illustrative only. At a minimum, you need to supply the certificate identification information described in "Certificate Identification Information" on page 225 and "Extended Certificate Identification Information" on page 230. Depending on the situation, you may also be able to specify additional certificate components of the type described in “Certificate Extensions” on page 246.

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 that you wrote the key to open and update it immediately with the certificate:

```
CRYPT_CERTIFICATE cryptCertificate;

/* Send the certification request to the CA and obtain the returned
   certificate */
/* ... */

/* Import the certificate and check its validity */
cryptImportCert( cert, certLength, cryptUser /* CRYPT_UNUSED */,
   &cryptCertificate );
cryptCheckCert( cryptCertificate, caCertificate );

/* Update the still-open keyset with the certificate */
cryptAddPublicKey( cryptKeyset, cryptCertificate );

/* Clean up */
cryptKeysetClose( cryptKeyset );
cryptDestroyCert( cryptCertificate );
```

Again, the Visual Basic equivalent for this is:

```
Dim cryptCertificate As Long

' Send the certification request to the CA and obtain the
' returned certificate
' ...

' Import the certificate and check its validity
cryptImportCert cert, certLength, cryptUser, cryptCertificate
cryptCheckCert cryptCertificate, caCertificate

' Update the still-open keyset with the certificate
cryptAddPublicKey cryptKeyset, cryptCertificate

' Clean up
cryptKeysetClose cryptKeyset
cryptDestroyCert cryptCertificate
```

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 cryptCertificate;
CRYPT_KEYSET cryptKeyset;

/* Obtain the returned certificate from the CA */
/* ... */

/* Import the certificate and check its validity */
cryptImportCert( cert, certLength, cryptUser /* CRYPT_UNUSED */,
   &cryptCertificate );
cryptCheckCert( cryptCertificate, caCertificate );

/* Open the keyset for update and add the certificate */
cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, fileName, CRYPT_KEYOPT_NONE );
cryptAddPublicKey( cryptKeyset, cryptCertificate );
cryptKeysetClose( cryptKeyset );

/* Clean up */
cryptDestroyCert( cryptCertificate );
```

The Visual Basic equivalent is:

```
Dim cryptCertificate As Long
Dim cryptKeyset As Long

' Obtain the returned certificate from the CA
' ...

' Import the certificate and check its validity
cryptImportCert cert, certLength, cryptUser, cryptCertificate
cryptCheckCert cryptCertificate, caCertificate

' Open the keyset for update and add the certificate
cryptKeysetOpen cryptKeyset, cryptUser, CRYPT_KEYSET_FILE, fileName, _
   CRYPT_KEYOPT_NONE
cryptAddPublicKey cryptKeyset, cryptCertificate
cryptKeysetClose cryptKeyset

' Clean up
cryptDestroyCert cryptCertificate
```

Again, device updates work in the same manner.

A final case involves self-signed certificates that are typically CA root certificates. Since self-signed CA certificates can be created on the spot, you can immediately update the still-open keyset with the self-signed certificate without any need to go through the usual certification process. When you create a CA certificate you need to set the CRYPT_CERTINFO_CA attribute to true (any nonzero value) to indicate that the certificate (and by extension the private key associated with it) is a CA certificate. If you don't do this and then try to sign a certificate using the key, cryptlib will return CRYPT_ERROR_INVALID to indicate that the key can't sign certificates because it isn't a CA key. To create a self-signed CA certificate you would do the following:

```
CRYPT_CERTIFICATE cryptCertificate;

/* Create a self-signed CA certificate */
cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_SELFSIGNED, 1 );
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_CA, 1 );
/* ... */

/* Sign the certificate with the private key and update the still-open
   keyset with it */
cryptSignCert( cryptCertificate, cryptContext );
cryptAddPublicKey( cryptKeyset, cryptCertificate );

/* Clean up */
cryptKeysetClose( cryptKeyset );
cryptDestroyCert( cryptCertificate );
```

When you sign a certificate for which the CRYPT_CERTINFO_CA attribute has been set, cryptlib will enable the key usages CRYPT_KEYUSAGE_KEYCERTSIGN and CRYPT_KEYUSAGE_CRLSIGN to indicate that the key can be used to sign certificates and CRLs. Since this is a CA key it will by default only be usable for these purposes and not for any other purpose such as encryption or general-purpose signing. You can override this by setting the key usage yourself, however CA keys shouldn't really be used for a purpose other than one or both of certificate and/or CRL signing.

Simple Certificate Creation

The process of creating a certificate is a rather complicated task that can be somewhat daunting when all you want to do is exchange a public key with someone. In order to simplify the process, cryptlib provides a facility to create simplified certificates that don't require you to go through all of the steps outlined in the following sections. These simplified certificates are valid for any type of usage (including encryption, signing, use in TLS servers and S/MIME, and issuing other certificates and CRLs) and have a long enough lifetime that you don't have to worry about them expiring or becoming invalid while you're still using them.

To create one of these simplified certificates, you first add the public key and then set the CRYPT_CERTINFO_XYZZY attribute to tell cryptlib to create a simplified certificate (this has to be added after the public key information so that cryptlib can auto-configure the certificate as a signature and/or encryption certificate depending on the key that's added), add a name via the CRYPT_CERTINFO_-COMMONNAME attribute (and an email address via the CRYPT_CERTINFO_-RFC822NAME attribute if you plan to use the certificate for email purposes), and finally sign the certificate. The name is usually the name of the certificate owner, but if you want to use it with an TLS server then it's the name of the TLS server. For example to create a simplified certificate for Dave Smith you would first generate a public/private key pair into a `cryptContext` as described in "The Certification Process" on page 157 and then use:

```
CRYPT_CERTIFICATE cryptCertificate;

/* Create a certificate and add the public key */
cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCertificate,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext );

/* Indicate that it's a simplified certificate, add the certificate
   owner name, and sign the certificate with the private key */
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_XYZZY, 1 );
cryptSetAttributeString( cryptCertificate, CRYPT_CERTINFO_COMMONNAME,
   "Dave Smith", 10 );
cryptSignCert( cryptCertificate, cryptContext );
```

To create a simplified certificate for the TLS server www.tlsserver.com you would go through the same steps but give the server name instead of the user's name:

```
/* ... */
cryptSetAttributeString( cryptCertificate, CRYPT_CERTINFO_COMMONNAME,
   "www.tlsserver.com", 17 );
/* ... */
```

Finally, if you wanted to use the certificate for email purposes you also need to add the certificate owner's email address:

```
/* ... */
cryptSetAttributeString( cryptCertificate, CRYPT_CERTINFO_RFC822NAME,
   "dave@smith.com", 14 );
/* ... */
```

The same operation in Java or C# is:

```
/* Create a certificate and add the public key */
int cryptCertificate = crypt.CreateCert( cryptUser /* crypt.UNUSED */,
   crypt.CERTTYPE_CERTIFICATE );
crypt.SetAttribute( cryptCertificate,
   crypt.CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext );

/* Indicate that it's a simplified certificate, add the certificate
   owner name, and sign the certificate with the private key */
crypt.SetAttribute( cryptCertificate, crypt.CERTINFO_XYZZY, 1 );
crypt.SetAttributeString( cryptCertificate, crypt.CERTINFO_COMMONNAME,
   "Dave Smith" );
crypt.SignCert( cryptCertificate, cryptContext );
```

The Visual Basic version is:

```
Dim cryptCertificate As Long

' Create a certificate and add the public key
cryptCreateCert cryptCertificate, cryptUser,
   CRYPT_CERTTYPE_CERTIFICATE
cryptSetAttribute cryptCertificate,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext

' Indicate that it's a simplified certificate, add the certificate
' owner name, and sign the certificate with the private key
cryptSetAttribute cryptCertificate, CRYPT_CERTINFO_XYZZY, 1
cryptSetAttributeString cryptCertificate, CRYPT_CERTINFO_COMMONNAME, _
   "Dave Smith", 10
cryptSignCert cryptCertificate, cryptContext
```

Since these certificates can be used for any purpose and (effectively) never expire, you can use them without having to worry about certificate requests, communicating with (and paying money to) a CA, proof of possession protocols, X.500 distinguished names, key usages, certificate extensions, and all the other paraphernalia that comes with X.509 certificates.

In order to distinguish these simplified certificates from normal certificates, cryptlib indicates that they were issued under a simplified-certificate policy using the certificatePolicies attribute, which is described in more detail in "Certificate Policies, Policy Mappings, Policy Constraints, and Policy Inhibiting" on page 249.

Obtaining Certificates using CMP

The discussion so far has covered the means of communicating with the CA in very general terms. Typically the message exchange is carried out via HTTP or through some other, unspecified mechanism. In addition to these very flexible communications options, cryptlib also supports the Certificate Management Protocol (CMP), which defines a mechanism for communicating with a CA to obtain certificates and request the revocation of existing certificates. CMP makes use of session objects as described in "Secure Sessions" on page 102, the following description assumes that you're familiar with the operation and use of cryptlib session objects.

The general process involved in a CMP session is a two-step one of which the first step is creating the appropriate request, for example a request for a new, updated, or additional certificate or a revocation of an existing certificate, and the second step is submitting it to a CA for processing. The result of the processing (typically a signed certificate) is returned at the end of the session:

```
create a CMP request;
fill in the request details;
sign the request;

create a CMP session;
add the CMP server address and request type;
add user name and password or signature key;
add the issuing CA's certificate;
add the CMP request;
activate the CMP session;
obtain the result from the CMP session;
destroy the CMP session;
```

The process involved in creating a request for use in CMP is mostly identical to creating a normal certification request (although the formats are incompatible cryptlib hides the details so the programming interface is identical) and is covered below.

cryptlib also implements a full CMP server that allows you to issue certificates using CMP. This process is described in "Managing a CA using CMP or SCEP" on page 186.

CMP Operation Types

The CMP protocol provides for a variety of certificate issue operations, which are as follows::

Operation	Description
CRYPT_REQUESTTYPE_-PKIBOOT	Request for CA information such as the CA's certificates, protected by a user name and password supplied by the CA.
CRYPT_REQUESTTYPE_-INITIALISATION	Initial request to a CA, protected by a user name and password supplied by the CA.
CRYPT_REQUESTTYPE_-CERTIFICATE	Subsequent request to the CA, protected by a signature created with an existing CA-certified key. This message is used to request a new certificate.

Operation	Description
CRYPT_REQUESTTYPE_-KEYUPDATE	Subsequent request to the CA, protected by a signature created with an existing CA-certified key. This message is used to request an update of an existing certificate.
CRYPT_REQUESTTYPE_-REVOCATION	Request for revocation of an existing certificate, protected either by a password supplied by the CA or by a signature created with an existing CA certified key.

When you submit a CMP request, you need to specify the request type before you activate the session. If it's a PKIBoot, Initialisation, or (for some CAs) Revocation request then the session is authenticated using a user name and password that was previously obtained from the CA. If it's a Certificate or Key Update or (for some CAs) Revocation request then the session is authenticated using a signature created with a key that was previously certified by the CA.

The difference between a Certificate request and a Key Update request is that a Certificate request is used to request a certificate for a new key while a Key Update request is used to request an updated certificate for an existing key. Since a Key Update reuses existing certificate data, you need to change something in the certificate request like its validity date(s) in order to avoid it being rejected as a request for a duplicate certificate.

The typical CMP message flow consists of a PKIBoot to obtain the CA information followed by an Initialisation request to get the initial certificate, and then a standard Certificate request to obtain subsequent certificates.

The PKIBoot request isn't a standard certificate request but simply returns CA parameters like the CA's certificates and other data. This is normally handled transparently as part of cryptlib's plug-and-play PKI, described in "Plug-and-play PKI" on page 156, but you can also submit an explicit PKIBoot request yourself in order to obtain or update CA and other certificates. Since the PKIBoot process is protected by the same cryptographic mechanisms used to protect other operations, there's no risk of someone slipping in fake CA certificates since the cryptographic mechanisms used ensure that the certificates that you're getting came from the real CA. This makes PKIBoot an ideal mechanism for provisioning devices with certificates, as discussed in "Plug-and-play PKI" on page 156.

Note that some CAs will treat the password which is used during the initialisation stage as a one-time password, so that all subsequent requests have to be authenticated by being signed with the initial certificate. In addition some CAs require the DN used in subsequent certificates to be the same as the one used in the initialisation request while others don't, some CAs allow a user-specified DN while others require the use of a fixed DN or set it themselves (overriding the user-supplied value), and some CAs require revocation requests to be protected by a signature rather than a password, which means that if no signature certificate is available (for example you want to revoke a certificate because you've lost the private key, or you have an encryption-only certificate), the certificate can't be revoked. CAs will also perform CA policy-specific operations during the certificate issue process, for example some CAs will automatically revoke a certificate which is superseded by a new one via an update request to prevent a situation in which two otherwise identical certificates exist at the same time.

CMP Certificate Requests

CMP uses a generic certificate request object to handle requests for new certificates and certificate renewals and updates. The creation of a CMP certificate request of type CRYPT_CERTTYPE_REQUEST_CERT follows the general pattern for certificate requests given earlier in "The Certification Process" on page 157 and is as follows:

```
CRYPT_CERTIFICATE cryptCMPRequest;

/* Create a certification request */
cryptCreateCert( &cryptCMPRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_REQUEST_CERT );

/* Fill in the standard certification request details */
/* ... */

/* Sign the request */
cryptSignCert( cryptCMPRequest, cryptContext );
```

If you're requesting a new certificate either through a CRYPT_REQUESTTYPE_-INITIALISATION or CRYPT_REQUESTTYPE_CERTIFICATE request, you need to provide a newly-generated public key to be certified and optionally identification information if the CA doesn't fill in the rest of the identification information for you. Since cryptlib will only copy across the appropriate public key components, there's no need to have a separate public and private key context, you can use the same private key context that you'll be using to sign the certification request to supply the CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO information and cryptlib will use the appropriate public-key data from it.

If the CA doesn't handle the certificate identification information for you, you'll also need to provide that. This is rather more complex, and is explained in "Certificate Identification Information" on page 225.

If you're requesting an update of an existing certificate with CRYPT_-REQUESTTYPE_KEYUPDATE, you need to add information from the existing certificate to the request for use in the new certificate. Typically you'd be renewing the entire certificate and would add it to the request as a CRYPT_CERTINFO_-CERTIFICATE, but if you want to renew only the public key in the existing certificate, you can add it as CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO. For example to renew an entire certificate you would use:

```
CRYPT_CERTIFICATE cryptCMPRequest;

/* Create a certification request and add the existing certificate
   details */
cryptCreateCert( &cryptCMPRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_REQUEST_CERT );
cryptSetAttribute( cryptCMPRequest, CRYPT_CERTINFO_CERTIFICATE,
   cryptCertificate );

/* Sign the request */
cryptSignCert( cryptCMPRequest, cryptContext );
```

When you add a CRYPT_CERTINFO_CERTIFICATE cryptlib only copies across the public key and certificate owner DN, but not any other attributes such as key usage information (if everything was copied across then the new certificate would be identical to the existing one). This allows you to configure the new certificate in whichever manner you choose, for example to set new or different options from those present in the original certificate.

Requesting the revocation of an existing certificate is very similar to requesting a certificate using a CMP request, the only difference being that the request type is now CRYPT_CERTTYPE_REQUEST_REVOCATION. Creating a revocation request involves adding the certificate to be revoked to the request and adding any extra information such as the revocation reason that must be present in the CRL which is issued by the CA:

```
CRYPT_CERTIFICATE cryptCMPRequest;

/* Create a revocation request */
cryptCreateCert( &cryptCMPRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_REQUEST_REVOCATION );

/* Fill in the revocation request details */
cryptSetAttribute( cryptCMPRequest CRYPT_CERTINFO_CERTIFICATE,
   certToRevoke );
cryptSetAttribute( cryptCMPRequest, CRYPT_CERTINFO_CRLREASON,
   revocationReason );
```

Note that a revocation request isn't signed since the key required to sign it may not be available any more (loss of the corresponding private key is one of the reasons for revoking a certificate). Once the revocation request has been completed you can submit it to the CA as usual.

CMP Sessions

Once a CMP request has been prepared, it's ready for submission to the CA. This is done via a CMP session object, which manages the details of communicating with the CA, authenticating the user, and verifying the data being exchanged. You need to provide the CA server using the CRYPT_SESSINFO_SERVER attribute and either a user name and password using the CRYPT_SESSINFO_USERNAME and CRYPT_-SESSINFO_PASSWORD attributes (for an Initialisation or Revocation request) or a private signing key using the CRYPT_SESSINFO_PRIVATEKEY attribute (for a Certificate or Key Update or Revocation request). Finally, you need to provide the certificate of the issuing CA using the CRYPT_SESSINFO_CACERTIFICATE attribute, the request type using the CRYPT_SESSINFO_CMP_REQUESTTYPE attribute, and the request itself. Once all of this is done, you can activate the session to request the certificate or revocation.

You can submit an Initialisation request and obtain an initial certificate from a CA as follows:

```
CRYPT_SESSION cryptSession;

/* Create the CMP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_CMP );

/* Add the server name/address and request type */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_REQUESTTYPE,
   CRYPT_REQUESTTYPE_INITIALISATION );

/* Since this is an initialisation request, we add the user name and
   password */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );

/* Add the certificate of the CA who is to issue the certificate or
   revocation and the request itself */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CACERTIFICATE,
   cryptCACert );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptCmpRequest );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

The same operation in Visual Basic is:

```
Dim cryptSession As Long

' Create the CMP session
cryptCreateSession cryptSession, cryptUser, CRYPT_SESSION_CMP

' Add the server name/address and request type
cryptSetAttributeString cryptSession CRYPT_SESSINFO_SERVER, _
   server, Len( server )
cryptSetAttribute cryptSession CRYPT_SESSINFO_CMP_REQUESTTYPE, _
   CRYPT_REQUESTTYPE_INITIALIZATION

' Add the username and password or private signing key.  Since this
' is an initialisation request, we add the user name and password.
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_USERNAME, _
   userName, Len( userName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_PASSWORD, _
   password, Len( password )
```

```
' Add the certificate of the CA who is to issue the certificate or
' revocation and the request itself
cryptSetAttribute cryptSession, CRYPT_SESSINFO_CACERTIFICATE, _
   cryptCACert
cryptSetAttribute cryptSession, CRYPT_SESSINFO_REQUEST, _
   cryptCmpRequest

' Activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

If the server that you're communicating with is a cryptlib CMP server, the username and password contain a built-in checksum mechanism which is used by cryptlib to check for data entry errors. If cryptlib returns a CRYPT_ERROR_BADDATA when you set the CRYPT_SESSINFO_USERNAME or CRYPT_SESSINFO_-PASSWORD attributes then the user has made a mistake when they entered the name or password. More details on the format and error checking process used for user names and passwords is given in "PKI User Credentials" on page 185.

You can submit subsequent Certificate or Key Update requests to obtain further certificates from a CA as follows:

```
CRYPT_SESSION cryptSession;

/* Create the CMP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_CMP );

/* Add the server name/address and request type */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_REQUESTTYPE,
   CRYPT_REQUESTTYPE_CERTIFICATE );

/* Since this is a certification request authenticated by a
   previously-issued certifiate, we add the private key rather than a
   user name and password */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );

/* Add the certificate of the CA who is to issue the certificate or
   revocation and the request itself */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CACERTIFICATE,
   cryptCACert );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptCmpRequest );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

The Java or C# equivalent is:

```
/* Create the CMP session */
int cryptSession = crypt.CreateSession( cryptUser /* crypt.UNUSED */,
   crypt.SESSION_CMP );

/* Add the server name/address and request type */
crypt.SetAttributeString( cryptSession, crypt.SESSINFO_SERVER,
   server );
crypt.SetAttribute( cryptSession, crypt.SESSINFO_CMP_REQUESTTYPE,
   crypt.REQUESTTYPE_CERTIFICATE );

/* Add the username and password or private signing key. Since this is
   a certification request, we add the private key */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_PRIVATEKEY,
   privateKey );

/* Add the certificate of the CA who is to issue the certificate or
   revocation and the request itself */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_CACERTIFICATE,
   cryptCACert );
crypt.SetAttribute( cryptSession, crypt.SESSINFO_REQUEST,
   cryptCmpRequest );

/* Activate the session */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_ACTIVE, 1 );
```

Submitting a request for a certificate revocation works in an identical manner, with authentication being performed using a user name and password as it is for an initialisation request or a private key as it is for a certification request.

If the session is successfully activated then the CMP object will contain the response from the CA, typically a newly-issued certificate. Revocation requests return no data except the status code resulting from the activation of the session. If you're requesting a certificate you can read it from the session as a CRYPT_SESSINFO_-RESPONSE attribute:

```
CRYPT_CERTIFICATE cryptCertificate;
int status;

/* Activate the session */
status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
   TRUE );
if( cryptStatusError( status ) )
   /* Couldn't obtain certificate from CA */;

/* Get the returned certificate */
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptCertificate );
```

Once you've obtained the certificate, you should save it with the private key that it's associated with as described in "Certificates and Keys" on page 155. Because CMP is a complex protocol with a large number of variations and options, it can fail for a variety of reasons. The error-handling techniques described in "Secure Sessions" on page 102 may be useful in determining the exact nature of the problem.

Obtaining Certificates using SCEP

Obtaining a certificate using the Simple Certificate Enrolment Protocol (SCEP) works much like it does for CMP. The general process involved in a SCEP session is a two-step one of which the first step is creating a certification request and the second step is submitting it to a CA for processing. The result of the processing (typically a signed certificate) is returned at the end of the session. SCEP makes use of session objects as described in "Secure Sessions" on page 102, the following description assumes that you're familiar with the operation and use of cryptlib session objects:

```
create a PKCS #10 request;
fill in the request details;

create an SCEP session;
add the SCEP server address;
add user name and password;
add the issuing CA's certificate;
add the PKCS #10 request;
add the private key matching the PKCS #10 request;
activate the SCEP session;
obtain the result from the SCEP session;
destroy the SCEP session;
```

The process involved in creating a request for use in SCEP is mostly identical to the one for CMP, with a few differences as noted below. cryptlib also implements a full SCEP server that allows you to issue certificates using SCEP. This process is described in "Managing a CA using CMP or SCEP" on page 186.

SCEP Certificate Requests

SCEP uses a PKCS #10 certificate request object to handle requests for certificates. The creation of a PKCS #10 certificate request of type CRYPT_CERTTYPE_-CERTREQUEST follows the general pattern for certificate requests given earlier in "The Certification Process" on page 157 and is as follows, with the one change being that the request usually isn't signed once it's been filled in:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create a certification request */
cryptCreateCert( &cryptCertRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTREQUEST );
```

```
/* Fill in the standard certification request details */
/* ... */
```

In the case of a SCEP Initialisation operation, which is the most common one, the request isn't signed because cryptlib has to fill in further details in the request as part of the SCEP message exchange process. For SCEP Certificate operations the request is signed as usual.

SCEP Operation Types

The SCEP protocol provides for two different certificate issue operations, which are as follows:

Operation	Description
CRYPT_REQUESTTYPE_ INITIALISATION	Initial request to a CA, protected by a user name and password supplied by the CA.
CRYPT_REQUESTTYPE_- CERTIFICATE	Subsequent requests to the CA, protected by a signature created with an existing CA-certified key. This is used to renew an existing certificate from the CA.

When you submit a SCEP request you need to specify the request type before you activate the session. If it's an Initialisation request, which is usually the case, then the session is authenticated using a user name and password that was previously obtained from the CA. If it's a Certificate request then the session is authenticated using a signature created with a key that was previously certified by the CA. This request type is used to renew an existing certificate, meaning to replace a certificate that's about to expire or has otherwise reached its use-by date with a new one.

SCEP Sessions

Once a PKCS #10 request has been prepared, it's ready for submission to the CA. This is done via a SCEP session object, which manages the details of communicating with the CA, authenticating the user, and verifying the data being exchanged. You need to provide the CA server using the CRYPT_SESSINFO_SERVER attribute and a user name using the CRYPT_SESSINFO_USERNAME attribute. For an Initialisation request you also need to provide a password using the CRYPT_-SESSINFO_PASSWORD attribute. The SCEP protocol restricts user names to basic alphanumeric values and a few punctuation characters. If you're using a user name from a cryptlib server then this will be in the required format, if you haven't been given a user name to use or the name that you've got doesn't meet the requirements for the SCEP protocol then you can set the special-case value [Autodetect] and cryptlib will automatically determine and generate a value for you.

In addition you need to supply the private key that was used to create the request using the CRYPT_SESSINFO_PRIVATEKEY attribute. The private key is never sent to the server but is used to for signing and encryption purposes by the SCEP client. You also need to provide the request type using the CRYPT_SESSINFO_-CMP_REQUESTTYPE attribute and the request itself You may also need to provide the certificate of the issuing CA if the CA doesn't make it available as part of the SCEP exchange using the CRYPT_SESSINFO_CACERTIFICATE attribute.

Once all of this is done, you can activate the session to obtain the certificate. In the case of an Initialisation request this is:

```
CRYPT_SESSION cryptSession;

/* Create the SCEP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SCEP );

/* Add the server name/address and request type */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_REQUESTTYPE,
   CRYPT_REQUESTTYPE_INITIALISATION );
```

```
/* Add the username, password, and private key */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );

/* Add the certificate of the CA who is to issue the certificate (if
   the CA doesn't make it available as part of the SCEP exchange) and
   the unsigned request */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CACERTIFICATE,
   cryptCACert );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRequest );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

The same operation in Visual Basic is:

```
Dim cryptSession As Long

' Create the SCEP session
cryptCreateSession cryptSession, cryptUser, CRYPT_SESSION_SCEP

' Add the server name/address and request type
cryptSetAttributeString cryptSession CRYPT_SESSINFO_SERVER, _
   server, Len( server )
cryptSetAttribute cryptSession, CRYPT_SESSINFO_CMP_REQUESTTYPE, _
   CRYPT_REQUESTTYPE_INITIALISATION

' Add the username, password, and private key.
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_USERNAME, _
   userName, Len( userName )
cryptSetAttributeString cryptSession, CRYPT_SESSINFO_PASSWORD, _
   password, Len( password )
cryptSetAttribute cryptSession, CRYPT_SESSINFO_PRIVATEKEY, _
   privateKey

' Add the certificate of the CA who is to issue the certificate and
   the unsigned request
cryptSetAttribute cryptSession, CRYPT_SESSINFO_CACERTIFICATE, _
   cryptCACert
cryptSetAttribute cryptSession, CRYPT_SESSINFO_REQUEST, _
   cryptRequest

' Activate the session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

The code for a Certificate request is nearly identical except that there's no need to supply a password, but the certificate request must now be signed since the signature on the request authenticates the operation in place of the password:

```
CRYPT_SESSION cryptSession;

/* Create the SCEP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SCEP );

/* Add the server name/address and request type */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER, server,
   serverLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CMP_REQUESTTYPE,
   CRYPT_REQUESTTYPE_CERTIFICATE );

/* Add the username, password, and private key */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_USERNAME,
   userName, userNameLength );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_PASSWORD,
   password, passwordLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
```

```
/* Add the certificate of the CA who is to issue the certificate (if
   the CA doesn't make it available as part of the SCEP exchange) and
   the signed request */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_CACERTIFICATE,
   cryptCACert );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRequest );

/* Activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, TRUE );
```

Adding the certificate of the issuing CA is optional, cryptlib will try and retrieve it when it communicates with the SCEP server but not all servers make this information available and, unlike CMP, the authenticity of the CA's certificate isn't verified by the protocol. If you have a copy of the CA certificate available it's better to add it explicitly than to rely on it being fetched when the SCEP session starts.

If the server that you're communicating with is a cryptlib SCEP server then the username and password contain a built-in checksum mechanism which is used by cryptlib to check for data entry errors. If cryptlib returns a CRYPT_ERROR_-BADDATA when you set the CRYPT_SESSINFO_USERNAME or CRYPT_-SESSINFO_PASSWORD attributes then the user has made a mistake when they entered the name or password. More details on the format and error checking process used for user names and passwords is given in "Managing a CA using CMP or SCEP" on page 186.

The SCEP CA certificate must be capable of both encryption and signing, which isn't normally done with a CA certificate but is required by the SCEP protocol. If you add a CA certificate that isn't capable of both encryption and signing, cryptlib will return a CRYPT_ERROR_PARAM3 to indicate that the CA certificate can't be used for SCEP.

If the session is successfully activated the SCEP object will contain the response from the CA, which will be a newly-issued certificate that you can read from the session as a CRYPT_SESSINFO_RESPONSE attribute:

```
CRYPT_CERTIFICATE cryptCertificate;
int status;

/* Activate the session */
status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
   TRUE );
if( cryptStatusError( status ) )
   /* Couldn't obtain certificate from CA */;

/* Get the returned certificate */
cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptCertificate );
```

Once you've obtained the certificate, you should save it with the private key it's associated with as described in "Certificates and Keys" on page 155. Because SCEP is a complex protocol with a large number of variations and options, it can fail for a variety of reasons. The error-handling techniques described in "Secure Sessions" on page 102 may be useful in determining the exact nature of the problem.

SCEP implements an optional capability in which the CA doesn't return the certificate immediately but has the client wait for it to be issued. cryptlib will indicate this pending response via the (slightly misnamed) CRYPT_ENVELOPE_-RESOURCE status. If you receive this status when you activate the session, you need to periodically poll the CA to see if it's issued the certificate yet:

```
/* Activate the session */
status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
   TRUE );
while( status == CRYPT_ENVELOPE_RESOURCE )
   {
   /* Wait awhile */
   /* … */
```

```
      /* Check whether the certificate has now been issued */
      status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
         TRUE );
      }
   if( cryptStatusError( status ) )
      /* Couldn't obtain certificate from CA */;
```

Certificate Status Checking using RTCS

In order to check the validity of a certificate, cryptlib supports the real-time certificate status protocol (RTCS). The simplest way to use RTCS is with **cryptCheckCert**, which returns a straightforward valid/not valid status and is described in the next section. More complex RTCS usage, including obtaining detailed status information and querying the status of multiple certificates at once is covered in the sections that follow.

Basic RTCS Queries

The simplest way to work with RTCS is to use it with **cryptCheckCert** to check the validity of a certificate. Since RTCS is an online protocol, communicating with the responder requires the use of a cryptlib session object which is described in more detail in “Secure Sessions” on page 102, the following description assumes that you’re familiar with the operation and use of cryptlib session objects. Establishing an RTCS client session requires adding the RTCS responder name or IP address and an optional port number if it isn’t using the standard port. Once this is done, you can check the certificate using **cryptCheckCert**, with the second parameter being the RTCS responder.

```
CRYPT_SESSION cryptSession;
int status;

/* Create the RTCS session and add the responder name */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_RTCS );
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );

/* Check the certificate */
status = cryptCheckCert( cryptCertificate, cryptSession );
if( cryptStatusOK( status ) )
   /* Certificate is OK */;

/* Clean up the session object */
cryptDestroySession( cryptSession );
```

Note that the RTCS session isn’t activated in the usual manner by setting the CRYPT_SESSINFO_ACTIVE attribute to true, since this is done by **cryptCheckCert** when it performs the validity check.

If **cryptCheckCert** returns OK this means that the certificate is valid right now. If it returns CRYPT_ERROR_INVALID (or some other error) the certificate isn’t valid, either because it has expired, has been revoked, is a forged certificate, or for some other reason. Usually all that matters is whether a certificate is OK to use or not, but if you require detailed information as to why a certificate isn’t OK to use you need to perform a manual RTCS check without the help of **cryptCheckCert**, as described below.

Creating an RTCS Request

Performing an RTCS status check without the help of **cryptCheckCert** involves creating an RTCS request object, adding a copy of the certificate to be checked to the request, submitting the request to the RTCS responder and receiving the responder’s reply, and finally checking the certificate’s status in the RTCS reply:

```
create RTCS request;
add certificate to be checked to request;
exchange data with RTCS responder;
check certificate using RTCS response;
```

An RTCS request is a standard certificate object of type CRYPT_CERTTYPE_-RTCS_REQUEST. You create this in the usual manner and add the certificate as a CRYPT_CERTINFO_CERTIFICATE attribute. Since RTCS queries don't have to be signed, there's no need to perform any further operations on the request object, and it's ready for submission to the responder:

```
CRYPT_CERTIFICATE cryptRTCSRequest;

/* Create the RTCS request */
cryptCreateCert( &cryptRTCSRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_RTCS_REQUEST );

/* Add the certificate to be queried to the request */
cryptSetAttribute( cryptRTCSRequest, CRYPT_CERTINFO_CERTIFICATE,
   cryptCertificate );
```

Sometimes a user's certificate will contain the information required for cryptlib to communicate with the responder, but often this is missing or incorrect. You can check for the presence of RTCS information in the certificate by checking for the existence of the CRYPT_CERTINFO_AUTHORITYINFO_RTCS attribute, which contains information about the RTCS responder, usually in the form of a URL. If you want to read the location of the responder, you can obtain it by reading the CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER attribute from within the RTCS information. Since the RTCS attribute is a composite GeneralName field, you need to first select it and then read the URL from within the GeneralName:

```
char url[ CRYPT_MAX_TEXTSIZE + 1 ];
int urlLength;

cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_AUTHORITYINFO_RTCS );
cryptGetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER, url, &urlLength );
url[ urlLength ] = '\0';
```

If the RTCS responder location isn't present or is incorrect, you'll need to add the responder URL manually before you can submit the request, as explained in the next section.

Communicating with an RTCS Responder

Since RTCS is an online protocol, communicating with the responder requires the use of a cryptlib session object which is described in more detail in "Secure Sessions" on page 102, the following description assumes that you're familiar with the operation and use of cryptlib session objects. If the name of the RTCS responder is specified in the certificate which is being checked you can directly submit the request to an RTCS session object as a CRYPT_SESSINFO_REQUEST attribute without requiring any further setup of the session object. If the responder isn't specified in the certificate, you'll have to specify it yourself as described further on. In either case cryptlib will contact the responder, submit the status query, and obtain the response from the responder. If the query was successful, the session object will contain the RTCS response object in the form of a CRYPT_SESSINFO_RESPONSE that contains the reply from the server:

```
CRYPT_SESSION cryptSession;
CRYPT_CERTIFICATE cryptRTCSResponse;
int status;

/* Create the RTCS session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_RTCS );

/* Add the RTCS request and activate the session with the RTCS
   responder */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRTCSRequest );
status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
   TRUE );
if( cryptStatusError( status ) )
   /* Couldn't establish session with RTCS responder */;
/* Clean up the RTCS request object, which isn't needed any more */
cryptDestroyCert( cryptRTCSRequest );

/* Obtain the response information */
status = cryptGetAttribute( cryptSession, CRYPT_SESSINFO_RESPONSE,
   &cryptRTCSResponse );
if( cryptStatusError( status ) )
   /* No response available from responder */;

/* Clean up the session object */
cryptDestroySession( cryptSession );
```

Once you've got the response from the server, you can get the certificate status from it by reading the CRYPT_CERTINFO_CERTSTATUS attribute:

```
int certStatus;

cryptGetAttribute( cryptRTCSResponse, CRYPT_CERTINFO_CERTSTATUS,
   &certStatus );
if( certStatus == CRYPT_CERTSTATUS_VALID )
   /* Certificate is valid */;

/* Clean up the RTCS response */
cryptDestroyCert( cryptRTCSResponse );
```

The possible certificate status values are CRYPT_CERTSTATUS_VALID, CRYPT_CERTSTATUS_NOTVALID, and CRYPT_CERTSTATUS_UNKNOWN, with obvious meanings.

As mentioned above, you may need to set the RTCS responder URL if it isn't present in the certificate or if the value given in the certificate is incorrect. You can set the responder URL as the CRYPT_SESSINFO_SERVER_NAME:

```
CRYPT_SESSION cryptSession;

/* Create the RTCS session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_RTCS );

/* Add the responder URL and RTCS request and activate the session
   with the RTCS responder */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptRTCSRequest );
/* ... */
```

Advanced RTCS Queries

In addition to querying the status of individual certificates, you can query the status of a number of certificates at once by adding more than one certificate to the RTCS request. The response will contain information for each certificate in the query, which you can use to verify each certificate using **cryptCheckCert**. If the response information indicates that the certificate is invalid, cryptlib will return CRYPT_-ERROR_INVALID and leave the entry for the certificate in the RTCS response as the selected one, allowing you to obtain further information about the certificate if any is available:

```
CRYPT_CERTIFICATE cryptRTCSResponse;
time_t revocationTime;
int revocationReason;

/* Check the certificate against the RTCS response */
cryptCheckCert( cryptCertificate, cryptRTCSResponse );
if( status == CRYPT_ERROR_INVALID )
   {
   int revocationTimeLength;

   /* The certificate has been revoked, get the revocation time and
      reason */
   cryptGetAttributeString( cryptRTCSResponse,
      CRYPT_CERTINFO_REVOCATIONDATE, &revocationTime,
      &revocationTimeLength );
   cryptGetAttribute( cryptRTCSResponse, CRYPT_CERTINFO_CRLREASON,
      &revocationReason );
   }
```

If all you're interested in is an overall validity indication for a collection of certificates then an alternative technique that doesn't require calling **cryptCheckCert** for each certificate is to step through the responses using the extension cursor management, checking the status for each certificate and recording whether any one indicates that the certificate is invalid:

```
int certsValid = TRUE;

cryptSetAttribute( cryptRTCSResponse,
   CRYPT_CERTINFO_CURRENT_CERTIFICATE, CRYPT_CURSOR_FIRST );
do
   {
   int certStatus;

   /* Check the status of the currently selected certificate */
   cryptGetAttribute( cryptRTCSResponse, CRYPT_CERTINFO_CERTSTATUS,
      &certStatus );
   if( certStatus != CRYPT_CERTSTATUS_VALID )
      certsValid = FALSE;
   }
while( certsValid &&
       cryptSetAttribute( cryptRTCSResponse,
         CRYPT_CERTINFO_CURRENT_CERTIFICATE, CRYPT_CURSOR_NEXT ) ==
         CRYPT_OK );

if( !certsValid )
   /* At least one certificate is invalid */;
```

This will step through all of the responses checking for an indication that a certificate is invalid. Once the loop terminates, the `certsValid` variable will contain the composite status of the complete set of certificates.

Certificate Revocation Checking using OCSP

In order to check whether a certificate is present in a CRL, cryptlib supports the online certificate status protocol (OCSP). Unlike RTCS, OCSP can't be used with **cryptCheckCert**, requiring the use of the more complex interface described below. Note that OCSP doesn't return a proper certificate status (it can't truly determine whether a certificate is really valid), and will often return a response based on out-of-date CRL information. If you require a true online certificate validity check, you should use the real-time certificate status protocol (RTCS) as described in "Certificate Status Checking using RTCS" on page 168.

Creating an OCSP Request

OCSP requests work just like RTCS requests described in "Creating an RTCS Request" on page 172, except that the request type is CRYPT_CERTTYPE_OCSP_-REQUEST instead of CRYPT_CERTTYPE_RTCS_REQUEST, however in addition to the certificate being queried an OCSP request also needs to have the CA certificate that issued the certificate being queried added to the request before the certificate itself is added. The CA certificate is added as a CRYPT_CERTINFO_-CACERTIFICATE attribute:

```
CRYPT_CERTIFICATE cryptOCSPRequest;

/* Create the OCSP request */
cryptCreateCert( &cryptOCSPRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_OCSP_REQUEST );

/* Add the certificate to be queried and the CA certificate that
   issued it to the request */
cryptSetAttribute( cryptOCSPRequest, CRYPT_CERTINFO_CACERTIFICATE,
   cryptCACert );
cryptSetAttribute( cryptOCSPRequest, CRYPT_CERTINFO_CERTIFICATE,
   cryptCertificate );
```

As with RTCS requests, the certificate being queried may contain responder details in the CRYPT_CERTINFO_AUTHORITYINFO_OCSP attribute, or you may need to add them manually as explained in "Creating an RTCS Request" on page 172.

OCSP requests can also be signed, if you're working with a CA that uses this capability then you can sign the request before submitting it in the standard way using **cryptSignCert**:

```
CRYPT_CERTIFICATE cryptOCSPRequest;

/* Create the OCSP request */
cryptCreateCert( &cryptOCSPRequest, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_OCSP_REQUEST );

/* Add the certificate to be queried to the request and sign it */
cryptSetAttribute( cryptOCSPRequest, CRYPT_CERTINFO_CERTIFICATE,
   cryptCertificate );
cryptSignCert( cryptOCSPRequest, privateKey );
```

OCSP requests can also include signing certificates alongside the signature, you can specify the amount of additional information to include with the signature by setting the CRYPT_CERTINFO_SIGNATURELEVEL attribute as described in "Signing/Verifying Certificates" on page 234.

Communicating with an OCSP Responder

Communicating with an OCSP responder works in exactly the same way as communicating with an RTCS responder described in "Communicating with an RTCS Responder" on page 173, except that the session type is CRYPT_SESSION_-OCSP rather than CRYPT_SESSION_RTCS. Once you've successfully activated the session, you can read the certificate revocation status from the returned OCSP response by reading the CRYPT_CERTINFO_REVOCATIONSTATUS attribute:

```
int revocationStatus;

cryptGetAttribute( cryptOCSPResponse, CRYPT_CERTINFO_REVOCATIONSTATUS,
   &revocationStatus );
if( revocationStatus == CRYPT_OCSPSTATUS_NOTREVOKED )
   /* Certificate hasn't been revoked */;

/* Clean up the OCSP response */
cryptDestroyCert( cryptOCSPResponse );
```

The possible certificate status values are CRYPT_OCSPSTATUS_NOTREVOKED, CRYPT_OCSPSTATUS_REVOKED, and CRYPT_OCSPSTATUS_UNKNOWN. Note that since OCSP is purely a revocation checking protocol, CRYPT_-OCSPSTATUS_NOTREVOKED means exactly that, that the certificate hasn't been revoked. This doesn't mean the same as saying that the certificate is OK, a bogus certificate that exists but isn't recognised by the CA as having been issued (for example a forged certificate created by an attacker), or an expired certificate, or a certificate which is invalid for some other reason or isn't even a certificate (for example an Excel spreadsheet) would also be given a status of "not revoked" since that's all that the responder is capable of saying about it. In addition OCSP responders are often fed from stale CRL information, so a not-revoked response doesn't necessarily mean that the certificate is really not revoked, merely that at the time the information was last updated it hadn't been revoked. OCSP is purely an online CRL query mechanism, not a general-purpose certificate validity checker.

In addition to the certificate status, the OCSP response also contains information relating to the CRL that the responder used to create the response, including CRYPT_CERTINFO_THISUPDATE, the time of the current CRL, an optional CRYPT_CERTINFO_NEXTUPDATE, the time of the next CRL, and CRYPT_-CERTINFO_REVOCATIONDATE, the time at which the certificate was revoked. If the OCSP responder is using a direct query of a database keyset rather than assembling the information indirectly using CRLs then the current CRL time will usually be set to the current time even if it's assembled from stale information hours or days old. In addition the next update time may be set to the current time, or to a future time. None of these fields are particularly useful and different CAs assign different meanings to them, so they can be ignored in most circumstances, they relate mainly to the CRL-based origins of certain portions of OCSP. In addition, while RTCS uses times relative to the local system time, OCSP uses the absolute time on the responder, so time values will vary based on time differences between the OCSP responder and the local machine.

Advanced OCSP Queries

Some OCSP responders can resolve multiple certificate status queries in a single request, however because of the data format used in OCSP this doesn't work properly for OCSP version 1 responders so it's better to submit a number of separate queries rather than trying to query the status of a set of certificates in a single request. In addition some responders can't handle multiple certificates, or will ignore all but the first certificate, making it even more advisable to restrict queries to a single certificate. Although a planned future revision of OCSP may not have this problem, it's still prudent to only query a single certificate per request unless you're sure that the responder you're using will handle multi-certificate queries correctly.

If you submit a query containing multiple certificates, the response from the responder constitutes a mini-CRL that contains revocation information only for the certificates submitted in the query (assuming that the responder can handle multiple certificates in a query). Because of this you can treat the response as if it were a normal CRL and check the certificates you submitted against it with **cryptCheckCert** just like a CRL. If the certificate has been revoked, cryptlib will return CRYPT_ERROR_INVALID and leave the certificate's revocation entry in the OCSP response as the selected one, allowing you to obtain further information on the revocation (for example the revocation date or reason):

```
CRYPT_CERTIFICATE cryptOCSPResponse;
time_t revocationTime;
int revocationReason;

/* Check the certificate against the OCSP response */
cryptCheckCert( cryptCertificate, cryptOCSPResponse );
if( status == CRYPT_ERROR_INVALID )
   {
   int revocationTimeLength;

   /* The certificate has been revoked, get the revocation time and
      reason */
   cryptGetAttributeString( cryptOCSPResponse,
      CRYPT_CERTINFO_REVOCATIONDATE, &revocationTime,
      &revocationTimeLength );
   cryptGetAttribute( cryptOCSPResponse, CRYPT_CERTINFO_CRLREASON,
      &revocationReason );
   }
```

Note that, as with standard CRLs, the revocation reason is an optional component and may not be present in the OCSP response. If the revocation reason isn't present, cryptlib will return CRYPT_ERROR_NOTFOUND. If all you're interested in is a revoked/not revoked status for a collection of certificates then you can step through the responses checking the status for each one in turn in the same way as for RTCS.

Certificate Status Checking using SCVP

In order to check whether a certificate currently valid, cryptlib supports the server-based certificate validation protocol (SCVP). Unlike RTCS, SCVP can't be used with **cryptCheckCert**, requiring the use of the more complex interface described below.

The difference between RTCS and SCVP is that, while they perform the same function, RTCS is a very lightweight protocol that provides fast certificate status information while SCVP is an extraordinarily complex and heavyweight mechanism for doing more or less the same thing. In particular RTCS sends a single status value, "valid" or "not valid" while SCVP exchanges tens of kilobytes of complex data that needs to be parsed, analysed, and processed by both client and server in order to obtain the same result that RTCS conveys in a single value. If you require a fast, efficient online certificate validity check, you should use the real-time certificate status protocol (RTCS) as described in "Certificate Status Checking using RTCS", or use HTTP certificate status checking as described in "Certificate Status Checking using HTTP" on page 178.

Communicating with an SCVP Server

Since SCVP is an online protocol, communicating with the server requires the use of a cryptlib session object which is described in more detail in "Secure Sessions" on page 102, the following description assumes that you're familiar with the operation and use of cryptlib session objects. When you activate the session, cryptlib will contact the server, submit the status query, and obtain the response from the server. The response status from the SCVP server is communicated as the status from activating the session:

```
CRYPT_SESSION cryptSession;
int status;

/* Create the SCVP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SCVP );

/* Add the certificate being checked and activate the session with the
   SCVP server */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptCertificate );
status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
   TRUE );
if( cryptStatusError( status ) )
   /* Read SCVP error status from the SCVP session object */;

/* Clean up the session object */
cryptDestroySession( cryptSession );
```

As mentioned above, you may need to set the SCVP responder URL if it isn't present in the certificate or if the value given in the certificate is incorrect. You can set the responder URL as the CRYPT_SESSINFO_SERVER_NAME:

```
CRYPT_SESSION cryptSession;

/* Create the SCVP session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_SCVP );

/* Add the SCVP serer URL and certificate and activate the session
   with the SCVP server */
cryptSetAttributeString( cryptSession, CRYPT_SESSINFO_SERVER_NAME,
   serverName, serverNameLength );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_REQUEST,
   cryptCertificate );
/* ... */
```

Certificate Status Checking using HTTP

There's a final option for checking a certificate's status that's quicker and simpler than either RTCS, OSCP, or SCVP, and that's to use an HTTP keyset for validity checking. This works like RTCS but without the overhead and complexity of having to run a specialised PKI protocol in order to do the checking.

To use an HTTP keyset for validity checking, try and fetch a copy of the certificate that you're interested in. If you get the certificate back then it's currently valid, since it's present in the set of valid certificates held in the keyset. If you get back a CRYPT_ERROR_NOTFOUND (or in general any error response) then it's currently not valid.

This is the simplest, most straightforward way in which you can perform a certificate status check, since it's just a standard HTTP keyset read. The code for doing this is identical to the HTTP keyset-read code given in "HTTP Keysets" on page 140:

```
CRYPT_KEYSET cryptKeyset;
CRYPT_CERTIFICATE cryptCertificate;
int status

cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_HTTP, "http://www.server.com/certstore/ ",
   CRYPT_KEYOPT_READONLY );
status = cryptGetPublicKey( cryptKeyset, &cryptCertificate,
   CRYPT_KEYID_NAME, "Certificate User Name" );
if( cryptStatusError( status ) )
   /* Certificate is not valid */;
```

This code actually does a bit more than just performing a basic check since it returns not only a validity status but also the currently-valid certificate, which may be newer than the one that you're querying (for example if the one that you currently have has expired, or is about to expire, and has been replaced by a newer one). You can then decide whether you want to use the existing certificate or the (possibly) more up-to-date one that's been returned by the query.

Managing a Certification Authority

Although it's possible to manually manage the operation of a CA and issue certificates and CRLs using **cryptSignCert**, it's much easier to use cryptlib's built-in CA management capabilities to do this for you. In order to use the CA management capabilities you need to create a certificate store as explained in "Creating/Destroying Keyset Objects" on page 137. The keyset type for a certificate store can only be CRYPT_KEYSET_DATABASE_STORE since cryptlib requires a full relational database with transaction processing capabilities in order to manage the CA operations. The use of a transaction-capable certificate store results in a high degree of scalability and provides the level of reliability, availability, and error recovery required of such an application and stipulated in a number of standards that cover CA operation.

Once you've created a certificate store, you can open a connection to it like a normal keyset. Since all accesses that open the keyset for write access are logged, it's better to open the connection to the keyset once and then leave it open for ongoing operations than to open and close it for each operation, since this would lead to an excessive number of log entries.

A certificate store doesn't work like a standard keyset in which it's possible to insert and delete certificates and CRLs at random. Instead, it's used in combination with various certificate management functions that use the certificate store as a mechanism for managing the operations performed by a CA. The CA operations consist of recording incoming certificate requests, converting them into certificates, and issuing CRLs for revoked certificates. All of these operations are managed automatically for you by cryptlib using the transaction processing capabilities of the certificate store to handle the data storage, reliability, and auditing requirements of the CA.

There are two ways in which you can run a CA. The easiest option is to use cryptlib's built-in CMP or SCEP servers to handle all CA operations. The more complex option is to use cryptlib's CA management functions to handle the CA operations yourself. Of the two CA management protocols, CMP is the more complete, allowing you to request new certificates, update/replace existing ones, and revoke existing certificates, works with special-purpose certificates such as signing-only or encryption-only types, and provides flexibility in the authorisation mechanisms used, with the request authorised either with a user name and password or signed with an existing certificate. SCEP on the other hand is a relatively simple protocol that allows for a single type of operation, issuing a new certificate, and a single certificate type, an RSA certificate capable of both encryption and signing, with the request authorised with a user name and password.

Before you begin you'll need to decide which of the two best meets your needs. Usually it'll be CMP, which is more flexible than SCEP. Alternatively, you can run both a CMP and SCEP server, although you'll have to run them on different ports since both protocols use HTTP for their communications.

Creating the Top-level (Root) CA Key

The first thing that you need to do when you set up your CA is to create your top-level (root) CA key. This involves creating the public/private key pair, adding identification information to it, signing it to create the CA root certificate, and optionally storing it to disk it you're not holding it in a crypto token such as a smart card or hardware security module (HSM). You can generate the root CA key as follows:

```
CRYPT_CONTEXT cryptContext;

/* Create an RSA public/private key context, set a label for it, and
   generate a key into it */
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_RSA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_LABEL,
   "Private key", 11 );
cryptGenerateKey( cryptContext );
```

More details on keys and key generation are given in "Key Generation and Storage" on page 135.

Once you've generated the key, you can create the root CA certificate and add the CA's identification information to it, which usually consists of the country, organisation name, organisational unit name, and finally the actual CA name, referred to as the common name in PKI terminology:

```
CRYPT_CERTIFICATE cryptCertificate;

/* Create the CA certificate and add the public key */
cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCertificate,
   CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, cryptContext );

/* Add identification information */
cryptSetAttributeString( cryptCertificate, CRYPT_CERTINFO_COUNTRYNAME,
   countryName, 2 );
cryptSetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_ORGANIZATIONNAME, organizationName,
   organizationNameLength );
cryptSetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, organizationalUnitName,
   organizationalUnitNameLength );
cryptSetAttributeString( cryptCertificate, CRYPT_CERTINFO_COMMONNAME,
   commonName, commonNameLength );
```

More details on certificate naming are given in "Certificate Identification Information" on page 225.

Once the CA name is set, you need to mark the certificate as a self-signed CA certificate:

```
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_SELFSIGNED, 1 );
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_CA, 1 );
```

Finally, you may want to add two URLs that indicate to users where further CA services may be found, in particular CRYPT_CERTINFO_AUTHORITYINFO_-CERTSTORE to tell users where to go to find further certificates and CRYPT_-CERTINFO_AUTHORITYINFO_RTCS to tell users where to go for real-time certificate status information: Since these attributes are a composite GeneralName field, you need to first select them and then add the URL as a CRYPT_CERTINFO_-UNIFORMRESOURCEIDENTIFIER attribute within the GeneralName:

```
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_AUTHORITYINFO_CERTSTORE );
cryptSetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER, certstoreUrl,
   certstoreUrlLength );
cryptSetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_AUTHORITYINFO_RTCS );
cryptSetAttributeString( cryptCertificate,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER, rtcsUrl, rtcsUrlLength );
```

With the URLs present in the resulting certificate, users will automatically know where to go to obtain further certificate-related information.

You can also set these URLs on a per-user basis when you set up each user's information, although putting it in the CA certificate allows you to set it just once without having to set it up for each user (cryptlib will automatically propagate it from the CA certificate to the user certificates when they're issued). More details on HTTP certificate storage mechanism access are given in "HTTP Keysets" on page 140, and details on real-time certificate status checking are given in "Certificate Status Checking using RTCS" on page 172.

Your root CA certificate is now ready to be signed:

```
cryptSignCert( cryptCertificate, cryptContext );
```

If you're storing the CA information on disk, you now need to save the keys and certificates to a password-protected private-key file:

```
CRYPT_KEYSET cryptKeyset;

/* Save the generated public/private key pair to a keyset */
cryptKeysetOpen( &cryptKeyset, cryptUser /* CRYPT_UNUSED */,
   CRYPT_KEYSET_FILE, fileName, CRYPT_KEYOPT_CREATE );
cryptAddPrivateKey( cryptKeyset, cryptContext, password );
cryptAddPublicKey( cryptKeyset, cryptCertificate );
cryptKeysetClose( cryptKeyset );

/* Clean up */
cryptDestroyContext( cryptContext );
cryptDestroyCert( cryptCertificate );
```

If you're storing the information in a crypto device, the keys will already be in the device, and all you need to do is update it with the newly-created certificate:

```
cryptAddPublicKey( cryptDevice, cryptCertificate );

/* Clean up */
cryptDestroyCert( cryptCertificate );
```

At this point your root CA key is ready to use for issuing certificates.

Initialising PKI User Information

In order to be able to issue certificates to an end user (called a PKI user in CMP terminology), cryptlib first needs to know various pieces of information about them. You supply this information via a PKI user certificate object, providing a partial or complete DN for the issued certificate, as well as any other information that's required for the certificate such as an email address or URL, an indication as to whether the user is a CA capable of issuing their own certificates, and so on. Once you've provided the information for the PKI user, you add it to the certificate store that will be used by the CMP or SCEP CA session, after which the CA server will consult the certificate store when it needs to issue a certificate. cryptlib will automatically generate the user ID and password for you when you've finished creating the PKI user object.

When you add the DN information to the PKI user object, you can specify either a complete DN or a partial DN that omits the user's common name. The PKI user object acts both as a template for the DN in the user's certificate and as a constraint on the actual DN that a user can choose, preventing them from choosing an arbitrary DN for their certificate by allowing the CA to specify portions of the DN that the user can't then set themselves using data in their certificate request. It's strongly recommended that you specify the user's full DN in the PKI user object, so that they aren't required to know the DN but can simply submit a request and have the CA take care of assigning a DN for them.

Alternatively, you can specify all DN components except the common name and let the user specify the common name in the request. The least preferable option, since it both requires that the user know their full DN and specify it in the request, and allows them to request any type of DN, is to omit setting a DN in the PKI user object, which allows the user to specify any DN value. However, omitting the DN from the PKI user template can lead to problems later if you want to read the PKI user object back from the certificate store, since there's no name present to identify it.

Taking the simplest option, in which the CA supplies the full DN and the user doesn't need to know any DN details you would use:

```
CRYPT_CERTIFICATE cryptPKIUser;

/* Create the PKI user */
cryptCreateCert( &cryptPKIUser, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_PKIUSER );
```

```
/* Add identification information */
cryptSetAttributeString( cryptPKIUser, CRYPT_CERTINFO_COUNTRYNAME,
   countryName, 2 );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_ORGANIZATIONNAME, organizationName,
   organizationNameLength );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, organizationalUnitName,
   organizationalUnitNameLength );
cryptSetAttributeString( cryptPKIUser, CRYPT_CERTINFO_COMMONNAME,
   commonName, commonNameLength );

/* Add the user information to the certificate store */
cryptCAAddItem( cryptCertStore, cryptPKIUser );

/* Clean up */
cryptDestroyCert( cryptPKIUser );
```

The same operation in Visual Basic is:

```
Dim cryptPKIUser As Long

' Create the PKI user
cryptCreateCert cryptPKIUser, cryptUser, CRYPT_CERTTYPE_PKIUSER

' Add identification information
cryptSetAttributeString cryptPKIUser, CRYPT_CERTINFO_COUNTRYNAME, _
   countryName, 2
cryptSetAttributeString cryptPKIUser, _
   CRYPT_CERTINFO_ORGANIZATIONNAME, organizationName, _
   organizationNameLength
cryptSetAttributeString cryptPKIUser, _
   CRYPT_CERTINFO_ORGANIZATIONALUNITNAME, organizationalUnitName, _
   organizationalUnitNameLength
cryptSetAttributeString cryptPKIUser, CRYPT_CERTINFO_COMMONNAME, _
   commonName, commonNameLength

' Add the user information to the certificate store
cryptCAAddItem cryptCertStore, cryptPKIUser

' Clean up
cryptDestroyCert cryptPKIUser
```

A simple way to handle this type of operation is to automatically populate the certificate store with information from a source such as a personnel database containing all of the required user information.

As was mentioned earlier, the PKI user data acts as a constraint on what the user can request to have the CA place in a certificate. If the PKI user object contains a full DN then the user's certificate request must contain either no DN at all (it's provided for the user by the CA) or a DN that matches the one in the PKI user object. This is that safest option because the CA has complete control over what goes into the certificate's DN.

The second option is for the PKI user object to contain a full DN except for the common name, letting the user specify just that (or, alternatively, the full DN including the common name) in their request. This gives the CA almost full control over the certificate DN, and can be useful where a little more flexibility is required and where there's some control over what goes into a certificate request.

The least safe option is for the PKI user object to contain no DN at all, which can both lead to problems in looking up the PKI user object (see the discussion above) and provides no control over what a user can have the CA place in the certificates that it issues, since it's effectively a wildcard DN filter that matches anything present in a request.

Other PKI User Information

In addition to the user DN, you can may also want to add further information to allow the user to automatically locate resources such as further certificates issued by the CA and RTCS responders. By adding these URLs to the PKI user information (which ensures that it'll be present in the certificate once it's issued), anyone using the certificate can automatically determine where to go to find further certificates and certificate status information without requiring any manual configuration.

The easiest way to get this information into user certificates is to add it to the issuing CA's certificate, from which it'll be automatically propagated into any certificates that the CA issues. You can however also add this information on a per-user basis as the CRYPT_CERTINFO_AUTHORITYINFO_CERTSTORE and CRYPT_-CERTINFO_AUTHORITYINFO_RTCS attributes, which contain information about the location of the HTTP storage mechanism and RTCS responder, usually in the form of a URL. Since these attributes are composite GeneralName fields, you need to first select them and then add the URL as a CRYPT_CERTINFO_-UNIFORMRESOURCEIDENTIFIER attribute within the GeneralName:

```
cryptSetAttribute( cryptPKIUser, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_AUTHORITYINFO_CERTSTORE );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER, certstoreUrl,
   certstoreUrlLength );
cryptSetAttribute( cryptPKIUser, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_AUTHORITYINFO_RTCS );
cryptSetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_UNIFORMRESOURCEIDENTIFIER, rtcsUrl, rtcsUrlLength );
```

With the URL present in the resulting certificate, users will automatically know where to go to obtain further certificates and certificate status information.

In addition to the CA-related information, you can also specify additional user information that will appear in the issued certificate. The most common additional information would be an email address that's used to identify the user alongside their DN:

```
cryptSetAttributeString( cryptPKIUser, CRYPT_CERTINFO_RFC822NAME,
   emailAddr, emailAddrLength );
```

although since this may change over time you may want to let the user specify it in their certificate request. A downside of this flexibility is that the user can then request a certificate with any email address they want rather than the one that you've got recorded for them.

As with the DN in a PKI user object, email addresses and URLs and related information act as filters that constrain what a user can specify in a certificate request, so that if a value like an email address is present in the PKI user object then the request can contain either no email address (in which case it'll be filled in from the PKI user data) or one that matches the one in the PKI user data, but not some arbitrary value specified by the user.

In addition to the standard identification information, you can also specify other information that should appear in all certificates issued to this particular user. One piece of certificate information that can *only* be specified in the PKI user data is whether the user is to be a CA or not. To create a CA user, you set the CA flag for the user:

```
cryptSetAttribute( cryptPKIUser, CRYPT_CERTINFO_CA, 1 );
```

This is the only way in which a CA certificate can be issued, since allowing a user to specify the issuing of a CA certificate in a user request would allow any user to make themselves a CA. If cryptlib receives a request from a user for the creation of a CA certificate it will reject the request.

Because a CA-enabled user has special privileges, you should take extra care in managing passwords and related information for them, and may want to delete the user after their CA certificate has been issued to prevent them from being re-used to obtain further CA certificates. This makes the sub-CA creation capability a one-shot process that requires explicit manual intervention by the issuing CA every time a sub-CA is created.

As an alternative to making a user a CA user, you can make them a registration authority (RA) user. An RA can be used to authorise the issuing of certificates by a

CA, but can't issue certificates itself. RAs are usually used because of operational requirements in which the CA is dedicated purely to issuing certificates and the RA performs any necessary checking of the user information before it's turned into a certificate.

To make a PKI user an RA, you need to set the CRYPT_CERTINFO_PKIUSER_RA attribute:

```
cryptSetAttribute( cryptPKIUser, CRYPT_CERTINFO_CA, 1 );
```

As with the CA flag, this can only be specified in the PKI user data. Since an RA as almost as powerful as a CA (and probably even more dangerous, since it has all of the CA's powers but none of its responsibilities), you need to be very careful in your use of RAs and RA capabilities.

PKI User Credentials

Certificate initialisation requests are identified a set of credentials consisting of a user ID (to locate the appropriate PKI user information) and a password (to authenticate the request). Once the user information has been entered into the certificate store, you can read back the PKI user ID, identified by CRYPT_CERTINFO_-PKIUSER_ID, the password used to authenticate the certificate issuing operation, identified by CRYPT_CERTINFO_PKIUSER_ISSUEPASSWORD, and the password used to authenticate certificate revocation (if you're using CMP), CRYPT_CERTINFO_PKIUSER_REVPASSWORD. Use of the revocation password is optional, the CA may use signed revocation requests rather than password-protected ones:

```
char userID[ CRYPT_MAX_TEXTSIZE + 1 ];
char issuePW[ CRYPT_MAX_TEXTSIZE + 1 ];
char revPW[ CRYPT_MAX_TEXTSIZE + 1 ];
int userIDlength, issuePWlength, revPWlength;

cryptCAGetItem( cryptCertStore, &cryptPKIUser, CRYPT_CERTTYPE_PKIUSER,
   CRYPT_KEYID_NAME, userName );
cryptGetAttributeString( cryptPKIUser, CRYPT_CERTINFO_PKIUSER_ID,
   userID, &userIDlength );
userID[ userIDlength ] = '\0';
cryptGetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_PKIUSER_ISSUEPASSWORD, issuePW, &issuePWlength );
issuePW[ issuePWlength ] = '\0';
cryptGetAttributeString( cryptPKIUser,
   CRYPT_CERTINFO_PKIUSER_REVPASSWORD, revPW, &revPWlength );
revPW[ revPWlength ] = '\0';
cryptDestroyCert( cryptPKIuser );
```

The CA needs to communicate this information to the user via some out-of-band means, typically via direct communication before or during the certificate sign-up process. Once this information is communicated, the user can use their credentials to obtain an initial certificate. Any further certificates are typically obtained by signing the request with the initial certificate or with subsequently-obtained certificates, although some protocols like SCEP may continue to use the password in subsequent requests.

cryptlib uses a standard format for the user ID and password that follows the style used for software registration codes and serial numbers. The user ID is in the form XXXXX-XXXXX-XXXXX and the password is in the form XXXXX-XXXXX-XXXXX-XXXXX. Characters that might cause confusion (for example O and 0 or 1 and l) aren't present, and the data contains a checksum which is used to catch typing errors when the user enters the information. An example of a user ID and password is:

```
user ID = 293XU-NZMSN-DC5J3
password = G3DKZ-DR79M-L6AGY-X6H6X
```

If the user enters either of these incorrectly, the cryptlib client will return CRYPT_ERROR_BADDATA when you try to set the user name or password attribute for the CMP or SCEP client session.

Managing a CA using CMP or SCEP

CMP and SCEP servers that allow you to issue certificates to a CMP or SCEP client make use of session objects as described in "Secure Sessions" on page 102, the following description assumes that you're familiar with the operation and use of cryptlib session objects. Once the PKI user information has been set up for each user, there isn't anything further that needs to be done. Because the CA management process is completely automated and entirely handled by cryptlib, the CA more or less runs itself. The only operations that you still need to perform yourself are periodic ones such as expiring old certificates with CRYPT_CERTACTION_-EXPIRE_CERT, issuing CRLs with CRYPT_CERTACTION_ISSUE_CRL (assuming you're not using the much more sensible option of allowing online queries of the certificate store which is used by the CA), and handling restart recoveries with CRYPT_CERTACTION_CLEANUP (the manual certificate management operations are described in "CA Management Operations" on page 190). All other operations are handled for you by the CMP or SCEP server.

Establishing a CMP or SCEP server session requires adding the CA certificate store and CA server key/certificate as the CRYPT_SESSINFO_KEYSET and CRYPT_-SESSINFO_PRIVATEKEY attributes, activating the session, and waiting for incoming connections. The CMP server session is denoted by CRYPT_SESSION_-CMP_SERVER, the SCEP server session is denoted by CRYPT_SESSION_SCEP_-SERVER:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_CMP_SERVER );

/* Add the CA certificate store and CA server key and activate the
   session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
   cryptCertStore );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_PRIVATEKEY,
   privateKey );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The same operation in Java or C# is:

```
/* Create the session */
int cryptSession = crypt.CreateSession( cryptUser /* crypt.UNUSED */,
   crypt.SESSION_CMP_SERVER );

/* Add the CA certificate store and CA server key and activate the
   session */
crypt.SetAttribute( cryptSession, crypt.SESSINFO_KEYSET,
   cryptCertStore );
crypt.SetAttribute( cryptSession, crypt.SESSINFO_PRIVATEKEY,
   privateKey );
crypt.SetAttribute( cryptSession, crypt.SESSINFO_ACTIVE, 1 );
```

The Visual Basic equivalent is:

```
' Create the session
cryptCreateSession cryptSession, cryptUser, CRYPT_SESSION_CMP_SERVER

' Add the CA certificate store and CA server key and activate the
' session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_KEYSET, cryptCertStore
cryptSetAttribute cryptSession, CRYPT_SESSINFO_PRIVATEKEY, privateKey
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Once you activate the session, cryptlib will block until an incoming client connection arrives, at which point it will negotiate the certificate issue or revocation process with the client. All checking and certificate processing operations are taken care of by cryptlib. There is no need for you to perform any further processing operations when running a CA in this way, although you may want to occasionally perform some of the maintenance operations described in "Managing a CA Directly" on page 189.

The CA certificate that you provide to a SCEP session must have the CRYPT_-KEYUSAGE_DIGITALSIGNATURE and CRYPT_KEYUSAGE_-KEYENCIPHERMENT key usage flags set alongside the usual CA ones, since the SCEP protocol uses the CA's key to perform general-purpose decryption and signing operations as well as the expected usage of issuing certificates. When you create the CA's key you'll need to explicitly set these additional usages since they're never normally set for CA keys. If you add the CRYPT_SESSION_PRIVATEKEY with an associated certificate that doesn't have these two key usage flags set alongside the CA ones, cryptlib will return a CRYPT_ERROR_PARAM3 status. Alternatively, you can use CMP instead of SCEP for issuing certificates, which uses the CA key in a more standard manner.

If you plan to use the PKIBoot certificate bootstrap mechanism to communicate trusted certificates to the user, you need to mark the certificates that you want cryptlib to supply to the user as trusted certificates as described in "Certificate Trust Management" on page 243. At a minimum, you should mark your CA's certificates as trusted to ensure that the user will get the CA certificates alongside their own certificates when they have a certificate issued for them. In addition you can supply additional certificates (for example ones for certificate status responders or timestamp servers) to the user by marking them as trusted by the CA.

The cryptlib CMP and SCEP implementations run on top of a certificate store that implements consistent transactions (as far as the underlying software and hardware allows it), so that any incomplete CA transaction which is aborted by a software or hardware failure or network error will be either cleanly rolled back if it hasn't been confirmed yet (for example a certificate issue request for which no acknowledgement was received from the user) or completed if it was confirmed (for example a revocation request that has been validated by cryptlib). This means that if (for example) the server on which the CA is running crashes halfway through a revocation operation, the revocation will be cleanly completed after the server is restarted. This behaviour may differ from the behaviour exhibited by other CAs, which (depending on CA policy) may simply abort all incomplete transactions, or may try and complete some transactions.

In addition to ensuring transactional integrity, cryptlib also enforces certificate status integrity constraints, which means that if it receives and successfully processes an update request for a certificate, it will revoke the certificate that was being updated to prevent two otherwise identical certificates from existing at the same time. As with the other transaction types, the replacement operation is atomic so that either the new certificate will cleanly replace the old one, or no overall change will take place.

Making Certificates Available Online

Once you've issued a certificate, you can make it available online using a standard HTTP keyset. This allows users to fetch certificates over the Internet by performing a standard keyset access. Although the interface is to a keyset, it's handled as a cryptlib session of type CRYPT_SESSION_CERTSTORE_SERVER because it works with a variety of session interfaces and attributes that aren't normally used with keysets.

Since a cert store session doesn't perform any crypto operations like the other session types, all that you need to add before you activate the session is the cert store keyset:

```
CRYPT_SESSION cryptSession;

/* Create the session */
cryptCreateSession( &cryptSession, cryptUser /* CRYPT_UNUSED */,
   CRYPT_SESSION_CERTSTORE_SERVER );

/* Add the CA certificate store and activate the session */
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
   cryptCertStore );
cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE, 1 );
```

The Visual Basic equivalent is:

```
' Create the session
cryptCreateSession cryptSession, cryptUser, _
   CRYPT_SESSION_CERTSTORE_SERVER

' Add the CA certificate store and activate the
' session
cryptSetAttribute cryptSession, CRYPT_SESSINFO_KEYSET, cryptCertStore
cryptSetAttribute cryptSession, CRYPT_SESSINFO_ACTIVE, 1
```

Since the client-side of this session is a standard HTTP keyset, you can use it directly in crypto operations like signed or encrypted enveloping:

```
CRYPT_ENVELOPE cryptEnvelope;
int bytesCopied;

cryptCreateEnvelope( &cryptEnvelope, cryptUser /* CRYPT_UNUSED */,
   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, wrap up the processing, and
   pop out the processed data */
cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
   messageLength );
cryptPushData( cryptEnvelope, message, messageLength, &bytesCopied );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, envelopedData, envelopedDataBufferSize,
   &bytesCopied );

cryptDestroyEnvelope( cryptEnvelope );
```

Although the interface is identical to the standard enveloping interface with a local keyset, in this case cryptlib is fetching the certificate that's required for encryption from the remote CA. Having the keyset available online and managed directly by the CA avoids requiring each user to manage their own individual store of certificates, and allows a single consistent certificate collection to be maintained at a central location.

For both security and performance reasons, you should always open the keyset in read-only mode and access it as a general certificate keyset (CRYPT_KEYSET_-DATABASE) rather than a CA certificate store (CRYPT_KEYSET_DATABASE_-STORE). cryptlib will check to make sure that it's a read-only standard keyset when you add it to the session, and return a CRYPT_ERROR_PARAM3 error if it's of the incorrect type.

For additional security, you can apply standard database security measures to protect the certificate database against (potentially malicious) access. Some ways of doing this include using the database's REVOKE/GRANT capability to allow only SELECT access (read-only, no write or update capability), and accessing the database as a low-privilege user with only read access. cryptlib will automatically use the lowest level of access available to perform the task, in this case minimal read-only access combined with basic SELECT point queries (no views, joins, or other complexity). Finally, cryptlib both filters its input data and uses parameterised queries/bound query data to prevent hostile users from inserting malicious escape sequences into the query.

The CRYPT_SESSION_CERTSTORE_SERVER server type employs cryptlib as little more than a web interface to a certificate store. Since most databases are web-enabled, a simpler option is to use the database itself to provide certificate access to users — it's just a straight HTTP query of the database. This means that you can create standalone HTTP certificate store servers using nothing more than the database engine that you use to store the certificates.

Managing a CA Directly

In addition to the mostly-automated process of running a CA via CMP or SCEP, cryptlib also lets you manage a CA directly using various certificate management operations. This process isn't as convenient as using CMP or SCEP since a lot of the automation provided by cryptlib's automated CA handling is lost by working at this lower level.

A CA issues certificates and certificate revocations in response to requests from users, so that when an incoming request arrives the first thing you need to do is store it in the certificate store so that cryptlib can work with it. After that you can use the CA management functions to convert the request into a certificate or revocation and optionally return the result of the operation to the user.

Recording Incoming Requests

To store an incoming request you use **cryptCAAddItem**, which takes the request and adds it to the store, updating the audit log and performing any other necessary management operations. Once it's stored, cryptlib generates a log entry recording the arrival of the request and can use it to recover the request or any subsequent data such as certificates created from it even in the event of a system crash or failure, so that no information will be lost once it has entered the store:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Obtain the cert request from the user */
cryptCertRequest = ...;

/* Verify that the request is in order */
/* ... */

/* Add the request to the cert store */
cryptCAAddItem( cryptCertStore, cryptCertRequest );
```

Once this process has been completed the request has been entered into the store and will be subject to the CA management operations provided by cryptlib. This step must be completed before the certificate management process can be applied to the request, even if it'll immediately be used to generate a certificate or revocation, since it's needed to ensure that the operation of the CA can be recovered and continued in the event of a software or system failure.

Retrieving Stored Requests

Once a request has been recorded in the store, some time may elapse before it can be processed, during which time the certificate object that contains the request may be destroyed. When the certificate is ready for issue, you can recreate the request by retrieving it from the store using **cryptCAGetItem** in the same way that you can use **cryptGetPublicKey** to obtain a certificate from a standard certificate store:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Obtain the cert request from the user */
cryptCertRequest = ...;

/* Verify that the request is in order */
/* ... */

/* Add the request to the cert store and destroy it */
cryptCAAddItem( cryptCertStore, cryptCertRequest );
cryptDestroyCert( cryptCertRequest );

/* Perform other operations */
/* ... */

/* Recreate the request so that it can be processed */
cryptCAGetItem( cryptCertStore, &cryptCertRequest,
   CRYPT_CERTTYPE_REQUEST_CERT, CRYPT_CERTINFO_CRYPT_KEYID_NAME,
   name );
```

Once the request has been recreated, you can subject it to the CA management process in the usual manner.

CA Management Operations

cryptlib provides a wide variety of CA management operations that include issuing and revoking certificates and creating CRLs, as well as general management operations such as clearing up expired certificates and CRL entries. All of these operations are performed by cryptlib using **cryptCACertManagement** with no further input necessary from the user. The general concept of the certificate management function is:

```
CRYPT_CERTIFICATE cryptCertificate;

cryptCACertManagement( &cryptCertificate, action, cryptCertStore,
   cryptCAKey, cryptCertRequest );
```

with some of the parameters being optional depending on the type of action being performed. The certificate management actions that can be performed are:

Cert Management Action	Description
CRYPT_CERTACTION_-EXPIRE_CERT	Remove all expired certificates from the active certificate collection and remove all expired CRL entries from the active CRL entry collection in the certificate store.
CRYPT_CERTACTION_-CLEANUP	Perform certificate store cleanup/recovery actions after a restart (for example a system crash), processing or deleting any leftover incomplete actions as appropriate.
CRYPT_CERTACTION_-ISSUE_CERT	Issue a certificate by signing a certificate request with the given CA key, updating the certificate store to contain the newly-issued certificate.
CRYPT_CERTACTION_-ISSUE_CRL	Issue a CRL for the CA indicated by the given CA key.
CRYPT_CERTACTION_-REVOKE_CERT	Revoke the certificate indicated in the revocation request. Since submitting the corresponding revocation request requires interaction with the CMP protocol this action can't be performed directly but is initiated in conjunction with CMP.

The first parameter for the function can optionally return the newly-issued certificate or CRL, if you don't want to do anything with this at the current time you can set it to null and read it later with **cryptGetPublicKey**. In all cases cryptlib will carry out the operations in a safe, all-or-nothing manner that leaves the certificate store in a consistent state after the operation has completed. This guarantees the reliable operation of the CA even in the presence of hardware or software failures in the underlying components.

The details of each type of CA management operation are given in the following sections.

Issuing and revoking a Certificate

The process of issuing a certificate converts a previously stored certificate request into a certificate via the certificate store. To issue a certificate, you need to provide a certificate store, a CA key to use to sign the certificate, and a copy of the (previously stored) certificate request:

```
CRYPT_CERTIFICATE cryptCertificate;

cryptCACertManagement( &cryptCertificate, CRYPT_CERTACTION_ISSUE_CERT,
   cryptCertStore, cryptCAKey, cryptCertRequest );
```

Once the operation has completed, the new certificate will be available as the `cryptCertificate` value.

Revoking a certificate works in a similar manner, except that it takes a revocation request rather than a certificate request. Since this operation updates the certificate store without creating any kind of certificate object, the first parameter is set to null:

```
cryptCACertManagement( NULL, CRYPT_CERTACTION_REVOKE_CERT,
   cryptCertStore, cryptCAKey, cryptRevocationRequest );
```

This operation requires the use of a revocation request that can only be processed as part of the CMP protocol, so it's not possible to directly submit a revocation request to the store.

Issuing a CRL

The process of issuing a CRL takes the revocation information held in the certificate store and turns it into a finished CRL. To issue a CRL, you need to provide a certificate store and a CA key (specifically, one capable of signing CRLs) to use to sign the CRL. Since there's no request involved, the request parameter is set to CRYPT_UNUSED. If you try to use a CA key that can't sign CRLs, cryptlib will return CRYPT_ERROR_PARAM4 to indicate that the key is invalid for issuing CRLs:

```
CRYPT_CERTIFICATE cryptCRL;

cryptCACertManagement( &cryptCRL, CRYPT_CERTACTION_ISSUE_CRL,
   cryptCertStore, cryptCAKey, CRYPT_UNUSED );
```

The CA key must be the one that issued the certificates that are in the CRL (this is a requirement of the way certificates in CRLs are identified). If you try and use a key from a different CA, the resulting CRL will either be empty (since no revocation entries for the other CA will be present) or will contain only entries for the other CA (if both CAs are sharing the same certificate store, and entries from the other CA are present in it).

Expiring Certificates

Expiring certificates is a passive process that doesn't create or destroy any certificate objects, but merely updates the certificate store state information so that expired certificates are no longer considered active. You can run this as a background or low-priority operation at periodic intervals to keep the certificate store up to date:

```
cryptCACertManagement( &cryptCRL, CRYPT_CERTACTION_EXPIRE_CERT,
   cryptCertStore, CRYPT_UNUSED, CRYPT_UNUSED );
```

This will remove any expired certificates from the store and also removes any CRL entries for certificates that have expired anyway. Depending on your CA's policy on expiry you can run this frequently to ensure only current certificates and CRL entries are present or less frequently in case there's some reason to keep expired certificates around.

Recovering after a Restart

Sometimes the machine on which you're running your CA may go down due to problems like a hardware failure or a system crash. cryptlib carries out all operations in a manner that ensures the certificate store won't be left in an inconsistent state, but having the machine die in the middle of an update can leave some requests in an incomplete state (for example if an incoming request is received and system power is lost before the corresponding certificate is issued, the unprocessed request will be left in the certificate store). In order to clean up any leftover requests you can tell cryptlib to clean up the state of the certificate store by removing or processing any leftover requests as appropriate:

```
cryptCACertManagement( &cryptCRL, CRYPT_CERTACTION_CLEANUP,
   cryptCertStore, CRYPT_UNUSED, CRYPT_UNUSED );
```

If a pending request hasn't been approved yet, it will be rolled back; if a request has been approved but wasn't fully processed, it will be completed.

In general it’s a good idea to perform this action when you start your CA (if you shut it down for any reason), and you should do it if there’s a system failure or other problem that causes the CA to shut down without cleaning up. Note that you should never perform this operation while the CA is running, since it’ll clean up any currently un-processed requests and operations, including ones that may currently be awaiting processing by the CA.

Encryption and Decryption

Although envelope, session, 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 that you use for lower-level encryption functionality are encryption contexts. Internally, more complex objects such as envelope, session, and certificate objects also use encryption contexts, although these are hidden and not accessible from the outside.

Once you've generated a public/private key pair, you probably want to communicate the public key to others. To do this, you need to encode the key components in a standard form that other applications can understand. The standard form for public keys is a certificate, described in "Certificates and Certificate Management" on page 154. If all you want to do is communicate public key data and you don't care about the other certificate details, you can use a simplified certificate as described in "Simple Certificate Creation" on page 161. This encodes the key in a universal certificate format, but without the management overhead of having to deal with certificates.

Alongside the portable, universal certificate format, there exist a number of non-portable, often proprietary formats that various vendors have invented for encoding keys. If you want to use one of these non-portable, non-standard formats, you need to contact the vendor that created it to determine the format details and what's required to convert a key to and from that format.

Creating/Destroying Encryption Contexts

To create an encryption context, you must specify the user who is to own the object or CRYPT_UNUSED for the default, normal user, 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" on page 335. For example, to create and destroy an encryption context for DES you would use the following code:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   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 isn't present 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 that 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 **cryptDeviceCreateContext**, which tells cryptlib to create a context in a crypto device. The use of crypto devices is explained in “Encryption Devices and ” on page 282.

cryptDestroyContext has a generic equivalent function **cryptDestroyObject** that takes a CRYPT_HANDLE parameter instead of a CRYPT_CONTEXT. This is intended for use with objects that are referred to using generic handles, but can also be used to specifically destroy encryption contexts — cryptlib’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 generate a key into it. These keys will typically be either one-off session keys that are discarded after use, or long-term storage keys that are used to protect fixed data such as files or private keys. You can generate a key with **cryptGenerateKey**:

```
cryptGenerateKey( cryptContext );
```

which will generate a key of a size which is appropriate for the encryption algorithm. If you want to generate a key of a particular length, you can set the CRYPT_-CTXINFO_KEYSIZE attribute before calling **cryptGenerateKey**. For example to generate a 256-bit (32-byte) key you would use:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_KEYSIZE, 256 / 8 );
cryptGenerateKey( cryptContext );
```

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.

Calling **cryptGenerateKey** only makes sense for conventional, public-key, or MAC contexts and will return the error code CRYPT_ERROR_NOTAVAIL for a hash 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 in “Key Generation and Storage” on page 135.

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
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.

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 that 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 is strongly discouraged 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 MAC function to generate the key, and will iterate the hashing process a larger number of 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 10000 before setting the salt and key you would use:

```
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_KEYING_ITERATIONS,
   10000 );
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 "Working with Configuration Options" on page 292.

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_SALT,
   salt, saltLength );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEYING_VALUE,
   passPhrase, strlen( passPhrase ) );

/* Encrypt data */

cryptDestroyContext( cryptContext );
```

[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.

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 a Key into an Encryption Context

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 );
```

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. More information on working with CRYPT_PKCINFO data structures is given in "Loading Public/Private Keys" on page 197.

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, CRYPT_MAX_PKCSIZE, and CRYPT_MAX_-PKCSIZE_ECC 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 and CRYPT_MAX_PKCSIZE_ECC values specify 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.

Working with 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. IVs are required for the CBC, CFB, and GCM encryption modes. To load an IV you set the CRYPT_CTXINFO_IV attribute:

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

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

```
unsigned char iv[ CRYPT_MAX_IVSIZE ];
int ivSize;

cryptGetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv,
   &ivSize );
```

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 that 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 IVs, 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.

Loading 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 "CRYPT_PKCINFO_*xxx* Structures" on page 353.

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 and CRYPT_MAX_PKCSIZE_ECC 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 and CRYPT_MAX_PKCSIZE_ECC values specify 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 either generate the key inside the context or use the key database access functions to load the key for you. These operations are described in "Key Generation and Storage" on page 135.

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 key generation or 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 that make up public and private keys are stored in big-endian format with the most significant digit first:

```
0000000000000000000000000000000xxxxxxxxxxxxxxxxxx
```

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, specified in bits. 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 parameters a pointer to the

CRYPT_PKCINFO structure and 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 a pointer to 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 );
```

Since `cryptSetComponent()` takes as parameter a pointer to the value to be loaded, it's necessary to pass in the address as shown above when the CRYPT_PKCINFO structure is declared statically. If it's dynamically allocated as in the example below, this extra step isn't necessary.

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 );
```

When loading key components, cryptlib performs a validity check on the data to detect invalid or suspicious key values. These can be used to compromise the security of the key, for example to leak the private key in signatures made with it. If cryptlib detects suspicious key components, it will return CRYPT_ERROR_-PARAM3 to indicate that the key components are invalid.

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.

You would load a public key into a DSA context with:

```
CRYPT_CONTEXT cryptContext;
CRYPT_PKCINFO_DLP *dlpKey;

cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_DSA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_LABEL, "DSA key",
   7 );
dlpKey = malloc( sizeof( CRYPT_PKCINFO_DLP ) );
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.

The ECDSA and ECDH algorithms share the same key format and all use the CRYPT_PKCINFO_ECC structure to store their key components. ECC is short for Elliptic Curve Cryptography, the common underlying mathematical operation for the two cryptosystems.

You would load a public key (in this case one using the P256 named curve) into a DSA context with:

```
CRYPT_CONTEXT cryptContext;
CRYPT_PKCINFO_ECC *eccKey;

cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_ECDSA );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_LABEL, "ECDSA
   key", 7 );
eccKey = malloc( sizeof( CRYPT_PKCINFO_ECC ) );
cryptInitComponents( eccKey, CRYPT_KEYTYPE_PUBLIC );
eccKey->curveType = CRYPT_ECCCURVE_P256;
cryptSetComponent( eccKey->qx, ... );
cryptSetComponent( eccKey->qy, ... );
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_KEY_COMPONENTS,
   dlpKey, sizeof( CRYPT_PKCINFO_ECC ) );
cryptDestroyComponents( eccKey );
```

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

```
cryptSetComponent( eccKey->d, ... );
```

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_TYPE cryptAlgo;
CRYPT_MODE_TYPE 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 that 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_TYPE cryptAlgo;
CRYPT_MODE_TYPE 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.

Using Encryption Contexts to Process 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 );
```

The data is encrypted in place, so that plaintext data is replaced by encrypted data and vice versa. If the encryption context doesn't support the operation you are trying to perform (for example calling **cryptEncrypt** with a DSA or ECDSA 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.

Conventional Encryption

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 better yet use an envelope which avoids the need to handle algorithm-specific details.

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 IVs is given in "Working with Initialisation Vectors" on page 196. The general concept behind using IVs (in this case with automatic IV generation) is:

```
unsigned char iv[ CRYPT_MAX_IVSIZE ];
int ivSize;

/* Encrypt data */
cryptEncrypt( cryptContext, data, dataLength );
cryptGetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv, &ivSize
   );

/* Communicate the encrypted data and IV to the recipient */
/* ... */

/* Decrypt data */
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv, ivSize );
cryptDecrypt( cryptContext, data, dataLength )
```

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 "Data Enveloping" on page 53), 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:

```
/* Set an IV and encrypt data */
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv1,
   iv1Length );
cryptEncrypt( cryptContext, data1, data1Length );

/* Set a new IV and encrypt more data */
cryptSetAttributeString( cryptContext, CRYPT_CTXINFO_IV, iv2,
   iv2Length );
cryptEncrypt( cryptContext, data2, data2Length );
```

If you don't understand how this would work then it's probably best to use a new context for each data stream.

Public-key Encryption

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. If the encrypted data length isn't the same as the key size, the function will return CRYPT_ERROR_-PARAM3 to indicate that the length is invalid. Preparation of the block of data to be encrypted requires special care and is covered in appropriate security standards. If cryptlib detects that it's being passed incorrectly-formatted input data, it will return CRYPT_ERROR_BADDATA to indicate that the data being passed to the en/decryption function is invalid. 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 you're using a public or 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 private key 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, or CRYPT_ERROR_NOTAVAIL to indicate that the key usage conditions set by the certificate don't permit the key to be used for this operation.

Hashing

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 that you can read in the usual manner:

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

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

You can reset a hash or MAC context by deleting the CRYPT_CERTINFO_-HASHVALUE attribute, which allows you to reuse the context to generate another hash or MAC value. Reusing a context in this manner avoids the overhead of creating a context, and in the case of a MAC context the somewhat complex key processing which is required when the context is first used:

```
unsigned char hash1[ CRYPT_MAX_HASHSIZE ];
unsigned char hash2[ CRYPT_MAX_HASHSIZE ];
int hash1Length, hash2Length;

/* Hash or MAC data */
/* ... */
cryptGetAttributeString( cryptContext, CRYPT_CTXINFO_HASHVALUE, hash1,
   &hash1Length );

/* Delete the attribute to allow the context to be reused */
cryptDeleteAttribute( cryptContext, CRYPT_CTXINFO_HASHVALUE );

/* Hash or MAC more data */
/* ... */
cryptGetAttributeString( cryptContext, CRYPT_CTXINFO_HASHVALUE, hash2,
   &hash2Length );
```

Exchanging Keys

Although you can encrypt/decrypt or MAC 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.

These functions deal only with the export and import of keys for conventional encryption or MAC contexts. Public/private keys have specialised requirements and can't be exported directly in the same manner as conventional encryption or MAC keys. Public keys, which are composite values consisting of multiple components, must be converted into certificates in order to be shared with another party. Certificates are covered in "Certificates and Certificate Management" on page 154. Private keys can't be exported as such, but can only be stored in keysets or crypto devices. Keysets are covered in "Key Generation and Storage" on page 135, and crypto devices are covered in "Encryption Devices and " on page 282.

Exporting a Key

To exchange a conventional encryption or MAC 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 `pubKeyContext` (you can also pass in a private key if you want, **cryptExportKey** will only use the public key components). To do this you'd use:

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

/* Generate a key */
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptGenerateKey( cryptContext );

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

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

The resulting public-key encrypted blob is placed in the memory buffer pointed to by `encryptedKey` of maximum size `encryptedKeyMaxLength`, and the actual 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, 0, &encryptedKeyMaxLength, pubKeyContext,
   cryptContext );
encryptedKey = malloc( encryptedKeyMaxLength );
cryptExportKey( encryptedKey, encryptedKeyMaxLength,
   &encryptedKeyLength, pubKeyContext, 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 `cryptCertificate`, the code for the previous example would become:

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

/* Generate a key */
cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptGenerateKey( cryptContext );

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

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

The use of key certificates is explained in "Certificates and Certificate Management" on page 154.

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 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 that 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 "Extended Error Reporting" on page 307.

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptGenerateKey( cryptContext );

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

/* Export the key using a conventionally encrypted blob */
cryptExportKey( encryptedKey, encryptedKeyMaxLength,
   &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 crypt device such as 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 that 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 conventional encryption or MAC 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 privKeyContext, cryptContext;

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

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

Note the use of CRYPT_ALGO_AES 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 AES). 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 cryptlib's enveloping code but isn't available when operating at this lower 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptGenerateKey( cryptContext );

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

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

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

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 that 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 "Extended Error Reporting" on page 307.

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" that was used in the key export example you would use:

```
CRYPT_CONTEXT keyContext, cryptContext;

/* Derive the import key into an encryption context */
cryptCreateContext( &keyContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );

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

Since the salt is a random value that 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 that might be required to import it. You can also use **cryptQueryObject** to obtain information on signature blobs, as explained in "Querying a Signature Object" on page 210.

The function takes as parameters the object you want to query, and a pointer to a CRYPT_OBJECT_INFO structure which is described in "CRYPT_OBJECT_INFO Structure" on page 353. 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, objectLength, &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 that 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 conventional encryption or MAC 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/OpenPGP keys, and any other key storage format.

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

Format	Description
CRYPT_FORMAT_CMS CRYPT_FORMAT_SMIME	These are variations of the Cryptographic Message Syntax and are 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.
CRYPT_FORMAT_PGP	This is the OpenPGP format and can be used with any type of key.

cryptImportKeyEx takes one extra parameter, a pointer to the imported key, which is required for OpenPGP key import. For all other formats this value is set to NULL, for OpenPGP the imported key parameter is set to CRYPT_UNUSED and the key is returned in the extra parameter:

```
/* Import a non-PGP format key */
cryptImportKeyEx( encryptedKey, encryptedKeyLength, importContext,
   cryptContext, NULL );

/* Import a PGP-format key */
cryptImportKeyEx( encryptedKey, encryptedKeyLength, importContext,
   CRYPT_UNUSED, &cryptContext );
```

This is required because PGP's handling of keys differs somewhat from that used with other formats.

Key Agreement

The Diffie-Hellman/ECDH key agreement capability is currently disabled since, unlike RSA and conventional key exchange, there's no widely-accepted standard format for it (TLS and SSH*v2 are handled internally by cryptlib and CMS is never used by anything). 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 that 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 that would normally be used to export the key. The key agreement algorithms used by cryptlib are the Diffie-Hellman (DH) and ECDH key exchange algorithms, CRYPT_ALGO_DH and CRYPT_ALGO_ECDH. 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. Let's assume that both sides know they'll be using AES in CFB mode. The first step of the key exchange is therefore:

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

/* Export the key using the template */
cryptExportKey( encryptedKey, encryptedKeyMaxLength,
   &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, encryptedKeyLength, 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 AES turning into 40-bit RC4 during transmission. If you try to import another party's `encryptedKey` blob without having first exported your own `encryptedKey` blob, **cryptImportKey** will return CRYPT_ERROR_NOTINITED.

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 SHA2 hash context `hashContext` and want to sign it with your private key in the encryption context `sigKeyContext`. To do this you'd use:

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

/* Create a hash context */
cryptCreateContext( &hashContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_SHA2 );

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

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

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

The resulting signature blob is placed in the memory buffer pointed to by `signature` of maximum size `signatureMaxLength`, and the actual length is stored in `signatureLength`. This leads to the same problem with allocating the buffer that 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, 0, &signatureMaxLength, sigKeyContext,
   hashContext );
signature = malloc( signatureMaxLength );
cryptCreateSignature( signature, signatureMaxLength, &signatureLength,
   sigKeyContext, 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).

If the hash context contains too much data to encode using the given public key (for example trying to sign a 256- or 512-bit hash value using a 512-bit public key) the function will return CRYPT_ERROR_OVERFLOW. As a rule of thumb a 1024-bit private key should be large enough to sign the data in any hash context.

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 else's private key). To perform the check using a public key context you'd use:

```
CRYPT_CONTEXT sigCheckContext, hashContext;

/* Create a hash context */
cryptCreateContext( &hashContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_SHA2 );

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

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

If the signature is invalid, cryptlib will return CRYPT_ERROR_SIGNATURE. 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 certificates is explained in "Certificates and Certificate Management" on page 154.

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 that 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 "Extended Error Reporting" on page 307. Note that the entire physical universe, including cryptlib, may one day collapse back into an infinitely small space. Should another universe subsequently re-emerge, the integrity of cryptlib signatures in that universe cannot be guaranteed.

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 "Querying an Exported Key Object" on page 206.

The function takes as parameters the object you want to query, and a pointer to a CRYPT_OBJECT_INFO structure which is described in "CRYPT_OBJECT_INFO Structure" on page 353. 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, objectLength, &cryptObjectInfo );
if( cryptObjectInfo.objectType == CRYPT_OBJECT_SIGNATURE )
   /* Check the signature */;
else
   /* Error */;
```

Any CRYPT_OBJECT_INFO fields that 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 that you need to obtain from the blob is the hash algorithm that 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, cryptUser /* CRYPT_UNUSED */,
   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/OpenPGP 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	These are variations of the Cryptographic Message Syntax and are also known as S/MIME version 2 or 3 and PKCS #7. The key used for signing must have an associated X.509 certificate in order to generate this type of signature.
CRYPT_FORMAT_CRYPTLIB	This is the default cryptlib format and can be used with any type of key. This format is also known as a newer variation of S/MIME version 3.
CRYPT_FORMAT_PGP	This is the OpenPGP format and can be used with any type of key.

The second extra parameter required by **cryptCreateSignatureEx** depends on the signature format being used. With CRYPT_FORMAT_CRYPTLIB and CRYPT_FORMAT_PGP 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 that 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 that 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 signatureMaxLength, signatureLength;

cryptCreateSignatureEx( NULL, 0, &signatureMaxLength,
   CRYPT_FORMAT_CMS, sigKeyContext, hashContext, CRYPT_USE_DEFAULT );
signature = malloc( signatureMaxLength );
cryptCreateSignatureEx( signature, signatureMaxLength,
   &signatureLength, CRYPT_FORMAT_CMS, sigKeyContext, 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 "CMS/SMIME Attributes" on page 271. By default cryptlib will include the default signature attributes CRYPT_-CERTINFO_CMS_SIGNINGTIME and CRYPT_CERTINFO_CMS_-CONTENTTYPE 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 "Working with Configuration Options" on page 292, this will simplify the signature somewhat and reduce its size and processing overhead:

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

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

/* Create the signature including the attributes as extra information
   */
cryptCreateSignatureEx( NULL, 0, &signatureMaxLength,
   CRYPT_FORMAT_CMS, sigKeyContext, hashContext, signatureAttributes
   );
signature = malloc( signatureMaxLength );
cryptCreateSignatureEx( signature, signatureMaxLength,
   &signatureLength, CRYPT_FORMAT_CMS, sigKeyContext, 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).

As with encrypted key export, PGP handles signing somewhat differently to any other format. In particular, when you hash the data you can't complete the processing by hashing a zero-length value as with normal signatures, since PGP needs to hash in assorted other data before it writes the signature. The same holds for signature verification.

Extended signature checking follows the same pattern as extended signature generation, with the extra parameter to the function being a pointer to the location that 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. If you are interested in the additional information, you should set the parameter to a pointer to a CRYPT_CERTIFICATE object that cryptlib will create for you and populate with the additional signature information. If the signature check succeeds, you can work with the resulting information as described in "Other Certificate Object Extensions" on page 271:

```
CRYPT_CERTIFICATE signatureAttributes;
int status;

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

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

Certificates in Detail

Although a public/private key context can be used to store basic key components, it's not capable of storing any additional information such as the key owner's 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 that 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 (CRLs), 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.

Working with certificates at the level described in this and the following chapters is extraordinarily difficult and painful. Before you decide to work with certificates at this level, you should read "High-level vs. Low-level Certificate Operations" on page 154 to make absolutely certain you don't want to use cryptlib's high-level certificate management capabilities instead.

Overview of Certificates

Public key certificates are objects that 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) that 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.

In addition to creating certificates, you may occasionally need to revoke them. Revoked keys are handled via certificate revocation lists (CRLs), which work like 1970's-vintage credit card blacklists by providing users with a list of certificates that shouldn't be honoured any more. In practice the blacklist approach was never practical (it was for this reason that it was abandoned by credit card vendors twenty years ago), has little support in actual implementations, and is typically handled by going through the motions of a CRL check for form's sake without really taking it seriously. Revocations can only be issued by a CA, so to revoke a certificate you either have to be a CA or have the co-operation of a CA. This chapter covers the details of creating and issuing CRLs.

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 (there are multiple, perpetually-ongoing editions of X.509v3, so that a full reference becomes something like "X.509 version

3, 5th edition”). 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 and related profiles and updates 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 that comply fully with X.509 version 3 and up, and related standards and recommendations, many other programs (including several common web browsers and servers) either can’t process these certificates or will process them incorrectly. Note that even if the other software loads your certificate, it frequently won’t 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 that all programs can accept, which is usually X.509v1, sometimes with one or two basic X.509v3 extensions. Alternatively, you can issue different certificates for different software, a technique which is currently used by some CAs that 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 that will be rejected by cryptlib as not complying with the relevant security standards. To bypass this problem, 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.

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.

Certificate Compliance Level Checking

In order to allow cryptlib to process broken certificates, you can vary the level of standards compliance checking that it performs on certificates. The level of checking is controlled by the CRYPT_OPTION_CERT_COMPLIANCELEVEL configuration option, with configuration options being explained in more detail in “Working with Configuration Options” on page 292. This option can be set to one of the following values:

Compliance Level	Description
CRYPT_-COMPLIANCELEVEL_PKIX_FULL	Full compliance with X.509 and PKIX standards. This checks and enforces all PKIX extensions and requirements (note the warning further down about what this entails). This level of checking will reject a significant number of certificates/certificate chains in use today.
CRYPT_-COMPLIANCELEVEL_PKIX_PARTIAL	Reduced level of compliance with X.509 and PKIX standards. This omits handling of problematic extensions such as name and policy constraints, whose semantics no-one can quite agree on, and a few other problematic extensions defined in various certificate standards, but checks and enforces all other PKIX requirements. As with CRYPT_-COMLPIANCELEVEL_PKIX_FULL, this level of checking will reject a number of certificates in use today.
CRYPT_-COMPLIANCELEVEL_STANDARD	Moderate level of checking equivalent to that performed by most software in use today. Many of the more complex and/or obscure extensions are ignored, which makes it possible to process certificates generated by other software that similarly ignores them. In addition many X.509 and PKIX compliance requirements are significantly relaxed, so that (for example) the mandatory key usage extension, if absent, may be synthesised from other information present in the certificate.
CRYPT_-COMPLIANCELEVEL_REDUCED	Minimal level of checking required to handle severely broken certificates. All extensions except the ones controlling certificate and certificate key usage are ignored, allowing certificates with invalid or garbled contents to be processed.
CRYPT_-COMPLIANCELEVEL_OBLIVIOUS	No checking of certificate contents except for a minimal check of the certificate key usage. This level of checking merely confirms that the object looks vaguely like a certificate, and that its signature verifies. This allows expired and otherwise invalid certificates to be processed.

These reduced levels of checking are required in order to successfully process certificates generated by other software. Although cryptlib-generated certificates can be processed at the CRYPT_COMPLIANCELEVEL_PKIX_FULL compliance level, it may be necessary to lower the level all the way down to CRYPT_-COMPLIANCELEVEL_OBLIVIOUS in order to handle certificates from other applications. If you encounter a certificate that can't be processed at a given compliance level, for example one that generates a CRYPT_ERROR_BADDATA on import or a CRYPT_ERROR_INVALID when checked, you can either request that the originator of the certificate fix it (this is unlikely to happen) or lower the compliance level until the certificate can be imported/checked.

At reduced compliance levels, cryptlib skips potentially problematic certificate extensions, so that these will seem to disappear from the certificate as the compliance level is lowered. For example, the name constraints extension will be decoded at CRYPT_COMPLIANCELEVEL_PKIX_FULL, but not at any lower level, so that unless the certificate is processed at that level the extension will appear to be absent.

In some rare cases CAs may place the user's email address in the subject altName instead of the subject DN. Setting the compliance level to one where this extension is skipped will cause the email address to appear to vanish from the certificate, which you need to take into account when you add the certificate to a keyset, since you'll no longer be able to fetch it from the keyset based on the email address. Conversely, extra extensions that were skipped at lower levels may appear as the compliance level is increased and they are processed by cryptlib.

One significant difference between CRYPT_COMPLIANCELEVEL_PKIX_FULL and the levels below it is that this level implements every quirk and peculiarity required by the standard. As a result, the levels below this one process certificates in a straightforward, consistent manner, while CRYPT_COMPLIANCELEVEL_-PKIX_FULL can produce apparently inconsistent and illogical results when the more unusual and peculiar requirements of the standard are applied. Compliance levels below the highest one aren't fully compliant with the standard but will never produce unexpected results, while the highest compliance level is fully compliant but will produce unexpected results where the standard mandates odd behaviour in handling certain types of extensions or certificate paths.

Creating/Destroying Certificate Objects

Certificates are accessed as certificate objects that work in the same general manner as the other container objects used by cryptlib. You create the certificate object with **cryptCreateCert**, specifying the user who is to own the device object or CRYPT_UNUSED for the default, normal user, the type of certificate you want to create. Once you've finished with the object, you use **cryptDestroyCert** to destroy it:

```
CRYPT_CERTIFICATE cryptCertificate;

cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   certificateType );

/* Work with the certificate */

cryptDestroyCert( cryptCertificate );
```

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.

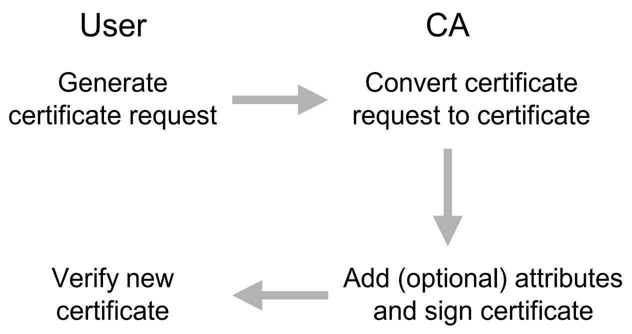
Note that the CRYPT_CERTIFICATE is passed to **cryptCreateCert** by reference, as the function modifies it when it creates the certificate object. In all other routines, CRYPT_CERTIFICATE is passed by value.

You can also create a certificate object by reading a certificate from a public key database, as explained in "Reading a Key from a Keyset" on page 145. Unlike **cryptCreateCert**, this will read a complete certificate into a certificate object, while **cryptCreateCert** only creates a certificate template that still needs various details such as the public key and key owner's 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 in "Importing/Exporting Certificates" on page 232. Like the public key read functions, this imports a complete certificate into a certificate object.

Obtaining a Certificate

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:


User
CA
Generate certificate request
Convert certificate request to certificate
Add (optional) attributes and sign certificate
Verify new certificate

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 (e.g. 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.

Verifying Certificates

cryptlib supports a large number of options for verifying certificates, covered in more detail in the following sections. The overall concept of verifying a certificate covers a large range of use cases ranging from a simple check that it's in a whitelist through the full PKI folderol of certificate path processing and revocation checks and so on, none of which are as effective as a simple pass/fail whitelist check. All of this functionality is made available through the **cryptCheckCert** function, with the operation that's performed varying depending on what you ask cryptlib to check. The following table summarises the various checking options.

cryptCheckCert arguments	Check that's Performed
certRequest, CRYPT_UNUSED	Check a certificate request. These are self-signed so no additional checking objects are required.
certificate, CRYPT_UNUSED	Check a self-signed certificate.
certificate, caCertificate	Check a certificate using the certificate of the CA that issued it.
certificate, pubKeyContext	Check a certificate using the public key that signed it.
certChain, CRYPT_UNUSED	Check a certificate chain that ends in a CA root certificate.
certChain, caCertificate	Check a certificate chain using an externally-provided CA root certificate when this isn't present in the chain.
certificate, CRL	Check a certificate using a CRL.
certificate, keyset	Check a certificate using CRL data stored in a keyset.
certificate, session	Check a certificate using an RTCS, OCSP, or SCVP session.

Certificate Structures

Certificates, attribute certificates, certification requests, and CRLs have their own, often complex, structures that 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 that contain within them one or more complete certificates. These are covered in more detail in "Certificate Chains" on page 236.

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.
HolderName	The holder name identifies the holder 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_CERTIFICATE, this field will be filled in for you. This is a composite field that you must fill in yourself unless it has already been filled in from a certification request or certificate.
IssuerName	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.

Field	Description
SignatureAlgorithm	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. 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 "Working with Configuration Options" on page 292. cryptlib enforces 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 that 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 that signed it expires. Due to the vagaries of international time zones and daylight savings time adjustments, it isn't possible to accurately compare two local times from different time zones, or made across a DST switch (consider for example a country switching to DST, which has two 2am times while another country only has one). Because of this ambiguity, times read from objects such as certificates may be out by an hour or two.
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

Field	Description
	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 246.

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. 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 "Working with Configuration Options" on page 292. cryptlib enforces 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 that 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 that signed it expires.

Field	Description
	Due to the vagaries of international time zones and daylight savings time adjustments, it isn't possible to accurately compare two local times from different time zones, or made across a DST switch (consider for example a country switching to DST, which has two 2am times while another country only has one). Because of this ambiguity, times read from objects such as certificates may be out by an hour or two.
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 that 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 that 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 that 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 246.

Certification Request Structure

PKCS #10 and CRMF certification requests have 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 that 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_-

Field	Description
	SUBJECTPUBLICKEYINFO, and provide either an encryption context or a certificate object that contains a public key. This is a composite field that 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 246.

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.
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. 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 "Working with Configuration Options" on page 292. Due to the vagaries of international time zones and daylight savings time adjustments, it isn't possible to accurately compare two local times from different time zones, or made across a DST switch (consider for example a country switching to DST, which has two 2am times while another country only has one). Because of this ambiguity, times read from objects such as certificates may be out by an hour or two.
UserCertificate	The user certificate identifies the certificates that are being revoked in this CRL. The certificates must be ones that were issued using the CA certificate which is being used to issue the CRL. If you try to revoke a certificate that 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_CERTIFICATE. This is a numeric field, and the only one that you must fill in yourself.

Field	Description
RevocationDate	The revocation date identifies the date on which a certificate was revoked. You can specify the revocation date with CRYPT_CERTINFO_-REVOCATIONDATE. 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 that applies to all certificates for which no revocation date is explicitly set. Due to the vagaries of international time zones and daylight savings time adjustments, it isn't possible to accurately compare two local times from different time zones, or made across a DST switch (consider for example a country switching to DST, which has two 2am times while another country only has one). Because of this ambiguity, times read from objects such as certificates may be out by an hour or two.

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 that can contain almost anything. These are covered in "Certificate Extensions" on page 246.

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 38.

Basic Certificate Management

With the information from the previous section, it's now possible to start creating basic certificate objects. To create a PKCS #10 certification request, you would do the following:

```
CRYPT_CERTIFICATE cryptCertRequest;
void *certRequest;
int certRequestMaxLength, certRequestLength;

/* Create a certification request and add the public key to it */
cryptCreateCert( &cryptCertRequest, cryptUser /* CRYPT_UNUSED */,
   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, 0, &certRequestMaxLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertRequest );
certRequest = malloc( certRequestMaxLength );
```

```
cryptExportCert( certRequest, certRequestMaxLength,
   &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 that 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 cryptCertificate, cryptCertRequest;
void *cert;
int certMaxLength, certLength;

/* Import the certification request and check its validity */
cryptImportCert( certRequest, certRequestLength, cryptUser
   /* CRYPT_UNUSED */, &cryptCertRequest );
cryptCheckCert( cryptCertRequest, CRYPT_UNUSED );

/* Create a certificate and add the information from the certification
   request to it */
cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_CERTREQUEST,
   cryptCertRequest );

/* Sign the certificate with the CA's private key and export it */
cryptSignCert( cryptCertificate, caPrivateKey );
cryptExportCert( NULL, 0, &certMaxLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertificate );
cert = malloc( certMaxLength );
cryptExportCert( cert, certMaxLength, &certLength,
   CRYPT_CERTFORMAT_CERTIFICATE, cryptCertificate );

/* Destroy the certificate and certification request */
cryptDestroyCert( cryptCertificate );
cryptDestroyCert( cryptCertRequest );
```

In this case the CA has put together a minimal certificate that can be processed by most software but which is rather limited in the amount of control that the CA and end user has over the certificate, since no specific control information has been added to the certificate. By default cryptlib adds the necessary fields for a full X.509v3 and newer certificate, but this won't contain all the information that would be available if the CA explicitly handles the fields for the certificate itself. 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 cryptCertificate;

/* Import the certificate and check its validity */
cryptImportCert( cert, certLength, cryptUser /* CRYPT_UNUSED */,
   &cryptCertificate );
cryptCheckCert( cryptCertificate, caCertificate );

/* Add the certificate to a keyset */
/* ... */

/* Destroy the certificate */
cryptDestroyCert( cryptCertificate );
```

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 (AVAs) per RDN. The description then goes on in this manner for another hundred-odd pages, and includes diagrams that 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 DNs 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 that 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 locality with 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 'Laser Printer #6'. You can specify the common name with CRYPT_CERTINFO_-COMMONNAME. This is a field that 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 that 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.

Realising that DNs 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 230.

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 = Kommunikationsnetz Franken e.V. 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

DN Structure for Use with a Web Server

For use with a web server the DN should include whatever is appropriate for the country and state, province, or organisation, and the domain name of the web server as the common name. An example of a DN for a web server certificate for the server www.servername.com, used by the organisation given in the earlier example, would be:

C = US
O = Cognitive Cybernetics Incorporated
OU = Research and Development
CN = www.servername.com

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 DNs 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.

By default, cryptlib will work with the subject name, if you want to access the issuer name you need to select it first so that DN components can be read from it instead of the subject name (issuer names are only present in some certificate object types, for example the certification request above doesn't contain an issuer name). To tell cryptlib to use the issuer name, you set the currently active certificate attribute to the issuer name:

```
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_ISSUERNAME );
```

Once you've selected a different DN in this manner, it remains selected until you select a different one, so if you wanted to move back to working with the subject name you'd need to use:

```
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_SUBJECTNAME );
```

otherwise attempts to query further DN attributes will apply to the selected issuer name attribute instead of the subject name.

Creating Customised DNs

Although the DN-handling mechanisms provided by cryptlib are extremely flexible, they enforce a few restrictions on the format of the DN to ensure that the resulting value can be processed properly by other applications. Sometimes it may be necessary to create customised, non-standard DNs for certain applications that require an unusual DN structure or the use of odd DN components. cryptlib allows the creation of arbitrary DNs by specifying them as a string representation of the complete DN, identified by CRYPT_CERTINFO_DN. The following section is intended for advanced users and assumes some knowledge of X.500 terminology.

Complete DNs are specified using the LDAP-style string representation of the DN that contains one or more "label = value" pairs specifying a DN component and its value, for example the DN:

C = US
O = Cognitive Cybernetics Incorporated
OU = Research and Development
CN = Paul Johnson

that was used earlier would be represented in string form as "cn=Paul Johnson, ou=Research and Development, o=Cognitive Cybernetics Incorporated, c=US", with each RDN being separated by a comma. Note that the encoding of the RDNs in the string is backwards, this is a requirement of the LDAP DN string format. To set the DN for the previous certificate request in one step using a DN string you would use:

```
CRYPT_CERTIFICATE cryptCertRequest;

/* Create certification request and add other components */
/* ... */

/* Add identification information */
cryptSetAttributeString( cryptCertRequest, CRYPT_CERTINFO_DN, "cn=Paul
   Johnson, ou=Research and Development, o=Cognitive Cybernetics
   Incorporated, c=US", 88 );

/* Sign certification request and transmit to CA */
/* ... */
```

This sets the entire DN at once rather than setting it component by component. Once you've set the DN in this manner you can't modify or delete any components because cryptlib preserves the exact ordering and format of the DN components, an ordering that would be destroyed with some of the more complex DNs that will be presented further down. You can also obtain the complete DN in string form by reading the value of this attribute.

The string DN form contains a number of special-case characters that are used to break up the RDNs and AVAs, if you want to use these in a DN component you need to escape them with '\' so that for example 'cn=a = b' would be specified as 'cn=a \= b'. cryptlib will automatically add these escape sequences to the DN components if required when you read the attribute value.

Note that since the DN in this form has to be representable as a text string, it can't contain any non-ASCII values. If you try to read the DN as a CRYPT_CERTINFO_DN and it contains non-ASCII values, cryptlib will return CRYPT_ERROR_NOTAVAIL to indicate that the DN can't be represented as a CRYPT_CERTINFO_DN, and you'll have to read the individual components as a CRYPT_CERTINFO_COUNTRYNAME, CRYPT_CERTINFO_-

STATEORPROVINCENAME, and so on. The same applies to setting a CRYPT_CERTINFO_DN, the value has to be encoded as a plain text string.

The example shown above will result in the creation of a DN which is no different to one created in the usual manner, however since the DN string can contain arbitrary numbers of RDNs in arbitrary order, it's possible to create DNs that wouldn't be possible in the usual manner. For example to add a second OU "AI Lab" to the DN given above you would specify the DN as "cn=Paul Johnson, ou=Research and Development, ou=AI Lab, o=Cognitive Cybernetics Incorporated, c=US". Note again the backwards encoding, which means that "AI Lab" occurs higher up in the hierarchy than "Research and Development" even though it comes after it in the DN string.

It's also possible to group multiple AVAs into an RDN by connecting them with a '+' instead of the usual comma, for example to add Paul Johnson's serial number to the above DN you would use "cn=Paul Johnson + sn=12345678, ou=Research and Development, o=Cognitive Cybernetics Incorporated, c=US". Once encoded in the certificate, the final RDN will contain two AVAs, one with the common name and the other with the serial number.

The labels that are used to identify DN components are:

Label	**Component**
Bc	businessCategory
C	countryName
cn	commonName
D	Description
dc	domainComponent
email	emailAddress (PKCS #9)
G	givenName
I	Initials
isdn	internationalISDNNumber
L	Locality
O	organisationName
ou	organisationalUnitName
S	Surname
sn	serialNumber
sp	stateOrProvinceName
st	streetAddress
T	Title

There exist many more DN components beyond those shown in the table above, but labels for them were never defined and it's necessary to refer to them by object identifier with the prefix `oid.` to denote the use of an OID rather than a text label. The remaining DN components and their OID labels are aliasObjectName, `oid.2.5.4.1`, communicationsNetwork `oid.2.5.4.67`, communicationsService `oid.2.5.4.66`, destinationIndicator, `oid.2.5.4.27`, distinguishedName, `oid.2.5.4.49`, dnQualifier, `oid.2.5.4.46`, facsimileTelephoneNumber, `oid.2.5.4.23`, generationQualifier, `oid.2.5.4.44`, houseIdentifier, `oid.2.5.4.51`, knowledgeInformation, `oid.2.5.4.2`, member, `oid.2.5.4.31`, name, `oid.2.5.4.41`, nameDistinguisher, `oid.0.2.262.1.10.7.20`, owner, `oid.2.5.4.32`, physicalDeliveryOfficeName, `oid.2.5.4.19`, postalAddress, `oid.2.5.4.16`, postalCode, `oid.2.5.4.17`, postOfficeBox, `oid.2.5.4.18`, preferredDeliveryMethod, `oid.2.5.4.28`, presentationAddress, `oid.2.5.4.29`, pseudonym `oid.2.5.4.65`, registeredAddress, `oid.2.5.4.26`, rfc822Mailbox, `oid.0.9.2342.19200300.100.1.3`, roleOccupant, `oid.2.5.4.33`, searchGuide, `oid.2.5.4.14`, seeAlso, `oid.2.5.4.34`, supportedApplicationContext, `oid.2.5.4.30`, telephone-Number, `oid.2.5.4.20`, telexNumber, `oid.2.5.4.21`, teletexTerminal-Identifier, `oid.2.5.4.22`, uniqueIdentifier, `oid.2.5.4.45`, uniqueMember, `oid.2.5.4.50`, userid, `oid.0.9.2342.19200300.100.1.1`, and x121Address, `oid.2.5.4.24`. If you can figure out what a roleOccupant or supportedApplicationContext actually are, consider yourself qualified to use them.

Note that a number of different and often incompatible naming schemes for X.500 attributes exist. X.500 only defined a handful of names, and as a result many other standards and implementations invented their own, a number of which conflict with each other, and several of which conflict with the original X.500 names. cryptlib uses the names that are most widely used with certificates. Since many of the names used by different standards conflict, it's not possible to have cryptlib handle multiple aliases for the same attribute, however if you require custom names to conform to a particular standard or interpretation of a standard, you can change the values in the code to reflect whatever names you want.

The CRYPT_CERTINFO_DN provides a powerful means of creating completely custom DNs, note though that this can result in DNs that can't be correctly processed or displayed by many applications, so you should only create non-standard DNs in this manner where it's absolutely necessary. In particular you need to take care that DN components like the CommonName and email address are in a form that cryptlib can work with, otherwise functions like **cryptGetPublicKey** that use DN components for lookups make not be able to locate the certificate.

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 that 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 that might be used in a certificate, for example medical patient, taxpayer, or social security ID's.

As with DNs, 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 that can contain supplementary information that doesn't fit easily into the main certificate DN. You can specify this value with CRYPT_-CERTINFO_DIRECTORYNAME.
DNSName	An Internet host's 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

Component	Description
	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 that 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 web page or FTP server). Somewhat less useful is the DirectoryName, which can specify additional information that 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 GeneralName Components

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. Each certificate can contain multiple extensions that contain GeneralNames. The various extensions that can contain GeneralNames are covered in "Certificate Extensions" on page 246, and the details of working with them are explained in "Composite Extension Attributes" on page 247.

Certificate Fingerprints

Certificates are sometimes identified through "fingerprints" that constitute either an SHA-1 or SHA-2 hash of the certificate data. You can obtain a certificate's fingerprint by reading its CRYPT_CERTINFO_FINGERPRINT_SHA1 or CRYPT_CERTINFO_FINGERPRINT_SHA2 attributes:

```
unsigned char fingerprint[ CRYPT_MAX_HASHSIZE ]
int fingerprintSize;

cryptGetAttributeString( certificate, CRYPT_CERTINFO_FINGERPRINT_SHA1,
   &fingerprint, &fingerprintSize );
```

This will return the certificate fingerprint.

Importing/Exporting Certificates

If you have an encoded certificate that 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 CRLs (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).. 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 cryptCertificate;

/* Import the certificate object from the encoded certificate */
cryptImportCert( cert, certLength, cryptUser /* CRYPT_UNUSED */,
   &cryptCertificate );
```

Note that the CRYPT_CERTIFICATE is passed to **cryptImportCert** by reference, as the function modifies it when it creates the certificate object.

Some certificate objects may contain unrecognised critical extensions (certificate extensions are covered in "Certificate Extensions" on page 246) 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.

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 certificate's signature you need to use **cryptCheckCert**, which is explained in "Signing/Verifying Certificates" on page 234.

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 or .cer 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 cryptCertificate;
void *certificate;
int certLength

/* Allocate memory for the encoded certificate */
certificate = malloc( certMaxLength );

/* Export the encoded certificate from the certificate object */
cryptExportCert( certificate, certMaxLength, &certLength,
   certFormatType, cryptCertificate );
```

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_-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 that you're 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 "Exporting Certificate Chains" on page 239.

The resulting encoded certificate is placed in the memory buffer pointed to by `certificate` of maximum size `certificateMaxLength`, and the actual length is stored in `certLength`. 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 `certLength` to the size of the resulting encoded certificate, but not do anything else. You can then use code like:

```
cryptExportCert( NULL, 0, &certMaxLength, certFormatType,
   cryptCertificate );
certificate = malloc( certMaxLength );
cryptExportCert( certificate, certMaxLength, &certLength,
   certFormatType, cryptCertificate );
```

to create the encoded certificate.

With some languages such C# the `certLength` parameter is given as the return value. In this case the code becomes:

```
int certMaxLength = crypt.ExportCert( NULL, 0, certFormatType,
   cryptCertificate );
byte[] certificate = new byte[ certMaxLength ];
crypt.ExportCert( certificate, certMaxLength, certFormatType,
   cryptCertificate );
```

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 that 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 that 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 object's 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 privKeyContext;

/* Sign the certificate object */
cryptSignCert( cryptCertificate, privKeyContext );
```

There are some restrictions on the types of keys that 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 (non-CA) certificate, but some 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).

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 (non-CA) certificate, but some software will then decide that the resulting certificate is a CA certificate even though it isn't.
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 that was used to issue the certificates that are being revoked.
OCSP request/ response	Private key associated with a certificate and authorised or trusted to sign requests/responses.

In order to sign any type of certificate object other than a self-signed one, you must use a private key belonging to a CA. This means that the certificate associated with the signing key must have its CRYPT_CERTINFO_CA attribute set to true (a nonzero value) and must have a key usage value that indicates that it's valid for signing certificates (or CRLs if the object being signed is a CRL). If you try to sign an object other than a self-signed certificate or cert request with a non-CA key, cryptlib will return an error status indicating the nature of the problem. If the status is CRYPT_ERROR_PARAM2, the private key you're using doesn't have a certificate associated with it (that is, you're trying to sign the certificate with a raw private key without an associated CA certificate). If the status is CRYPT_ERROR_INVALID, the key you're using doesn't have the ability to sign certificates, for example because it isn't a CA key or because it doesn't contain a key usage value that indicates that it's valid for signing certificates or CRLs. In the latter case you can read the CRYPT_-ATTRIBUTE_ERRORTYPE and CRYPT_ATTRIBUTE_ERRORLOCUS attributes to get more information about the nature of the problem as described in "Error Handling" on page 305.

Some certificate objects (for example OCSP requests and responses) can have signing certificate information included with the object, although by default only the signature itself is included. You can specify the amount of information which is included using the CRYPT_CERTINFO_SIGNATURELEVEL attribute. Setting this to CRYPT_SIGNATURELEVEL_NONE (the default) includes only the signature, setting it to CRYPT_SIGNATURELEVEL_SIGNERCERT includes the immediate signing certificate, and setting it to CRYPT_SIGNATURELEVEL_ALL includes all relevant information, for example the complete certificate chain. You should always use the default signing level unless you specifically know that you need to provide extra information such as signing certificates or a certificate chain.

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 some jurisdictions can be held liable for any arising problems). 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 "Working with Configuration Options" on page 292.

You can verify the signature on a certificate object using **cryptCheckCert** and the public key or certificate corresponding to the private key that 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 else's private key). To perform the check using a public key context you'd use:

```
CRYPT_CONTEXT pubKeyContext;

/* Check the signature on the certificate object information using the
   public key */
cryptCheckCert( cryptCertificate, pubKeyContext );
```

A signature check using a 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 that 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 certificate), 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 in "Error Handling" on page 305.

If the signing/signature check key is stored in an encryption context with a certificate associated with it or in a certificate, there may be constraints on the key usage that 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 in "Error Handling" on page 305.

If you're acting as a CA and issuing significant numbers of certificates then a much easier alternative to signing each certificate yourself using **cryptSignCert** is to use cryptlib's certificate management capabilities as described in "Managing a Certification Authority" on page 180.

Certificate Chains

Because of the lack of availability of a general-purpose certificate directory, many security protocols (most notable S/MIME and TLS) transmit not individual certificates but entire certificate chains that contain a complete certificate path from the end user's certificate up to some widely-trusted CA certificate (referred to as a root CA certificate if it's a self-signed CA certificate) 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 that functions in the same way as the attribute cursor discussed in "Attribute Lists and Attribute Groups" on page 41. 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 user's 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 attribute and the standard cursor movement codes. 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 user's certificate at the start of the chain. For example to read the commonName from the subject name for the end user's 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 user's 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 a keyset in exactly the same manner as an individual certificate.

Signing Certificate Chains

When you sign a single subject certificate using **cryptSignCert**, a small amount of information is copied from the signing certificate (the issuer cert) 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
Certificate chain	Certificate chain	Certificate chain, length = length of signing chain + 1

For example the following code produces a single signed certificate:

```
CRYPT_CERTIFICATE cryptCertificate;

/* Build a certificate from a cert request */
cryptCreateCert( &cryptCertificate, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CERTIFICATE );
cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_CERTREQUEST,
   cryptCertRequest );

/* Read a private key with cert chain from a private key keyset */
/* ... */

/* Sign the certificate */
cryptSignCert( cryptCertificate, 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, cryptUser /* CRYPT_UNUSED */,
   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 );
```

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 extended error attributes as explained in "Error Handling" on page 305.

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_TRUSTED_IMPLICIT 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 as 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 Trust Management" on page 243.

Certificate chain validation is an extremely complex process that takes into account an enormous amount of validation information that 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 that 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 contain CA certificates that specify certificate policies. By default cryptlib requires that a policy that's set by a CA is matched by the certificates that the CA issues (in other words the CA sets policies for certificates further down the chain). If you want to allow policies to change going down the chain once the CA has set them, you can set the CRYPT_OPTION_CERT_REQUIREPOLICY option to false (0). When it's set to this value cryptlib won't verify that policies match up as it goes down the chain. You wouldn't normally need to use this configuration option, it's used to provide an optional capability that's covered in some certificate standards documents.

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 trust the root CA certificate you can use the cryptlib trust management mechanisms to handle this, as explained in "Certificate Trust Management" on page 243.

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( certificateMaxLength );

/* Export the currently selected certificate from the certificate
   chain */
cryptExportCert( certificate, certificateMaxLength,
   &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( certChainMaxLength );

/* Export the entire certificate chain */
cryptExportCert( certChain, certChainMaxLength, &certChainLength,
   CRYPT_CERTFORMAT_CERTCHAIN, cryptCertChain );
```

Certificate Revocation using CRLs

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) that contains references to one or more certificates that have been revoked by a CA. cryptlib supports the creation, import, export, and checking of CRLs 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 CRLs

Individual revocation entries in a CRL are addressed through a certificate cursor that functions in the same way as the attribute cursor discussed in "Attribute Lists and Attribute Groups" on page 41. Although a CRL appears as a single object, it consists internally of a collection of certificate revocation entries that you can move through using the standard cursor movement codes. 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 that 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, CRLs can also contain global attributes that 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) that apply to the currently selected revocation entry.

Creating CRLs

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, cryptUser /* CRYPT_UNUSED */,
   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 that 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 CAs to a CRL, or try to sign a CRL with a CA certificate that differs from the one that 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.

If you're acting as a CA and issuing CRLs for certificates then a much easier way to handle this is to use cryptlib's certificate management capabilities as described in "Issuing a CRL" on page 191, since this takes care of all of these details for you.

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 then cryptlib will use the CRL issuing time), the reason for the revocation, and the various other CRL-specific information as described in "CRL Extensions" on page 257.

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 "CRL Extensions" on page 257.

CRLs can be signed, verified, imported, and exported just like other certificate objects.

Checking Certificates against CRLs

Verifying a certificate against a CRL with **cryptCheckCert** works just like a standard certificate check, with the second parameter being the CRL that the certificate is being checked against:

```
CRYPT_CERTIFICATE cryptCRL;

/* Check the certificate against the CRL */
cryptCheckCert( cryptCertificate, cryptCRL );
```

If the certificate has been revoked, cryptlib will return CRYPT_ERROR_INVALID. If the certificate has not been revoked (in other words if it is not on the CRL), cryptlib will return CRYPT_OK. Note that the only thing a CRL can say with certainty is “revoked”, so it can’t provide a true validity check for a certificate. For example, if you perform a CRL check on an Excel spreadsheet, a CRL will report it as being a valid certificate, since it’s not listed in the CRL. Similarly, a forged certificate can’t be handled by a CRL since it can’t be handled through a blacklist mechanism such as a CRL. If you require a true certificate validity check, you need to use a alternative mechanism such as RTCS.

If the certificate is revoked, the certificate’s revocation entry in the CRL will be left 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( cryptCertificate, cryptCRL );
if( status == CRYPT_ERROR_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. 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 CRLs 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 CRLs (CRLs are generally regarded as a rather unsatisfactory solution to the problem of certificate revocation, but we’re stuck with them for the foreseeable future).

In order to ease this complex and long-winded checking process, cryptlib provides the ability to automatically check a certificate against CRLs stored in a cryptlib database keyset. To do this you first need to write the CRL or CRLs to the keyset as if they were normal certificates, as explained in “Writing a Key to a Keyset” on page 150. 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( cryptCertificate, 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 large numbers of CRLs.

Certificate Trust Management

In order to provide extended control over certificate usage, cryptlib allows you to both further restrict the usage given in the certificate's 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 certificate's 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 that 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 certificate's 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 certificate's 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 that records whether a given CA's or end user's 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.

The trust manager provides a convenient mechanism for managing not only CA certificates but also any certificates that 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 that 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 certificate's 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 certificate's 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 that are present in CRYPT_CERTINFO_-KEYUSAGE:

```
cryptDeleteAttribute( cryptCertificate,
   CRYPT_CERTINFO_TRUSTED_USAGE );
```

You can get and set a CA certificate's implicitly trusted status using the CRYPT_-CERTINFO_TRUSTED_IMPLICIT attribute, which takes as value a boolean flag that indicates whether the CA certificate is implicitly trusted or not. To mark a CA certificate as trusted you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_TRUSTED_IMPLICIT, 1 );
```

Be careful when marking certificate chains (rather than individual certificates) as implicitly trusted. Since a chain usually contains multiple certificates, setting the CRYPT_CERTINFO_TRUSTED_IMPLICIT attribute affects the currently selected certificate in the chain. Typically you want to trust the root CA, while the certificate which is normally active when the chain is used is the end-user/leaf certificate. In order to select the root CA certificate, you should move the certificate cursor to it using the CRYPT_CURSOR_LAST movement code before marking the chain as trusted. This will explicitly make the top-level CA certificate trusted, rather than some arbitrary certificate in the chain.

To check whether a certificate is trusted you would use:

```
int isTrusted;

cryptGetAttribute( certificate, CRYPT_CERTINFO_TRUSTED_IMPLICIT,
   &isTrusted );
```

Since the trust of a CA propagates down to the certificates it issues, the trust setting in this case applies to the whole chain rather than just one certificate in it. In other words if the chain is signed by a trusted CA, the entire chain beyond that point will be regarded as trusted.

If the result is set to true (a nonzero value) then the CA 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.

Marking a CA certificate as untrusted doesn't mean that it can never be trusted, but merely that its 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 CA certificate is really a certificate whose precise trust level has yet to be determined rather than a certificate which is explicitly not trusted. If you want to explicitly not trust a certificate for one or more types of usage, you can do this using the CRYPT_CERTINFO_TRUSTED_USAGE attribute.

Making Trust Settings Persistent

The certificate trust settings are part of cryptlib's configuration options, which are explained in more detail in "Working with Configuration Options" on page 292. 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 CA certificates and want the new trust values to be applied when you use cryptlib in the future you would use code like:

```
/* Mark various CA 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 );
```

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 that it recognises, and through **cryptAddCertExtension**, **cryptGetCertExtension**, and **cryptDeleteCertExtension** for general extensions that 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 that you define yourself.

Because of the high level of complexity of many of these extensions combined with the fact that most are rarely, if ever used, processing of the more complicated and/or obscure extensions are disabled by default, see "Certificate Compliance Level Checking" on page 214 for more detail on the different levels of processing. For each of the following extensions, the description lists which level it's available at.

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 contains three main pieces of information:

Field	**Description**
Type	The extension type, a unique identifier called an object identifier. This is given as a sequence of numbers that 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 that 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. 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 implementations process it. As usual, you should check to see whether your intended target correctly processes the extensions that you plan to use.
Value	The extension data, corresponding to a cryptlib attribute group for more complex composite extensions, or a single cryptlib attribute for a few very simple extensions.

For the extensions that cryptlib recognises, the handling of the critical flag is automatic. For extensions that 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

Certificate extensions correspond to cryptlib attribute groups, with individual components of each certificate extension being represented by attributes within the group. Since this section applies specifically to certificates, the certificate-specific terminology referring to extensions rather than the general term attribute group will be used here.

cryptlib can identify attributes in extensions/attribute groups in one of three ways:

1. Through an extension identifier that denotes the entire extension/attribute group. For example CRYPT_CERTINFO_CERTPOLICIES denotes the certificatePolicies extension/attribute group.
2. Through an attribute identifier that denotes a particular attribute within an extension/attribute group. For example CRYPT_CERTINFO_CERTPOLICY denotes the policyIdentifier attribute contained within the certificatePolicies extension/attribute group.

   Some extensions/groups only contain a single attribute, 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 the attribute cursor mechanism that allows you to step through a set of extensions extension by extension or attribute by attribute. Attribute cursor management is explained in more detail in "Attribute Lists and Attribute Groups" on page 41.

You can use the extension/group 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/group are handled in the usual manner, for example to retrieve the value of the basicConstraints CA attribute (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 that 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. Because the basicConstraints extension is a fundamental X.509v3 extension, cryptlib will in fact always add this by default even if you don't explicitly specify it.

Composite Extension Attributes

Attributes that contain complete GeneralNames and/or DNs are composite attributes that have further items within them. These are handled in the standard way using the attribute cursor: You first move the cursor to the attribute that contains the GeneralName or DN that you want to work with and then get, set, or delete attributes within it:

```
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_PERMITTEDSUBTREES );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_RFC822NAME,
   rfc822Name, rfc822NameLength );
cryptSetAttributeString( certificate, CRYPT_CERTINFO_DNSNAME, dnsName,
   dnsNameLength );
```

This code first moves the cursor to the nameConstraints permittedSubtrees GeneralName and then sets the GeneralName attributes as usual. Since a GeneralName contains its own DN, moving the attribute cursor onto a GeneralName means that any DN accesses will now refer to the DN in the GeneralName rather than the certificate subject or issuer name:

```
/* Select the permittedSubtrees GeneralName */
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_PERMITTEDSUBTREES );

/* Set the DN components within the GeneralName */
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 and then sets the DN components as usual. cryptlib uses this mechanism to access all DNs and GeneralNames, although this is usually hidden from you — when you modify a certificate object's 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 another one or move the attribute cursor off it, 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_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_SUBJECTNAME );
```

otherwise attempts to add, delete, or query further DN (or GeneralName) attributes will apply to the selected nameConstraints excludedSubtrees attribute instead of the subject name. Conversely, if you move the attribute cursor off the GeneralName that you're working with, subsequent attempts to work with GeneralName or DN fields will fail with a CRYPT_ERROR_NOTFOUND, since there's no GeneralName currently selected.

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 that 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 that aren't suitable for the main certificate DNs. 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 230. This extension is valid in certificates, certification requests and CRLs, is available at all certificate processing levels, 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, is available at all certificate processing levels, 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, Policy Constraints, and Policy Inhibiting

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, is valid in certificates, and is available at all certificate processing levels.

The certificate policies attribute is a complex extension that 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 are ignored by many implementations:

Attribute/Description	Type
CRYPT_CERTINFO_CERTPOLICYID The object identifier that 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 CRYPT_CERTINFO_CERTPOLICY_NOTICENUMBERS CRYPT_CERTINFO_CERTPOLICY_EXPLICITTEXT These attributes contain further qualifiers, modifiers, and text information that amend the certificate policy information. Refer to certificate standards documents for more information on these attributes.	String Numeric String

Since various CAs that would like to accept each other's certificates may have differing policies, there is an extension that 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 identified by CRYPT_CERTINFO_POLICYMAPPINGS, is valid in certificates, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

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, is valid in certificates, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_REQUIREEXPLICITPOLICY See certificate standards documents.	Numeric
CRYPT_CERTINFO_INHIBITPOLICYMAPPING See certificate standards documents.	Numeric

Finally, a CA can inhibit the use of the special-case anyPolicy policy. The inhibitAnyPolicy extension is identified by CRYPT_CERTINFO_-INHIBITANYPOLICY, is valid in certificates, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_INHIBITANYPOLICY See certificate standards documents.	Numeric

CRL Distribution Points/Freshest CRL and Subject/Authority Information Access

These extensions specify how to obtain CRL information and information on the CA that issued a certificate. The cRLDistributionPoint extension is valid in certificates, is available at all certificate processing levels, and is identified by CRYPT_CERTINFO_CRLDISTRIBUTIONPOINT:

Attribute/Description	Type
CRYPT_CERTINFO_CRLDIST_FULLNAME The location at which CRLs may be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_CRLDIST_REASONS CRYPT_CERTINFO_CRLDIST_CRLISSUER See certificate standards documents.	Numeric 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.

If you plan to use this extension, you should be aware of the fact that it exists solely as a kludge created to work around problems involved in finding CRLs in X.500 directories, and thus presents a rather poor mechanism for distributing and obtaining revocation information. Unless it's absolutely imperative that you use this extension, it's better to use RTCS, OCSP, or SCVP as explained in "Certificate Status Checking using RTCS" on page 168, "RTCS Server Sessions" on page 124, "Certificate Revocation Checking using OCSP" on page 168, "OCSP Server Sessions" on page 124, "SCVP Server Sessions" on page 125, and "Certificate Status Checking using SCVP" on page 177.

The freshestCRL extension is valid in certificates, is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level, and is identified by

CRYPT_CERTINFO_FRESHESTCRL. The structure is identical to cRLDistributionPoint, with the subfields named with FRESHESTCRL instead of CRLDIST. As with cRLDistributionPoint, this is a kludge used to work with delta CRLs.

The subjectInfoAccess extension is valid in certificates, is available at all certificate processing levels, and is identified by CRYPT_CERTINFO_-SUBJECTINFOACCESS:

Attribute/Description	**Type**
CRYPT_CERTINFO_SUBJECTINFO_CAREPOSITORY The location at which the CA publishes certificates and CRLs, if the certificate is for a CA. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_SUBJECTINFO_SIGNEDOBJECT The location at which resource PKI (RPKI) objects are published. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_SUBJECTINFO_-SIGNEDOBJECTREPOSITORY The location at which resource PKI (RPKI) objects are published. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_SUBJECTINFO_RPKIMANIFEST The location at which resource PKI (RPKI) objects are published. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_SUBJECTINFO_TIMESTAMPING The location at which timestamping services using the timestamp protocol (TSP) are available. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName

The authorityInfoAccess extension is valid in certificates and CRLs, is available at all certificate processing levels, and is identified by CRYPT_CERTINFO_AUTHORITYINFOACCESS:

Attribute/Description	**Type**
CRYPT_CERTINFO_AUTHORITYINFO_CAISSUERS The location at which information on CAs located above the CA that issued this certificate can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AUTHORITYINFO_CERTSTORE The location at which further certificates issued by the CAs that issued this certificate can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AUTHORITYINFO_CRLS The location at which further certificates issued by the CAs that issued this certificate can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AUTHORITYINFO_OCSP The location at which certificate revocation information can be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AUTHORITYINFO_RTCS The location at which certificate validity information 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_SUBJECTDIRECTORY-ATTRIBUTES, allows additional X.500 directory attributes to be specified for a certificate. This extension is valid in certificates, is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SUBJECTDIR_TYPE The object identifier that 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 certificate type

These extensions specify the allowed usage for the key contained in this certificate. The keyUsage attribute is a standard extension identified by CRYPT_CERTINFO_-KEYUSAGE and is used to specify general-purpose key usages 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 or ECDSA 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 TLS server authentication.

The Netscape certificate type attribute 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, is available at all certificate processing levels, and contains a bit flag that 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 and ECDH.
CRYPT_KEYUSAGE_-KEYCERTSIGN CRYPT_KEYUSAGE_-CRLSIGN	The key can be used to sign certificates and CRLs. 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 usually different to CRYPT_KEYUSAGE_-DIGITALSIGNATURE and is interpreted in various incompatible ways by different standards and profiles.

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 that specify a specific type of extended usage that 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, is available at all certificate processing levels, and can contain any of the following values:

Value	Used in
CRYPT_CERTINFO_EXTKEY_-ANYKEYUSAGE	No-op wildcard value used to work around extended key-usage validation bugs in some software.
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 file system 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_-OCSPSIGNING	OCSP response signing.
CRYPT_CERTINFO_EXTKEY_-SERVERAUTH CRYPT_CERTINFO_EXTKEY_-CLIENTAUTH	TLS 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 TLS server authentication you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_EXTKEY_SERVERAUTH,
   CRYPT_UNUSED );
```

Like the keyUsage extension, the Netscape certificate 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, is available if processing of obscure and obsolete extensions are enabled, and contains a bit flag that 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	TLS 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 basicConstraints CA value to be set.

This extension is obsolete and is supported as a read-only attribute by cryptlib. If you try to set this extension cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you can't set this attribute value.

Name Constraints

The nameConstraints extension is used to constrain the certificate's 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 CAs 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, is valid in certificates, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

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 DNs, it is possible to avoid the excludedSubtrees for DNs by choosing unusual (but perfectly valid) string encodings that 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 that was only permitted to issue certificates for Cognitive Cybernetics' marketing division, it would set DN name constraints with:

```
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_PERMITTEDSUBTREES );
cryptSetAttribute( certificate, CRYPT_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_DIRECTORYNAME );
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 attribute 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 that 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_ATTRIBUTE_CURRENT,
   CRYPT_CERTINFO_PERMITTEDSUBTREES );
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, is valid in certificates, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above. 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	Time
CRYPT_CERTINFO_PRIVATEKEY_NOTAFTER	Time
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.	

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, is available at the CRYPT_-COMPLIANCELEVEL_PKIX_PARTIAL processing level and above, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_AUTHORITY_KEYIDENTIFIER	Binary data
Binary data identifying the public key in the certificate that 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 that 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.	

The subjectKeyIdentifier is identified by CRYPT_CERTINFO_-SUBJECTKEYIDENTIFIER, is available at all certificate processing levels, and contains binary data identifying the public key in the certificate.

CRL Extensions

CRLs have a number of CRL-specific extensions that are described below.

CRL Reasons, CRL Numbers, Delta CRL Indicators

These extensions specify various pieces of information about CRLs. The reasonCode extension is used to indicate why a certificate was revoked. The cRLNumber extension provides a serial number for CRLs. The deltaCRLIndicator indicates a delta CRL that contains changes between a base CRL and a delta-CRL (this is used to reduce the overall size of CRLs).

The reasonCode extension is identified by CRYPT_CERTINFO_CRLREASON, is valid in CRLs, and is available at all certificate processing levels. The extension has a single numeric attribute with the same identifier as the extension itself (CRYPT_CERTINFO_CRLREASON) which contains a bit flag that 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 CRYPTCRLREASON_-AACOMPROMISE	The CA or attribute authority that 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 that requires the use of the certificate.
CRYPT_CRLREASON_-KEYCOMPROMISE	The key for the certificate was compromised.
CRYPT_CRLREASON_-PRIVILEGEWITHDRAWN	The privilege granted in an attribute certificate is no longer valid.
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, is valid in CRLs, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_-PARTIAL processing level and above. The extension has a single attribute with the same identifier as the extension itself (CRYPT_CERTINFO_CRLNUMBER) which contains a monotonically increasing sequence number set by the CA for each CRL issued. This allows an application to check that it has received and processed each CRL that was issued.

The deltaCRLIndicator extension is identified by CRYPT_CERTINFO_-DELTACRLINDICATOR, is valid in CRLs, and is available at the CRYPT_-COMPLIANCELEVEL_PKIX_PARTIAL processing level and above. 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 CRLs).

Base Update Time and Delta Information

These extension specify various pieces of information about delta CRLs that were added in newer versions of X.509v3. The baseUpdateTime extension is used to indicate the time after which this delta CRL provides updates to the revocation status. The deltaInfo extension is used in non-delta CRLs to indicate where delta CRL information can be found.

The baseUpdateTime extension is identified by CRYPT_CERTINFO_-BASEUPDATETIME, is valid in CRLs, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_BASEUPDATETIME Time after which this delta CRL provides updates to the revocation status.	Time

The deltaInfo extension is identified by CRYPT_CERTINFO_DELTAINFO, is valid in CRLs, is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_DELTAINFO_LOCATION The location where the delta CRL can be found.	GeneralName
CRYPT_CERTINFO_DELTAINFO_NEXTDELTA The issue time for the next delta CRL.	Time

CRL Stream Identifier and Ordered List

These extension specify various pieces of information about CRLs that were added in newer versions of X.509v3. The crlStreamIdentifier extension is used to identify the context within which the CRL number is unique. The orderedList extension indicates that the list of revoked certificates in a CRL is sorted by serial number or revocation data.

The crlStreamIdentifier extension is identified by CRYPT_CERTINFO_-CRLSTREAMIDENTIFIER, is valid in CRLs, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_CRLSTREAMIDENTIFIER The context within which the CRL number is unique.	Numeric

The orderedList extension is identified by CRYPT_CERTINFO_ORDEREDLIST, is valid in CRLs, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_ORDEREDLIST Zero if the CRL entries are sorted by serial number, one if the CRL entries are sorted by revocation date.	Numeric

Hold Instruction Code

This extension contains a code that specifies what to do with a certificate that 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 CRLs, is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level, 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 that 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 );
```

You shouldn't use this extension (or the CRYPT_CRLREASON_-CERTIFICATEHOLD reasonCode) unless you really need to because although a mechanism was defined for placing a certificate on hold, no-one ever defined one for removing it from this state, so once it's on hold it's revoked no matter what the reasonCode says.

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, is valid in CRLs, and is available at all certificate processing levels:

Attribute/Description	Type
CRYPT_CERTINFO_INVALIDITYDATE The date on which the key identified in a CRL became invalid.	Time

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 and aaIssuingDistributionPoint extensions are identified by CRYPT_CERTINFO_ISSUINGDISTRIBUTIONPOINT and CRYPT_CERTINFO_AAISSUINGDISTRIBUTIONPOINT, are valid in CRLs, and are available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above:

Attribute/Description	Type
CRYPT_CERTINFO_ISSUINGDIST_FULLNAME The location at which CRLs 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

Attribute/Description	Type
CRYPT_CERTINFO_AAISSUINGDIST_FULLNAME The location at which CRLs may be obtained. You should use the URL component of the GeneralName for this, avoiding the other possibilities.	GeneralName
CRYPT_CERTINFO_AAISSUINGDIST_ SOMEREASONSONLY	Numeric
CRYPT_CERTINFO_AAISSUINGDIST_INDIRECTCRL	Boolean
CRYPT_CERTINFO_AAISSUINGDIST_USERATTRCERTS	Boolean
CRYPT_CERTINFO_AAISSUINGDIST_AACERTS	Boolean
CRYPT_CERTINFO_AAISSUINGDIST_SOACERTS See certificate standards documents.	Boolean

Note that the CRYPT_CERTINFO_ISSUINGDIST_SOMEREASONSONLY and CRYPT_CERTINFO_AAISSUINGDIST_SOMEREASONSONLY attributes have 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. Note also that although the extensions serve the same purposes (one is for standard certificates and the other is for attribute certificates), some of the otherwise-identical fields have different names and are in different places (again, because X.509 says so).

The certificateIssuer extension contains the certificate issuer for an indirect CRL. The extension is identified by CRYPT_CERTINFO_CERTIFICATEISSUER, is valid in CRLs, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_-PARTIAL processing level and above:

Attribute/Description	Type
CRYPT_CERTINFO_CERTIFICATEISSUER See certificate standards documents.	GeneralName

To Be Revoked, Revoked Groups, and Expired Certs in CRL

These extension specify various pieces of additional revocation information that were added in newer versions of X.509v3. The toBeRevoked extension is used to indicate that certificates will be revoked in the future. The revokedGroups extension is used to identify a set of certificates that have been revoked. The expiredCertsOnCRL extension is used to indicate that the CRL also contains expired certificates.

The toBeRevoked extension is identified by CRYPT_CERTINFO_-TOBEREVOKED, is valid in CRLs, and is available at the CRYPT_-COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_TOBEREVOKED_CERTISSUER The issuer of the certificates to be revoked.	GeneralName
CRYPT_CERTINFO_REVOKEDGROUPS_ REASONCODE The reason code for the revocation.	Numeric
CRYPT_CERTINFO_TOBEREVOKED_- REVOCATIONTIME The revocation time.	Time
CRYPT_CERTINFO_TOBEREVOKED_- CERTSERIALNUMBER The serial number of the certificate to be revoked.	Binary data

The revokedGroups extension is identified by CRYPT_CERTINFO_-REVOKEDGROUPS, is valid in CRLs, and is available at the CRYPT_-COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_REVOKEDGROUPS_CERTISSUER The issuer of the revoked group of certificates.	GeneralName
CRYPT_CERTINFO_REVOKEDGROUPS_ REASONCODE The reason code for the revocation of a group of certificates.	Numeric
CRYPT_CERTINFO_REVOKEDGROUPS_ INVALIDITYDATE The revocation time for the group of certificates	Time
CRYPT_CERTINFO_REVOKEDGROUPS_ STARTINGNUMBER The serial number of the first certificate in the group being revoked.	Binary data
CRYPT_CERTINFO_REVOKEDGROUPS_ ENDINGNUMBER The serial number of the last certificate in the group being revoked.	Binary data

The expiredCertsOnCRL extension is identified by CRYPT_CERTINFO_-EXPIREDCERTSONCRL, is valid in CRLs, and is available at the CRYPT_-COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_EXPIREDCERTSONCRL The time at which the certificates expired.	Time

Digital Signature Legislation Extensions

Various digital signature laws specify extensions beyond the X.509v3 ones that are described below.

Additional Information

The German signature law (SigG) specifies an extension containing any other general free-form information that may be included with the certificate. The extension is identified by CRYPT_CERTINFO_SIGG_ADDITIONALINFORMATION, is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_ ADDITIONALINFORMATION Free-form text string containing further information to be included with the certificate.	String

Admissions

The German signature law (SigG) specifies an extension containing information on admissions granted to professional associations, chambers, unions, administrative bodies, and companies. The extension is identified by CRYPT_CERTINFO_-SIGG_ADMISSIONS, is available at the CRYPT_COMPLIANCELEVEL_PKIX_-PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_ADMISSIONS_AUTHORITY The authority granting the admission.	GeneralName
CRYPT_CERTINFO_SIGG_ADMISSIONS_ NAMINGAUTHID The ASN.1 object identifier for the naming authority that administers the title granted by the admission.	String
CRYPT_CERTINFO_SIGG_ADMISSIONS_ NAMINGAUTHURL The URL for the naming authority that administers the title granted by the admission.	String
CRYPT_CERTINFO_SIGG_ADMISSIONS_ NAMINGAUTHTEXT Text describing the naming authority that administers the title granted by the admission.	String
CRYPT_CERTINFO_SIGG_ADMISSIONS_ PROFESSIONITEM Text describing the admission being granted.	String
CRYPT_CERTINFO_SIGG_ADMISSIONS_ PROFESSIONOID ASN.1 object identifier for the admission being granted.	String
CRYPT_CERTINFO_SIGG_ADMISSIONS_ REGISTRATIONNUMBER Registration number within the admission being granted.	String

Certificate Hash

The German signature law (SigG) specifies an OCSP extension containing the hash of the certificate in an OCSP response. Note that this is actually an OCSP extension even though it's specified in certificate digital signature legislation. The extension is identified by CRYPT_CERTINFO_SIGG_CERTHASH, is available at the CRYPT_-COMPLIANCELEVEL_PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_CERTHASH See SigG standards documents.	Binary data

Certificate Generation Date

The German signature law (SigG) specifies an extension containing the date at which the certificate was generated. This is necessary for post-dated 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, is available at the CRYPT_COMPLIANCELEVEL_-PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_DATEOFCERTGEN The date on which the certificate was issued.	Time

Declaration of Majority

The German signature law (SigG) specifies an extension containing a declaration of majority for the certificate holder. The extension is identified by CRYPT_-CERTINFO_SIGG_DECLARATIONOFMAJORITY, is available at the CRYPT_-COMPLIANCELEVEL_PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, and contains the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_SIGG_ DECLARATIONOFMAJORITY_COUNTRY The two-letter country code for which the declaration of majority applies.	String

Reliance Limit

The German signature law (SigG) 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, is available at the CRYPT_COMPLIANCELEVEL_PKIX_-PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, 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 × 10^{exponent}.	Integer

Restrictions

The German signature law (SigG) specifies an extension containing any other general free-form restrictions that may be imposed on the certificate. The extension is identified by CRYPT_CERTINFO_SIGG_RESTRICTION, is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, 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

Signature Delegation

The German signature law (SigG) 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, is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above if processing of obscure and obsolete extensions is enabled, 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

Qualified Certificate Extensions

Qualified certificates contain additional extensions beyond the X.509v3 ones that are described below.

Biometric Info

The biometricInfo extension contains biometric information in the form of a hash of a biometric template. The extension is identified by CRYPT_CERTINFO_-BIOMETRICINFO, is valid in certificates and certification requests, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_BIOMETRICINFO_TYPE The type of the biometric data, see certificate standards documents.	Numeric
CRYPT_CERTINFO_BIOMETRICINFO_HASHALGO The object identifier for the hash algorithm used to hash the biometric template.	String
CRYPT_CERTINFO_BIOMETRICINFO_HASH The hash of the biometric template.	String
CRYPT_CERTINFO_BIOMETRICINFO_URL An optional URL at which the biometric data may be found.	String

QC Statements

The qcStatements extension contains defined statements for a qualified certificate. The extension is identified by CRYPT_CERTINFO_QCSTATEMENT, is valid in certificates and certification requests, and is available at the CRYPT_-COMPLIANCELEVEL_PKIX_FULL processing level:

Attribute/Description	Type
CRYPT_CERTINFO_QCSTATEMENT_SEMANTICS An object identifier identifying the defined statement for this certificate.	String
CRYPT_CERTINFO_QCSTATEMENT_-REGISTRATIONAUTHORITY See certificate standards documents.	String

Resource PKI Extensions

Resource PKI (RPKI) certificates contain additional extensions beyond the X.509v3 ones that cover allocations of IPv4 and IPv6 address space and autonomous system (AS) numbers. These certificates grant the use of the IP address space or AS number space to the certificate holder, and are used for securing the Internet routing infrastructure.

IP Address Blocks

The ipAddrBlocks extension contains information on IP address space allocated to the certificate holder. The extension is identified by CRYPT_CERTINFO_-IPADDRESSBLOCKS, is valid in certificates and certification requests, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above:

Attribute/Description	Type
CRYPT_CERTINFO_IPADDRESSBLOCKS_-ADDRESSFAMILY A two-byte address family identifier (AFI), either `00 01` for IPv4 or `00 02` for IPv6.	String
CRYPT_CERTINFO_IPADDRESSBLOCKS_PREFIX An IP address block prefix for an IP address range controlled by the certificate holder.	String
CRYPT_CERTINFO_IPADDRESSBLOCKS_MIN CRYPT_CERTINFO_IPADDRESSBLOCKS_MAX The start and end of an address range controlled by the certificate holder.	String String

Note that a single ipAddrBlocks extension may contain dozens or even hundreds of these values.

The encoding of the address block ranges is done using a peculiar format specified in the RPKI documentation which requires, instead of the usual 1.2.3.4/28-style encoding, the use of a non-standard ASN.1 BIT STRING encoding to convey the address prefix or address ranges. Because the resulting BIT STRING isn't in any standard form, you need to provide it to cryptlib as a pre-encoded blob starting with an `03` type specifier, a one-byte length, a one-byte unused-bits count, and then the IP address prefix as a variable-length bit string. See the RPKI standards documentation for exact details on how to encode this.

Autonomous Systems IDs

The autonomousSysIds extension contains information on AS numbers allocated to the certificate holder. The extension is identified by CRYPT_CERTINFO_-AUTONOMOUSSYSIDS, is valid in certificates and certification requests, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above:

Attribute/Description	Type
CRYPT_CERTINFO_AUTONOMOUSSYSIDS_-ASNUM_ID An AS number controlled by the certificate holder.	Numeric
CRYPT_CERTINFO_AUTONOMOUSSYSIDS_-ASNUM_MIN CRYPT_CERTINFO_AUTONOMOUSSYSIDS_-ASNUM_MAX The start and end of a range of AS numbers controlled by the certificate holder.	Numeric Numeric

Note that a single autonomousSysIds extension may contain dozens or even hundreds of these values.

SET Extensions

SET specifies a number of extensions beyond the X.509v3 ones that 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, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled. 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, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled:

Attribute/Description	Type
CRYPT_CERTINFO_SET_MERACQUIRERBIN	String
CRYPT_CERTINFO_SET_MERAUTHFLAG	Boolean
CRYPT_CERTINFO_SET_MERCOUNTRY	Numeric
CRYPT_CERTINFO_SET_MERID	String
Merchant's 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
Merchant's language, name, city, state or province, postal code, and country name.	

SET Certificate Type, Hashed Root Key, and Tunnelling

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, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled. The extension contains 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 that 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, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled. The extension contains a single attribute:

Attribute/Description	Type
CRYPT_CERTINFO_SET_ROOTKEYTHUMBPRINT Binary string containing the root key thumbprint (see the SET standards documents).	Binary data

You can obtain the 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 tunnelling extension contains a tunnelling indicator and algorithm identifier. The extension is identified by CRYPT_CERTINFO_SET_TUNNELING, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled.

Attribute/Description	Type
CRYPT_CERTINFO_SET_TUNNELINGFLAG	Boolean
CRYPT_CERTINFO_SET_TUNNELINGALGID See SET standards documents.	String

Application-specific Extensions

Various applications such as certificate management protocols have their own extensions that extend or complement the X.509 ones. These are described below.

OCSP Extensions

These extensions specify various pieces of certificate management information used in the OCSP certificate management protocol.

The noCheck extension indicates that the certificate should be automatically trusted when used to sign OCSP responses. The extension is identified by CRYPT_-CERTINFO_OCSP_NOCHECK, is valid in certificates and certification requests, and is available at the CRYPT_COMPLIANCELEVEL_PKIX_PARTIAL processing level and above. The extension contains a numeric attribute with the same identifier as the extension itself (CRYPT_CERTINFO_OCSP_NOCHECK) which is always set to CRYPT_UNUSED since it has no inherent value associated with it.

Attribute/Description	Type
CRYPT_CERTINFO_OCSP_NOCHECK See OCSP standards documents.	Numeric

Vendor-specific Extensions

A number of vendors have defined their own extensions that extend or complement the X.509 ones. These are described below.

Netscape Certificate Extensions

Netscape defined a number of extensions that mostly predate the various X.509v3 extensions that now provide the same functionality. The various Netscape certificate extensions are available if processing of obscure and obsolete extensions is enabled and are:

Extension/Description	Type
CRYPT_CERTINFO_NS_BASEURL A base URL which, if present, is added to all partial URL's in Netscape extensions to create a full URL.	String
CRYPT_CERTINFO_NS_CAPOLICYURL The URL at which the certificate policy under which this certificate was issued can be found.	String
CRYPT_CERTINFO_NS_CAREVOCATIONURL The URL at which the revocation status of a CA certificate can be checked.	String
CRYPT_CERTINFO_NS_CERTRENEWALURL The URL at which a form allowing renewal of this certificate can be found.	String
CRYPT_CERTINFO_NS_COMMENT A comment which should be displayed when the certificate is viewed.	String
CRYPT_CERTINFO_NS_REVOCATIONURL The URL at which the revocation status of a server certificate can be checked.	String
CRYPT_CERTINFO_NS_SSLSERVERNAME A wildcard string containing a shell expression that matches the hostname of the TLS server using this certificate.	String

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 that allows the use of certificates with secure extranets. This extension is identified by CRYPT_CERTINFO_-STRONGEXTRANET, is valid in certificates and certification requests, and is available if processing of obscure and obsolete extensions is enabled:

Attribute/Description	Type
CRYPT_CERTINFO_STRONGEXTRANET_ZONE	Numeric
CRYPT_CERTINFO_STRONGEXTRANET_ID Extranet zone and ID.	Binary data

Generic Extensions

Beyond the standardised extensions listed above there exist any number of obscure or non-standard certificate extensions. cryptlib allows you to work with these extensions using **cryptAddCertExtension**, **cryptGetCertExtension**, and **cryptDeleteCertExtension**, which allow you to add, retrieve, or delete a complete encoded extension identified by its ASN.1 object identifier. The extension data must be a complete DER-encoded ASN.1 object without the OCTET STRING wrapper which is used for all extensions (cryptlib will add this itself). For example if you wanted to add a 4-byte UTF8 string as an extension the data would be `0C 04` *`xx xx xx xx`*. If you pass in extension data to **cryptAddCertExtension** that isn't a valid ASN.1-encoded object, cryptlib will return CRYPT_ERROR_PARAM4 to indicate that the data is in an invalid format.

If a certificate object contains a non-standard extension, cryptlib won't include it in the object when you sign it unless you set the CRYPT_OPTION_CERT_-SIGNUNRECOGNISEDATTRIBUTES option to true. This is to avoid problems where a CA could end up signing arbitrary data in an unrecognised certificate extension.

If the extension you are trying to add is already handled as a standard extension, cryptlib will return CRYPT_ERROR_PERMISSION to indicate that you can't add the extension in this manner but have to add it using **cryptSetAttribute**/ **cryptSetAttributeString**.

Other Certificate Object Extensions

Certificate objects other than certificates and CRLs can also contain extensions. In the following descriptions only the generally useful attributes have been described. The full range of attributes is enormous and will probably never be used in real life. These attributes are marked with "See 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).

CMS/SMIME Attributes

The CMS and S/MIME standards specify various attributes that can be included with signatures. In addition there are a variety of proprietary and vendor-specific attributes that are also handled by cryptlib. In the following description only the generally useful attributes have been described, the full range of attributes is enormous and requires a number of standards specifications (often followed by cries for help on mailing lists) to interpret them. These attributes 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_DATA	Plain data.
CRYPT_CONTENT_-SIGNEDDATA	Signed data.
CRYPT_CONTENT_-ENVELOPEDDATA	Data encrypted using a password or public-key or conventional encryption.
CRYPT_CONTENT_-SIGNEDANDENVELOPED-DATA	Data which is both signed and enveloped (this is an obsolete composite content type that shouldn't be used).
CRYPT_CONTENT_-DIGESTEDDATA	Hashed data.
CRYPT_CONTENT_-ENCRYPTEDDATA	Data encrypted directly with a session key.
CRYPT_CONTENT_-COMPRESSEDDATA	Compressed data.
CRYPT_CONTENT_TSTINFO	Timestamp token generated by a timestamp authority (TSA).
CRYPT_CONTENT_-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_CONTENT_ENVELOPEDDATA (rather than CRYPT_CONTENT_-

ENCRYPTEDDATA, which is slightly different), if it's signed then use CRYPT_-CONTENT_SIGNEDDATA, and if it's anything else then use CRYPT_-CONTENT_DATA. For example to identify the data you're signing as encrypted data you would use:

```
cryptSetAttribute( cmsAttributes, CRYPT_CERTINFO_CMS_CONTENTTYPE,
    CRYPT_CONTENT_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 that 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 Description

This CMS attribute contains a short text message with an additional description of the data being signed. For example if the signed message was a response to a received signed message, the signing description might contain an indication of the type of message it's being sent in response to. Note that CMS has a number of special-purpose signing attributes such as message receipt information that allow automated processing of messages that contain them, so you should only use this free-form human-readable attribute for cases that aren't covered by special-case attributes designed for the purpose.

The attribute is identified by CRYPT_CERTINFO_CMS_SIGNINGDESCRIPTION:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGNINGDESCRIPTION Free-form text annotation for the message being signed.	String

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.	Time

Extended CMS/SMIME Attributes

The attributes given above are the standard CMS attributes. Extending beyond this are further attributes that are defined in additional standards documents and that 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 attributes that 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.	Numeric
CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIAL-CODESIGNING The data was signed by a commercial organisation.	Numeric

The opus info CMS attribute, identified by CRYPT_CERTINFO_CMS_-SPCOPUSINFO, is used to identify program details for AuthentiCode use, and has the following attributes:

Attribute/Description	**Type**
CRYPT_CERTINFO_CMS_SPCOPUSINFO_NAME Program name/version.	String
CRYPT_CERTINFO_CMS_SPCOPUSINFO_URL AuthentiCode information URL.	String

Note that the CRYPT_CERTINFO_CMS_SPCOPUSINFO_NAME attribute is a Unicode string, as used by Windows.

For example to indicate that the data was signed by an individual you would use:

```
cryptSetAttribute( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIALCODESIGNING, CRYPT_UNUSED );
```

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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_CERTTYPE_CMS_ATTRIBUTES );
cryptSetAttributeString( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCAGENCYURL,
   "http://homepage.organisation.com", 32 );
cryptSetAttribute( cmsAttributes,
   CRYPT_CERTINFO_CMS_SPCSTMT_COMMERCIALCODESIGNING, CRYPT_UNUSED );

/* 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 that 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 that 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

DOMSEC Attributes

The domain security (DOMSEC) attributes are used to handle delegated signing by systems such as mail gateways. The signature type CMS attribute, identified by CRYPT_CERTINFO_CMS_SIGTYPEIDENTIFIER, is used to identify the signature type, and has the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGTYPEID_-ADDITIONALATTRIBUTES Additional attributes for a domain signature.	Numeric
CRYPT_CERTINFO_CMS_SIGTYPEID_DOMAINSIG Domain signature by a gateway on behalf of a user.	Numeric
CRYPT_CERTINFO_CMS_SIGTYPEID_ORIGINATORSIG Indication that the signer is the originator of the message. This attribute isn't normally used, since it corresponds to a standard (non-DOMSEC) signature..	Numeric
CRYPT_CERTINFO_CMS_SIGTYPEID_REVIEWSIG Review signature to indicate that the domain signer has reviewed the message.	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.	Time
CRYPT_CERTINFO_CMS_MLEXP_NONE CRYPT_CERTINFO_CMS_MLEXP_INSTEADOF CRYPT_CERTINFO_CMS_MLEXP_INADDITIONTO This attribute can have one of the three values specified above, and is used to indicate a receipt policy that overrides the one given in the original message. See the S/MIME standards documents for more information.	— General- Name

Nonce

This CMS attribute nonce is used to prevent replay attacks. The attribute is identified by CRYPT_CERTINFO_CMS_NONCE:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_NONCE Nonce to prevent replay attacks.	Binary data

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

SCEP Attributes

The Simple Certificate Enrolment Protocol uses a variety of protocol-specific attributes that are attached to CMS signed data and are used to manage the operation of the protocol. These attributes are not normally used with CMS but are provided for use by cryptlib's SCEP implementation. The SCEP attributes are:

Attribute/Description	Type
CRYPT_CERTINFO_SCEP_MESSAGETYPE The SCEP message type.	String
CRYPT_CERTINFO_SCEP_PKISTATUS The processing status of an SCEP request.	String
CRYPT_CERTINFO_SCEP_FAILINFO Extended error information if the SCEP processing status indicates that an error occurred.	String
CRYPT_CERTINFO_SCEP_SENDERNONCE CRYPT_CERTINFO_SCEP_RECIPIENTNONCE Nonce values used to protect against message replay attacks. Note that these values duplicate the more usual CRYPT_CERTINFO_CMS_NONCE attribute, which should be used in place of these attributes unless they're specifically being used for SCEP.	Binary data
CRYPT_CERTINFO_SCEP_TRANSACTIONID A value that uniquely identifies the entity requesting a certificate.	String

In addition to these attributes, SCEP also uses an additional attribute which is added to PKCS #10 requests even though it's a CMS attribute. It therefore acts as a certificate attribute rather than a CMS attribute. The attribute is identified by CRYPT_CERTINFO_CHALLENGEPASSWORD:

Attribute/Description	Type
CRYPT_CERTINFO_CHALLENGEPASSWORD Password used to authorise certificate issue requests.	String

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 that 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 that 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 CRYPT_CERTINFO_CMS_SECLABEL_CATVALUE See S/MIME standards documents.	String 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 that lie outside this range:

Value
CRYPT_CLASSIFICATION_UNMARKED
CRYPT_CLASSIFICATION_UNCLASSIFIED
CRYPT_CLASSIFICATION_RESTRICTED
CRYPT_CLASSIFICATION_CONFIDENTIAL
CRYPT_CLASSIFICATION_SECRET
CRYPT_CLASSIFICATION_TOP_SECRET

Signature Policy

This CMS attribute is used to identify the policy under which a signature was generated, and is identified by CRYPT_CERTINFO_CMS_-SIGNATUREPOLICYID. The signature policies extension allows a signer to provide information on the policies governing a signature, and to control the way in which a signature can be interpreted. For example it allows you to check that a signature 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 a complex extension that allows for all sorts of qualifiers and additional modifiers (several of them exist only because this extension was a cut & paste of a similar-looking extension that's used with certificates). In general you should only use the policyIdentifier attribute in this extension, since the other attributes are difficult to support in user software and are ignored by many implementations:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGPOLICYID	String
The object identifier that identifies the policy under which this certificate was issued.	
CRYPT_CERTINFO_CMS_SIGPOLICYHASH	Binary data
The hash algorithm identifier and hash of the signature policy, see signature standards documents.	
CRYPT_CERTINFO_CMS_SIGPOLICY_CPSURI	String
The URL for the certificate practice statement (CPS) for this signature policy.	
CRYPT_CERTINFO_CMS_SIGPOLICY_ORGANIZATION	String
CRYPT_CERTINFO_CMS_SIGPOLICY_-NOTICENUMBERS	Numeric
CRYPT_CERTINFO_CMS_SIGPOLICY_EXPLICITTEXT	String
These attributes contain further qualifiers, modifiers, and text information that amend the signature policy information. Refer to signature standards documents for more information on these attributes.	

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 algorithm(s) and mechanism(s), as well as other S/MIME-related preferences. The attribute is identified by CRYPT_CERTINFO_CMS_SMIMECAPABILITIES and can contains any one or more of the following values:

Value	Description
CRYPT_CERTINFO_CMS_-SMIMECAP_3DES CRYPT_CERTINFO_CMS_-SMIMECAP_AES CRYPT_CERTINFO_CMS_-SMIMECAP_CAST128 CRYPT_CERTINFO_CMS_-SMIMECAP_DES CRYPT_CERTINFO_CMS_-SMIMECAP_IDEA	The sender supports the use of these encryption algorithms. When encoding them, cryptlib will order them by algorithm strength so that AES will be preferred over triple DES which will be preferred over DES. In addition only those algorithms that are currently enabled in cryptlib will be encoded.
CRYPT_CERTINFO_CMS_-SMIMECAP_SHA1 CRYPT_CERTINFO_CMS_-SMIMECAP_SHA2	The sender supports the use of these hash algorithms, with the same encoding procedure as for encryption algorithms above.
CRYPT_CERTINFO_CMS_-SMIMECAP_HMAC_SHA1 CRYPT_CERTINFO_CMS_-SMIMECAP_HMAC_SHA2	The sender supports the use of these MAC algorithms, with the same encoding procedure as for encryption algorithms above.
CRYPT_CERTINFO_CMS_-SMIMECAP_AUTHENC128 CRYPT_CERTINFO_CMS_-SMIMECAP_AUTHENC256	The sender supports the use of these authenticated encryption modes, with the same encoding procedure as for encryption algorithms above.
CRYPT_CERTINFO_CMS_-SMIMECAP_DSA_SHA1 CRYPT_CERTINFO_CMS_-SMIMECAP_ECDSA_SHA1 CRYPT_CERTINFO_CMS_-SMIMECAP_ECDSA_SHA2 CRYPT_CERTINFO_CMS_-SMIMECAP_RSA_SHA1 CRYPT_CERTINFO_CMS_-SMIMECAP_RSA_SHA2	The sender supports the use of these signature mechanisms, with the same encoding procedure as for encryption algorithms above.
CRYPT_CERTINFO_CMS_-SMIMECAP_-PREFERSIGNEDDATA	The sender would prefer to be sent signed data.
CRYPT_CERTINFO_CMS_-SMIMECAP_-PREFERBINARYINSIDE	The sender would prefer to be sent binary MIME messages rather than base64-encoded ones (this is a S/MIME-specific option that doesn't apply to the underlying CMS format).
CRYPT_CERTINFO_CMS_-SMIMECAP_-CANNOTDECRYPTANY	The sender can't handle any form of encrypted data.

To indicate that you can support messages encrypted with AES and CAST-128 you would use:

```
cryptSetAttribute( certificate, CRYPT_CERTINFO_CMS_SMIMECAP_AES,
   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 no need to set these attributes yourself.

Signing Certificate and Signing Certificate V2

These CMS attributes provides additional information about the certificate used to sign a message, are identified by CRYPT_CERTINFO_SIGNINGCERTIFICATE and CRYPT_CERTINFO_SIGNINGCERTIFICATEV2, and contain the following attributes:

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGNINGCERT_ESSCERTID See S/MIME standards documents.	Binary data
CRYPT_CERTINFO_CMS_SIGNINGCERT_POLICIES The object identifier for the policy that applies to the signing certificate.	String

Attribute/Description	Type
CRYPT_CERTINFO_CMS_SIGNINGCERTV2_ ESSCERTIDV2 See S/MIME standards documents.	Binary data
CRYPT_CERTINFO_CMS_SIGNINGCERTV2_ POLICIES The object identifier for the policy that applies to the signing certificate.	String

OCSP Attributes

Like certificates, OCSP requests and responses can contain extensions that contain additional information relating to the request or response. The ocspNonce extension is used to prevent replay attacks on OCSP requests and is set automatically by cryptlib. The ocspArchiveCutoff extension indicates the time limit to which an OCSP responder will store revocation information for a certificate. The ocspResponseType extension indicates the type of response you'd like to receive from a responder.

The ocspNonce extension is identified by CRYPT_CERTINFO_OCSP_NONCE and is valid in OCSP requests and responses. The extension has a single binary data attribute with the same identifier as the extension itself (CRYPT_CERTINFO_-OCSP_NONCE). Since cryptlib sets this value automatically, you can't set it yourself:

Attribute/Description	Type
CRYPT_CERTINFO_OCSP_NONCE Nonce to prevent replay attacks.	Binary data

The ocspArchiveCutoff extension is identified by CRYPT_CERTINFO_OCSP_-ARCHIVECUTOFF and is valid in OCSP responses:

Attribute/Description	Type
CRYPT_CERTINFO_OCSP_ARCHIVECUTOFF The date beyond which revocation information will no longer be archived by the responder.	Time

The ocspResponseType extension is identified by CRYPT_CERTINFO_-OCSP_RESPONSE and is valid in OCSP requests. This extension contains a collection of one or more values that indicate the type of response which is being requested from the OCSP responder. The values are:

Value	**Description**
CRYPT_CERTINFO_OCSP_-RESPONSE_OCSP	OCSP response containing only revocation information but no actual certificate status.
CRYPT_CERTINFO_OCSP_-RESPONSE_RTCS	RTCS response containing OK/not OK certificate status.
CRYPT_CERTINFO_OCSP_-RESPONSE_RTCS_-EXTENDED	Extended RTCS response containing certificate status and additional information such as revocation information.

In addition to OCSP-specific attributes, OCSP responses can also contain the CRL attributes reasonCode, holdInstructionCode, invalidityDate, and certificateIssuer, which are described in "CRL Extensions" on page 257.

Encryption Devices and Encryption Hardware

cryptlib's standard cryptographic functionality is provided through its built-in implementations of the required algorithms and mechanisms, however in some cases you may want 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
- Cryptographic smart cards
- Dallas iButtons
- Datakeys
- PCMCIA crypto cards such as Fortezza cards
- PKCS #11 crypto tokens
- Software encryption modules

In addition cryptlib supports a general-purpose cryptographic hardware capability that allows you to replace any of the built-in software algorithms with encryption hardware capabilities present as on-chip or in-system cryptographic hardware. These capabilities are typically present on some of the more advanced ARM, MIPS, and PPC cores.

The most common use for a hardware device is one where the hardware provides secure key storage and management functions, or where it provides specific algorithms or performance that may not be available in software. Using an external implementation involves conceptually plugging in the external hardware or software or on-board cryptographic hardware alongside or in place of the built-in capabilities provided by cryptlib and then creating cryptlib objects (for example encryption contexts) via the device.

Onboard cryptographic hardware has slightly different requirements from other encryption device types since it's directly integrated into cryptlib, and is covered in "On-Chip Encryption Hardware and Crypto Cores" on page 291. In the discussion of general-purpose encryption devices that follows, the sections on user authentication and key storage don't apply to on-chip and in-system cryptographic hardware.

Creating/Destroying Device Objects

Devices are accessed as device objects that work in the same general manner as other cryptlib objects. You open a connection to a device using **cryptDeviceOpen**, specifying the user who is to own the device object or CRYPT_UNUSED for the default, normal user, 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, cryptUser /* CRYPT_UNUSED */,
   deviceType, deviceName );

/* Use the services provided by the device */

cryptDeviceClose( cryptDevice );
```

The available device types are:

Device	Description
CRYPT_DEVICE_FORTEZZA	Fortezza PCMCIA card.

Device	Description
CRYPT_DEVICE_HARDWARE	On-chip cryptographic hardware or dedicated crypto core.
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, cryptUser /* CRYPT_UNUSED */,
   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.

Some devices have built-in real-time clocks, if cryptlib detects that the device has a built-in clock it'll use the device clock to obtain the time for operations such as creating signed timestamps. Since device clocks can drift over time, cryptlib will perform a consistency check of the device time against the system time and will fall back to using the system time if the device time is too far out of step. In addition the debug build will throw an exception if it detects a problem with the device time.

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 (this isn't generally required for onboard cryptographic hardware accessed as CRYPT_DEVICE_HARDWARE, but is usually necessary for external crypto devices). 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 that may be required. In the case of user authentication, the additional information will consist of a PIN or password that enables access. Many devices recognise two types of access code, a user-level code that provides standard access (for example for encryption or signing) and a supervisor-level code that 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 re-enable its use after a user has been locked out.

Device Initialisation

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 that support supervisor access you need to supply the initialisation or initial supervisor PIN when you call this function:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_INITIALISE,
   initialPin, initialPinLength );
```

Once you've initialised the device, you may need to set the supervisor PIN if the device uses a distinct initialisation PIN:

```
cryptSetAttributeString( cryptDevice,
   CRYPT_DEVINFO_SET_AUTHENT_SUPERVISOR, supervisorPin,
   supervisorPinLength );
```

At this point you can carry out device-specific initialisation actions while the device is still in the supervisor state. For example if you're working with a Fortezza card, you would load the CA root (PAA) certificate at this point, since it can only be loaded when the card is first moved into the supervisor-initialised state. Since this is the ultimately-trusted certificate in the card, it can only be loaded when the card is in this state.

Once you've finished performing any optional further initialisation, you need to set a user PIN, unless the device uses a combined user/supervisor role:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_SET_AUTHENT_USER,
   userPin, userPinLength );
```

Finally, you'll need to log on as a user with the PIN you've just set if the device doesn't do this automatically when you initially set the PIN:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_AUTHENT_USER,
   userPin, userPinLength );
```

The exact initialisation details vary from device to device and driver to driver. Some devices don't distinguish between supervisor and user roles and so only have a single role and PIN. Some devices require a PIN to initialise the device and then set the supervisor PIN using a separate call, others set the supervisor PIN as part of the initialisation call. Some devices will automatically switch over to user mode when you set the user PIN while others require you to explicitly log on in user mode after setting the user PIN. Finally, some devices can't be initialised through PKCS #11 but require proprietary vendor software to initialise them.

When the device is initialised, it usually moves through a number of states going from uninitialised to supervisor initialised to user initialised, with strict restrictions on what can be done in each state. For example once a supervisor has set the user PIN, they can usually no longer change it, since the supervisor isn't supposed to be able to take on the user role and manipulate the device. This is why some devices automatically log the supervisor out once the user PIN has been set. In addition some maintenance operations such as loading initial trusted certificates can only be performed after the device has been initialised and is still in the initial supervisor-initialised state. Again, this prevents modification of trusted keys after the user has been given access to the device.

A general rule of thumb is that when you go through an initialisation you have to perform all of the steps in sequence without logging out in between, and once you've initialised the device you usually can't change any settings without re-initialising it and starting from scratch. Individual devices may diverge from this in places, but in general you shouldn't assume that you can go back later and change things once you've set them.

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 that 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 re-enable user access.

Device Zeroisation

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 some devices you may need to supply a zeroisation PIN or the initial supervisor PIN when you call this function, otherwise you should set the data value to an empty string:

```
cryptSetAttributeString( cryptDevice, CRYPT_DEVINFO_ZEROISE, "", 0 );
```

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 cryptlib's built-in functionality, cryptlib provides the **cryptDeviceCreateContext** and **cryptDeviceQueryCapability** functions that 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, cryptUser /* CRYPT_UNUSED */,
   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.

Note that, unlike the other functions that create cryptlib objects, **cryptDeviceCreateContext** doesn't require you to specify the identity of the user who is to own the context which is being created. This is because the device is already associated with a user, so there's no need to specify this again when creating an object within it.

For an example of how you might utilise external hardware, let's use a generic AES hardware accelerator (identified by the label "AES accelerator") accessed as a PKS #11 device. To use the AES hardware instead of cryptlib's built-in AES implementation you would use:

```
CRYPT_DEVICE cryptDevice;
CRYPT_CONTEXT cryptContext;

/* Activate the DES hardware and create a context in it */
cryptDeviceOpen( &cryptDevice, cryptUser /* CRYPT_UNUSED */,
   CRYPT_DEVICE_PKCS11, "AES accelerator" );
cryptDeviceCreateContext( cryptDevice, &cryptContext,
   CRYPT_ALGO_AES );

/* 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  */
cryptDeviceCreateContext( &privKeyContext, cryptUser
   /* CRYPT_UNUSED */, 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.

Key storage in crypto devices has additional special considerations that are covered in "Considerations when Working with Devices" on page 287. The most notable of these is that many devices don't allow direct key loads into devices, and virtually all don't allow them to be extracted, so that the key has to be generated inside the device (as the example code given earlier shows) and can't leave the device except (for conventional encryption keys) in encrypted form.

Querying Device Information

Crypto devices come in a wide range of configurations and with varying capabilities, which can include facilities that bypass the normal device-handling operations described here. For example a device may have a built-in keypad or other authentication mechanism that bypasses the need to provide a PIN or password from software. In this case it's not necessary to log in to the device because the login process is handled via an external mechanism. You can determine whether a device is already logged in, or doesn't require a login, by reading the CRYPT_DEVINFO_-LOGGEDIN attribute. If this is set to true (any nonzero value) then the device is already logged in, otherwise you need to provide a PIN or password to log in to the device:

```
int deviceLoggedIn;

/* Check whether we're logged in to the device and if not, log in */
cryptGetAttribute( cryptDevice, CRYPT_DEVINFO_LOGGEDIN,
   &deviceLoggedIn );
if( !deviceLoggedIn )
   /* Get PIN from user and log in */;
```

Since some devices represent removable tokens such as smart cards, it's possible for the user to unplug one token and plug in a new one in its place. To help you determine which token was plugged in at the time it was accessed with **cryptDeviceOpen**, you can read the device's CRYPT_DEVINFO_LABEL attribute, which returns the label or name of the token which is accessible via the device:

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

cryptGetAttributeString( cryptDevice, CRYPT_DEVINFO_LABEL, label,
   &labelLength );
label[ labelLength ] = '\0';
```

Once you've read the label you can use it to determine whether the required crypto token is available via the device.

Some readers and device interfaces aren't very good at detecting the removal of a crypto token, or the removal of a token and insertion of a new one. For example, many smart card readers only have a simple sensor to detect whether there's something present in the reader, but can't tell whether what's present is the original smart card or a piece of cardboard. In addition some low-level reader drivers can't report the presence (or absence) of a card to the higher-level code. cryptlib will try to contact the crypto token to check whether it's still present and active, but can only go as far as the underlying hardware and software will let it.

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 crypto devices can also complicate key management, since keys generated or loaded into the device usually can't be extracted again afterwards. This is a security feature that 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 that 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 that needs to be taken into account is the data processing speed of the device. In most cases it's preferable to use cryptlib's built-in implementation of an algorithm rather than the one provided by the device because the built-in implementation will be much faster. For example when hashing data prior to signing it, cryptlib's 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 clocked at several gigahertz 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 data it's generally preferable to use cryptlib's 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. As a general rule of thumb, if your system processor is running at 500 MHz or higher then it's always faster to perform the crypto in software rather than using crypto hardware. Because of this it's usual to only perform private-key operations in the crypto device.

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 through which all data must pass (possibly augmented by the ability to store multiple encryption keys in the device). 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).

As a result of this, 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 device's crypto engine), a single digital signature context (using the device's public-key engine), and a single hash context (using the device's CPU or hash engine, 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 complete 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 are given in "Activating and Controlling Cryptographic Devices" on page 283. Once the card is in the SSO initialised state (after you've set the SSO PIN), you should install the CA root (PAA) certificate in the card, since this operation is only permitted in the SSO initialised state. The use of PAA certificates is somewhat specific to the use of Fortezza's by the US Government, you may want to simply load a dummy certificate at this point and use standard CA certificates with any keys that you'll be storing on the card.

Note that the Fortezza control firmware requires that all of the steps in the initialisation/programming process be performed in a continuous sequence of operations, without removing the card or closing the device. If you interrupt the process halfway through, you'll 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 Fortezza CAW.

PKCS #11 Devices

Although most of the devices that cryptlib interfaces with have specialised, single-purpose interfaces, PKCS #11 provides a general-purpose interface that 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 cryptlib's 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. Once you've specified the module name, you need to commit the changes in order for cryptlib to use them the next time it's loaded. For example to use the Gemplus GemSAFE driver you would use:

```
cryptSetAttributeString( CRYPT_UNUSED,
   CRYPT_OPTION_DEVICE_PKCS11_DVR01, "w32pk2ig.dll", 12 );
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_CONFIGCHANGED, FALSE );
```

The first line of code updates the configuration information to point to the PKCS #11 driver DLL, and the second line makes the changes permanent by flushing the configuration information to disk. For safety reasons cryptlib will always load drivers from the Windows system directory (%SystemDirectory%) unless you specify an absolute path to the driver. This is to prevent the loading of arbitrary files from the current directory as drivers.

Since the drivers are dynamically loaded on start-up 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 start-up, whereupon 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 that you should use to access it using **cryptDeviceOpen**.

Some devices don't implement all of their crypto functionality in the device but instead emulate it in software on the host PC. If you have a PKCS #11 module that does then it's better to use cryptlib's native crypto capabilities because they'll be more efficient than those in the driver and possibly more secure as well, depending on how carefully the driver has been written. In order to use only the real device capabilities (rather than those emulated on the host PC), you can set the configuration option CRYPT_OPTION_DEVICE_PKCS11_HARDWAREONLY to true (any nonzero value) as explained in "Working with Configuration Options" on page 292. If this option is set, cryptlib will only use capabilities that are provided by the crypto token any not any that are emulated in software.

Accessing PKCS #11 Devices

PKCS #11 devices are identified by the device name, for example the Litronix PKCS #11 driver identifies itself as "Litronix CryptOki Interface" so you would create a device object of this type with:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, cryptUser /* CRYPT_UNUSED */,
   CRYPT_DEVICE_PKCS11, "Litronix CryptOki Interface" );
```

If you don't know the device name or there's only one device present, you can use the special device name [Autodetect] to have cryptlib auto-detect the device for you. If there's more than one device present, cryptlib will use the first one it finds:

```
CRYPT_DEVICE cryptDevice;

cryptDeviceOpen( &cryptDevice, cryptUser /* CRYPT_UNUSED */,
   CRYPT_DEVICE_PKCS11, "[Autodetect]" );
```

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 that 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, cryptUser /* CRYPT_UNUSED */,
   CRYPT_DEVICE_PKCS11, "Litronix CryptOki Interface::Signing smart
   card " );
```

Some PKCS #11 devices and drivers have special-case requirements that need to be taken into account when you use them. For example some removable tokens may require special handling for token changes if the reader doesn't support automatic insertion detection, some drivers may have problems if the application forks (under Unix), and so on. You should consult the vendor documentation for the crypto device and drivers that you'll be using to check for any special requirements that you need to meet when you use the device.

TPMs

cryptlib supports TPMs as crypto devices, however unlike other device types like PKCS #11 devices TPMs have extremely limited capabilities, being re-purposed smart cards intended for use for DRM rather than standard crypto devices. TPMs can only store a handful of keys and an equally small number of certificates, with a lot of processing being emulated by software on the host PC, making the end result no different from running a standard software implementation. For example Windows Bitlocker, despite appearing to use TPMs for encryption, only ever encrypts and decrypts a single 16- or 32-byte value, the Volume Master Key (VMK), using the TPM's Storage Root Key (SRK). The VMK and the Full Volume Encryption Key (FVEK) are stored and used outside the TPM, with all processing being done in software on the host system. If the TPM were used to perform the decryption needed to load Windows, the system would take around three weeks to boot.

In addition to this, since TPMs are re-purposed smart cards intended for DRM use rather than standard crypto devices they don't act as general-purpose devices but need extensive manual configuration in order to make them work. This typically takes the form of multi-page XML scripts that configure the TPM, and the functionality that cryptlib uses then needs to be matched to how the TPM has been configured since there's no mechanism for automatically querying what's available. For example TPMs refer to objects such as keys by hardcoded paths that may or may not include algorithm parameters and other options, with the details being specific to the TPM and/or driver being used. In addition TPMs only support a small subset of algorithms and specific key sizes, sometimes requiring trial-and-error to figure out what's available.

Since there's no way to know in advance what the various restrictions imposed by the TPM will be, you'll need to adapt the options in `device/tpm_pkc.c` to match the specific configuration of your TPM. In addition because TPMs are repurposed smart cards emulating a crypto device, some operations that look like they're supported, typically ones around provisioning and key creation, are similarly emulated and may fail in unusual ways when the emulation metaphor breaks down. As with PKCS #11 tokens backed by smart cards, it's best to use the vendor's tools to provision the device rather than relying on the TPM software stack to be able to correctly do it. So the process for setting up a TPM would be:

1. Use the vendor tools to provision the TPM and set up keys.
2. Use the same vendor tools to verify that the keys are set up correctly and can be used.
3. Copy the device-specific information configured by the vendor tools such as object paths into `device/tpm.c` and `device/tpm_pkc.c`.
4. Use the provisioned keys through cryptlib's TPM interface.

CryptoAPI

The following section is intended for forwards-compatibility with future versions of cryptlib. Although some portions of this interface may be implemented, you shouldn't rely on them in applications.

The CryptoAPI interface provides access to the encryption, signature, and hashing capabilities of the underlying CryptoAPI implementation. All of these facilities are already provided by cryptlib, so it's primary purpose is to provide access to PKCS #12/PFX private keys and certificates held in Windows' internal (proprietary) key store, and by extension keys imported to it from other applications. Using the CryptoAPI interface provides full access to all keys generated by and stored inside Windows, while still allowing the use of all standard cryptlib functionality and facilities.

Since CryptoAPI is a software implementation managed entirely by the host operating system, there is no need to perform any initialisation, user authentication, or other operations like zeroisation, when using a CryptoAPI device. Initialisation was performed when the operating system was installed, and authentication is performed when the user logs in or the dæmon or service that uses the keys is activated. This means that using the CryptoAPI device consists of no more than creating the device object and then utilising it in subsequent crypto operations. All keys and certificates that are accessed through the device will be ones stored in CryptoAPI, giving cryptlib full access to the host operating system's keys and crypto capabilities.

On-Chip Encryption Hardware and Crypto Cores

Alongside its support for external cryptographic devices cryptlib also provides a general-purpose interface to the cryptographic capabilities present in a number of the processors typically used in embedded systems. This allows you to use cryptlib as a high-level security abstraction layer for the raw encryption hardware in the device, interfacing with it at the level of S/MIME or TLS rather than hardware registers and DMA transfers. Interfacing your custom crypto hardware to cryptlib requires creating a lightweight shim to provide a hardware abstraction layer (HAL) between cryptlib's native capabilities and your hardware. You can find a sample crypto HAL in the file devices/hw_dummy.c, which provides the shim for an implementation of a sample block cipher, AES, a sample public-key encryption/signature algorithm, RSA, and a sample hash function, SHA-1, as well as hardware key storage for the algorithms used.

Your HAL needs to provide five functions for cryptlib to call, one to initialise the hardware `hwInitialise()`, one to return a list of the hardware capabilities `hwGetCapabilities()`, and three other functions to handle key storage and cryptographic random number generation `hwGetRandom()`, `hwLookupItem()` and `hwDeleteItem()`.

Key Storage and Personalities

Encryption, signing, and MAC keys are typically stored in the crypto hardware and can't be accessed outside it. cryptlib abstracts these through the concept of personalities, where a personality corresponds to an encryption, signing, or MAC key and any associated metadata like certificates. The keys are held in the crypto hardware in whatever format the hardware requires, and everything else is managed by cryptlib.

In order to identify a personality cryptlib uses an opaque identifier of up to 20 bytes in length that your HAL needs to be able to map to an integer handle that's used to refer to that personality in future operations. You can use as little or as many of the 20 bytes in your lookup as your require.

In response to a request to locate a personality based on its opaque identifier your personality should return a handle (an integer value) for each distinct personality (key) contained in the hardware, which it uses in crypto operations to identify the personality or key to use for that operation rather than having to pass in the full-length identifier for each operation. These handles work just like file handles and are an opaque value that your HAL uses internally but that cryptlib just treats as an arbitrary integer to pass to the HAL.

The personality-lookup operation is performed using the function `hwLookpuItem()`:

```
status = hwLookupItem( keyID, keyIDlength, &keyHandle );
```

If the lookup succeeds the function returns status CRYPT_OK and sets `keyHandle` to the handle value to be used in future operations. If it fails it returns an appropriate status code, for example CRYPT_ERROR_NOTFOUND if the personality isn't present in the crypto hardware. Future crypto operations are then performed using the keyHandle, following the pattern '`fileHandle = open( … ); read( fileHandle, … );`'.

More details on implementing your own HAL can be found in the sample file `devices/hw_dummy.c`.

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 "CRYPT_QUERY_INFO Structure" on page 354.

You can interrogate cryptlib about the details of a particular encryption algorithm and mode using **cryptQueryCapability**:

```
CRYPT_QUERY_INFO cryptQueryInfo;

cryptQueryCapability( algorithm, &cryptQueryInfo );
```

If you just want to check whether a particular algorithm is available (without obtaining further information on them), you can set the query information parameter to null:

```
cryptQueryCapability( algorithm, NULL );
```

This will simply return a status value without trying to return algorithm information.

Working with Configuration Options

In order to allow extensive control over its security and operational parameters, cryptlib provides a configuration database that can be used to tune its operation for different environments using portable configuration files that 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 1536-or 2048 bit public keys on a company-wide basis instead of unsafe 768-bit keys), 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 that affect encryption parameter settings are automatically applied by cryptlib to operations such as key generation and data encryption and signing.

The configuration database can be used to tune the way cryptlib works, with options ranging from algorithms and key sizes through to 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:

Note that these values may change over time as new algorithms are introduced. The default settings for the parameters can be informally stated as "the strongest mechanism that provides good interoperability with other implementations", which is a moving target (for example the default encryption algorithm for many years was triple DES, while now it's AES). If you absolutely must use a particular algorithms or mechanism then you should explicitly set it yourself rather than relying on a default value remaining constant over many years.

Value	Type	Description
CRYPT_OPTION_CERT_-SIGNUNRECOGNISED-ATTRIBUTES	B	Whether to sign a certificate containing unrecognised attributes. If this option is set to false, the attributes will be omitted from the certificate when it is signed. Default = false.
CRYPT_OPTION_CERT_-COMPLIANCELEVEL	N	The amount of checking for standards-compliance to apply to certificates, certificate requests, and other certificate objects. Default = CRYPT_-COMPLIANCELEVEL_-STANDARD,
CRYPT_OPTION_CERT_-REQUIREPOLICY	B	Whether to require matching certificate policies for certificates in a cert chain once a CA sets a policy. Default = true.
CRYPT_OPTION_CERT_-UPDATEINTERVAL	N	The update interval in days for CRLs. Default = 90.
CRYPT_OPTION_CERT_-VALIDITY	N	The validity period in days for certificates. Default = 365.
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). Default = true.
CRYPT_OPTION_-CONFIGCHANGED	B	Whether any configuration options have been changed from their original settings (see note below).
CRYPT_OPTION_DEVICE_-PKCS11_DVR01 ... CRYPT_OPTION_DEVICE_-PKCS11_DVR05	S	The module names of any PKCS #11 drivers that cryptlib should load on start-up.
CRYPT_OPTION_DEVICE_-PKCS11_HARDWAREONLY	B	Whether cryptlib should use only the hardware capabilities of the device and not capabilities emulated in software on the host PC by the PKCS #11 driver. Default = false.
CRYPT_OPTION_ENCR_ALGO	N	Encryption algorithm given as a conventional-encryption CRYPT_ALGO_TYPE. Default = CRYPT_ALGO_AES.
CRYPT_OPTION_ENCR_HASH	N	Hash algorithm given as a hash CRYPT_ALGO_TYPE. Default = CRYPT_ALGO_SHA2.

Value	Type	Description
CRYPT_OPTION_ENCR_-HASHPARAM	N	Optional hash algorithm parameter for variable-width algorithms like SHA-2. Default = 32 bytes / 256 bits for SHA-2/256.
CRYPT_OPTION_ENCR_MAC	N	MAC algorithm given as a MAC CRYPT_ALGO_TYPE. Default = CRYPT_ALGO_HMAC_-SHA2.
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.
CRYPT_OPTION_KEYING_ALGO	N	Key processing algorithm given as a hash CRYPT_ALGO_TYPE. Default = CRYPT_ALGO_SHA2.
CRYPT_OPTION_KEYING_-ITERATIONS	N	Number of times to iterate the key-processing algorithm. Note that key processing when used for private-key encryption uses a much higher value than this general-purpose value. Default depends on CPU speed, typically 10,000 – 50,000.
CRYPT_OPTION_KEYS_LDAP_-CACERTNAME CRYPT_OPTION_KEYS_LDAP_-CERTNAME CRYPT_OPTION_KEYS_LDAP_-CRLNAME CRYPT_OPTION_KEYS_LDAP_-EMAILNAME CRYPT_OPTION_KEYS_LDAP_-FILTER 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 start-up until the initialisation has completed. Default = true.

Value	Type	Description
CRYPT_OPTION_MISC_-SIDECHANNELPROTECTION	B	Whether to perform additional operations that add protection against some obscure (and rather unlikely) side-channel attacks on private keys. Enabling this option will slow down all private-key operations by up to 10%. Default = false.
CRYPT_OPTION_NET_HTTP_-PROXY	S	HTTP proxy used for accessing web pages. Default = none.
CRYPT_OPTION_NET_SOCKS_-SERVER CRYPT_OPTION_NET_SOCKS_-USERNAME	S	Socks server and user name used for Internet access. Default = none.
CRYPT_OPTION_NET_-CONNECTTIMEOUT CRYPT_OPTION_NET_-READTIMEOUT CRYPT_OPTION_NET_-WRITETIMEOUT	N	Timeout in seconds when connecting to a remote server and when transferring data after a connection has been established. Default = 30 seconds for the connect timeout, 0 seconds for the read timeout, 2 seconds for the write timeout.
CRYPT_OPTION_PKC_ALGO	N	Public-key algorithm given as a public-key CRYPT_ALGO_-TYPE. Default = CRYPT_-ALGO_RSA.
CRYPT_OPTION_PKC_KEYSIZE	N	Public-key encryption key size in bytes. Default = 192 (1536 bits).
CRYPT_OPTION_SELFTESTOK	B	The current algorithm self-test status (see note below).

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 that 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 or flash memory. Writing this value back to FALSE forces the current in-memory values to be committed to permanent storage so that the two match up again.

CRYPT_OPTION_SELFTESTOK also has special significance, controlling cryptlib's built-in self-test functionality. If you want to perform a self-test of cryptlib's algorithms and crypto mechanisms you can set this attribute to TRUE to trigger the self-test. To obtain the status of the completed tests, read back the CRYPT_OPTION_SELFTESTOK attribute, a value of TRUE indicates that the tests completed successfully:

```
/* Initiate the self-test */
cryptSetAttribute( cryptEnvelope, CRYPT_OPTION_SELFTESTOK, TRUE );

/* Check the self-test status */
status = cryptGetAttribute( CRYPT_UNUSED, CRYPT_OPTION_SELFTESTOK,
   &value );
if( cryptStatusError( status ) || value != TRUE )
   /* Self-test failed*/;
```

In addition to these manually-triggered self-tests cryptlib automatically tests its built-in SHA1, SHA2, and AES implementation and random number generator every time that it starts, and won't start if there's a problem with any of them since they're fundamental algorithms used in most of the security protocols supported by cryptlib.

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_TYPE cryptAlgo;

cryptGetAttribute( CRYPT_UNUSED, CRYPT_OPTION_ENCR_ALGO, &cryptAlgo );
```

To set the default encryption algorithm to CAST-128 you would use:

```
cryptSetAttribute( CRYPT_UNUSED, CRYPT_OPTION_ENCR_ALGO,
   CRYPT_ALGO_CAST );
```

Some configuration options which contain values that apply to individual objects can also be set for that one object type rather than as a global setting. These options include timeouts for session objects, key size and key setup parameters for encryption contexts, and encryption and hash algorithms for envelopes. For example to set the encryption algorithm to be used when enveloping data in one particular envelope to IDEA you would use:

```
cryptSetAttribute( cryptEnvelope, CRYPT_OPTION_ENCR_ALGO,
   CRYPT_ALGO_IDEA );
```

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/cryptlib.p15
DOS OS/2	./cryptlib.p15
MVS VM/CMS	CRYPTLIB P15
Tandem	$system.system.cryptlib
Windows 3.x	Windows/cryptlib/cryptlib.p15
Windows Windows CE	\Documents and Settings\user_name\Application Data\cryptlib\cryptlib.p15 or \Windows\All Users\Application Data\cryptlib\cryptlib.p15 or \Windows\Profiles\user_name\Application Data\cryptlib.p15 or \Users\user_name\AppData\-Roaming (this varies depending on the OS type and version, and is determined by the Windows application data CSIDL)

Someone who can manipulate the cryptlib configuration file can change the way that cryptlib operates through it, so if you consider this to be a threat then you should ensure that only authorised users can access and modify the file. 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 with NTFS the file ACLs 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 that you can use to obtain information about the version of cryptlib that you're working with.

These options are:

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 3.1 the major version number is 3 and the minor version number is 1. For beta release 2 the stepping is 2.
CRYPT_OPTION_INFO_-DESCRIPTION	S	A text string containing a description of cryptlib.
CRYPT_OPTION_INFO_-COPYRIGHT	S	The cryptlib copyright notice.

The cryptlib Threat Model

cryptlib makes certain assumptions about the environment in which it operates, most of which are common-sense ones such as an attacker not having operating-system-level control of the system on which cryptlib is running. In terms of trust boundaries, cryptlib assumes that data like cryptlib keysets and configuration files stored on the system can only be modified by a source trusted at the same level that cryptlib is operating at. The remaining aspects of cryptlib's threat model are given below.

Manipulation of configuration files

cryptlib protects its configuration data with OS-level security mechanisms such as ACLs on the configuration file and the directory that contains it, however an attacker may be able to bypass this and modify the configuration data. Because of this cryptlib treats it as a potentially untrusted data source in terms of the checks that it applies to the contents, and won't use the configuration data file on a multi-user system if it's world-writeable.

Most of the configuration options are preferences that have little to no security impact, however the configuration data can also store trusted CA certificates to which an attacker that can bypass OS-level security measures could add otherwise untrusted certificates.

Manipulation of private-key keysets

As with configuration data, cryptlib protects native private-key keysets with OS-level security mechanisms such as ACLs. In addition the private keys themselves are protected using authenticated encryption and the public-key data is cryptographically bound to the corresponding private key data.

Some key-related metadata such as identification information isn't protected outside of the OS-level security mechanisms, but it's unclear what an attacker who can bypass these controls would achieve by changing, for example, the label on a key.

Non-cryptlib native keysets like PGP and PKCS #12 that come from an external source don't have this same level of protection since they're not created or controlled by cryptlib. cryptlib assumes that the user has verified the data in them before it uses it.

Manipulation of public-key databases

cryptlib stores certificates in database keysets implemented through a range of database software, and can optionally fetch them from external storage locations such as LDAP directories and HTTP URLs. These are treated as generally untrusted data stores (at the discretion of the user, who may choose to trust a local access-controlled database), requiring certificate verification via the user's preferred PKI mechanism.

cryptlib manages CA operations through a trusted CA database keyset, which is distinct from a standard database keyset. Since this controls the issuance of certificates by the CA, an attacker who can manipulate it can compromise the CA's operations, as well as altering logs to hide certificate mis-issuance and other malfeasance.

Attacks on encryption algorithms and protocols

cryptlib implements a range of encryption algorithms and security protocols. Without writing an entire book on the topic it's not possible to enumerate every possible threat, both real and imagined, to these. To mitigate unexpected surprises, cryptlib takes an extremely conservative approach to implementing and deploying new algorithms and protocols, and the developers keep abreast of publications of new attacks and act to mitigate them if and as required.

cryptlib protects cryptovariables and cryptographic operations but can't protect against calling application or user misuse, for example use of hardcoded credentials, posting credentials to Github, failing to verify and/or ignoring failed signature or MAC operations, and so on.

Manipulation of entropy sources

cryptlib utilises a large number of diverse entropy sources to seed (and continuously reseed) the PRNG used for the generation of cryptovariables. An attacker who can control or manipulate those sources can degrade the amount of entropy present in the PRNG. However in order to manipulate enough of these sources to cause a problem an attacker would need either complete control of the system on which cryptlib is running or the ability to modify cryptlib's operation, at which point they can simply compromise cryptlib directly.

Spoofing attacks on multiuser systems

cryptlib uses absolute paths when loading drivers and shared libraries, as well as mitigating other common risks like DLL/shared library hijacking, use of symlinks, handle overloading, and so on.

Malicious content in data fields

cryptlib treats all content as opaque `( value, length )` pairs and shouldn't be affected by maliciously-created data fields like ones containing control characters, embedded nul characters, and similar. When reporting problems, cryptlib will sanitise values in error messages to only display ASCII characters.

Since cryptlib passes the data through to the caller as opaque blobs, applications that expect to see (for example) valid text strings may not be able to handle unexpected malicious content.

Sensitive content in error messages

When built with error messages enabled, cryptlib returns text-form error information alongside the standard error status codes. cryptlib will never return sensitive information such as passwords or cryptovariables in an error message, however other information on the exact nature of the error may be of use to an attacker if they gain access to it.

Denial of Service attacks

cryptlib enforces strict bounds on both data values and running time, with static upper bounds for all data items and loop iterations. It should not be possible for an attacker to feed cryptlib malformed data that causes it to go into an infinite (or even excessively long) loop or induce excessive memory consumption. Conversely, cryptlib also takes steps to mitigate slowloris-type attacks that induce a denial-of-service through very low data rates.

Since the operation of a cryptlib client or server is controlled by the calling application, an application that doesn't rate-limit the number of server instances that it starts, busy-waits on network I/O, or performs other suboptimal network operations, can still be vulnerable to a denial of service attack.

Stateless nature of cryptlib

cryptlib is stateless, so that an instantiation of an object such as an SSH server is independent from any other instantiation of that server object. This means that defences like rate-limiting in conventional stateful servers will be undermined if the calling application recycles server objects at a rapid rate.

Data vs. control interface access

cryptlib contains two interfaces to the world, the control interface used by the calling application and the data interface that processes data coming from external, typically untrusted sources. These can be thought of as the internal interface, which handles commands from the calling application, and the external interface, which handles data from external sources. cryptlib assumes that the control interface is trusted, in other words that the calling application won't act maliciously, while treating the the data interface as untrusted and potentially malicious.

An attacker who can compromise the control interface via the calling application can compromise security, for example by setting malicious or unsafe attribute values or by exfiltrating plaintext data after decryption.

Faults

cryptlib contains extensive mitigations for both memory and code faults, including checksumming of cryptovariables, pairwise consistency tests of important crypto operations (for example each signature generation is followed by a signature verification), double-indexing of all loop variables with a consistency check on each loop iteration, checksumming of all non-ephemeral pointers to detect pointer corruption, control-flow integrity checks of important encryption-related operations, checksumming of bitflags and use of complex values for booleans to ensure that a single bitflip can't affect operations, overflow-checking integer maths where possible, explicit checking of all parameters passed to any function including the contents of complex values like structures, and the checks mentioned below in the buffer overflow section. cryptlib also provides the option to store critical data in TMR storage, users interested in this capability should contact the cryptlib developers.

Some compilers, most notably gcc, will remove some of these checks or rewrite the code to render them ineffective. Users concerned with operation in environments where faults are expected or the code needs to operate continuously over long periods without restarts/rejuvenation should build the code with a compiler that preserves its semantics as much as possible.

Buffer overflows

Alongside the fault-mitigation checks, cryptlib bounds-checks all length values that refer to data, as well as bounds-checking all memory-manipulation operations. Allocated data buffers are bounded by canaries that are checked for buffer under/overruns, and the checks mentioned above in the faults section are applied to data values relating to buffers.

Since cryptlib returns data values to the caller as `( value, length )` pairs, callers who allocate fixed-length buffers for data without performing a length check may be vulnerable to buffer overflows if cryptlib returns more data than will fit in the fixed-length buffer.

Side-channel Attacks

Although cryptlib takes care to mitigate software side-channels, it will never be possible to eliminate them all, as the neverending stream of side-channel conference papers show. On the other hand there has never been a single recorded case of a real-world attacker using a software side-channel attack because every other attack is much, much easier to perform and far more likely to succeed. For this reason cryptlib does not consider software side-channel attacks as a legitimate threat until such time as they're actually demonstrated by real-world attackers instead of conference papers.

Supply chain attacks

cryptlib is hosted in a public Github repository, however this is only a copy of the master copy of the code which is held on isolated systems and saved to write-once media. This means that any compromise of the code in the repository can only ever affect a copy but not the original.

An attacker who can compromise the Github repository will be able to distribute modified copies of the code until the compromise is detected and the code replaced with a known-good copy.

Unsafe behaviour mandated by standards

Many security standards encourage or even mandate unsafe behaviour in implementations, typically through specifying excessively general or unnecessarily complex operations as part of the specification. Examples of this are TLS' use of a state machine that can be exploited in attacks and SSH's unnecessarily complex negotiation of authentication and other protocol details after the crypto handshake has completed. cryptlib mitigates this by recasting the TLS state machine into a safe form as a ladder diagram that only allows one path through the process, and severely limiting the operations allowed during an SSH handshake to the minimum required in order to authenticate safely.

As a variation of this, the security protocols that cryptlib implements that are designed by standards bodies typically have an attack surface that ranges from moderate through to extremely large. This isn't because of the cryptography that's used, which no attacker bothers with, but with the often extraordinarily complex mechanisms that surround it and the awkward data formats that typically accompany them. To try and deal with this, cryptlib interprets the standards in a manner that limits their generality as much as possible, implementing the minimum required for interoperability in order to limit the amount of freedom that attackers have to exploit the complexity of the underlying protocols. It also disables by default protocol capabilities that are likely to be unsafe, one example being in SSH for which connection multiplexing, port forwarding, and subsystems are disabled by default unless the high-risk `USE_SSH_EXTENDED` configuration option is explicitly enabled.

In some cases this may lead to cryptlib rejecting connections or data provided by implementations that take advantage of peculiarities in the standard. These issues can be addressed by the developers on a case-by-case basis.

In terms of the protocols implemented by cryptlib, the ranking by attack surface from largest to smallest is:

- SSH: SSH messages mix binary data and comma-delimited text strings, and the protocol allows complex back-and-forth negotiations and state changes during the handshake process. SSH has the largest attack surface of any protocol that cryptlb implements, and if extended functionality is enabled by building with the `USE_SSH_EXTENDED` define mentioned above the attack surface is extended even further.

- TLS 1.3: TLS 1.3 reinvents standard TLS using extensions in the client hello, which requires complex reconciliation to resolve all of the protocol details up-front. This has the second-largest attack surface of any cryptlib protocol.
- PGP: PGP's packet format strings data packets together in an arbitrary fashion, with the packets themselves being composed of a sequence of implicitly-typed data fields without any encapsulation. This makes parsing of messages, and figuring out what's coming next, difficult, and leads to a considerable attack surface.
- TLS classic: TLS before TLS 1.3 has a standard request/response negotiation that cryptlib implements as a ladder diagram as mentioned above. This makes it relatively safe and reduces the attack surface significantly.
- CMS / S/MIME: CMS uses an encoding format that can be statically type-checked as the data is read, making it the safest of the protocols supported by cryptlib. This also extends implicitly to the various PKI protocols that use the same format.

Unsafe behaviour dictated by audit tools

Vulnerability scanners and similar checkbox-based security auditing systems will report every issue that they know about, including ones that aren't actually real vulnerabilities (see the section "Quantum computing" below for one example). This creates two problems when they report a non-vulnerability, the first being that it may trigger a governance process that requires a risk management exercise resulting in a written report to explain that there's no risk, and the second being that a standard response in order to silence the scanner is to disable the crypto mechanism or protocol that it's triggering on, resulting in a fallback to a far less secure mechanism. An example of this is the fact that at one point a small number of buggy TLS implementations got a crypto mechanism called DHE trivially wrong, resulting in some implementations disabling it altogether and falling back to RSA instead, the least secure key exchange mechanism there is.

Unfortunately there is no easy fix for this problem. One option would be to have cryptlib detect the operation of a particular audit tool and report only what the tool wants to see. If you're experiencing this type of problem please contact the cryptlib developers.

Quantum computing

As with early computers being imaginatively described as "electronic brains", so the devices marketed as "quantum computers" are actually physics experiments that have been specially set up to confirm a single already-known result. In particular, these experiments rely on the use of so-called sleight-of-hand numbers, values specifically chosen to make them practical with a physics experiment (such values are never used in actual cryptography). At the time of writing, after more than twenty years of work and hundreds of millions of dollars spent, the total number of integer factorisations using Shor's algorithm of a value where the result wasn't known and pre-set in advance, the benchmark used in press releases for quantum cryptanalysis, is zero (again, cryptanalysis is built on the premise that the attacker doesn't already know the solution before they begin).

In addition the problem that needs to be solved to attack security protocols like SSH and TLS, alongside various others like Signal, WhatsApp, WireGuard, and many more, isn't integer factorisation but a completely different problem called the discrete logarithm problem, which no-one has ever claimed to be able to attack because they haven't managed to construct the required sleight-of-hand numbers to set up a physics experiment for it. Going further, no-one even knows *how* one would set up a physics experiment to target the values used in SSH, TLS, and similar.

The threat from PQC then isn't of an actual attack but of regulatory non-compliance, complicated by the fact that the requirements for PQC are still being debated by standards bodies so the field is in a state of flux, and that more than half of all proposed PQC algorithms have already been broken with no doubt more breaks to come in the future. Because of this, cryptlib will implement the PQC algorithms required for regulatory compliance but in line with its conservative deployment of new, unproven cryptography will disable them by default to avoid unpleasant surprises in the future.

Threats out of Scope

There are a number of attacks published in conference papers that assume extremely unrealistic conditions, for example one that requires that the user encrypt nearly 800 gigabytes of carefully-chosen data supplied for them by an attacker all under the same encryption key in order to recover an HTTP cookie. Although it's difficult to explicitly specify what constitutes a realistic threat and what doesn't, unrealistic attacks like this aren't considered part of the threat model.

cryptlib defaults to a configuration that's as secure as reasonably possible, however since it's supplied in source code form it's possible through the creation of custom builds, for example via code changes or changes in the build process, to create insecure configurations. These custom configurations are outside the scope of the threat model.

Random Numbers

Several cryptlib functions require access to a source of cryptographically strong random numbers. 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 randomness 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 );
```

In addition to user-supplied and built-in randomness sources, cryptlib will check for a `/dev/random`, EGD, or PRNGD-style style randomness driver (which continually accumulates random data from the system) and will use this as a source of randomness. If running on a system with a hardware random number source (provided by some CPUs and chipsets), cryptlib will also make use of the hardware random number source. cryptlib can also make use of additional entropy seeding information on embedded systems without inherent entropy sources, see "Embedded Systems" on page 324 for more information.

cryptlib includes in its built-in generator an ANSI X9.17 / ANSI X9.31 generator for FIPS 140 certification purposes. Full technical details of the generator are given in the reference in "Recommended Reading" on page 13.

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 that returns immediately and retrieves a moderate amount of random information, and a slow poll that may take some time but that 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 and Windows CE 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 function such as envelopes, secure sessions, 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 for use by cryptlib. On most systems cryptlib will perform a non-blocking randomness poll, so you can usually do this by calling the slow poll routine when your program starts. This ensures that the random information will have accumulated by the time you need it:

```
/* Program start-up */

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 that 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 that produces byte-oriented output (for example a block cipher employed in a stream mode like CFB). 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 what the buffer initially contains. For example you can use the AES algorithm in CFB mode to generate random data with:

```
CRYPT_CONTEXT cryptContext;

cryptCreateContext( &cryptContext, cryptUser /* CRYPT_UNUSED */,
   CRYPT_ALGO_AES );
cryptSetAttribute( cryptContext, CRYPT_CTXINFO_MODE, CRYPT_MODE_CFB );
cryptGenerateKey( cryptContext )
cryptEncrypt( cryptContext, randomDataBuffer, randomDataLength );
cryptDestroyContext( cryptContext );
```

This will fill the data buffer with the required number of random bytes.

Working with Newer Versions of cryptlib

Your software can automatically support new encryption algorithms as they are added to cryptlib if you check for the range of supported algorithms instead of hard-coding in the values that 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_TYPE cryptAlgo;

for( cryptAlgo = CRYPT_ALGO_FIRST_CONVENTIONAL;
     cryptAlgo <= CRYPT_ALGO_LAST_CONVENTIONAL;
     cryptAlgo++ )
   if( cryptStatusOK( cryptQueryCapability( cryptAlgo, 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, hash, and MAC algorithms.

If your code follows these guidelines, it will automatically handle any new encryption algorithms that are added in newer versions of cryptlib. If you are using the shared library or DLL form of cryptlib, your software's 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 each of the built-in encryption algorithms can perform a self-test procedure that 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, cryptUser
   /* CRYPT_UNUSED */, CRYPT_ALGO_IDEA );
if( cryptStatusError( status ) )
   /* Perform error processing */;
```

The error codes that can be returned are grouped into a number of classes that 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 that 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 attribute that 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 TLS).
CRYPT_ERROR_-NOTINITED	The object or attribute that 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_-COMPLETE	An operation that consists of multiple steps (such as a message hash) is complete and cannot be continued.
CRYPT_ERROR_-INCOMPLETE	An operation that 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 or attribute is invalid for this type of operation.
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 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 attribute).
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 that had been loaded or derived from the data on the smart card would return CRYPT_ERROR_-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_-TIMEOUT	The operation timed out, either because of a general timeout while accessing an object such as a network connection or data file, or because the object was in use for another operation such as a key database lookup.
CRYPT_ERROR_-WRONGKEY	The key being used to decrypt or verify the signature on a piece of data is incorrect.

The next group contains errors related to the higher-level encryption functions such as enveloping, secure session, and 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 an enveloping function, you need to call **cryptPopData** before you can add any more data to the envelope.
	For a certificate function this means the amount of data you have supplied is more than what is allowed for the field you are trying to store it in.
	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**).

Error code	Description
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 or session 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) couldn't be opened, either because it wasn't found or because the open operation failed.
CRYPT_ERROR_-READ	The requested item couldn't 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 that you can encounter. This is particularly true for complex objects like certificates, keysets, envelopes, and secure sessions. In order to handle the more complex error conditions, cryptlib supports two types of extended error information, one for errors arising from operations on object attributes and one for errors arising from use of the object itself. For example if you try to set a certificate or envelope attribute and get an error then you'd look at the attribute-related extended error information, and if you try to envelope or de-envelope data and get an error then you'd look at the object-operation extended error information.

Extended error information on attribute-related errors are reported through the CRYPT_ATTRIBUTE_ERRORLOCUS attribute, which reports the locus of the error (the attribute that caused the problem) and the CRYPT_ATTRIBUTE_-ERRORTYPE attribute, which identifies the type of problem that 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 type and locus information relates to errors with attribute-related object usage that have been identified by cryptlib. In addition keyset, envelope, and session objects can often provide text error messages describing problems encountered while performing an operation. The error message text is accessible through the CRYPT_-ATTRIBUTE_ERRORMESSAGE attribute.

The attribute-related 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_ERRTYPE_-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_-ISSUERCONSTRAINT	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.

To obtain more information on why an attempt to sign a certificate failed you would use:

```
CRYPT_ATTRIBUTE_TYPE errorLocus;
CRYPT_ERRTYPE_TYPE errorType;

status = cryptSignCert( cryptCertificate, cryptCAKey );
if( cryptStatusError( status ) )
   {
   cryptGetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_ERRORLOCUS,
      &errorLocus );
   cryptGetAttribute( cryptCertificate, CRYPT_ATTRIBUTE_ERRORTYPE,
      &errorType );
   }
```

To obtain more information on why an attempt to read a key from a key 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, errorStringLength;
   char *errorString;

   errorString = malloc( ... );
   cryptGetAttributeString( cryptKeyset, CRYPT_ATTRIBUTE_ERRORMESSAGE,
      errorString, &errorStringLength );
   }
```

Note that in some cases the error message being returned is passed up from underlying software or hardware, and will be specific to the implementation. For example if the software that underlies a keyset database is SQL Server then the data returned will be an SQL Server error message. Since the returned data contains low-level error information coming from the underlying software and will often be information provided by a third-party or remote client or server application, the contents of the error message can vary somewhat but the will typically contain some indication of what the problem is.

In some cases an object access or operation will be blocked by the cryptlib security kernel and never get to the object itself. This typically occurs when cryptlib returns a CRYPT_ERROR_PERMISSION error, in which the kernel has prevented a disallowed access type. In this case none of the extended error information will be set, since the object never saw the access attempt.

Complete Code Samples

This section contains more complete code examples than the excerpts that are provided throughout the manual. These perform specific (although somewhat common) tasks, so that you can use them as the basis for further code. Note though that they're intended as code samples to illustrate one way of accomplishing a particular task, if you want to perform something a bit more specific than what's illustrated here then you should consult the relevant section of the manual that covers what you want to do.

Utility Functions

The first few code sample are utility functions that are used in the rest of the code. **createDeviceSession** opens a session with a crypto device like a smart card or HSM:

```
int createDeviceSession( CRYPT_DEVICE *userDevice,
                         const char *devicePassword )
   {
   CRYPT_DEVICE cryptDevice;
   int status;

   /* Clear the return value */
   *userDevice = -1;

   /* Open a session with the device */
   status = cryptDeviceOpen( &cryptDevice, CRYPT_UNUSED,
      CRYPT_DEVICE_PKCS11, "[Autodetect]" );
   if( cryptStatusError( status ) )
      return( status );

   /* Log on to the device */
   status = cryptSetAttributeString( cryptDevice,
      CRYPT_DEVINFO_AUTHENT_USER, devicePassword,
      strlen( devicePassword ) );
   if( cryptStatusError( status ) )
      {
      cryptDeviceClose( cryptDevice );
      return( status );
      }

   /* Return the device object to the caller */
   *userDevice = cryptDevice;

   return( CRYPT_OK );
   }
```

publishPublicKey publishes a user's public key and/or certificate to a public-key keyset for use by anything that requires access to a collection of public keys/certificates. This assumes that the keyset has already been created and initialised using CRYPT_KEYOPT_CREATE:

```
int publishPublicKey( const char *publicKeysetName,
                      const CRYPT_CERTIFICATE cryptCertificate )
   {
   CRYPT_KEYSET cryptKeyset;
   char name[ 128 ];
   int nameLen, status;

   /* Open a connection to the public-key keyset */
   status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
      CRYPT_KEYSET_DATABASE, publicKeysetName, CRYPT_KEYOPT_NONE );
   if( cryptStatusError( status ) )
      return( status );

   /* Clear any existing certificates under this name that may already
      be present.  This is useful for the sample code when re-running
      it multiple times and creating new keys on each run */
   status = cryptGetAttributeString( cryptCertificate,
      CRYPT_CERTINFO_COMMONNAME, name, &nameLen );
   if( cryptStatusOK( status ) )
      {
      name[ nameLen ] = '\0';
      ( void ) cryptDeleteKey( cryptKeyset, CRYPT_KEYID_NAME, name );
      }

   /* Publish the certificate to the keyset */
   status = cryptAddPublicKey( cryptKeyset, cryptCertificate );
   cryptKeysetClose( cryptKeyset );

   return( status );
   }
```

getPublicKey fetches a user's public key/certificate from a file keyset or device:

```
int getPublicKey( CRYPT_CERTIFICATE *userCertificate,
                  const char *pubKeysetName,
                  const char *pubKeyLabel,
                  const char *devicePassword )
   {
   CRYPT_CERTIFICATE cryptCertificate;
   int status;

   /* Clear the return value */
   *userCertificate = -1;

   /* Fetch the public key/certificate.  If a filename is given then
      we assume that the source is a public-key keyset, otherwise it's
      a device */
   if( pubKeysetName != NULL )
      {
      CRYPT_KEYSET cryptKeyset;

      /* Fetch the public key/certificate from the keyset */
      status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
         CRYPT_KEYSET_DATABASE, pubKeysetName,
         CRYPT_KEYOPT_READONLY );
      if( cryptStatusOK( status ) )
         {
         status = cryptGetPublicKey( cryptKeyset, &cryptCertificate,
            CRYPT_KEYID_NAME, pubKeyLabel );
         cryptKeysetClose( cryptKeyset );
         }
      }
   else
      {
      CRYPT_DEVICE cryptDevice;

      /* Fetch the public key/certificate from the device */
      status = createDeviceSession( &cryptDevice, devicePassword );
      if( cryptStatusOK( status ) )
         {
         status = cryptGetPublicKey( cryptDevice, &cryptCertificate,
            CRYPT_KEYID_NAME, pubKeyLabel );
         cryptDeviceClose( cryptDevice );
         }
      }
   if( cryptStatusError( status ) )
      return( status );

   /* Return the public key/certificate to the caller */
   *userCertificate = cryptCertificate;

   return( CRYPT_OK );
   }
```

getPrivateKey fetches a user's private key from a file keyset or device:

```
int getPrivateKey( CRYPT_CONTEXT *privateKeyContext,
                   const char *privKeysetName,
                   const char *privKeyLabel,
                   const char *privKeyPassword )
   {
   CRYPT_CONTEXT cryptContext;
   int status;

   /* Clear the return value */
   *privateKeyContext = -1;
```

```
    /* Fetch the private key.  If a filename is given then we assume
       that the source is a file keyset, otherwise it's a device */
    if( privKeysetName != NULL )
       {
       CRYPT_KEYSET cryptKeyset;

       /* Fetch the private key from the keyset */
       status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
          CRYPT_KEYSET_FILE, privKeysetName, CRYPT_KEYOPT_READONLY );
       if( cryptStatusOK( status ) )
          {
          status = cryptGetPrivateKey( cryptKeyset, &cryptContext,
             CRYPT_KEYID_NAME, privKeyLabel, privKeyPassword );
          cryptKeysetClose( cryptKeyset );
          }
       }
    else
       {
       CRYPT_DEVICE cryptDevice;

       /* Fetch the private key from the device */
       status = createDeviceSession( &cryptDevice, privKeyPassword );
       if( cryptStatusOK( status ) )
          {
          status = cryptGetPrivateKey( cryptDevice, &cryptContext,
             CRYPT_KEYID_NAME, privKeyLabel, NULL );
          cryptDeviceClose( cryptDevice );
          }
       }
    if( cryptStatusError( status ) )
       return( status );

    /* Return the private key to the caller */
    *privateKeyContext = cryptContext;

    return( CRYPT_OK );
    }
```

createClientSession creates a client network session:

```
 int createClientSession( CRYPT_SESSION *clientSession,
                          const CRYPT_SESSION_TYPE sessionType,
                          const char *serverName )
    {
    CRYPT_SESSION cryptSession;
    int status;

    /* Clear the return value */
    *clientSession = -1;

    /* Create a client session */
    status = cryptCreateSession( &cryptSession, CRYPT_UNUSED,
       sessionType );
    if( cryptStatusError( status ) )
       return( status );

    /* Add the name of the server to connect to */
    status = cryptSetAttributeString( cryptSession,
       CRYPT_SESSINFO_SERVER_NAME, serverName, strlen( serverName ) );
    if( cryptStatusError( status ) )
       {
       cryptDestroySession( cryptSession );
       return( status );
       }

    /* Return the session object to the caller */
    *clientSession = cryptSession;

    return( CRYPT_OK );
    }
```

createServerSession creates a server network session:

```
int createServerSession( CRYPT_SESSION *serverSession,
                         const CRYPT_SESSION_TYPE sessionType,
                         const char *privKeysetName,
                         const char *privKeyLabel,
                         const char *privKeyPassword )
   {
   CRYPT_SESSION cryptSession;
   CRYPT_CONTEXT cryptContext;
   int status;

   /* Clear the return value */
   *serverSession = -1;

   /* Create a server session */
   status = cryptCreateSession( &cryptSession, CRYPT_UNUSED,
      sessionType );
   if( cryptStatusError( status ) )
      return( status );

   /* Add the server's key to the session */
   status = getPrivateKey( &cryptContext, privKeysetName,
      privKeyLabel, privKeyPassword );
   if( cryptStatusOK( status ) )
      {
      /* Add the key (and associated certificate) to the server
         session */
      status = cryptSetAttribute( cryptSession,
         CRYPT_SESSINFO_PRIVATEKEY, cryptContext );
      cryptDestroyContext( cryptContext );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Return the session object to the caller */
   *serverSession = cryptSession;

   return( CRYPT_OK );
   }
```

Key/Certificate Examples

The next set of code samples use the utility functions above to create keys and certificates. **createSimplifiedCert** creates a simplified certificate for a user with the given name, optional email address, and optional server DNS name. Note that this isn't a standard CA-issued certificate but merely one intended for use as a convenient key container:

```
int createSimplifiedCert( CRYPT_CERTIFICATE *userCertificate,
                          const CRYPT_CONTEXT certKey,
                          const char *certOwnerName,
                          const char *certOwnerEmail,
                          const char *certOwnerDNSName )
   {
   CRYPT_CERTIFICATE cryptCertificate;
   int status;

   /* Clear the return value */
   *userCertificate = -1;

   /* Create a certificate and mark it as a simlified certificate */
   status = cryptCreateCert( &cryptCertificate, CRYPT_UNUSED,
      CRYPT_CERTTYPE_CERTIFICATE );
   if( cryptStatusError( status ) )
      return( status );
   status = cryptSetAttribute( cryptCertificate, CRYPT_CERTINFO_XYZZY,
      1 );
   if( cryptStatusError( status ) )
      {
      cryptDestroyCert( cryptCertificate );
      return( status );
      }
   /* Add the public key */
   status = cryptSetAttribute( cryptCertificate,
      CRYPT_CERTINFO_SUBJECTPUBLICKEYINFO, certKey );
   if( cryptStatusError( status ) )
      {
      cryptDestroyCert( cryptCertificate );
      return( status );
      }

   /* Add the various user identification components */
   status = cryptSetAttributeString( cryptCertificate,
      CRYPT_CERTINFO_COMMONNAME, certOwnerName,
      strlen( certOwnerName ) );
   if( cryptStatusOK( status ) && certOwnerEmail != NULL )
      {
      status = cryptSetAttributeString( cryptCertificate,
         CRYPT_CERTINFO_EMAIL, certOwnerEmail,
         strlen( certOwnerEmail ) );
      }
   if( cryptStatusOK( status ) && certOwnerDNSName != NULL )
      {
      status = cryptSetAttributeString( cryptCertificate,
         CRYPT_CERTINFO_DNSNAME, certOwnerDNSName,
         strlen( certOwnerDNSName ) );
      }
   if( cryptStatusOK( status ) )
      {
      /* Working with alternative name components like email addresses
         and DNS names moves the certificate cursor away from the DN,
         in case access to the DN will be required later on we move it
         back */
      status = cryptSetAttribute( cryptCertificate,
         CRYPT_ATTRIBUTE_CURRENT, CRYPT_CERTINFO_SUBJECTNAME );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyCert( cryptCertificate );
      return( status );
      }

   /* Sign the certificate with the private key */
   status = cryptSignCert( cryptCertificate, certKey );
   if( cryptStatusError( status ) )
      {
      cryptDestroyCert( cryptCertificate );
      return( status );
      }

   /* Return the newly-created certificate to the caller */
   *userCertificate = cryptCertificate;

   return( CRYPT_OK );
   }
```

generateKey generates a key and simplified certificate for a user with the given name, optional email address, and optional server DNS name. There are two versions of this, one that uses a standard file keyset and a second that uses an encryption device:

```
int generateKey( const char *keyOwnerName, const char *keyOwnerEmail,
                 const char *keyOwnerDNSName, const char *keyLabel,
                 const char *privKeysetName,
                 const char *privKeyPassword,
                 const char *pubKeysetName )
   {
   CRYPT_CONTEXT cryptContext;
   CRYPT_CERTIFICATE cryptCertificate;
   CRYPT_KEYSET cryptKeyset;
   int status;

   /* Generate a key */
   status = cryptCreateContext( &cryptContext, CRYPT_UNUSED,
      CRYPT_ALGO_RSA );
   if( cryptStatusError( status ) )
      return( status );
   status = cryptSetAttributeString( cryptContext,
      CRYPT_CTXINFO_LABEL, keyLabel, strlen( keyLabel ) );
   if( cryptStatusOK( status ) )
      status = cryptGenerateKey( cryptContext );
   if( cryptStatusError( status ) )
      {
      cryptDestroyContext( cryptContext );
      return( status );
      }

   /* Create a simplified certificate for it */
   status = createSimplifiedCert( &cryptCertificate, cryptContext,
      keyOwnerName, keyOwnerEmail, keyOwnerDNSName );
   if( cryptStatusError( status ) )
      {
      cryptDestroyContext( cryptContext );
      return( status );
      }

   /* Create a new keyset and write the key and certificate to it */
   status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
      CRYPT_KEYSET_FILE, privKeysetName, CRYPT_KEYOPT_CREATE );
   if( cryptStatusOK( status ) )
      {
      status = cryptAddPrivateKey( cryptKeyset, cryptContext,
         privKeyPassword );
      if( cryptStatusOK( status ) )
         status = cryptAddPublicKey( cryptKeyset, cryptCertificate );
      cryptKeysetClose( cryptKeyset );
      }
   if( cryptStatusOK( status ) )
      status = publishPublicKey( pubKeysetName, cryptCertificate );

   /* Clean up */
   cryptDestroyContext( cryptContext );
   cryptDestroyCert( cryptCertificate );

   return( status );
   }

int generateKeyDevice( const char *keyOwnerName,
                       const char *keyOwnerEmail,
                       const char *keyOwnerDNSName,
                       const char *keyLabel,
                       const char *devicePassword,
                       const char *pubKeysetName )
   {
   CRYPT_CONTEXT cryptContext;
   CRYPT_CERTIFICATE cryptCertificate;
   CRYPT_DEVICE cryptDevice;
   int status;

   /* Open a session with the device */
   status = createDeviceSession( &cryptDevice, devicePassword );
   if( cryptStatusError( status ) )
      return( status );

   /* Clear any existing key(s) with this label that may already be
      present in the device.  As is the case when publishing the
      public key to a public-key keyset, this is useful when re-
      running the sample code multiple times */
   ( void ) cryptDeleteKey( cryptDevice, CRYPT_KEYID_NAME, keyLabel );

   /* Generate a key in the device */
   status = cryptDeviceCreateContext( cryptDevice, &cryptContext,
      CRYPT_ALGO_RSA );
   if( cryptStatusError( status ) )
      {
      cryptDeviceClose( cryptDevice );
      return( status );
      }
   status = cryptSetAttributeString( cryptContext,
      CRYPT_CTXINFO_LABEL, keyLabel, strlen( keyLabel ) );
   if( cryptStatusOK( status ) )
      status = cryptGenerateKey( cryptContext );
```

```
	if( cryptStatusError( status ) )
		{
		cryptDestroyContext( cryptContext );
		cryptDeleteKey( cryptDevice, CRYPT_KEYID_NAME, keyLabel );
		cryptDeviceClose( cryptDevice );
		return( status );
		}

	/* Create a simplified certificate for it */
	status = createSimplifiedCert( &cryptCertificate, cryptContext,
		keyOwnerName, keyOwnerEmail, keyOwnerDNSName );
	if( cryptStatusError( status ) )
		{
		cryptDestroyContext( cryptContext );
		cryptDeleteKey( cryptDevice, CRYPT_KEYID_NAME, keyLabel );
		cryptDeviceClose( cryptDevice );
		return( status );
		}

	/* Update the device with the certificate */
	status = cryptAddPublicKey( cryptDevice, cryptCertificate );
	if( cryptStatusOK( status ) )
		status = publishPublicKey( pubKeysetName, cryptCertificate );

	/* Clean up */
	cryptDestroyContext( cryptContext );
	cryptDestroyCert( cryptCertificate );
	if( cryptStatusError( status ) )
		cryptDeleteKey( cryptDevice, CRYPT_KEYID_NAME, keyLabel );
	cryptDeviceClose( cryptDevice );

	return( status );
	}
```

Encryption/Decryption Examples

The next set of code samples encrypt and decrypt data. **encryptMessage** public-key encrypts data in a block of memory for the given recipient:

```
int encryptMessage( const void *inData, const int inDataLength,
                    void *outData, const int outDataMaxLength,
                    int *outDataLength, const char *pubKeysetName,
                    const char *recipientName,
                    const CRYPT_FORMAT_TYPE formatType )
	{
	CRYPT_ENVELOPE cryptEnvelope;
	CRYPT_KEYSET cryptKeyset;
	int bytesCopied, status;

	/* Clear the return value */
	*outDataLength = 0;

	/* Create an envelope to wrap the data */
	status = cryptCreateEnvelope( &cryptEnvelope, CRYPT_UNUSED,
		formatType );
	if( cryptStatusError( status ) )
		return( status );
	status = cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
		inDataLength );
	if( cryptStatusError( status ) )
		{
		cryptDestroyEnvelope( cryptEnvelope );
		return( status );
		}

	/* Add the public-key keyset and recipient name to the envelope */
	status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
		CRYPT_KEYSET_DATABASE, pubKeysetName, CRYPT_KEYOPT_READONLY );
	if( cryptStatusOK( status ) )
		{
		status = cryptSetAttribute( cryptEnvelope,
			CRYPT_ENVINFO_KEYSET_ENCRYPT, cryptKeyset );
		cryptKeysetClose( cryptKeyset );
		}
	if( cryptStatusOK( status ) )
		{
      status = cryptSetAttributeString( cryptEnvelope,
         CRYPT_ENVINFO_RECIPIENT, recipientName,
         strlen( recipientName ) );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Push the data to be encrypted into the envelope */
   status = cryptPushData( cryptEnvelope, inData, inDataLength,
      &bytesCopied );
   if( cryptStatusOK( status ) )
      {
      if( bytesCopied != inDataLength )
         status = -1; /* Not all data was processed */
      else
         status = cryptFlushData( cryptEnvelope );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Pop the encrypted result back out */
   status = cryptPopData( cryptEnvelope, outData, outDataMaxLength,
      outDataLength );
   cryptDestroyEnvelope( cryptEnvelope );

   return( status );
   }
```

decryptMessage decrypts a block of memory:

```
int decryptMessage( const void *inData, const int inDataLength,
                    void *outData, const int outDataMaxLength,
                    int *outDataLength, const char *privKeysetName,
                    const char *privKeyPassword )
   {
   CRYPT_ENVELOPE cryptEnvelope;
   int bytesCopied, status;

   /* Clear the return value */
   *outDataLength = 0;

   /* Create an envelope to unwrap the data */
   status = cryptCreateEnvelope( &cryptEnvelope, CRYPT_UNUSED,
      CRYPT_FORMAT_AUTO );
   if( cryptStatusError( status ) )
      return( status );

   /* Add the decryption keyset to the envelope.  If a filename is
      given then we assume that it's a file keyset, otherwise it's a
      device */
   if( privKeysetName != NULL )
      {
      CRYPT_KEYSET cryptKeyset;

      /* Add the decryption keyset to the envelope */
      status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
         CRYPT_KEYSET_FILE, privKeysetName, CRYPT_KEYOPT_READONLY );
      if( cryptStatusOK( status ) )
         {
         status = cryptSetAttribute( cryptEnvelope,
            CRYPT_ENVINFO_KEYSET_DECRYPT, cryptKeyset );
         cryptKeysetClose( cryptKeyset );
         }
      }
   else
      {
      CRYPT_DEVICE cryptDevice;

      /* Add the decryption device to the envelope */
      status = createDeviceSession( &cryptDevice, privKeyPassword );
      if( cryptStatusOK( status ) )
         {
         status = cryptSetAttribute( cryptEnvelope,
            CRYPT_ENVINFO_KEYSET_DECRYPT, cryptDevice );
         cryptDeviceClose( cryptDevice );
         }
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Push the encrypted data into the envelope */
   status = cryptPushData( cryptEnvelope, inData, inDataLength,
      &bytesCopied );
   if( cryptStatusOK( status ) && bytesCopied != inDataLength )
      status = -1;    /* Not all data was processed */
   if( status == CRYPT_ENVELOPE_RESOURCE )
      {
      int cryptEnvInfo;

      /* Find out what's required in order to continue */
      status = cryptGetAttribute( cryptEnvelope,
         CRYPT_ATTRIBUTE_CURRENT, &cryptEnvInfo );
      if( cryptStatusOK( status ) )
         {
         switch( cryptEnvInfo )
            {
            case CRYPT_ATTRIBUTE_NONE:
               /* We're using a device for decryption, there's nothing
                  left to do since decryption has been automatically
                  handled by the device */
               break;

            case CRYPT_ENVINFO_PRIVATEKEY:
               /* We're using a keyset for decryption, we need to
                  supply the password to decrypt the private key */
               status = cryptSetAttributeString( cryptEnvelope,
                  CRYPT_ENVINFO_PASSWORD, privKeyPassword,
                  strlen( privKeyPassword ) );
               break;

            default:
               /* Something unexpected happened */
               status = -1;
            }
         }
      }
   if( cryptStatusOK( status ) )
      status = cryptFlushData( cryptEnvelope );
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Pop the decrypted result back out */
   status = cryptPopData( cryptEnvelope, outData, outDataMaxLength,
      outDataLength );
   cryptDestroyEnvelope( cryptEnvelope );

   return( status );
   }
```

Signing/Verification Examples

The next set of code samples sign and verify data. **signMessage** signs data in a block of memory:

```
int signMessage( const void *inData, const int inDataLength,
                 void *outData, const int outDataMaxLength,
                 int *outDataLength, const char *privKeysetName,
                 const char *privKeyLabel,
                 const char *privKeyPassword,
                 const CRYPT_FORMAT_TYPE formatType )
```

```
   {
   CRYPT_ENVELOPE cryptEnvelope;
   CRYPT_CONTEXT cryptContext;
   int bytesCopied, status;

   /* Clear the return value */
   *outDataLength = 0;

   /* Create an envelope to sign the data */
   status = cryptCreateEnvelope( &cryptEnvelope, CRYPT_UNUSED,
      formatType );
   if( cryptStatusError( status ) )
      return( status );
   status = cryptSetAttribute( cryptEnvelope, CRYPT_ENVINFO_DATASIZE,
      inDataLength );
   if( cryptStatusOK( status ) && inDataLength > 16384 )
      {
      status = cryptSetAttribute( cryptEnvelope,
         CRYPT_ATTRIBUTE_BUFFERSIZE, inDataLength + 8192 );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Add the signing key to the envelope */
   status = getPrivateKey( &cryptContext, privKeysetName,
      privKeyLabel, privKeyPassword );
   if( cryptStatusOK( status ) )
      {
      status = cryptSetAttribute( cryptEnvelope,
         CRYPT_ENVINFO_SIGNATURE, cryptContext );
      cryptDestroyContext( cryptContext );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Push the data to be signed into the envelope */
   status = cryptPushData( cryptEnvelope, inData, inDataLength,
      &bytesCopied );
   if( cryptStatusOK( status ) )
      {
      if( bytesCopied != inDataLength )
         status = -1; /* Not all data was processed */
      else
         status = cryptFlushData( cryptEnvelope );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Pop the signed result back out */
   status = cryptPopData( cryptEnvelope, outData, outDataMaxLength,
      outDataLength );
   cryptDestroyEnvelope( cryptEnvelope );

   return( status );
   }
```

sigCheckMessage signature-checks data in a block of memory:

```
int sigCheckMessage( const void *inData, const int inDataLength,
                     void *outData, const int outDataMaxLength,
                     int *outDataLength, const char *pubKeysetName,
                     int *sigResult )
   {
   CRYPT_ENVELOPE cryptEnvelope;
   CRYPT_KEYSET cryptKeyset;
   int bytesCopied, status;
   /* Clear the return values */
   *outDataLength = 0;
   *sigResult = -1;

   /* Create an envelope to sig.check the data */
   status = cryptCreateEnvelope( &cryptEnvelope, CRYPT_UNUSED,
      CRYPT_FORMAT_AUTO );
   if( cryptStatusError( status ) )
      return( status );
   if( inDataLength > 16384 )
      {
      status = cryptSetAttribute( cryptEnvelope,
         CRYPT_ATTRIBUTE_BUFFERSIZE, inDataLength + 8192 );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Add the public-key keyset that's used for signature verification
      to tne envelope.  This is optional, if we're using an S/MIME
      signature then the signing cerificates will be included with the
      signed message */
   status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
      CRYPT_KEYSET_DATABASE, pubKeysetName, CRYPT_KEYOPT_READONLY );
   if( cryptStatusOK( status ) )
      {
      status = cryptSetAttribute( cryptEnvelope,
         CRYPT_ENVINFO_KEYSET_SIGCHECK, cryptKeyset );
      cryptKeysetClose( cryptKeyset );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroyEnvelope( cryptEnvelope );
      return( status );
      }

   /* Push the signed data into the envelope */
   status = cryptPushData( cryptEnvelope, inData, inDataLength,
      &bytesCopied );
   if( cryptStatusOK( status ) )
      {
      if( bytesCopied != inDataLength )
         status = -1; /* Not all data was processed */
      else
         status = cryptFlushData( cryptEnvelope );
      }
      if( cryptStatusError( status ) )
         {
         cryptDestroyEnvelope( cryptEnvelope );
         return( status );
         }

   /* Pop the sig-checked result back out */
   status = cryptPopData( cryptEnvelope, outData, outDataMaxLength,
      outDataLength );
   if( cryptStatusOK( status ) )
      {
      status = cryptGetAttribute( cryptEnvelope,
         CRYPT_ENVINFO_SIGNATURE_RESULT, sigResult );
      }
   cryptDestroyEnvelope( cryptEnvelope );

   return( status );
   }
```

Certificate Processing Examples

The next set of code samples work with certificates. **checkCertValidity** checks the validity of a certificate:

```
int checkCertValidity( const char *serverName,
                       const char *keyFileName,
                       const char *devicePassword,
                       const char *keyLabel, int *certStatus )
```

```
{
CRYPT_CERTIFICATE cryptCertificate;
CRYPT_SESSION cryptSession;
int status;

/* Clear the return value */
*certStatus = -1;

/* Create a client session */
status = createClientSession( &cryptSession, CRYPT_SESSION_RTCS,
   serverName );
if( cryptStatusError( status ) )
   return( status );

/* Fetch the certificate to be checked */
status = getPublicKey( &cryptCertificate, keyFileName, keyLabel,
   devicePassword );
if( cryptStatusError( status ) )
   {
   cryptDestroySession( cryptSession );
   return( status );
   }

/* Check the certificate's validity using the RTCS server */
status = cryptCheckCert( cryptCertificate, cryptSession );

/* Clean up */
cryptDestroyCert( cryptCertificate );
cryptDestroySession( cryptSession );

return( status );
}
```

runCertValServer runs a certificate-validation server:

```
int runCertValServer( const char *pubKeysetName,
                      const char *privKeysetName,
                      const char *privKeyLabel,
                      const char *privKeyPassword )
   {
   CRYPT_SESSION cryptSession;
   CRYPT_KEYSET cryptKeyset;
   int status;

   /* Create a server session and add the server's key to it */
   status = createServerSession( &cryptSession,
      CRYPT_SESSION_RTCS_SERVER, privKeysetName, privKeyLabel,
      privKeyPassword );
   if( cryptStatusError( status ) )
      return( status );

   /* Add the certificate keyset to tne envelope */
   status = cryptKeysetOpen( &cryptKeyset, CRYPT_UNUSED,
      CRYPT_KEYSET_DATABASE, pubKeysetName, CRYPT_KEYOPT_READONLY );
   if( cryptStatusOK( status ) )
      {
      status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_KEYSET,
         cryptKeyset );
      cryptKeysetClose( cryptKeyset );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Activate the session */
   status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
      1 );
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Clean up */
   cryptDestroySession( cryptSession );
   return( status );
   }
```

Client/Server Examples

The finals set of code samples work with secure client/server sessions. **fetchFromSecureServer** fetches data from a secure server:

```
int fetchFromSecureServer( const char *serverName,
                           const char *command, void *outData,
                           const int outDataMaxLength,
                           int *outDataLength )
   {
   CRYPT_SESSION cryptSession;
   int bytesCopied, status;

   /* Clear the return value */
   *outDataLength = 0;

   /* Create a client session */
   status = createClientSession( &cryptSession, CRYPT_SESSION_TLS,
      serverName );
   if( cryptStatusError( status ) )
      return( status );

   /* Activate the session */
   status = cryptSetAttribute( cryptSession, CRYPT_SESSINFO_ACTIVE,
      1 );
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Send a command to the remote system */
   status = cryptPushData( cryptSession, command, strlen( command ),
      &bytesCopied );
   if( cryptStatusOK( status ) )
      {
      if( bytesCopied != ( int ) strlen( command ) )
         status = -1; /* Not all data was processed */
      else
         status = cryptFlushData( cryptSession );
      }
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Read back the response */
   status = cryptPopData( cryptSession, outData, outDataMaxLength,
      outDataLength );
   cryptDestroySession( cryptSession );

   return( status );
   }
```

runSecureServer runs a secure server:

```
int runSecureServer( const char *privKeysetName,
                     const char *privKeyLabel,
                     const char *privKeyPassword, void *outData,
                     const int outDataMaxLength, int *outDataLength )
   {
   CRYPT_SESSION cryptSession;
   int status;

   /* Clear the return value */
   *outDataLength = 0;

   /* Create a server session and add the server's key to it */
   status = createServerSession( &cryptSession,
      CRYPT_SESSION_TLS_SERVER, privKeysetName, privKeyLabel,
      privKeyPassword );
   if( cryptStatusError( status ) )
      return( status );

   /* Activate the session */
   status = cryptSetAttribute( cryptSession,
                             CRYPT_SESSINFO_ACTIVE, 1 );
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Read a command from the client */
   status = cryptPopData( cryptSession, outData, outDataMaxLength,
      outDataLength );
   if( cryptStatusError( status ) )
      {
      cryptDestroySession( cryptSession );
      return( status );
      }

   /* Possible further operations here */
   /* … */

   cryptDestroySession( cryptSession );

   return( status );
   }
```

Embedded Systems

cryptlib has been designed to be usable in embedded designs that lack many facilities that are normally found on standard systems, both in terms of resources (memory, network I/O) and in system functionality (a filesystem, dynamic memory allocation). If you're running in a resource-constrained environment such as an embedded system, you first need to decide what cryptlib services you require and disable any unnecessary options, as described in "Customised and Cut-down cryptlib Versions" on page 25. This will reduce the cryptlib code footprint to the minimum required for your particular situation.

As a general rule of thumb if you're on a resource-constrained system you should turn off anything that uses networking, which includes secure sessions (USE_-SESSIONS), and HTTP and LDAP keyset access (USE_HTTP, USE_LDAP). You probably also want to turn off crypto devices, (USE_PKCS11 and USE_-FORTEZZA), since the embedded system is unlikely to have PKCS #11 crypto hardware attached to it. You probably won't be using database keysets (USE_-DBMS), and unless you're using PGP keyrings you can turn that off as well (USE_-PGPKEYS). PGP keyrings are particularly problematic because their structure requires that they be processed via a lookahead buffer because it's not possible to determine how much more data associated with the key is to follow. If you're running in a memory-constrained environment and are thinking of using PGP keys, you should consider using the PKCS #15 format (the cryptlib native keyset type) instead, since this doesn't have this problem.

For envelopes, you probably want to turn off compressed enveloping (USE_-COMPRESSION) since zlib needs to allocate a series of fairly sizeable buffers in order to operate (256KB for compression, 32KB for decompression, compared to only 8KB used for the envelope buffer itself). If you're not using PGP, you can turn that off as well (USE_PGP). Finally, there are a considerable range of other options that you can turn off to save memory and space, see misc/config.h for more details.

In addition to the code size tuning and if you're targeting a new embedded system that isn't already supported by cryptlib, you need to make any necessary system-specific adaptations to cryptlib to match the characteristics of the embedded device that you're working with. These adaptations are described following the discussion of supported systems below.

Embedded OS Types

Many embedded OSes, and in particular real-time OSes (RTOSes) are highly modular and are typically heavily customised to suit particular applications, with entire sections of the RTOS added or removed depending on configuration option settings. In addition each one will typically be running on custom hardware with a wide range of features, capabilities, peripherals, BSP variants, and build options. Because of the individuality of each hardware configuration and the high degree of customisability of each embedded OS cryptlib out of the box provides a generic-as-possible setup that should build across as many variants of each embedded OS/RTOS as possible. As a result the default cryptlib build for a particular embedded OS/RTOS uses a fairly minimal set of OS capabilities that should work in many configurations. If you have extended OS facilities available you can use the cryptlib configuration file `misc/config.h` to enable any additional capabilities that you may need.

It's a good idea to contact the cryptlib developers before you build cryptlib on one of the more modular, configurable embedded OSes since the wide range of configurations and hardware variants means that in some cases additional work may be required to accommodate specific configurations. Notes for individual OSes are given below.

AMX

AMX is a highly configurable kernel with most functionality set to the minimum level in the default build in order to conserve space. To run cryptlib you need to enable the time/date manager (so that cryptlib can check datestamps on the data that it's processing), and time-slicing if you're running multiple tasks within cryptlib. For task synchronisation cryptlib uses AMX semaphores, but doesn't require any further AMX facilities like mailboxes, event groups, or buffer pools.

ARINC 653

The ARINC 653 specification defines the APEX API that provides the kernel services used by cryptlib. No special operational considerations are required for cryptlib in this environment.

ChorusOS

ChorusOS provides a standard Posix file API and BSD sockets networking API that matches the one used by cryptlib's generic Unix configuration. No special operational considerations are required for cryptlib in this environment.

CMSIS-RTOS/mbed-rtos

CMSIS-RTOS/mbed-rtos is an embedded kernel targeted at restricted environments employing ARM Cortex processors. In its standard configuration it doesn't provide a networking stack or a filesystem, so the default .build disables file I/O and networking, and all associated services such as secure sessions and keyset storage. If you're using a Software Pack that provides either filesystem or network support then you need to enable this through the appropriate configuration options as described in "Embedded cryptlib Configuration Options" on page 328.

CMX

CMX provides the kernel services used by cryptlib. No special operational considerations are required for cryptlib in this environment.

DOS

DOS isn't strictly speaking an embedded OS but its facilities are limited enough that for cryptlib's purposes it functions as one. Since standard real-mode DOS has very little memory available, you should shrink the object table (via CONFIG_-CONSERVE_MEMORY and/or CONFIG_NUM_OBJECTS) to the smallest size that you can work with. In addition you should disable all unused functionality to conserve as much code and data space as possible. Finally, since DOS has no reliable entropy source, you should use the CONFIG_RANDSEED mechanism to enable the use of an external random number seed file.

eCOS

Unlike most other embedded OSes, eCOS requires that all data structures used by the kernel be statically allocated by the user. This means that cryptlib has to allocate storage for all semaphores, mutexes, tasks, and other eCOS objects either at compile time or (at the latest) when it's loaded/initialised. This entails allocating the storage required by eCOS for each object when cryptlib allocates its kernel object table, rather than allocating the storage on-demand when an object is created. If memory is at a premium, you should shrink the object table (via CONFIG_CONSERVE_-MEMORY and/or CONFIG_NUM_OBJECTS) to the smallest size that you can work with, since each object entry has to include space for eCOS kernel data.

Typical eCOS configurations include a full TCP/IP stack and file I/O services. cryptlib uses the Posix section 5/6 file I/O layer, the universal low-level I/O layer that's supported by all filesystem drivers. The TCP/IP stack is a standard BSD-derived stack and it's use is enabled by default in the eCOS build.

Embedded Linux

Embedded Linux is a standard Unix environment. No special operational considerations are required for cryptlib in this environment.

embOS

embOS provides a file API via the emFile interface and a networking API via the IP interface that's used by cryptlib. No special operational considerations are required for cryptlib in this environment.

FreeRTOS/OpenRTOS

FreeRTOS is an embedded realtime kernel that, in its standard configuration, doesn't provide a networking stack or a filesystem, so the default .build disables file I/O and networking, and all associated services such as secure sessions and keyset storage. If your FreeRTOS configuration contains either filesystem or network support then you need to enable this through the appropriate configuration options as described in "Embedded cryptlib Configuration Options" on page 328.

µITRON

µITRON has a file interface (ITRON/FILE) derived from the BTRON persistent object store interface, but the only documentation for this is for BTRON and it's only available in Japanese. Because of the inability to obtain either documentation or an implementation to code against, cryptlib only contains stubs for file I/O functionality. If your µITRON system provides this file interface, please contact the cryptlib developers.

µITRON also has a TCP/IP interface, but it doesn't seem to be widely used and the only documentation available is in Japanese. Because of this the use of TCP/IP under µITRON is disabled by default in misc/config.h, if you have a µITRON TCP/IP implementation you can use it to replace the existing TCP/IP interface in io/tcp.c.

Mongoose OS

Mongoose OS provides a standard Posix-style file API that matches the one used by cryptlib's generic Unix configuration. No special operational considerations are required for cryptlib in this environment.

QNX Neutrino

QNX Neutrino is a standard Unix environment, and in general no special operational considerations are required for cryptlib in this environment. The one exception is in the choice of networking environments. QNX Neutrino provides three network stack configurations, the standard TCP/IP stack, the enhanced TCP/IP stack, and a low-resource version of the standard stack. cryptlib works with all of these stacks, and will try and use the most sophisticated features provided by the system. If you're using one of the more restricted networking stacks (for example the tiny TCP/IP stack with no IPv6 support) you may need to change the settings in io/tcp.h to reflect this.

RTEMS

RTEMS provides a standard Posix file API and BSD sockets networking API that matches the one used by cryptlib's generic Unix configuration. No special operational considerations are required for cryptlib in this environment.

SMX

SMX provides a standard file API via the smxFS interface that matches the one used by cryptlib's generic Unix configuration. No special operational considerations are required for cryptlib in this environment.

ThreadX

ThreadX provides a standard file API via the FileX interface and BSD sockets networking API that matches the one used by cryptlib's generic Unix configuration. No special operational considerations are required for cryptlib in this environment.

uClinux

uClinux is an embedded OS intended for use on hardware without memory protection, allowing it to be run on systems that couldn't otherwise run a standard Linux build. To conserve memory, you may want to configure uClinux to use the `page_alloc2/kmalloc2` allocator instead of the somewhat wasteful standard power-of-two Linux allocator, which is intended for use on systems with virtual memory support. cryptlib's memory allocation strategy fits neatly with the `page_alloc2` allocator to minimise memory usage.

By default the uClinux toolchains tend to allocate extremely small stacks of only 4KB, which is inadequate for all but the most trivial applications. To provide an adequate stack, you need to either set `FLTFLAGS=-s` *`stacksize`* and export `FLTFLAGS` to the makefile before building your application, or run flthdrs –s *stacksize* on your executable after building it.

µC/OS-II

To run cryptlib under µC/OS-II you need to enable mutexes (`OS_MUTEX_EN` and `OS_MUTEX_DEL_EN`) for task synchronisation and tasks (`OS_TASK_CREATE_EN` and `OS_TASK_DEL_EN`) if you're running multiple tasks within cryptlib. µC/OS-II makes a task's priority do double duty as the task ID, so there's no way to uniquely identify a task over the long term. If you change a task's priority using `OSTaskChangePrio()`, you'll also change its task ID. This means that if you've bound a cryptlib object to a task for access control purposes (see "Object Security" on page 45), it'll no longer be accessible once the task priority change changes its task ID. If your tasks change their IDs in this manner, you shouldn't bind objects to particular task IDs.

Windows CE

Windows CE is a standard Windows environment. No special operational considerations are required for cryptlib in this environment.

VDK

Analog Devices' Visual DSP Kernel (VDK) is a minimal embedded kernel that doesn't provide a networking stack or a filesystem, so the default .build disables file I/O and networking, and all associated services such as secure sessions and keyset storage. If your VDK configuration contains either filesystem or network support then you need to enable this through the appropriate configuration options as described in "Embedded cryptlib Configuration Options" on page 328.

VxWorks

VxWorks includes a TCP/IP stack and file I/O services. cryptlib uses the ioLib file I/O mechanisms, the universal low-level I/O layer that's supported by all filesystem drivers. The VxWorks TCP/IP stack has changed somewhat over time and is sometimes replaced by more functional third-party alternatives or may not be present at all if VxWorks has been configured without it. Because of this, the use of TCP/IP services isn't enabled by default. If you need networking services, you can enable them in misc/config.h, and may need to perform VxWorks-specific network initialisation (for example calling `selectInit`) if your application doesn't already do so.

XMK

Xilinx XMK is highly configurable kernel with several functions disabled in the default build. To run cryptlib you need to enable mutexes (`config_pthread_mutex`) for thread synchronisation, the yield interface (`config_yield`) for thread management, and timers (`config_time`) for time handling. In addition if you're starting threads within cryptlib, you need to either increase the default thread stack size (`pthread_stack_size`) or set a larger stack size when you start the internal thread.

Xilinx XMK provides an emulated Posix filesystem API, however in order to reduce code size cryptlib uses the native XMK memory filesystem (MFS) interface to access stored data in RAM, ROM, or flash memory. If you need to store data such as configuration options or private keys, you need to enable MFS support in your XMK build.

XMK includes a minimal network stack (LibXilNet), however this only provides server functionality (so it's not possible to implement a network client) and doesn't support timers, so that each send or receive will block forever until data arrives or is sent. Because of these limitations, you need to use a third-party network stack in order to use cryptlib's networking capabilities under XMK.

Zephyr

Zephyr includes a TCP/IP stack and file I/O services. No special operational considerations are required for cryptlib in this environment.

Embedded cryptlib Configuration Options

You can use the standard cryptlib makefile to cross-compile the code for any of the embedded targets. If you're building for a new target type, you first need to add the new target type at the end of the makefile in the "Embedded Systems" section. The cryptlib naming convention for preprocessor symbols used to identify targets is to use *__target_name__*, which then enables system-specific behaviour in the code. For example if you were adding a new target type to build the code for an Atmel TDMI ARM core, you'd use –D__ATMEL__ as the necessary compile option (some compilers will define the necessary symbols automatically).

The cryptlib makefile and source code auto-detect various system parameters at compile time, if you're cross-compiling for a new target type that you've defined yourself you'll need to override this so that you're building with the parameters for your target rather than for the host system. In addition you can enable various build options for systems with limited resources as described earlier. The values that you may need to define to handle these system-specific options are:

Option	Description
__target_name__	The target type that you're building for.
CONFIG_LITTLE_ENDIAN CONFIG_BIG_ENDIAN	The CPU endianness of the target system.
CONFIG_CONSERVE_-MEMORY	Define if the target has limited memory available to reduce the default sizes of buffers and data structures. "Limited" means less than about 512KB of RAM.
CONFIG_CONSERVE_-MEMORY_EXTRA	Define to remove additional functionality beyond the standard CONFIG_-CONSERVE_MEMORY. In particular this will remove startup self-checks on the cryptlib kernel and other API checks, it's recommended that you enable this only if you really need every byte of memory, and run the code during development and testing without enabling this in order to catch possible problems.
CONFIG_DEBUG_MALLOC	Define to dump memory usage diagnostics to the console. You generally wouldn't use this option on the target system, but only on the host during development.

Option	**Description**
CONFIG_FILE_PATH	The base path used by cryptlib to locate its configuration file and the random number seed file. Embedded systems usually have minimal filesystems and so cryptlib will look for files in the root directory, if you have a more complex filesystem structure then you can specify the path to the required directory here.
CONFIG_NO_CERTIFICATES	Define to disable the use of certificate objects. If you define this you also need to disable the use of secure sessions, which requires certificates. Some envelope types and keysets that work with certificates will also be affected.
CONFIG_NO_DEVICES	Define to disable the use of crypto device objects.
CONFIG_NO_DYNALLOC	Define to change cryptlib's handling of on-demand memory allocation as described in "Devices without Dynamic Memory Allocation" on page 332.
CONFIG_NO_ENVELOPES	Define to disable the use of envelope objects. Some secure session types that work with envelopes will also be affected.
CONFIG_NO_ERRORMSG	Don't include long descriptive error messages in the code, which reduces code size.
CONFIG_NO_KEYSETS	Define to disable the use of keyset objects. This also disables the ability to store configuration options to persistent storage, since these are stored in a file keyset. Some secure session and envelope types that work with keysets will also be affected.
CONFIG_NO_SELFTEST	Define to disable the crypto algorithm and mechanism self-test carried out by cryptlib on initialisation. This can speed up cryptlib's startup on slower processors. Note that non-algorithm-related self tests such as basic checks on cryptlib kernel functioning are still performed, to disable these as well use CONFIG_CONSERVE_MEMORY_-EXTRA.
CONFIG_NO_SESSIONS	Define to disable the use of secure session objects.
CONFIG_NO_STDIO	Define if the target has no filesystem/stdio support.
CONFIG_NUM_OBJECTS=*n*	The number of objects that cryptlib reserves room for, defaulting to 512 without CONFIG_CONSERVE_-MEMORY defined or 64 with.

Option	Description
CONFIG_PKC_ALLOCSIZE	The amount of memory to pre-allocate for storage of public/private key components, which in turn sets CRYPT_-MAX_PKCSIZE, the maximum size of key components that can be processed. Setting this to 256 (2048-bit keys) rather than the default of 512 (4096-bit keys) will halve the amount of memory used to hold public/private key components, setting it to 128 (1024-bit keys) reduces it by 75%. Setting CONFIG_-CONSERVE_MEMORY sets CONFIG_PKC_ALLOCSIZE to 2048 unless you've explicitly set it yourself.
CONFIG_RANDSEED	Define to use external random seed data.
CONFIG_RANDSEED_-QUALITY	Define to set the value of the random seed data, as a percentage figure from 10-100 percent.
CONFIG_SLOW_CPU	Define to disables some of the more CPU-intensive self-tests that are performed on cryptlib start-up. The exact definition of a "slow" CPU is somewhat variable, but as a rule of thumb if you're using a 16-bit CPU or one clocked at under 100MHz or so then you probably want to enable this define to speed up the start-up process.

Finally, cryptlib includes a considerable amount of other configurability that you can take advantage of if you need to use it in an environment that imposes particular restrictions on resource usage. If you're working with an embedded system, you should contact the cryptlib developers with more details on any specific requirements that you may have.

Once you've got the necessary options set up, you can build the code. If you're building for a completely new target, cryptlib will detect this and print messages at the various locations in the code where you need to add system-specific adaptations such as support for reading/writing to flash memory segments in io/file.c. Alternatively, you can edit io/file.c before you try to build the code, look for all the locations where CONFIG_NO_STDIO is referenced, and add the necessary support there rather than having cryptlib warn you about it during the build process.

Debugging with Embedded cryptlib

Since you'll be using the same code on your host system as you will in the target, by far the easiest way to develop and debug your application is to do it on the host using your preferred development tools. By enabling the same build options as you would on the target (except for the CPU endianness override) you can exactly duplicate the conditions on the target embedded system and perform all of your application development on the host rather than having to cross-compile, upload code, and work with the target's debugging facilities (if there are any).

Special Considerations for Embedded Systems

Unlike standard computer systems, embedded systems often have resource limits that impose special constraints on the use of cryptlib. The three most noticeable resource issues that will affect your use of cryptlib are the need for a randomness source for key generation, the need for a real-time clock for processing items like certificates and signatures that require date stamps or verification of date stamps, and the need for a filesystem for storing keys and configuration information. A less likely problem is the lack of dynamic memory allocation, which fortunately is relatively uncommon as most embedded systems provide at least some form of memory management.

Random Data Sources

In order to generate keys for encryption and signing operations (which includes indirect key generation for higher-level protocols like PGP, S/MIME, TLS, and SSH) cryptlib needs to have a source of unpredictable random data available as described in "Random Numbers" on page 297. On many embedded systems there may not be enough random data available to safely generate these keys, so cryptlib provides the ability to provide this additional entropy through the use of the CONFIG_RANDSEED option, which enables the use of stored random data that contains additional random seed material. On embedded systems CONFIG_-RANDSEED is defined by default, in which case cryptlib will try and read the random seed data and use it as additional input to the internal randomness pool.

The seed material is stored in the same location as the cryptlib configuration data (see "Working with Configuration Options" on page 292 for more details), and isn't necessarily a file but can be a block of data in flash memory, data in battery-backed RAM, or whatever other mechanism your system uses for persistent storage. The storage location for the data is determined by your system configuration, if there's a file system available then it'll be stored in a file called randseed.dat, if not it'll be accessed via whatever persistent storage mechanism is configured for your system in io/file.c.

The random seed data must be at least 128 bits (16 bytes) long, something like 128 or 256 bytes is a better value, and should be unique and random for each device. This is extremely important since if two (or more) devices are configured with the same random seed data or if the data from one device is such that the values stored in other devices can be predicted from it then someone who compromises one device can compromise the key generation on every other device. When you build your embedded system you should install the per-device unique seed data from an external source, for example a hardware random number generator or a copy of cryptlib running on a secure system with a good source of randomness (the use of cryptlib to generate random data is covered in "Random Numbers" on page 297).

Since a significant portion of the input data for key generation will be determined by the seed data if there are no other randomness sources available (cryptlib will always get at least *some* randomness from the environment, so the value will change each time it's used), you should take as much care as possible to protect the seed data. As mentioned above you should always use different, unique seed data on each system, to prevent a compromise of one system from affecting any others. In addition if your system provides any protection mechanisms you should apply them to the seed data to try and safeguard it as much as possible. Finally, you should use the ability to add user-supplied randomness described in "Random Numbers" on page 297 to periodically add any situation-specific data that you may have available. For example if your embedded device is being used for voice or video transmission you can add segments of the compressed audio or video data, and if your device performs a sensor/monitoring function you can add the sensor data. Since most embedded devices have at least some interaction with the surrounding environment, there's usually at least some source of additional randomness available.

Once you have your seed data set up, you need to decide how much overall randomness it contributes to the system. You can set this value as a percentage between 10 and 100 percent via the CONFIG_RANDSEED_QUALITY configuration option. If you don't set a value, cryptlib will assume a figure of 80%, meaning that it needs to obtain an additional 20% of randomness from the environment before it'll generate keys. Note that this setting is merely a safety level, it doesn't mean that cryptlib will gather randomness until it reaches 100% and then stop (it never stops gathering randomness), merely that it won't generate keys when the randomness value is below 100%.

Devices without a Real-time Clock

Most high-level crypto protocols that involve digital signatures require a real-time clock, either to add a timestamp when the signature is created or to check the time on the signature when it's verified. For example certificates have a fixed validity period that has to be checked against a time source when the certificate is verified. In order to do this, cryptlib needs to have access to a correctly-set real-time clock.

Since a time source isn't necessarily available on embedded systems, cryptlib won't require that a clock be present as it would for a non-embedded system. However if you try to perform an operation that requires a time source such as creating or verifying a certificate or creating a PGP or S/MIME signature, cryptlib will return a CRYPT_ERROR_NOTAVAIL status to indicate that no time source was available for the signature-creation or checking operation. Note that if your system has a clock but it's not set correctly then signature verification-related operations will fail with time-related errors. If you plan to make use of time-related functionality then you need to ensure that there's an appropriate time source available.

The lack of a real-time clock won't affect most protocols like TLS, SSH, or PGP and S/MIME when used only for encryption, but will affect protocols that involve signatures and require timestamps. In particular running a CA or a timestamping server without a source of timestamp's isn't going to work very well.

Devices without a Filesystem

If the device you're working with lacks a filesystem, you'll need to work with io/file.c to add an adaptation layer to handle the underlying storage abstraction that you're using. In embedded devices this usually consists of blocks of flash memory or occasionally battery-backed RAM, identified either by name/label or an integer value or tag. cryptlib supports the use of named/tagged memory segments if you build it with the CONFIG_NO_STDIO option, and will assemble in-memory (RAM) pseudo-files on which it performs all I/O until the file is committed to backing store, whereupon it'll perform an atomic transfer of the pseudo-file to flash to minimise wear on the flash memory cells. It's thus possible to manipulate these (pseudo-)files arbitrarily without causing excessive wear on the underlying storage medium.

Devices without Dynamic Memory Allocation

If your system lacks dynamic memory allocation, or has so little memory that it's necessary to conserve it as much as possible, you first need to build cryptlib with the CONFIG_CONSERVE_MEMORY option. This reduces the default sizes of some buffers, and sets the initial size of cryptlib's internal object table to 128 objects instead of the usual 1024. You can further tune the amount of memory used by the system object table by setting the CONFIG_NUM_OBJECTS setting to the maximum number of objects that you'll need. This value must be a power of 2, and can't be less than 8. For single-purpose use in an embedded device (for example when used specifically for enveloping messages rather than as a general-purpose tool where anything is possible), you can usually get by with 32 or even 16 objects. Depending on other options such as whether you use certificate trust settings or not and whether your system has a 16- or 32-bit word size, the cryptlib kernel and built-in system objects consume between 6 and 12 KB of memory.

As a rough rule of thumb, each non-public-key encryption context consumes around 200 bytes (along with any extra memory needed by the algorithm's expanded encryption key), each public-key encryption context consumes around 1500 bytes (depending again on algorithm-specific parameters such as the algorithm type and key size), file keysets (which are buffered in memory as mentioned earlier) consume 600 bytes plus the size of the keyset file (usually around 1.3 KB for a standard 1024-bit RSA key and accompanying certificate and 3 KB for the key and a 3-certificate chain), envelopes consume 1.2KB plus 16 KB for enveloping and 8KB for de-enveloping (the extra size is due to the built-in envelope buffer), and certificates consume an amount of memory that isn't easily predictable in advance since they consist of an arbitrary number of arbitrarily-sized objects. This makes it very difficult to estimate their eventual memory usage, but a rule of thumb is about 2 KB used for a typical certificate. Note that the certificate object consumption has very little to do with the key size, but is mostly dependent on the number and size of all the other X.509 components that are present in the certificate.

Cryptlib Memory Management

cryptlib allocates memory in strict LIFO manner, so that creating an object and then destroying it again rolls back memory to the exact state it was in before the object was created. This ensures that it's possible to run cryptlib on a system without dynamic memory allocation by using a simple high-water-mark pointer that tracks the last memory position used, and falls back to its earlier position when the memory is "freed". Because of this memory usage strategy, cryptlib, although it does acquire memory as required, doesn't need real dynamic memory allocation and can function perfectly well if given a single fixed block of memory and told to use that.

cryptlib allocates either very little or no memory during its normal operation. That is, memory is allocated once at object creation or activation, after which cryptlib stays within the already-allocated bounds unless it encounters some object that it needs to store for later use. For example if it finds a certificate while processing S/MIME data it'll need to acquire a block of memory to store the certificate for later access by the caller.

cryptlib Memory Usage

Almost all of the information that cryptlib processes has the potential to include arbitrary-length data, and occasionally arbitrary amounts of arbitrary-length data. Certificates are a particular example of this, as mentioned earlier. cryptlib's strategy for handling these situations is to use stack memory to handle the data if possible, but if the item being processed exceeds a certain size, to temporarily grab a larger block of memory from the heap to allow the item to be processed, freeing it again immediately after use.

In normal use this overflow handling is never invoked, however since cryptlib can always run into data items of unusual size (constructed either accidentally or maliciously), you need to decide whether you want to allow this behaviour or not. Allowing it means that you can process unusual data items, but may make you vulnerable to deliberate resource-starvation attacks. Conversely, denying it makes you immune to excessive memory usage when trying to process data maliciously constructed to require extra memory to process, but will also make it impossible to process data that just happens to have unusual characteristics. In general, cryptlib will be able to process any normal data without requiring dynamically allocated memory, so if you know in advance which types of data you'll be processing and are concerned about possible resource-starvation attacks, you can disable the opportunistic allocation of larger working areas by using the CONFIG_NO_DYNALLOC build option.

cryptlib includes a number of internal lookup tables used for certificate decoding, algorithm information lookup, error parsing, and so on. These are all declared `static const` to tell the compiler to place them in the read-only code segment (held in ROM) rather than the initialised data segment (held in RAM). If your compiler doesn't automatically do this for you (almost all do), you'll need to play with compiler options to ensure that the tables are stored in ROM rather than RAM.

Many cryptlib functions store detailed error information as descriptive text strings that can be retrieved through the CRYPT_ATTRIBUTE_ERRORMESSAGE attribute. Since storage for these detailed text messages consumes ROM space, you may want to disable them to save space, or only enable them in the debug build but not the release build. To disable descriptive error messages (only error codes will be returned), define CONFIG_NO_ERRORMSG.

Tracking Memory Usage

In order to track memory usage and determine what'll be required on your target system, you can use the CONFIG_DEBUG_MALLOC option to dump diagnostics about memory usage to the console. This will allow you to see approximately how much memory a certain operation will require, and let you play with rearranging operations to reduce memory consumption. For example having two objects active simultaneously rather than using them one after the other will result in a total memory consumption equal to the sum of their sizes rather than only the size of the larger of the two objects.

The memory usage diagnostics will reveal the LIFO nature of the memory allocation that cryptlib uses to try to minimise its overall footprint. You can use the sequence numbers after each allocate and free to track the order in which things are used.

Database and Networking Plugins

In order to communicate with databases that are used as database keysets or certificate stores and with different network types, cryptlib uses a plugin interface that allows it to talk to any type of database back-end and network protocol. The database interface provides four functions that are used to interface to the back-end, two functions to open and close the connection to the back-end and two to send data to and read data from it. The network plugin interface provides five functions, two to initialise and shut down the connection, two to read and write data, and one to check that the networking interface provided by the interface has been correctly initialised. The network plugin allows cryptlib to use any kind of network interface, either a customised form of the built-in BSD sockets interface or a completely different network mechanism such as SNA or X.25.

The Database Backend Interface

The database backend interface is used when cryptlib receives a user request to access a database of type CRYPT_KEYSET_DATABASE or CRYPT_KEYSET_-DATABASE_STORE. The first thing that cryptlib does is call the `initDbxSession()` function in keyset/dbms.c, which connects the generic database type to the actual database backend-specific code (for example an Oracle, Sybase, or PostgreSQL interface).

The structure of the database interface is as follows:

```
#include "keyset/keyset.h"

/* Plugin functions: openDatabase(), closeDatabase(), performUpdate(),
   performQuery() */

int initDispatchDatabase( DBMS_INFO *dbmsInfo )
   {
   dbmsInfo->openDatabaseBackend = openDatabase;
   dbmsInfo->closeDatabaseBackend = closeDatabase;
   dbmsInfo->performUpdateBackend = performUpdate;
   dbmsInfo->performQueryBackend = performQuery;

   return( CRYPT_OK );
   }
```

keyset/keyset.h contains the keyset-related defines that are used in the code, and the dispatcher initialisation function sets up function pointers to the database access routines, which are explained in more detail below. State information about a session with the database is contained in the DBMS_STATE_INFO structure which is defined in keyset/keyset.h. This contains both shared information such as the last error code and the status of the session, and back-end -specific information such as connection handles and temporary data areas. When you create an interface for a new database type, you should add any variables that you need to the database-specific section of the DBMS_STATE_INFO structure. When cryptlib calls your interface functions it will pass in the DBMS_STATE_INFO that you can use to store state information.

Database Interface Functions

The database interface functions that you need to provide are as follows:

```
static int openDatabase( DBMS_STATE_INFO *dbmsInfo, const char *name,
   const int nameLen, const CRYPT_KEYOPT_TYPE options,
   int *featureFlags )
```

This function is called to open a session with the database. The parameters are the name of the database to open the session to and a set of option flags that apply to the session. The name parameter is a composite value that depends on the underlying database being used, usually this is simply the database name, but it can also contain a complete user name and password in the format `user:pass@server`. Other combinations are `user:pass` (only a database user name and password) or `user@server` (only a user name and server).

The option flags will be set to either CRYPT_KEYOPT_NONE or CRYPT_-KEYOPT_READONLY, many servers can optimise accesses if they know that no updates will be performed so your code should try and communicate this to the server if possible. The function should return a set of database feature flags indicating its capabilities in the featureFlags parameter. These will be either DBMS_FLAG_-BINARYBLOBS if the database can store binary data blobs rather than requiring that data be base64-encoded or DBMS_FLAG_NONE if it has no special capabilities. The interface code should provide binary blob access if the database supports this (almost all do) since this increases data handling efficiency and reduces storage requirements.

```
static void closeDatabase( DBMS_STATE_INFO *dbmsInfo )
```

This function is called to shut down the session with the database.

```
static int performUpdate( DBMS_STATE_INFO *dbmsInfo,
   const char *command, const int commandLength,
   const BOUND_DATA *boundData, DBMS_UPDATE_TYPE updateType )
```

This function is called to send data to the database. The parameters are an SQL command, an optional set of bound data items, and an update type indicator that indicates which type of update is being performed. If the `boundData` value is non-null then it contains one or more bound data items that are to be added as part of the SQL command. `boundData` is an array of BOUND_DATA items of type `boundData.type`, with the end of the array being denoted by `boundData.type == BOUND_DATA_NONE`. Other types can be BOUND_DATA_TIME (a date-and-time value), BOUND_DATA_STRING (a text string), and BOUND_DATA_BLOB (a binary blob). The date value needs to be converted into whatever format the database back-end expects for a DATETIME value. The exact format depends on the back-end, which is why it's not present in the SQL command but is provided as a bound data value.

For example the `performUpdate()` function can be called as:

```
performUpdate( …, "'INSERT INTO certificates VALUES ( '…', '…', …
   '…' )", …, NULL, … );
performUpdate( …, "INSERT INTO certificates VALUES ( '…', '…', … ? )",
   …, boundData1, … );
performUpdate( …, "INSERT INTO certificates VALUES ( ?, '…', … ? )",
   …, boundData2, … );
```

In the first case all data is contained in the SQL command, and `boundData` is null. In the second case there's a bound value that happen to be a binary data blob associated with the SQL command whose position is indicated by the '?' placeholder. After sending the SQL command to the database, you also need to send the { `boundData1[ 0 ].data, boundData1[ 0 ].dataLength` } value. In the third case there are two parameters that happen to be a binary data blob and a date value, associated with the SQL command, with the positions again indicated by the '?' placeholders. The types and values will be given in the `boundData` information so you don't need to worry about which parameter is of which type. In the third case, assuming that the SQL has the blob first and the date second, `boundData[ 0 ].type` will be BOUND_DATA_BLOB with { `boundData1[ 0 ].data, boundData1[ 0 ].dataLength` } being the blob and `boundData[ 1 ].type` will be BOUND_DATA_TIME with { `boundData1[ 1 ].data, boundData1[ 1 ].dataLength` } being the time as a standard `time_t` value. Since the order in the `boundData` matches the order in the SQL, all you have to do is iterate through the BOUND_DATA array binding in parameter data until you reach the BOUND_DATA_NONE marker at the end.

The update types are as follows:

Update Type	Description
DBMS_UPDATE_-ABORT	Abort a transaction. This state is communicated to the database through an SQL statement such as ABORT TRANSACTION or ROLLBACK or ABORT, or via a function call that indicates that the transaction begun earlier should be aborted or rolled back.
DBMS_UPDATE_-BEGIN	Begin a transaction. This state is communicated to the database through an SQL statement such as BEGIN TRANSACTION or BEGIN WORK or BEGIN, or via a function call that indicates that transaction semantics are in effect for the following SQL statements.
DBMS_UPDATE_-COMMIT	Commit a transaction. This state is communicated to the database through an SQL statement such as END TRANSACTION or COMMIT WORK or COMMIT, or via a function call that indicates that the transaction should be committed and that transaction semantics are no longer in effect after the statement has been submitted.
DBMS_UPDATE_-CONTINUE	Continue an ongoing transaction.
DBMS_UPDATE_-NORMAL	Standard data update.

The DBMS_UPDATE_BEGIN/CONTINUE/COMMIT combination is used to perform an atomic update on the database. The sequence of calls is as follows:

```
performUpdate( …, "INSERT INTO certificates VALUES ( … )", …,
   boundData, DBMS_UPDATE_BEGIN );
performUpdate( …, "INSERT INTO certLog VALUES ( … )", …, boundData,
   DBMS_UPDATE_CONTINUE );
performUpdate( …, "DELETE FROM certRequests WHERE keyID = keyID", …,
   NULL, DBMS_UPDATE_COMMIT );
```

The first call begins the transaction and submits the initial portion of the transaction, the ongoing calls submit successive portions of the transaction, and the final call submits the last portion and commits the transaction. If there's a problem then the last call in the transaction will use an update type of DBMS_UPDATE_ABORT. Note that it's important to ensure that `performUpdate()` itself is atomic, for example if there's an error inside the function then it needs to back out of the transaction (if one is in progress) rather than simply returning immediately to the caller. This requires careful tracking of the state of the transaction and handling of error conditions.

```
static int performQuery( DBMS_STATE_INFO *dbmsInfo,
   const char *command, const int commandLength, void *data,
   const int dataMaxLength, int *dataLength,
   const BOUND_DATA *boundData,
   const DBMS_CACHEDQUERY_TYPE queryEntry,
   const DBMS_QUERY_TYPE queryType )
```

This function is called to fetch data from the database. The parameters are an SQL command, an optional buffer of total size `dataMaxLength` to store the result, optional bound query data as for `performUpdate()`, a query caching indicator (explained further on) and a query type indicator that indicates which type of query is being performed. The query types are as follows:

Query Type	Description
DBMS_QUERY_-CANCEL	Cancel an ongoing query. This terminates an ongoing query begun by sending a DBMS_QUERY_START query.
DBMS_QUERY_-CHECK	Perform a presence check that simply returns a present/not present indication without returning any data. This allows the query to be optimised since there's no need to actually fetch any data from the back-end. All that's necessary is that a status indication be returned that indicates whether the requested data is available to be fetched or not.
DBMS_QUERY_-CONTINUE	Continue a previous ongoing query. This returns the next entry in the result set generated by sending a DBMS_QUERY_START query.
DBMS_QUERY_-NORMAL	Standard data fetch.
DBMS_QUERY_-START	Begin an ongoing query. This submits a query to the back-end without returning any data. The result set is read one entry at a time by sending DBMS_QUERY_CONTINUE messages.

The DBMS_QUERY_START/CONTINUE/CANCEL combination is used to fetch a collection of entries from the database. The sequence of calls is as follows:

```
performQuery( …, "SELECT certData FROM certificates WHERE key = ?", …,
   NULL, 0, NULL, boundData, DBMS_CACHEDQUERY_NONE,
   DBMS_QUERY_START );

do
   status = performQuery( …, NULL, …, buffer, bufferMaxLength,
      &length, NULL, DBMS_CACHEDQUERY_NONE, DBMS_QUERY_CONTINUE );
while( cryptStatusOK( status ) );
```

The first call submits the query and the ongoing calls fetch successive entries in the result set until an error status is returned (usually this is CRYPT_ERROR_-COMPLETE to indicate that there are no more entries in the result set).

In order to allow for more efficient execution of common queries, cryptlib allows them to be cached by the database back-end for re-use in the future. This allows the back-end to perform the task of SQL parsing and validation against the system catalogue, query optimisation, and access plan generation just once when the first query is executed rather than having to re-do it for each query. cryptlib provides hints about cached queries by specifying a cache entry number when it submits the query. Uncached queries are given an entry number of DBMS_CACHEDQUERY_-NONE (these will be little-used query types that it's not worth caching) while queries where caching are worthwhile are given an entry number from 1 to 5. The submitted SQL for these queries will never change over subsequent calls, so it's only necessary to perform the parsing and processing once when the query is submitted for the first time. Any subsequent requests can be satisfied using the previously parsed query held at the back-end. In the above example, if the query were submitted with a caching indicator of 2 you would prepare a query for the supplied SQL string the first time that the query is submitted and then re-use the prepared query every time another query with the caching indicator 2 was used.

Note that some databases may return a (potentially large) result set in response to a query for a single result using DBMS_QUERY_NORMAL, for example by returning further results after the first one is read or by disallowing further queries until all results have been processed. In this you need to limit the query response size either by setting a size limit before submitting the query or by explicitly cancelling a query if more than one result is returned. In addition since cryptlib expects all data to be SQL text strings (or binary data for certificate objects if the database supports it) you may need to convert some data types such as integer values to text equivalents when returning them in response to a query.

An example of a database backend interface is keyset/odbc.c, which implements the full functionality required by cryptlib. In addition to the standard functions included below, you may also need to include an SQL rewrite function that changes the contents of SQL queries to match the SQL dialect used by your database. This is a simple function that just substitutes one text string in the query for another. The most common conversion changes the name of the binary blob type (if the database supports it) from the built-in "BLOB" to whatever value is required by the database. Again, see keyset/odbc.c for an example of the SQL rewrite process.

The Network Plugin Interface

The network plugin interface is used to provide a transport-layer service to the higher-level cryptlib protocols that require network access capabilities. Network management is handled by the cryptlib I/O streams module io/stream.c. The stream I/O system implements a multi-layer architecture with the transport-layer service in the lowest layer, optional I/O buffering layered above that, optional application-layer handling (for example HTTP) above that, and finally the cryptlib protocols such as CMP, RTCS, SCEP, OCSP, SCVP, and TSP above that. Other protocols such as SSH and TLS, which don't require any of the intermediate layers, talk directly to the transport layer.

By replacing the transport-layer interface, you can run cryptlib communications over any type of transport interface. Currently cryptlib provides two types of built-in transport provider, a generic BSD sockets provider and a provider that uses a cryptlib session as the transport layer, making it possible to run (for example) RTCS over TLS or CMP over SSH. You can also use the plugin functionality to provide custom I/O handling that goes beyond that provided by the standard sockets-based interface. For example if you need to use event-based I/O or OS-specific mechanisms such as I/O completion ports, you can provide this capability through the use of custom I/O handlers in the network plugin interface.

The network plugin interface is handled through function pointers to the various transport-layer functions. By setting these to point to functions in the appropriate plugins, it's possible to use any type of networking or communications interface for the transport layer. To set these pointers, the cryptlib I/O stream system calls `setAccessMethodXXX()`, which in the case of BSD sockets is `setAccessMethodTCP()`.

When calling a transport-layer interface function, cryptlib passes in a STREAM structure which is defined in io/stream.h. This contains information which is required by the transport layer such as socket handles and network communications timeout information.

Network Plugin Functions

The network plugin functions that you need to provide are as follows:

```
static BOOLEAN transportOKFunction( void )
```

This function is called before using any transport-layer functions to query whether the transport layer is OK. It should return TRUE if it's safe to call the other transport-layer functions or FALSE otherwise, for example because the requested network interface drivers aren't loaded.

```
static int transportConnectFunction( STREAM *stream,
    const char *server, const int port )
```

This functions is called to established a connection, either by connecting to a remote system or by waiting for a connection from a remote system (the exact type depends on whether the stream is acting as a client or server stream). The `server` parameter is the name of the local interface or remote server, and the `port` parameter is the port number to listen on or connect to.

```
static void transportDisconnectFunction( STREAM *stream )
```

This function is called to shut down a connection with a remote client or server.

```
static int transportReadFunction( STREAM *stream, BYTE *buffer,
   const int length, const int flags )
```

This function is called to read data from a remote client or server. The behaviour of this function differs slightly depending on read timeout handling. For blocking reads, it should read as many bytes as are indicated in the length parameter, returning an error if less bytes are read. For non-blocking reads it should read as many bytes as are available (which may be zero) and return. In either case if the read succeeds it returns a byte count.

Normally the read should wait for data to appear for the number of seconds indicated by the timeout value stored in the stream I/O structure. However, it's possible to override this with the `flags` parameter, which can contain the following flags:

Flag	Description
TRANSPORT_FLAG_-NONBLOCKING	Perform a non-blocking read, overriding the timeout value in the stream I/O structure if necessary.
TRANSPORT_FLAG_-BLOCKING	Perform a blocking read, overriding the timeout value in the stream I/O structure if necessary.

These flags are used in cases where it's known that a certain number of bytes must be read in order to continue, or when the higher-level stream buffering functions want to perform a speculative read-ahead.

```
static int transportWriteFunction( STREAM *stream, const BYTE *buffer,
   const int length, const int flags )
```

This function is used to write data to a remote client or server. The flags parameter is currently unused and should be set to TRANSPORT_FLAG_NONE.

Algorithms and Standards Conformance

This chapter describes the characteristics of each algorithm used in cryptlib and any known restrictions on their use. Since cryptlib originates in a country that doesn't allow software patents, there are no patent restrictions on the code in its country of origin. Known restrictions in other countries are listed below and all possible care has been taken to ensure that no other infringing technology is incorporated into the code, however since the author is a cryptographer and not an IP lawyer users are urged to consult IP lawyers in the country of intended use if they have any concerns over potential restrictions.

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 algorithm standards that cryptlib follows are listed below. A further list of non-algorithm-related standards that cryptlib complies with are given at the start of this document.

AES

AES is a 128-bit block cipher with a 128-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_AES.

AES has been implemented as per:

FIPS PUB 197, "Advanced Encryption Standard", 2001.

The AES code has been validated against the test vectors given in:

FIPS PUB 197, "Advanced Encryption Standard", 2001.

CAST-128

CAST-128 is a 64-bit block cipher with a 128-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_CAST.

CAST-128 has been implemented 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 is a 64-bit block cipher with a 56-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_DES. Note that this algorithm is no longer considered secure and should not be used. It is present in cryptlib only for compatibility with legacy applications.

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_PARAM3 error if the key is a weak key. **cryptExportKey** will export the correct parity-adjusted version of the key.

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 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_PARAM3 error if the key is a weak key. **cryptExportKey** will export the correct parity-adjusted version of the key.

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

Diffie-Hellman is a key-agreement algorithm with a key size of up to 4096 bits and has the cryptlib algorithm identifier CRYPT_ALGO_DH.

Diffie-Hellman was formerly covered by a patent in the US, this has now expired.

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 is a digital signature algorithm with a key size of up to 1024 bits and has the cryptlib algorithm identifier CRYPT_ALGO_DSA.

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 world-wide. The US Department of Commerce is not aware of any other patents that 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.

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.

ECDSA

ECDSA is a digital signature algorithm with a key size of up to 521 bits and has the cryptlib algorithm identifier CRYPT_ALGO_ECDSA.

ECDSA has been implemented as per:

ANSI X9.62:2005, "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)", 2005.

Although cryptlib supports the ECC algorithms ECDSA and ECDH, enabling them in the code creates a number of issues. Due to the complexity of ECC, both the memory and code footprints of the crypto portion of cryptlib are noticeably increased when ECC support is enabled. In addition the ECC algorithms are extremely brittle, with the slightest issue potentially causing problems, and the algorithms having a distressing propensity for failing by leaking bits of the private key (this is a property of the crypto behind the algorithms, not an implementation issue). For these reasons you should only enable the use of ECC if it's absolutely essential, and if you fully understand the consequences of doing so.

ECDH

ECDH is a key-agreement algorithm with a key size of up to 521 bits and has the cryptlib algorithm identifier CRYPT_ALGO_ECDH.

ECDH has been implemented as per:

ANSI X9.63:2001, "Public Key Cryptography for the Financial Services Industry, Key Agreement and Key Transport Using Elliptic Curve Cryptography", 2001.

Although cryptlib supports the ECC algorithms ECDSA and ECDH, enabling them in the code creates a number of issues. Due to the complexity of ECC, both the memory and code footprints of the crypto portion of cryptlib are noticeably increased when ECC support is enabled. In addition the ECC algorithms are extremely brittle, with the slightest issue potentially causing problems, and the algorithms having a distressing propensity for failing by leaking bits of the private key (this is a property of the crypto behind the algorithms, not an implementation issue). For these reasons you should only enable the use of ECC if it's absolutely essential, and if you fully understand the consequences of doing so.

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.

Elgamal was formerly covered (indirectly) by a patent in the US, this has now expired.

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-SHA1
HMAC-SHA2

HMAC-SHA1 and HMAC-SHA2 are MAC algorithms with a key size of up to 1024 bits and have the cryptlib algorithm identifiers CRYPT_ALGO_HMAC_SHA1 and CRYPT_ALGO_HMAC_SHA2.

HMAC-SHA1 and HMAC-SHA2 have been implemented as per:

FIPS PUB 198, "The Keyed-Hash Message Authentication Code (HMAC)", 2002.

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 is a 64-bit block cipher with a 128-bit key and has the cryptlib algorithm identifier CRYPT_ALGO_IDEA.

IDEA was formerly covered by patents, but these have now all expired.

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 Techniques — 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.

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. Some weaknesses have been found in this algorithm, and it's proven to be extremely difficult to employ in a safe manner. For this reason it should not be used any more except for legacy application support, and is disabled by default.

The term "RC4" is trademarked in the US. It may be necessary to refer to it as "an algorithm compatible with RC4" in products that use RC4 and are distributed in the US. Common practice is to refer to it as ArcFour.

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.

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.

RSA was formerly covered by a patent in the US, this has now expired.

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.

SHA1

SHA1 is a message digest/hash algorithm with a digest/hash size of 160 bits and has the cryptlib algorithm identifier CRYPT_ALGO_SHA1.

The SHA1 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".

RFC 3174, "US Secure Hash Algorithm 1 (SHA1)", 2001

The SHA1 code has been validated against the test vectors given in:

FIPS PUB 180-1, "Secure Hash Standard", 1994.

SHA2/SHA256

SHA2/SHA256 is a message digest/hash algorithm with a digest/hash size of 256 bits and has the cryptlib algorithm identifier CRYPT_ALGO_SHA2.

The SHA2 code has been implemented as per:

FIPS PUB 180-2, “Secure Hash Standard”, 2002.

The SHA2 code has been validated against the test vectors given in:

FIPS PUB 180-2, “Secure Hash Standard”, 2002.

Data Types and Constants

This section describes the data types and constants used by cryptlib.

CRYPT_ALGO_TYPE

The CRYPT_ALGO_TYPE is used to identify a particular encryption algorithm. More information on the individual algorithm types can be found in "Algorithms" on page 305.

Value	Description
CRYPT_ALGO_AES	AES
CRYPT_ALGO_CAST	CAST-128
CRYPT_ALGO_DES	DES. This algorithm is no longer considered secure and should not be used except for legacy application support.
CRYPT_ALGO_3DES	Triple DES
CRYPT_ALGO_IDEA	IDEA
CRYPT_ALGO_RC4	RC4. This algorithm has security problems and should not be used except for legacy application support.
CRYPT_ALGO_DH	Diffie-Hellman
CRYPT_ALGO_DSA	DSA
CRYPT_ALGO_ECDSA	ECDSA
CRYPT_ALGO_ECDH	ECDH
CRYPT_ALGO_ELGAMAL	Elgamal
CRYPT_ALGO_RSA	RSA
CRYPT_ALGO_SHA1	SHA-1. This algorithm has poor long-term security prospects and should be deprecated in favour of SHA-2.
CRYPT_ALGO_SHA2	SHA-2/SHA-256
CRYPT_ALGO_HMAC_SHA1	HMAC-SHA1
CRYPT_ALGO_HMAC_SHA2	HMAC-SHA2
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.
CRYPT_ALGO_FIRST_HASH CRYPT_ALGO_LAST_HASH	First and last possible hash algorithm type.

Value	Description
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 38 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 exporting certificate objects is given in "Importing/Exporting Certificates" on page 232.

Value	Description
CRYPT_CERTFORMAT_-CERTCHAIN	Certificate object encoded as a PKCS #7 certificate chain. This encoding is only possible for objects that are certificates or certificate chains.
CRYPT_CERTFORMAT_-CERTIFICATE	Certificate object encoded according to the ASN.1 distinguished encoding rules (DER).
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 when used with **cryptCreateCert**. More information on certificates and certificate objects is given in "Certificates and Certificate Management" on page 154.

Value	Description
CRYPT_CERTTYPE_-ATTRIBUTE_CERT	Attribute certificate.
CRYPT_CERTTYPE_CERTCHAIN	PKCS #7 certificate chain.
CRYPT_CERTTYPE_-CERTIFICATE	Certificate.
CRYPT_CERTTYPE_-CERTREQUEST	PKCS #10 certification request.
CRYPT_CERTTYPE_CMS_-ATTRIBUTES	PKCS #7/CMS attributes.
CRYPT_CERTTYPE_CRL	CRL
CRYPT_CERTTYPE_OCSP_-REQUEST CRYPT_CERTTYPE_OCSP_-RESPONSE	OCSP request and response.
CRYPT_CERTTYPE_RTCS_-REQUEST	RTCS request and response.

Value	Description
CRYPT_CERTTYPE_RTCS_-RESPONSE	
CRYPT_CERTTYPE_PKIUSER	PKI user information.
CRYPT_CERTTYPE_REQUEST_-CERT CRYPT_CERTTYPE_REQUEST_-REVOCATION	CRMF certificate request/revocation request.

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 ” on page 282.

Value	Description
CRYPT_DEVICE_FORTEZZA	Fortezza card.
CRYPT_DEVICE_HARDWARE	Native on-chip/on-device cryptography support.
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 “Data Enveloping” on page 53, “Exchanging Keys” on page 203, and “Signing Data” on page 209.

Value	Description
CRYPT_FORMAT_CRYPTLIB	cryptlib native format.
CRYPT_FORMAT_PGP	PGP format.
CRYPT_FORMAT_CMS CRYPT_FORMAT_PKCS7	PKCS #7/CMS format.
CRYPT_FORMAT_SMIME	As CMS but with S/MIME-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 “Reading a Key from a Keyset” on page 145

Value	Description
CRYPT_KEYID_NAME	The name of the key owner.
CRYPT_KEYID_EMAIL	The email address of the key owner.

CRYPT_KEYOPT_TYPE

The CRYPT_KEYOPT_TYPE is used to contain keyset option flags passed to **cryptKeysetOpen**. 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 **cryptKeysetOpen**. More information on using these options when opening a connection to a keyset is given in "Creating/Destroying Keyset Objects" on page 137

Value	Description
CRYPT_KEYOPT_CREATE	Create a new keyset. This option is only valid for writeable keyset types, which includes keysets implemented as databases and cryptlib key files.
CRYPT_KEYOPT_NONE	No special access options (this option implies read/write access).
CRYPT_KEYOPT_READONLY	Read-only keyset access. This option is automatically enabled by cryptlib for keyset types that 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.

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 **cryptKeysetOpen**. Some keyset types may be unavailable on some systems. More information on keyset types is given in "Keyset Types" on page 136.

Value	Description
CRYPT_KEYSET_FILE	A flat-file keyset, either a cryptlib key file or a PGP/OpenPGP key ring.
CRYPT_KEYSET_HTTP	URL specifying the location of a certificate or CRL.
CRYPT_KEYSET_LDAP	LDAP directory service.
CRYPT_KEYSET_DATABASE	Generic database interface.
CRYPT_KEYSET_DATABASE_-STORE	As for the basic keyset types, but representing a certificate store for use by a CA rather than a simple keyset. The user who creates and updates these keyset types must be a CA user.

CRYPT_MODE_TYPE

The CRYPT_MODE_TYPE is used to identify a particular conventional encryption mode. More information on the individual modes can be found in "Algorithms" on page 305.

Value	Description
CRYPT_MODE_ECB	ECB
CRYPT_MODE_CBC	CBC

Value	Description
CRYPT_MODE_CFB	CFB
CRYPT_MODE_GCM	GCM

CRYPT_OBJECT_TYPE

The CRYPT_OBJECT_TYPE is used to identify the type of an exported key or signature object that has been created with **cryptExportKey** or **cryptCreateSignature**. More information on working with these objects is given in "Querying an Exported Key Object" on page 206, and "Querying a Signature Object" on page 210.

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.

CRYPT_SESSION_TYPE

The CRYPT_SESSION_TYPE is used to identify a secure session type when used with **cryptCreateSession**. More information on sessions is given in "Secure Sessions" on page 102.

Value	Description
CRYPT_SESSION_CMP CRYPT_SESSION_CMP_SERVER	CMP client/server session.
CRYPT_SESSION_SCEP CRYPT_SESSION_SCEP_SERVER	SCEP client/server session.
CRYPT_SESSION_RTCS CRYPT_SESSION_RTCS_SERVER	RTCS client/server session.
CRYPT_SESSION_OCSP CRYPT_SESSION_OCSP_SERVER	OCSP client/server session.
CRYPT_SESSION_SCVP CRYPT_SESSION_SCVP_SERVER	SCVP client/server session.
CRYPT_SESSION_SSH CRYPT_SESSION_SSH_SERVER	SSH client/server session.
CRYPT_SESSION_TLS CRYPT_SESSION_TLS_SERVER	TLS client/server session.
CRYPT_SESSION_TSP CRYPT_SESSION_TSP_SERVER	TSP client/server session.

Data Size Constants

The following values define various maximum lengths for data objects that 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.

Constant	Description
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_PKCSIZE_-ECC	Maximum ECC public-key component size in bytes. ECC keys have slightly different lengths than conventional public/private keys so this is given as a distinct value, otherwise it works just like CRYPT_-MAX_PKCSIZE.
CRYPT_MAX_TEXTSIZE	Maximum size of a text string (e.g. 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 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_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 section describes the data structures used by cryptlib.

CRYPT_OBJECT_INFO Structure

The CRYPT_OBJECT_INFO structure is used with **cryptQueryObject** to return information about a 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_TYPE cryptAlgo	Encryption/signature algorithm.
CRYPT_MODE_TYPE cryptMode	Encryption/signature mode.
CRYPT_ALGO_TYPE hashAlgo	The hash algorithm used to hash the data if the data object is a signature object, or used to derive the export/import key if the object is a conventionally encrypted key object.
unsigned char salt[CRYPT_MAX_-HASHSIZE] int saltLength	The salt used to derive the export/import key if the object is a conventionally encrypted key object.
int iterations	The number of iterations of hashing used to derive the export/import key if the object is a conventionally encrypted key object.

CRYPT_PKCINFO_*xxx* Structures

The CRYPT_PKCINFO_*xxx* 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_COMPONENTS attribute. All fields are multi-precision integer values that 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. Some DH and Elgamal keys don't use this parameter, in which case you should set it to an all-zero value of the appropriate size. Note that omitting the q parameter means that cryptlib can't perform certain key validity checks that it otherwise performs when q is present.
g	Element of order q mod p.
x	Private random integer.
y	Public random integer, g^x mod p.

The CRYPT_PKCINFO_ECC structure is used to load keys for elliptic-curve algorithms, which include keys for ECDSA and ECDH. The structure contains the following fields:

Field	Description
curveType	A named curve, one of CRYPT_ECCCURVE_P192, CRYPT_ECCCURVE_P224, CRYPT_ECCCURVE_-P256, CRYPT_ECCCURVE_P384, CRYPT_-ECCCURVE_P521, CRYPT_ECCCURVE_-BRAINPOOL_P256, CRYPT_ECCCURVE_-BRAINPOOL_P384 or CRYPT_ECCCURVE_-BRAINPOOL_P512. When an explicit named curve is used then the domain parameters p, a, b, gx, gy, n, and h must be left empty.
p	Prime defining Fq.
a	Element in Fq defining the curve.
b	Element in Fq defining the curve.
gx	Element in Fq defining the point on the curve.
gy	Element in Fq defining the point on the curve.
n	Order of the point
h	Optional cofactor
qx, qy	Point Q on the curve
d	Private random integer

The CRYPT_PKCINFO_RSA structure is used to load RSA public-key encryption keys and contains the following fields:

Field	Description
n	Modulus.
e	Public exponent.
d	Private exponent. Some keys don't include this parameter, in which case you should set it to an all-zero value of the appropriate size. Note that if the d parameter is absent then the e1 and e2 values must be present.
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).
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, SigG, 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 user's private key to a keyset.

int cryptAddPrivateKey(const CRYPT_KEYSET *keyset*, **const CRYPT_HANDLE** *cryptKey*, **const char** **password* **);**

Parameters

keyset
The keyset object to which to write the key.

cryptKey
The private key to write to the keyset.

password
The password used to encrypt the private key.

Remarks The use of a password to encrypt the private key is required when storing a private key to a keyset, but not to a crypto device such as a smart card or Fortezza card, since these provide their own protection for the key data.

See also **cryptAddPublicKey**, **cryptDeleteKey**, **cryptGetPrivateKey**, **cryptGetPublicKey**.

cryptAddPublicKey

The **cryptAddPublicKey** function is used to add a user's public key or certificate to a keyset.

int cryptAddPublicKey(const CRYPT_KEYSET *keyset*, **const CRYPT_CERTIFICATE** *certificate* **);**

Parameters

keyset
The keyset object to which to write 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.

cryptCAAddItem

The **cryptCAAddItem** function is used to add a certificate object to a certificate store. **cryptAddPublicKey** is used to add standard certificates, this CA-specific function can be used by CAs to add special items such as certificate requests and PKI user information.

int cryptCAAddItem(const CRYPT_KEYSET *keyset*, **const CRYPT_CERTIFICATE** *certificate* **);**

Parameters *keyset*
The certificate store to which the item will be added.

certificate
The item to add to the certificate store.

See also **cryptCACertManagement**, **cryptCAGetItem**.

cryptCACertManagement

The **cryptCACertManagement** function is used to perform a CA certificate management operation such as a certificate issue, revocation, CRL issue, certificate expiry, or other operation with a certificate store.

int cryptCACertManagement(CRYPT_CERTIFICATE **cryptCert*, **const CRYPT_CERTACTION_TYPE** *action*, **const CRYPT_KEYSET** *keyset*, **const CRYPT_CONTEXT** *caKey*, **const CRYPT_CERTIFICATE** *certRequest* **);**

Parameters *cryptCert*
The address of the certificate object to be created.

action
The certificate management operation to perform.

keyset
The certificate store to use to perform the action.

caKey
The CA key to use when performing the action, or CRYPT_UNUSED if no key is necessary for this action.

certRequest
The certificate request to use when performing the action, or CRYPT_UNUSED if no request is necessary for this action.

See also **cryptCAAddItem**, **cryptCAGetItem**.

cryptCAGetItem

The **cryptCAGetItem** function is used to read a certificate object from a certificate store. **cryptGetPublicKey** is used to read standard certificates, this CA-specific function can be used by CAs to obtain special items such as certificate requests and PKI user information. The item to be fetched is identified either through the key owner's name or their email address.

int cryptCAGetItem(const CRYPT_KEYSET *keyset*, **CRYPT_CERTIFICATE** **certificate*, **const CRYPT_CERTTYPE_TYPE** *certType*, **const CRYPT_KEYID_TYPE** *keyIDtype*, **const void** **keyID* **);**

Parameters *keyset*
The certificate store from which to obtain the item.

certificate
The address of the certificate object to be fetched.

certType
The item type.

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 item to read.

See also **cryptCACertManagement**, **cryptCAAddItem**.

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 that 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. If the certificate is to be verified online, this should be a session object for the server used to verify the certificate.

See also **cryptSignCert**.

cryptCheckSignature

The **cryptCheckSignature** function is used to check the digital signature on a piece of data.

int cryptCheckSignature(const void **signature*, **const int** *signatureLength*, **const CRYPT_HANDLE** *sigCheckKey*, **const CRYPT_CONTEXT** *hashContext* **);**

Parameters *signature*
The address of a buffer that contains the signature.

signatureLength
The length in bytes of the signature data.

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.

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.

int cryptCheckSignatureEx(const void **signature*, **const int** *signatureLength*, **const CRYPT_HANDLE** *sigCheckKey*, **const CRYPT_CONTEXT** *hashContext*, **CRYPT_HANDLE** **extraData* **);**

Parameters *signature*
The address of a buffer that contains the signature.

signatureLength
The length in bytes of the signature data.

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.

See also **cryptCheckSignature**, **cryptCreateSignature**, **cryptCreateSignatureEx**, **cryptQueryObject**.

cryptCreateCert

The **cryptCreateCert** function is used to create a certificate object that contains a certificate, certification request, certificate chain, CRL, or other certificate-like object.

int cryptCreateCert(CRYPT_CERTIFICATE **cryptCert*, **const CRYPT_USER** *cryptUser*, **const CRYPT_CERTTYPE_TYPE** *certType* **);**

Parameters *cryptCert*
The address of the certificate object to be created.

cryptUser
The user who is to own the certificate object or CRYPT_UNUSED for the default, normal user.

certType
The type of certificate item that 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_USER** *cryptUser*, **const CRYPT_ALGO_TYPE** *cryptAlgo* **);**

Parameters

cryptContext
The address of the encryption context to be created.

cryptUser
The user who is to own the encryption context or CRYPT_UNUSED for the default, normal user.

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_USER** *cryptUser*, **const CRYPT_FORMAT_TYPE** *formatType* **);**

Parameters

cryptEnvelope
The address of the envelope to be created.

cryptUser
The user who is to own the envelope object or CRYPT_UNUSED for the default, normal user.

formatType
The data format for the enveloped data.

See also **cryptDestroyEnvelope**.

cryptCreateSession

The **cryptCreateSession** function is used to create a secure session object for use in securing a communications link or otherwise communicating with a remote server or client.

int cryptCreateSession(CRYPT_SESSION **cryptSession*, **const CRYPT_USER** *cryptUser*, **const CRYPT_SESSION_TYPE** *sessionType* **);**

Parameters

cryptSession
The address of the session to be created.

cryptUser
The user who is to own the session object or CRYPT_UNUSED for the default, normal user.

sessionType
The type of the secure session.

See also **cryptDestroySession**.

cryptCreateSignature

The **cryptCreateSignature** function digitally signs a piece of data. The signature is placed in a buffer in a portable format that allows it to be checked using **cryptCheckSignature**.

int cryptCreateSignature(void **signature*, **const int** *signatureMaxLength*, **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.

signatureMaxLength
The maximum size in bytes of the buffer to contain the signature data.

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. The signature is placed in a buffer in a portable format that allows it to be checked using **cryptCheckSignatureEx**.

int cryptCreateSignatureEx(void **signature*, **const int** *signatureMaxLength*, **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.

signatureMaxLength
The maximum size in bytes of the buffer to contain the signature data.

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*, **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.

Remarks Public-key encryption and signature algorithms have special data formatting requirements that 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 **cryptDeleteCertExtension** 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, SigG, 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 owner's 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 that 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 **cryptDestroyContext**, **cryptDestroyCert, cryptDestroyEnvelope, cryptDestroySession, cryptKeysetClose**.

cryptDestroySession

The **cryptDestroySession** function is used to destroy a session object after use. This close the link to the client or server, erases all keying and security information used by the session, and frees up any memory it uses.

int cryptDestroySession(const CRYPT_SESSION *cryptSession* **);**

Parameters *cryptSession*
The session to be destroyed.

See also **cryptCreateSession**.

cryptDeviceClose

The **cryptDeviceClose** 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**.

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_TYPE** *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_USER** *cryptUser*, **const CRYPT_DEVICE_TYPE** *deviceType*, **const char** **name* **);**

Parameters

device
The address of the device object to be created.

cryptUser
The user who is to own the device object or CRYPT_UNUSED for the default, normal user.

deviceType
The device type to be used.

name
The name of the device, or null if a name isn't required.

See also **cryptDeviceClose**.

cryptDeviceQueryCapability

The **cryptDeviceQueryCapability** 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 cryptDeviceQueryCapability(const CRYPT_DEVICE *cryptDevice*, **const CRYPT_ALGO_TYPE** *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 that don't apply to the algorithm being queried are set to CRYPT_ERROR, 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**.

cryptEncrypt

The **cryptEncrypt** function is used to encrypt or hash data.

int cryptEncrypt(const CRYPT_CONTEXT *cryptContext*, **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.

Remarks Public-key encryption and signature algorithms have special data formatting requirements that 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**.

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*, **const int** *certObjectMaxLength*, **int** **certObjectLength*, **const CRYPT_CERTFORMAT_TYPE** *certFormatType*, **const CRYPT_CERTIFICATE** *certificate* **);**

Parameters

certObject
The address of a buffer to contain the encoded certificate.

certObjectMaxLength
The maximum size in bytes of the buffer to contain the exported 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 that 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*, **const int** *encryptedKeyMaxLength*, **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 *encryptedKeyLength* without actually exporting the key.

encryptedKeyMaxLength
The maximum size in bytes of the buffer to contain the exported key.

encryptedKeyLength
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) that the new shared session key is generated into.

See also **cryptExportKeyEx**, **cryptImportKey**, **cryptQueryObject**.

cryptExportKeyEx

The **cryptExportKeyEx** 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 extended control over the exported key format. The exported/shared key is placed in a buffer in a portable format that 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 cryptExportKeyEx(void **encryptedKey*, **const int** *encryptedKeyMaxLength*, **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 *encryptedKeyLength* without actually exporting the key.

encryptedKeyMaxLength
The maximum size in bytes of the buffer to contain the exported key.

encryptedKeyLength
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) that the new shared session key is generated into.

See also **cryptExportKey**, **cryptImportKey**, **cryptQueryObject**.

cryptFlushData

The **cryptFlushData** function is used to flush data through an envelope or session object, completing processing and (for session objects) sending the data to the remote client or server.

int cryptFlushData(const CRYPT_HANDLE *cryptHandle* **);**

Parameters *cryptHandle*
The envelope or session object to flush the data through.

See also **cryptPopData, cryptPushData**.

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 set the CRYPT_CTXINFO_KEYSIZE attribute before calling this function.

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 **cryptGetAttributeString** function is used to obtain text or binary strings or time values 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 text or binary string or time value.

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_CERTIFICATE *certificate*, **const char** **oid*, **int** **criticalFlag*, **void** **extension*, **const int** *extensionMaxLength*, **int** **extensionLength* **);**

Parameters *cryptObject*
The certificate or public/private key object from which to read the extension.

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.

extensionMaxLength
The maximum size in bytes of the buffer to contain the extension data.

extensionLength
The length in bytes of the extension data.

Remarks cryptlib directly supports extensions from X.509, PKIX, SET, SigG, 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 owner's 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 the context to be fetched.

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 owner's 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 the context or certificate to be fetched.

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 used with cryptlib functions.

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(const void **certObject*, **const int** *certObjectLength*, **const CRYPT_USER** *cryptUser*, **CRYPT_CERTIFICATE** **certificate* **);**

Parameters

certObject
The address of a buffer that contains the encoded certificate.

certObjectLength
The encoded certificate length.

cryptUser
The user who is to own the imported object or CRYPT_UNUSED for the default, normal user.

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 that 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(const void **encryptedKey*, **const int** *encryptedKeyLength*, **const CRYPT_CONTEXT** *importContext*, **const CRYPT_CONTEXT** *sessionKeyContext* **);**

Parameters *encryptedKey*
The address of a buffer that contains the exported key created by **cryptExportKey**.

encryptedKeyLength
The length in bytes of the encrypted key data.

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) that 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. This function should be called before any other cryptlib function is called.

int cryptInit(*void* **);**

Parameters None

See also **cryptEnd**.

cryptKeysetClose

The **cryptKeysetClose** 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**.

cryptKeysetOpen

The **cryptKeysetOpen** function is used to establish a connection to a key collection or keyset.

int cryptKeysetOpen(CRYPT_KEYSET **keyset*, **const CRYPT_USER** *cryptUser*, **const CRYPT_KEYSET_TYPE** *keysetType*, **const char** **name*, **const CRYPT_KEYOPT_TYPE** *options* **);**

Parameters

keyset
The address of the keyset object to be created.

cryptUser
The user who is to own the keyset object or CRYPT_UNUSED for the default, normal user.

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**.

cryptPopData

The **cryptPopData** function is used to remove data from an envelope or session object.

int cryptPopData(const CRYPT_HANDLE *envelope*, **void** **buffer*, **const int** *length*, **int** **bytesCopied* **);**

Parameters

envelope
The envelope or session 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 or session object.

int cryptPushData(const CRYPT_HANDLE *envelope*, **const void** **buffer*, **const int** *length*, **int** **bytesCopied* **);**

Parameters

envelope
The envelope or session 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_TYPE *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 that don't apply to the algorithm being queried are set to CRYPT_ERROR, 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 **cryptDeviceQueryCapability**.

cryptQueryObject

The **cryptQueryObject** function is used to obtain information about an exported key object created with **cryptExportKey** or a signature object created with **cryptCreateSignature**. It returns information such as the type and algorithms used by the object.

int cryptQueryObject(const void **objectData*, **const int** *objectDataLength*, **CRYPT_OBJECT_INFO** **cryptObjectInfo* **);**

Parameters *objectData*
The address of a buffer that contains the object created by **cryptExportKey** or **cryptCreateSignature**.

objectDataLength
The length in bytes of the object data.

cryptObjectInfo
The address of a CRYPT_OBJECT_INFO structure that contains information on the exported key or signature.

Remarks Any fields in the CRYPT_OBJECT_INFO structure that don't apply to the object being queried are set to CRYPT_ERROR, 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 text or binary strings or time values 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 text or binary string or time 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 that 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**.

Acknowledgements

Alexey Kirichenko provided information on NtQuerySystemInfo for randomness-gathering under WinNT/Win2K to avoid the need to access the buggy Windows registry performance counters.

Brian Gladman wrote the AES code.

Chris Wedgwood and Paul Kendall helped write the Unix random data gathering routines.

endergone Zwiebeltüte helped debug the TLS implementation.

Eric Young and the OpenSSL team wrote portions of the conventional encryption and hashing code and bignum library.

Gary Nebbett helped diagnose a problem with Windows NPS.

Jean-Loup Gailly and Mark Adler wrote the zlib compression code.

Joerg Plate did the Amiga port.

Markus F.X.J. Oberhumer did the 32-bit DOS port.

Matt Thomlinson and Blake Coverett helped fix up and debug the Win32 random data gathering routines.

Matthijs van Duin, Sascha Kratky, and Jeff Lamarche did the Macintosh Classic port.

Nathan Hammond did the MVS port.

Osma Ahvenlampi did the PPC BeOS port.

Sami Tolvanen implemented the cryptlib GUI interface.

Sriram Ramachandran did the Cygwin port.

Steve Landers provided the Tcl bindings, with financial support from Eolas Technologies.

Stuart Woolford and Mario Korva did the OS/2 port.

Thomas Pornin did the ECDSA and ECDH code.

Trevor Perrin did the C#, Java, and Python bindings.

Wolfgang Gothier did the Delphi and Visual Basic bindings and tracked down a number of *really* obscure probl^H^H^H^H^Hundocumented features.