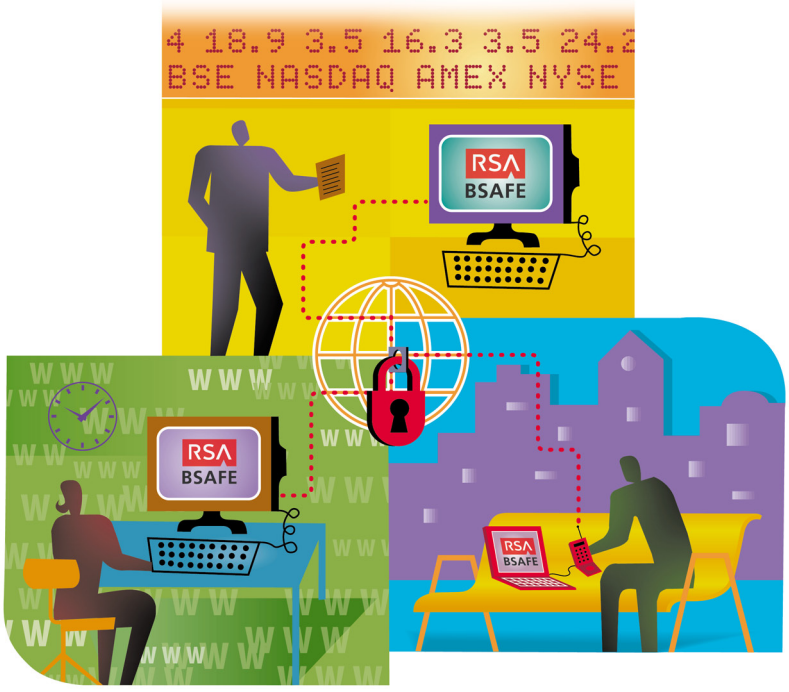
RSA BSAFE®
Cert-C
Basic Developer's Guide

Version 2.7

Certificate management components for C



RSA SECURITY®

Contact Information

See our Web sites for regional Customer Support telephone and fax numbers.

RSA Security Inc.
www.rsasecurity.com

RSA Security Ireland Limited
www.rsasecurity.ie

Trademarks

License Agreement

Third Party Licenses

This product may include software developed by parties other than RSA Security. The text of the license agreements applicable to third party software in this product may be viewed in the `thirdpartylicense.pdf` file.

Note on Encryption Technologies

This product may contain encryption technology. Many countries prohibit or restrict the use, import or export of encryption technologies and current use, import and export regulations should be followed when exporting this product.

Distribution

Limit distribution of this document to trusted personnel.

RSA Security Notice

The RC5® Block Encryption Algorithm With Data-Dependent Rotations is protected by U.S. Patent #5,724,428 and #5,835,600.

Compaq MultiPrime™ technology is protected by U.S. Patent #5,848,159 and is the subject of patent applications in other countries.

This product includes patented technology licensed from Entrust Technologies Inc. (US Patent# 5,699,431).

First printing: March, 2003

001-046014-270-001-000 (SAP P/N: 3904)

Contents

Preface

Dear Cert-C Developer,

Congratulations on your purchase of RSA BSAFE® Cert-C 2.7 (Cert-C), the software development toolkit (SDK) that enables you to quickly and efficiently integrate Public-Key Infrastructure components into your applications. This SDK enables you to develop certificate management engines for a wide range of purposes, including electronic commerce, enterprise security, and certificate issuing. Cert-C is built on top of RSA Security's cryptographic engine: RSA BSAFE Crypto-C.

Cert-C is written in C and is intended to be completely portable. It is available on a number of platforms and can be ported to most platforms with a minimum of effort. Cert-C is an SDK, not an application. It is intended to be integrated into your application. Therefore, you have a modest amount of work ahead of you. We have tried to make this task as clear as possible without constraining your alternatives. This *RSA BSAFE Cert-C Basic Developer's Guide*, with its examples, is the best place to start.

Please feel free to share with us any suggestions you have, bugs you find, or improvements you would suggest for the next version of the *Basic Developer's Guide*. E-mail your comments to `bsafeuserdocs@rsasecurity.com`. Any comments will help future Cert-C application developers.

Thanks, and welcome to the RSA Security family.

Sincerely,

The RSA BSAFE Cert-C Development Team, RSA Security.

How This Book Is Organized

This book is organized into the following chapters and appendixes:

- Chapter 1, “Introduction,” includes an overview of the Cert-C architecture.
- Chapter 2, “RSA Security Concepts,” introduces some security protocols and PKI concepts.
- Chapter 3, “Cert-C Setup,” provides installation, build, and deployment information.
- Chapter 4, “Getting Started,” introduces the Cert-C objects and outlines Cert-C programmatic information you need to know before you start developing an application that uses Cert-C.
- Chapter 5, “Cert-C Context and Services,” explains how to initialize the Cert-C context and the various Cert-C services.
- Chapter 6, “Using the List Object,” presents the list object and its APIs, and provides examples to explain how to use the list object.
- Chapter 7, “Using the Name and Attributes Objects,” presents the name and attributes objects, along with their respective APIs, and provides examples that show you how to create a name object and an attributes object.
- Chapter 8, “Creating a Certificate Request,” discusses the PKCS #10 certificate request and the PKI request message formats.
- Chapter 9, “Creating a PKI Message,” presents the PKI message object and the various types of PKI request and response objects, and provides a general example for creating a PKI message.
- Chapter 10, “Creating an X.509 Certificate,” introduces the certificate object, and provides examples that show you how to build an X.509 certificate.
- Chapter 11, “Verifying Certificates and CRLs,” provides examples that show you how to verify the signature on a certificate and a CRL.
- Chapter 12, “Storing and Retrieving Certificates, CRLs, and Private Keys,” provides examples that show you how to store and retrieve a certificate, a key, and a CRL.
- Chapter 13, “Retrieving Certificate Information,” provides examples that show you how to retrieve information from a certificate object, an attributes object, and an extensions object.
- Chapter 14, “CRL and CRL Entries,” discusses the CRL and CRL entries objects, and explains how to create a CRL and CRL entries object.

- Chapter 15, “Extensions,” presents the extensions object and its APIs, and provides examples that show you how to create an extensions object and user-defined extensions.
- Appendix A, “Using BSAFE Crypto-C,” briefly explains how to use Crypto-C functionality with Cert-C.
- Appendix B, “BCERT Compatibility,” discusses issues developers need to consider when migrating BCERT applications to Cert-C.
- Appendix C, “References,” lists the standards to which the Cert-C SDK conforms.

Cert-C Documentation Redesign

In Cert-C 2.5, the documentation set was completely redesigned. The reference information in now online, in HTML format, (the *RSA BSAFE Cert-C API Reference*), there is a new *Basic Developer's Guide*, as well as the existing *RSA BSAFE Cert-C Developer's Guide*, now called the *RSA BSAFE Cert-C Advanced Developer's Guide*. The old *RSA BSAFE Cert-C Service Provider Manual* is also online. It is included in the *API Reference*.

The *API Reference* will increase your ease of use in developing your products. You can look up a function's usage while implementing Cert-C into your application. For example, while programming, if you need to know more about a function, call up the help file and access the function's description and usage online. You can look up any function by header file name or alphabetically from the global list. If you need to know more about a particular structure that a function uses, then you simply click on the structure name to navigate to the structure's description.

The *Basic Developer's Guide* is designed to get you using Cert-C with your application faster than ever. It presents essential information, without overloading you with more advanced topics.

The *Advanced Developer's Guide* concentrates on more advanced topics and includes more sophisticated examples and samples.

Cert-C Documentation Map

This book is written for developers and independent software vendors who want to use Cert-C to add Public-Key Infrastructure (PKI) client functionality to applications.

Core Documentation

All users of Cert-C need the following core documentation.

Basic Developer's Guide

This book, the *RSA BSAFE Cert-C Basic Developer's Guide*, introduces the Cert-C architecture and background PKI concepts. It presents the essential information you need to start integrating Cert-C with your application. It provides basic, step-by-step examples to help you get started. It takes you through initializing the Cert-C context and services, and shows you the basic steps for general certificate management tasks. For example, how to build a certificate request, send a certificate request, or store a certificate. This book is the best place for a developer to start working with Cert-C.

Advanced Developer's Guide

The *RSA BSAFE Cert-C Advanced Developer's Guide*, introduces more sophisticated Cert-C functionality. It explains how to use advanced Cert-C PKI functions and services, and provides example and sample code with tutorials.

API Reference

The *RSA BSAFE Cert-C API Reference* contains complete documentation for the Cert-C API library, including certificate function calls. It describes the Cert-C API functions, structures, types, parameters, and return values.

The *API Reference* also includes a "Service Provider" section. This section defines the Cert-C service providers and their associated utility programs. Each instance of a Cert-C service-provider type behaves in a specific manner to provide a specific service.

Additional Documentation

Additional documentation helps advanced developers customize the Cert-C API and build additional services.

Crypto-C Developer's Guide & API Reference

The *RSA BSAFE Crypto-C Developer's Guide* introduces the developer to RSA Security's cryptographic architecture and background concepts. It describes the basics of encryption and decryption, the steps involved in choosing algorithms and keys, message digesting, and random number generation.

The *RSA BSAFE Crypto-C API Reference* describes the Crypto-C API library in detail.

Readme

The `readme-certc.txt` file contains the very latest information about the Cert-C product, and is located in the `Cert-C2.7` folder. This information is limited to bugs that were found close to software release time. See the *Release Notes* for more detailed release information.

Release Notes

The `certc_27_releasenotes.pdf` file contains detailed information about this Cert-C software release. It is located in the `Cert-C2.7` folder and in the `doc` directory. It covers known software bugs and their recommended workarounds, and other information specific to this release. The very latest information can be found in the `readme-certc.txt` file, located in the `Cert-C2.7` folder.

Cert-C Installation Guide

The `certc_27_install.pdf` file contains all the information you need to install Cert-C, and is located in the `Cert-C2.7` folder. See chapter 3, "Cert-C Setup" on page 49 of this book, the *Basic Developer's Guide*, for information about how to customize and use the Cert-C SDK.

How to Contact RSA Security

RSA Security Web Site

You can visit the RSA Security Web site at www.rsasecurity.com. It contains the latest RSA Security news, security bulletins, and information about coming events.

RSA BSAFE product information is available at www.rsasecurity.com/products/bsafe.

RSA Laboratories' Cryptography FAQ can also be found at www.rsasecurity.com/rsalabs/faq.

Getting Support and Service

You can get technical support as follows:

SecurCare® Online

www.rsasecurity.com/support/securcare

Technical Support Information

www.rsasecurity.com/support

Chapter 1

Introduction

What Is RSA BSAFE Cert-C?

The task of integrating Public-Key Infrastructure (PKI) security with server and end-user applications can be a complex and lengthy process. The application developer would need expertise in several areas, including X.509 certificate management and cryptography. The RSA BSAFE Cert-C (Cert-C) SDK addresses this complexity by providing a secure, ready-made certificate management engine and cryptographic engine to the application developer. Cert-C enables the application developer to add digital signature, certificate, and key-management functions to any application secured with RSA Security products—quickly, easily, and with confidence.

Cert-C is the state-of-the-art PKI API software library, written in C, that combines private key, trust, and certificate databases to request, retrieve, verify, store, and manage private keys and certificates for signing or encrypting messages.

Cert-C contains the cryptographic support necessary to generate certificate requests, sign certificates, and create and distribute Certificate Revocation Lists (CRLs). Cert-C is built upon the RSA BSAFE Crypto-C cryptographic engine.

The Cert-C Components

Cert-C provides the following components:

- Cert-C core functionality—The internal Cert-C static library.
- Cert-C API—The interface between your application and the PKI functionality.
- Cert-C SPI—PKI functionality to create your own service provider.
- Cert-C Service Providers—RSA Security's implementation of the Cert-C SPI.
- Cert-C Utilities—Specialized routines to help create, run, and test your PKI applications.

In addition, Cert-C includes the Crypto-C cryptographic component.

- Crypto-C API—Cryptographic functionality to perform the underlying cryptographic algorithms required by the public-key infrastructure.

Cert-C Features

The Cert-C SDK provides the following certificate management features:

- Generate a public/private key pair, utilizing Crypto-C.
- Certify a public/private key pair and store the resulting digital certificate. This functionality is implemented through the various Cert-C PKI service providers.
- Certificate management protocols—CRS, CMP, and SCEP.
  - CRS—Certificate requests and responses to CAs that implement the CRS protocol. This functionality is implemented through the Cert-C CRS PKI service provider according to the *VeriSign CRS Profile Specification*, available directly from VeriSign.
  - CMP—Certificate requests and responses, certificate revocation, key archival, and key update requesting to CAs that implement the CMP protocol. This functionality is implemented through the Cert-C CMP PKI service provider according to the profile outlined in *RFC 2510* and *RFC 2511* for CMP version 1 messages, and `draft-ietf-pkix-rfc2510bis-06.txt` and `draft-ietf-pkix-rfc2511bis-04.txt` for CMP version 2 messages.
  - SCEP—Certificate requests and responses to CAs that implement the Cisco Systems' *Simple Certificate Enrollment Protocol* certificate request mechanism. This functionality is implemented through the Cert-C SCEP PKI and Cert-C SCEP Database service providers.
- Maintain a data store for private keys, personal digital certificates, digital certificates belonging to others, and certificate revocation information. This functionality is implemented through the various Cert-C database service providers.
- Determine trust in the certificate authority (CA) or chain of CAs for a given certificate. Includes PKIX support for policy mapping and CRL distribution points and related extensions in path validation and certificate status checking. This functionality is implemented through the Cert-C Certificate Path Processing service provider according to the profiles outlined in X.509 v1, *RFC 2459*, and *RFC 3280*.
- Check certificate revocation status—CRL and OCSP.
  - CRL—Certificate revocation status checking using certificate revocation lists. This functionality is implemented through the Cert-C CRL Revocation Status service provider.

  - OCSP—Online certificate revocation status checking against a responder that implements the Online Certificate Status Protocol. This functionality is implemented through the Cert-C OCSP Revocation Status service provider.
- Encrypt, decrypt, sign, and verify data, utilizing Crypto-C.
- Import keys and certificates from other sources, according to the X.509, PKCS #7, PKCS #8, PKCS #11, and PKCS #12 standards.
- Export private keys and certificates to other sources, according to X.509, PKCS #7, PKCS #8, PKCS #11, and PKCS #12 standards.
- PKCS #11—Authenticated read-write access to certificates and private keys on PKCS #11 tokens, and the ability to perform cryptographic operations with those keys. This functionality is implemented through the Cert-C PKCS #11 Database and Cert-C Default Cryptographic service providers.

You can create applications, using Cert-C, that automatically and seamlessly interoperate with multiple existing PKI products available in the market today, including RSA Keon® Certificate Authority and VeriSign's OnSite certificate service. Cert-C takes a standards-based approach, which minimizes the effort needed to write applications for each PKI that needs to be supported. Cert-C supports PKCS #7, #8, #10, #11, and #12, LDAP, X.509, and the following PKIX standards: OCSP, SCEP, CRMF, CMP, and CRS.

Cert-C provides PKI functionality to your application through its application programming interface (API), which interfaces between your application and the internal Cert-C library or a service provider. The API enables your application to select a Cert-C service provider, a third-party service provider, or your own custom-built service provider. Cert-C also includes a Service Provider Interface (SPI), which enables you to create your own custom-built service provider.

New Features in Cert-C 2.7

RSA Security has added some new features to the Cert-C SDK. As always, RSA Security has developed these features according to industry standards so that you do not have to redevelop your applications when you decide to interoperate with a different PKI. The following is a list of the new features included in the Cert-C 2.7 SDK:

- Additional messaging objects and APIs for streaming PKCS #7 Data and EnvelopedData. These objects and APIs are documented in `cmsobj.h`. They can be used to stream PKCS #7 message input and output and to minimize memory use. An additional sample has also been provided to demonstrate this functionality.
- Performance enhancements to certificate path building and validation providers, including new APIs in `certlist.h` for adding certificate and CRL objects to `LIST_OBJ`s without the overhead of making deep copies of the added objects.
- Incorporation of RSA BSAFE Crypto-C 6.1 to provide the highest quality cryptographic technology for securing applications.
- The Cryptographic service provider now uses, by default, the full-blinding implementation of the RSA operation to help prevent against timing attacks.
- The Mozilla LDAP client libraries have replaced the iPlanet shared libraries on the HP 11.00 64-bit platform. All platforms now use the Mozilla LDAP client libraries.

Cert-C Architecture

The Cert-C SDK provides many essential programmatic components needed in a public-key infrastructure. Figure 1-1 shows the following layers of functionality that make up the Cert-C architecture:

- Application programming interface (API)
- Internal static libraries
- Service provider interface (SPI)
- Service Providers

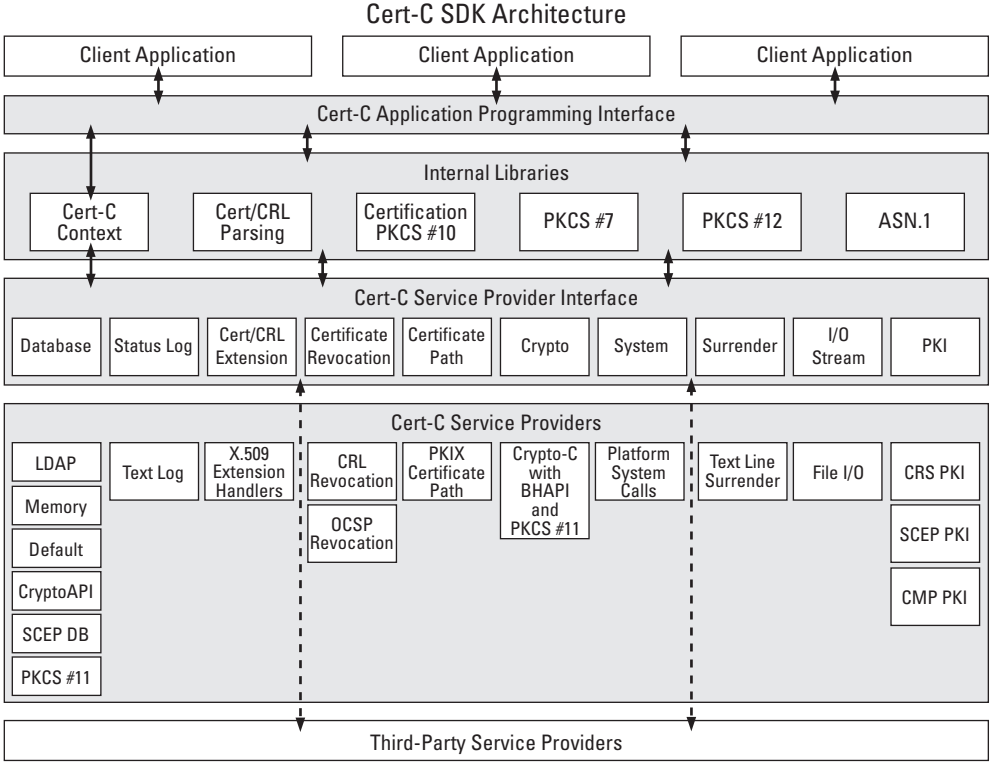


Figure 1-1 **Cert-C SDK Architecture**

Your Application

Your application sits on top of the Cert-C SDK, as Figure 1-2 shows. The Cert-C SDK provides an API layer; this API layer is the primary part of the Cert-C SDK with which your application needs to interface. Some APIs are the interface between your application and the internal Cert-C library, while others are the interface between your application and a selected service provider.

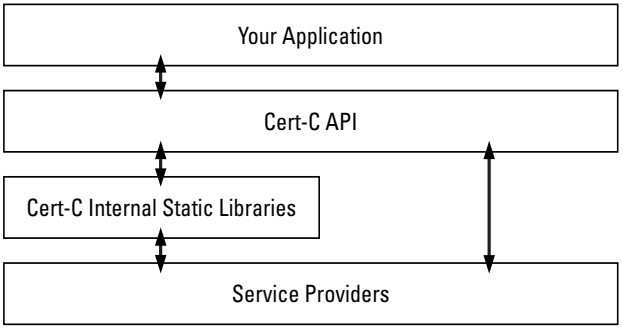


Figure 1-2 **Cert-C API**

Cert-C API

The Cert-C SDK provides PKI functionality to your application through its application programming interface; this API layer is the primary part of the Cert-C architecture with which your application needs to interface. This API can be categorized into two types of APIs. The first type of API gives your application an interface to the internal Cert-C library, where standard PKI functionality is provided. The second type of API provides additional PKI functionality, interfacing with service providers. This last type of API enables your application to select a Cert-C service provider, a third-party service provider, or a service provider created by you, therefore providing greater flexibility.

If your application uses a service provider, you might need to provide your application with information about the selected service provider. Also, not all service providers are linked to an API function call. Cert-C is designed with a context management component to assist applications in specifying and managing the numerous parameters and service providers.

Cert-C Context

The Cert-C context, shown in Figure 1-3, collects a number of common parameters and state variables together. It manages the Cert-C and service provider initialize and finalize functions. It also tracks the currently registered service providers, manages service-provider register and unregister functions, ordering and grouping of service providers, and binding and unbinding service providers with a service handle.

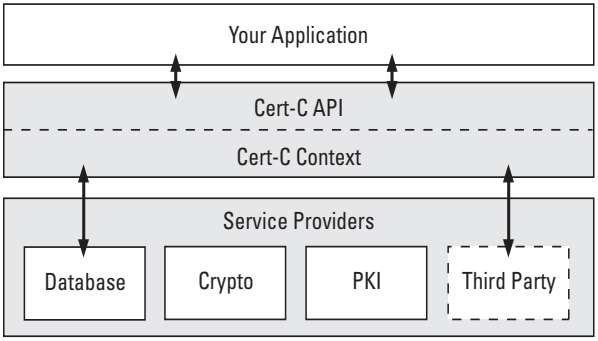


Figure 1-3 **Cert-C Context**

Note: When performing operations that require a Cert-C context, make sure you use the correct context. For more information about how to use a Cert-C context, see "Cert-C Context" on page 66.

Cert-C Initialization

The Cert-C context is established when your application calls the `C_InitializeCertC` function. This function allocates the application's context and initializes the internal fields of the context. It also initializes any service providers passed by the *`handlers`* parameter, defined as a `SERVICE_HANDLER` data structure. This data structure provides the Cert-C API with the service-provider information.

The `C_FinalizeCertC` function unregisters all currently registered service providers, frees all memory associated with the context, and sets the context handle to `NULL_PTR`.

Cert-C Service Providers

In addition to the PKI functionality provided in the internal Cert-C library, the Cert-C SDK also provides PKI functionality through the use of service providers, as shown in

Figure 1-4. Your application chooses a service-provider type and a specific service provider within that type: either one of Cert-C's service providers or a third-party's service provider. Your application can register or dynamically bind the service provider. The service provider's specifications customize the API function calls that interface to your application. For more information about RSA Security's Cert-C service-provider implementations, see the "Service Provider" section of the *API Reference*.

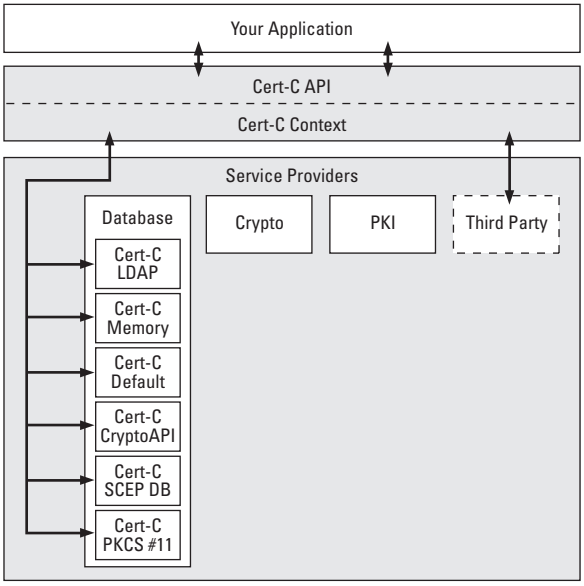


Figure 1-4 **Cert-C Service Providers**

Service-Provider Initialization

When the Cert-C context is established using the `C_InitializeCertC` function, all currently registered service providers are initialized. Service providers are initialized in order of type and in the specified order within each type. The service provider type order is `SPT_SURRENDER`, `SPT_LOG`, `SPT_CRYPTO`, `SPT_IO`, `SPT_DATABASE`, `SPT_DATABASE2`, `SPT_CERT_STATUS`, `SPT_CERT_PATH`, and `SPT_PKI`. There are also ways to initialize and uninitialize service providers dynamically. For more information, see "Dynamic Service-Provider Registration" on page 28.

Service-Provider Information

A service-provider instance is defined by its type, an instance name, and an initialization function. The SERVICE_HANDLER data structure contains the service-provider information. In general, distinct instances of a service provider may have the same type and initialization function, but the name and the initialization function's parameters must be unique. Note, however, that only a single instance (each) of type SPT_SURRENDER or SPT_CRYPTO may be registered.

Some Cert-C function calls require a SERVICE handle as an input parameter. The SERVICE handle can be bound to a single service-provider instance, or it can represent a sequence of service-provider instances, all of the same type. C_BindService is used to bind SERVICE to a service provider. If you want to bind SERVICE to more than one service provider, use the C_BindServices function. The SERVICE handle is useful when you want to define a subset of service providers for a particular operation.

Service-Provider Functions

Whenever a service provider is initialized, the service provider specifies a set of function pointers that are the service provider's entry points. The Cert-C context uses the SERVICE_FUNCS union to access the service provider's type-specific entry points.

Service-Provider Registering

When you register a new service provider, you can specify that the new service provider should be inserted before or after all other service providers of the same type. The service provider order affects the behavior of functions that use service providers of a given type. The SERVICE_ORDER_FIRST constant indicates the service provider should be inserted before others of the same type. The SERVICE_ORDER_LAST constant indicates the service provider should be inserted after others of the same type. The C_RegisterService function takes one of these constants as an input value in its *order* field.

Dynamic Service-Provider Registration

You can register additional service providers, subsequent to the Cert-C context initialization, by calling the C_RegisterService function. This function calls the service provider's initialization function and adds an entry for the service provider in the Cert-C context's internal list of service providers.

Service-Provider Unregistering

You can unregister a service provider before calling the `C_FinalizeCertC` function. When you do so, the service handler with the specified type and name is removed from the Cert-C context, the service provider's finalize function is called, and the memory associated with the context's copy of the service handler is freed.

Cert-C automatically unregisters all currently registered service providers associated with a particular Cert-C context when the Cert-C context is finalized. You do not need to call `C_UnregisterService` if the next Cert-C function call is `C_FinalizeCertC`.

You must be careful to ensure that the service provider you want to unregister is not bound to any service handles. Using a service handle that includes an unregistered service provider may cause the application to crash. You should call the `C_UnbindService` function before you unregister the service provider.

Service-Provider Binding

In some situations, you may want to bind one or more currently registered service providers to a service handle. A service handle is required for some Cert-C API functions calls that can be directed to a particular service provider or set of service providers. By calling either the `C_BindService` or `C_BindServices` function, you create a service reference (handle) that can be used as a parameter to Cert-C functions. This handle can target a specific service provider or set of service providers.

You should call `C_BindService` (as opposed to `C_BindServices`) in situations where a single, currently registered service provider must be bound to a service handle. The `C_BindServices` function binds one or more currently registered service providers to a service handle.

Some service provider types (for example, `SPT_DATABASE`) allow an ordered list of instances to be specified in the `C_BindService`'s *`name`* array (or `C_BindServices` *`names`* array). If a `NULL_PTR` is specified for the *`name`* (or *`names`*) array, all of the service provider instances of the given type are bound in registration order.

Service Provider Unbinding

You can unbind all service providers bound to a service handle by calling the `C_UnbindService` function. This function undoes a previous binding of service providers to the specified handle and frees any memory allocated by the corresponding `C_BindService(s)` function call.

Surrender Context

Some Cert-C functions are time-consuming. When an application calls one of these functions, it can appear as if the computer has crashed or frozen. A lengthy Cert-C function can tie up the computer, forcing other applications or programs to wait until the Cert-C function is finished before completing their execution. The Cert-C SDK includes a surrender context, `A_SURRENDER_CTX`; this gives you a way to enable Cert-C to surrender control. It contains a pointer to the application-specific `Surrender` callback function that Cert-C calls to surrender control to the application. You should call the `C_GetSurrenderCtx` function to return a pointer to a surrender context whose contents are defined by the currently registered surrender context service provider. For more information about `A_SURRENDER_CTX` and `C_GetSurrenderCtx`, see the *API Reference*.

Cert-C SPI

The Cert-C SDK includes Cert-C implementations for each type of service provider defined by the SPI. You can use a Cert-C service provider or choose a third-party service provider. Alternatively, you may want to create your own service provider. The Cert-C SDK provides an SPI for those who want to create a custom-built service provider. Cert-C also provides most of the source code for the Cert-C service providers; this source code can be used as a starting point for creating a custom service provider.

When initialized, your custom-built service provider's specifications customize the API function calls that interface to your application. If you are considering creating your own custom-built service provider, see "The SPI Architecture" section of the *API Reference* as your primary reference source.

Chapter 2

RSA Security Concepts

Before you use Cert-C, you should be familiar with the following basic cryptographic, certificate management, and security concepts.

Secret-Key Cryptography

Secret-key or symmetric-key cryptography uses only one key to encrypt and decrypt data. Therefore, the secret key must be known only to the originator and the individual with whom the originator wants to share the secret data. The advantage of secret-key cryptography is that it encrypts large amounts of data relatively efficiently. However, a disadvantage to using secret-key cryptography is finding a way to securely distribute the secret key.

Public-Key Cryptography

Public-key or asymmetric cryptography uses two keys, one public and the other private. These keys are otherwise known as a key pair. The private key must be kept a secret, while the public key can be transmitted in the clear to other parties. The private key and the public key are mathematically related. A message that is signed by a private key can be verified by the corresponding public key. Similarly, a message encrypted by the public key can be decrypted by the private key. This method ensures privacy because only the owner of the private key can decrypt the message. Both of

these methods can be combined to provide secrecy and to verify the origination of data.

Key Management

To sign messages or to send and receive encrypted messages, each user must have a key pair. Users may have more than one key pair; for example, one key pair for work and another key pair for personal use. Other entities may also have key pairs, including electronic entities such as a modem, workstation, or printer, and organizational entities such as a corporate department, hotel registration desk, or university registrar's office.

A corporation may require more than one key pair for communication. For example, one or more key pairs may be used for encryption and a single key pair may be used for authentication. The lengths of the encryption and authentication key pairs vary according to the desired level of security. Generally, longer key lengths provide greater security.

Users can generate their own key pairs or, depending on local policy, a security officer may generate key pairs for a group of users. There are advantages and disadvantages to both approaches. With the former approach, users must trust their copies of the key-generation software. With the latter approach, users must trust the security officer and private keys must be securely transferred to users.

Once generated, users must register their public keys with a central administrative body, called a certificate authority (CA). They accomplish this by generating a certificate request (which contains their public key) and then submitting it to the CA. The CA returns to each user a certificate that attests to the validity of the user's public key, along with other information. If a security officer generates the key pair, then the security officer can request the certificate for the user. Most users should get only one certificate for a key, so that bookkeeping tasks associated with the key remain uncomplicated.

Instead of registering their certificates with a CA, users may sign certificates themselves, which commonly occurs for trusted roots. This kind of certificate is called a self-signed certificate.

Private keys must be stored securely because their compromise can lead to loss of privacy and forgery. The measures taken to protect a private key must be, at a minimum, equal to the security measures taken when encrypting a message with the private key. A private key should never be stored anywhere in plaintext form. The simplest storage mechanism is to encrypt the private key under a password and store the result on a disk. However, because some passwords can easily be guessed,

passwords should be chosen very carefully. The Cert-C sample program `keywrap.c` demonstrates the use of password-based encryption (PBE) to secure a private key, as specified in Public-Key Cryptography Standard (PKCS) #8.

If an encrypted key is stored on a disk that is not accessible through a computer network, such as a floppy disk or a local hard disk, some security attacks are more difficult. It may be best to store the key on a computer that is not accessible to other users, or to store the key on removable media that users can take with them when they finish using a particular computer. Private keys can also be stored on portable hardware, such as smart cards. Users with extremely high security needs, such as CAs, should use these kinds of special hardware devices to protect their keys.

Digital Signatures

A digital signature is an electronic mark on data that identifies the signer and ensures the integrity of the signed data. It can be compared to a handwritten signature in that the mark can be produced by only one person, the signer. The digital signature also ensures that the signed data did not change from the time it was signed to the time it is checked. Figure 2-1 shows how a digital signature is created by performing the following two steps:

- Using a message-digest algorithm, compute the message digest of the data to be signed.
- Sign the message digest with the signer's private key.

A message-digest algorithm is similar to a checksum in that it always produces the same size output for any size input. A message-digest algorithm is cryptographically stronger than a checksum and makes it infeasible to find two meaningful messages with the same message digest.

The original data can now be transmitted and verified by anyone with knowledge of the signer's public key. The person receiving the signed data can verify the signer's signature and check the integrity of the data by performing the following three steps:

- Get the message digest from the signature using the signer's public key.
- Using the same message-digest algorithm, compute the message digest of the original data.

- Compare the decrypted message digest to the result of the message digest computed independently on the same data.

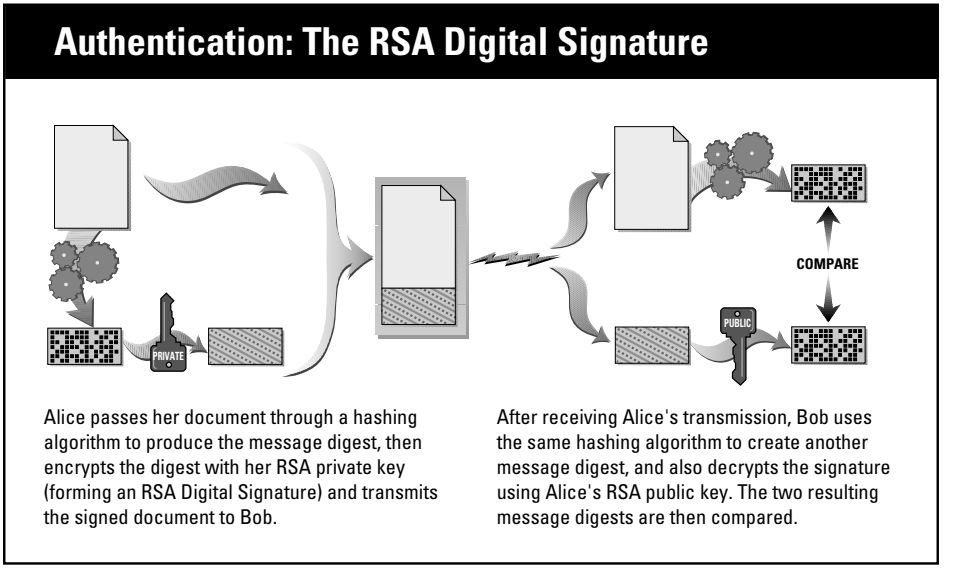


Figure 2-1 **Authentication: The RSA Digital Signature**

Digital Certificates

A digital certificate, or simply a certificate, is a digital document that attests to the identity of an individual or an entity. An entity can be an individual, an organization, a piece of software, or a hardware device. A certificate acts as the binding between the individual and the individual's public key; the private key of the individual is kept secret. Possession of the certificate's private key, which mathematically relates to the certificate's public key, verifies the individual's identity. In this way, a certificate helps to prevent someone from using an inauthentic key to impersonate someone else. The entity identified by a certificate is referred to as the certificate's subject or subscriber.

The purpose of a certificate is to verify the identity of an individual or entity; however, it can also be used to digitally sign or encrypt data, to control access to resources, or to implement nonrepudiation.

In its simplest form, a certificate contains an individual's public key and name, a validity period, the name of the CA that issued the certificate, a serial number, and a

signature algorithm identifier. Most importantly, the certificate contains the digital signature of the certificate's issuer.

Just as an individual's driver's license is issued by a trusted third party—in the United States, the Department of Motor Vehicles—a certificate must be issued by a trusted third party. This trusted third party is called a CA. A CA verifies a certificate requester's identity, creates a certificate, and then digitally signs the certificate with the CA's private key. The CA does this by computing the certificate's message digest and then signing it with its own private key. CAs also provide a way to distribute public keys or certificates in the public domain.

The most common form of authentication involves enclosing one or more certificates with a signed message. The recipient of the message first verifies the sender's certificate (or certificates) using the CA's public key and, now confident of the sender's public key, verifies the message's signature. The sender's certificate(s), in conjunction with one or more trusted certificates (or keys) already possessed by the recipient, form a hierarchical chain, where one certificate attests to the authenticity of the previous certificate. At the end of a certificate hierarchy chain is a top-level CA, or root certificate. The root certificate is trusted without a certificate from any other CA because it is self-signed. The public key of the top-level CA must be independently known, for example, by being widely published.

Even if no certificates are enclosed with a signed message, a verifier can still use a certificate chain to check the status of the public key. The verifier can simply look up the certificates; for example, in a data store. Specifically, each signature contains the certificate issuer's name and the certificate's serial number. (In a self-signed certificate, the issuer name is the same as the subject name.)

Extension Fields

Extension fields contain additional information, either critical or noncritical, about a certificate or CRL. An extension field has three parts: extension type, extension criticality, and extension value. The extension criticality instructs a certificate-using application on whether it may ignore an extension. If the extension criticality is set to critical and the extension is not recognized by an application, it should reject the certificate. On the other hand, if the extension criticality is set to noncritical and the application does not recognize the extension, it is safe for the application to ignore the extension and to use the certificate.

Extension fields provide a way to associate additional information with the user's identity and public key. Some of the fields can provide additional information about the user. Other fields can contain information on the intended use of the public key

(for example, the key pair used for authentication or digital envelopes). Additional fields can be used to locate other related certificates and certificate status information.

CRL Distribution Points

CRL distribution points is an extension that identifies how CRL information is obtained.

Attribute Fields

An attribute field is similar to an extension field in that it provides flexibility and scalability. However, the attribute field is used to request certificates within the constraints of PKCS #10, the *Certificate Request Syntax Standard*. The certificate request usually includes the Distinguished Name (DN) and public key of the user, along with a set of attributes. Each attribute has an attribute type and a set of one or possibly more values. An attribute type such as the time at which a message is signed has only one value, whereas an attribute type such as a postal address may have multiple values. PKCS #7 Signed-Data messages can also have attribute fields. PKCS #9 and X.520 specify some of these standard attribute types.

Digital Envelopes

A digital envelope is a way to send a message privately from sender to recipient, while also providing authentication of the sender.

A digital envelope combines the advantages of symmetric key and public-key cryptography. Public-key algorithms are generally slower than symmetric-key ciphers; for some applications they may be too slow to be practical. Symmetric-key ciphers, however, present the problem of transmitting the key securely. As Figure 2-2 shows, a digital envelope provides a solution to this dilemma.

The sender encrypts the message using a symmetric-key encryption algorithm, and then encrypts the symmetric key using the recipient's public key. The recipient then decrypts the symmetric key using the appropriate private key and decrypts the message with the symmetric key. In this way, a fast encryption method processes

large amounts of data, yet secret information is never transmitted unencrypted.

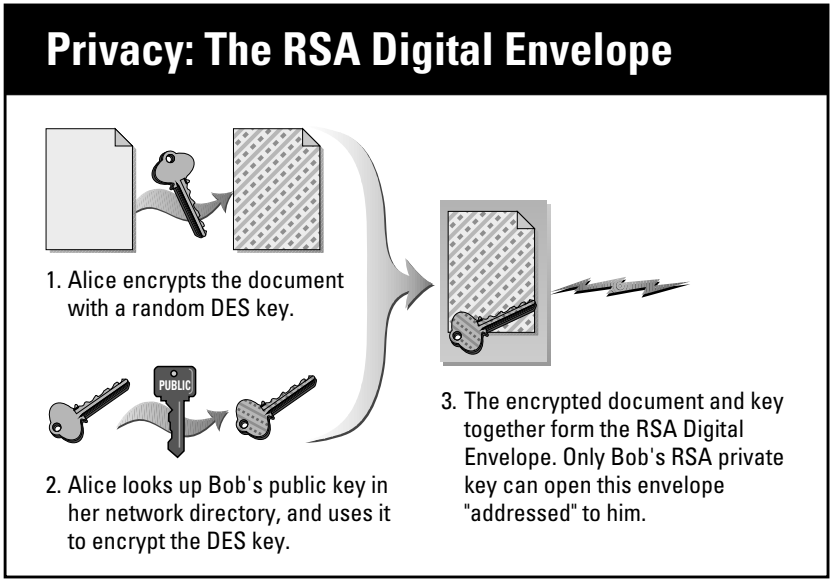


Figure 2-2 **Privacy: The RSA Digital Envelope**

Certificate Authority

A CA is any trusted central administrative body that accepts certificate applications from entities, authenticates the entity, vouches for the identities of these entities by issuing certificates, and maintains status information about these certificates. A company may issue certificates to its employees, a university may issue certificates to its students, and a town may issue certificates to its citizens.

To prevent forged certificates, the CA's public key must be trustworthy. A CA must either publicize its public key or provide a certificate from a higher-level CA that attests to the validity of its public key. The latter solution involves a CA hierarchy. The root key is the public key associated with the top of a certificate hierarchy. Unlike other public keys within certificates in the certificate chain, a root key can be trusted by some means other than a certificate. For example, a root key may be widely published in a major periodical or standards document. A root key may also be published as a self-signed root certificate.

An example of certificate issuance proceeds as follows. Alice generates her own key pair and sends the public key to an appropriate CA with some proof of her

identification. The CA checks the identification and takes any other steps necessary to ensure that the request did come from Alice. The CA then sends her a certificate that attests to the binding between Alice and her public key, along with a certificate hierarchy, or certificate chain, that verifies the CA's public key. Alice can present this certificate chain whenever desired to demonstrate the legitimacy of her public key.

Because the CA must check for proper identification, an organization usually finds it convenient to act as a CA for its own members or employees. Different CAs may have different identification requirements. One CA may require presentation of driver's licenses, another may want certificate request forms to be notarized, and yet another may want the fingerprints of those who request certificates.

To notify users as to whether their private keys have been compromised, CAs may regularly issue CRLs. Each CA should publish its own identification requirements and standards in a policy statement so that verifiers can attach the appropriate level of confidence in the certified name-key bindings. CAs with lower levels of identification requirements produce certificates with lower assurance.

Public-key certificates, such as X.509 certificates, determine the skeletal structure of trust within a distributed public-key cryptosystem. By signing a certificate, a certificate issuer binds together an entity's public key with the entity's name and other information. By verifying the signature on the certificate, someone who trusts the certificate issuer can develop trust in the entity's public key.

Because certificates are an essential part of an interoperable public-key standard, PKCS adopted the use of X.509 certificates, which maintains compatibility with other users of the X.509 standard.

Certificate Chaining

Certificate chaining is a method used to verify the binding between an entity and the entity's public key. To gain trust in a certificate, a certificate-using application must verify the following about each certificate until it reaches a trusted root:

- Each certificate in the chain is signed by the public key of the next certificate in the chain.
- Each certificate is not expired or revoked.
- Each certificate conforms to a set of criteria defined by certificates higher up in the chain.

By verifying the trusted root for the certificate, a certificate-using application that trusts the certificate issuer can develop trust in the entity's public key.

Push Model Versus Pull Model

The chaining described in the previous section relies on individuals having access to all the certificates in the chain. There are two ways to get these certificates: the push model and the pull model. In the push model, the sender pushes an entire chain of certificates when sending one certificate to the recipient, and the recipient can immediately verify all the certificates. In the pull model, the sender pushes only the sender's certificate, and then leaves it up to the recipient to pull in the CA's certificate.

Because each certificate contains the issuer name, the recipient knows where to go. X.509 v3 certificates offer more opportunities to place information in a certificate to make searches easier; see "X.509 v3 Certificate Extensions" on page 253 for more information. Even with the push model, some recipient chaining may be necessary. If two people trying to communicate both have certificates, but each individual's chain leads to a different trusted root, they will have to somehow find a link between the two hierarchies. As users and CAs receive more certificates, they will probably want to keep them in a database. That way, future certificate checks can be made more easy. In practice, the push model is commonly used with entire certificate chains sent in PKCS #7 Signed-Data messages.

Trusted Root

A trusted root is a root certificate, or top-level CA certificate, which can be trusted by a certificate-using application. A CA must either publicize its public key, also known as a root key, or provide a certificate from a higher-level CA that attests to the validity of its public key. A certificate-using application may import, store, and use the trusted root keys of several CAs.

When a certificate-using application verifies a certificate, it follows the certificate's certificate chain path to its root certificate. This root certificate acts as the final point of trust when verifying a certificate. A root certificate's root key, unlike other public keys, is not followed by another certificate to verify its trust. The root key is sometimes trusted by some means other than a certificate. For example, a root key may be widely published in a major periodical or standards document. Or, more commonly, a root key may also be published as a self-signed root certificate.

Certificate Revocation List

A Certificate Revocation List (CRL) is a digitally signed document that a CA publishes listing revoked certificates. A CA may revoke a certificate for many reasons. For example, an entity may no longer work for the organization that requested the

certificate, or a certificate's private key may have been compromised. Generally, the entity that requests the certificate also instigates the revocation of the certificate. However, a CA can revoke a certificate if the entity uses the certificate in a way that violates the CA's policies.

Protocol Considerations

RSA Security is committed to providing maximum interoperability in its PKI software components. Toward this goal, the Cert-C toolkit incorporates several standards for certificate and CRL creation, extension, importing, and exporting.

What Are the X.509 Standards?

The International Telecommunications Union, ITU-T (formerly known as CCITT), is a multinational union that provides standards for telecommunication equipment and systems, including X.500 directory, X.509 certificates, and Domain Names (DNs).

DNs are the standard form of naming. A DN is composed of one or more relative distinguished names (RDNs), and each RDN is composed of one or more attribute-value assertions (AVAs). Each AVA consists of an attribute identifier and its corresponding value information—for example, `CountryName = US`.

DNs are intended to identify entities in the X.500 directory tree. An RDN is the path from one node to a subordinate node. The entire DN traverses a path from the root of the tree to an end node that represents a particular entity. A goal of the directory is to provide an infrastructure to uniquely name every communications entity everywhere (thus the distinguished in DN). Because of this, names in X.509 certificates are perhaps more complex than one might like (for example, compared to an e-mail address). Nevertheless, for business applications, DNs are worth the complexity, because they are closely coupled with legal name registration procedures. Simple names, such as e-mail addresses, do not offer this.

ITU-T Recommendation X.509 specifies the authentication service for X.500 directories, as well as the widely adopted X.509 certificate syntax. The initial version of X.509 was published in 1988, v 2 was published in 1993, and v 3 was proposed in 1994 and considered for approval in 1995. V 3 addresses some of the security concerns and limited flexibility that were issues in v 1 and v 2.

Directory authentication in X.509 can be carried out using either secret-key or public-key techniques. The latter is based on public-key certificates. The standard does not specify a particular cryptographic algorithm, although an informative annex of the standard describes the RSA algorithm.

An X.509 certificate consists of the following fields:

- Version
- Serial number
- Signature algorithm ID
- Issuer name
- Validity period
- Subject (user) name
- Subject public-key information
- Issuer unique identifier (v 2 and 3 only)
- Subject unique identifier (v 2 and 3 only)
- Extensions (v 3 only)
- Signature on the preceding fields

This certificate is signed by the issuer to authenticate the binding between the subject (user's) name and the subject's public key. The major difference between v 2 and v 3 is the addition of the extensions field. This field grants more flexibility because it can convey additional information beyond just the key and name binding. Standard extensions include, for example, subject and issuer attributes, certification policy information, and key usage restrictions.

X.509 also defines a syntax for CRLs. The X.509 standard is supported by a number of protocols, including PKCS and PKIX.

PKIX Profiles

The Internet Engineering Task Force, IETF, has a working group attempting to standardize several aspects of certificates, in accordance with the X.509 standard, from certificate profiling to alternative certificate revocation methods. The first of the PKIX standards, RFC 2459, profiles the X.509 v3 certificates and v2 CRLs for use on the Internet. This standard is now superseded by *RFC 3280*. The Cert-C API conforms to these PKIX standards. You can view the IETF RFCs at `http://www.ietf.org/`.

PKCS Messaging

RSA Security's Public-Key Cryptosystem provides cryptographic methods in its encryption technology. Along with X.509 certificates, the Public-Key Cryptography Standards (PKCS) provide the foundation of certificate security features, and outline a

set of standards for the secure and interoperable use of cryptography. These standards have been implemented in thousands of applications and are used by millions worldwide. RSA Security's Cert-C SDK utilizes the message formats specified and defined by PKCS #7 and PKCS #10, discussed in the next section.

Combined, these two standards provide the messaging and certification standards necessary for Cert-C and other certification applications.

PKCS #7, PKCS #8, PKCS #10, PKCS #11, and PKCS #12 are described in detail in the following sections, as some of the Cert-C APIs are directly devoted to implementing these standards. The Cert-C API includes support for those elements of PKCS #1, PKCS #5, and PKCS #9 that are referenced by these three main standards. For a complete description of the Public-Key Cryptography Standards, see the RSA Laboratories Web site (`http://www.rsasecurity.com/rsalabs/pkcs/`).

PKCS #7 and PKCS #10 Message Formats

PKCS #7 and PKCS #10 are particularly important to developers who are implementing certification.

The Cryptographic Message Syntax Standard, PKCS #7, specifies a syntax for data that may have cryptography applied to it, such as digital signatures and digital envelopes. The purpose of PKCS #7 is to provide a standard syntax and platform-independent digital representation for cryptographic data. Applications that conform to the standard can interoperate regardless of platform differences as the standard prescribes a way for them to read and compose messages. The Cert-C API supports five data message types: Data, Signed Data, Enveloped Data, Encrypted Data, and Digested Data.

The Certificate Request Syntax Standard, PKCS #10, specifies a syntax for composing certification requests for a CA. The CA transforms a request into an X.509 digital certificate. The certificate request usually includes the DN and the public key of the user, along with a set of attributes.

PKCS #8 Private-Key Syntax

The Private-key Information Syntax Standard, PKCS #8, describes a syntax for private-key information, including a private key for some public-key algorithm and a set of attributes. The standard also describes syntax for encrypted private keys. The intention of including a set of attributes is to provide a simple way for a user to establish trust in information such as a DN or a top-level CA's public key.

PKCS #11 Cryptographic Token Interface

The Cryptographic Token Interface Standard, PKCS #11, defines an API called Cryptoki to devices that hold cryptographic information and perform cryptographic functions. PKCS #11 addresses the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a cryptographic token.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable.

A number of cryptographic mechanisms (algorithms) are supported in the latest version. Cryptoki v2.1 is intended for cryptographic devices associated with a single user, so some features that might be included in a general-purpose interface are omitted. For example, Cryptoki v2.1 does not have a means of distinguishing multiple users. The focus is on a single user's keys and perhaps a small number of certificates related to them.

PKCS #12 Public/Private-Key Importing and Exporting

The Personal Information Exchange Syntax Standard, PKCS #12, specifies a portable format for storing or transporting a user's certificate and private-key information. This standard describes a transfer syntax for personal identity information, including private keys, certificates, miscellaneous secrets, and extensions. You can use the Cert-C function, `C_ImportPKCS12`, to import certificates contained in a PKCS #12 message from Netscape and Microsoft browsers, and `C_ExportPKCS12`, to export certificates, CRLs, and private keys into a PKCS #12 formatted file.

OCSP Certificate Status

The X.509 Internet Public-Key Infrastructure Online Certificate Status Protocol, OCSP, determines the current status of a digital certificate without requiring CRLs.

SCEP Certificate Request

The Cisco Simple Certificate Enrollment Protocol, SCEP, supports the secure issuance

of certificates to network devices in a scalable manner, using existing technology whenever possible. The protocol supports the following operations:

- CA and RA public-key distribution
- Certificate enrollment
- Certificate revocation
- Certificate query
- CRL query

Certificate and CRL access can be achieved by using the LDAP, or by using the query messages defined in SCEP.

CRS Certificate Request

The Internet PKI Certificate Request Syntax, CRS, specifies an interface to public-key certification products and services based on PKCS #7 and PKCS #10. A small number of additional services are defined to supplement the core certificate request service. Current industry practice regarding the use of PKCS #7 and PKCS #10 is also documented for the benefit of the Internet community. In general, the use of PKCS #7 in CRS is aligned to the Cryptographic Message Syntax (CMS), which provides a super-set of the PKCS #7 syntax.

CMP Certificate Management

The X.509 Public-Key Infrastructure Certificate Management Protocols, CMP, defines protocol messages for all relevant aspects of certificate creation and management. Management protocols are required to support online interactions between PKI components. For example, a management protocol might be used between a CA and a client system with which a key pair is associated, or between CAs that issue cross-certificates for each other.

Cert-C supports the following CMP protocol operations:

- Certificate request
- Key archival
- Key update
- Revocation request

ASN.1 BER and DER Encoding

Much of the data you deal with in cryptography is information passed between two or more individuals. Perhaps you need to send a CA a certificate request, or maybe a CA needs to send you a CRL. Not everyone uses the Cert-C SDK, and how information is represented in the Cert-C SDK may be different from another company's package. There needs to be a standard way to describe certain information. Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER) are two standards that can do this.

Open Systems Interconnection (OSI, described in ANSI X.200) is an internationally standardized architecture that governs the interconnection of computers from the physical layer up to the user application layer. OSI's method of specifying abstract objects is called Abstract Syntax Notation One (ASN.1 defined in X.680 and X.681), and one set of rules for representing such objects as strings of ones and zeros is called BER, defined in X.690. There is generally more than one way to BER-encode a given value, so another set of rules, called DER, which is a subset of BER, gives a unique encoding to each ASN.1 value. The RSA Laboratories Web site includes *A Layman's Guide to a Subset of ASN.1, BER, and DER*, which you will almost certainly find more readable than the actual standard. You can get a copy of this document at `http://www.rsasecurity.com/rsalabs/pkcs/`.

If your application must transfer information to another computer or software package, you may find it necessary to convert the data into a DER-encoded string before you send it. The BER or DER encoding of information is a simple concept; it is merely a way to parse information with standardized identifying marks. However, BER or DER encoding can be time consuming. The Cert-C SDK offers a way to encode information into DER format by using the `C_Get*DER` and `C_DEREncode*` routines. It is also possible to convert BER- or DER-encoded information into the Cert-C format with the `C_Set*BER` and `C_BERDecode*` routines. These routines perform general ASN.1 encoding and decoding. See the *API Reference* for additional information regarding the ASN.1 functions.

The Cert-C SDK generally encodes data using DER and decodes data using BER. (The only exception is when a standard specifies that DER must be used. For example, PKCS #10 states that the data signed in a certificate request must be DER-encoded, not BER-encoded.) The reason a Cert-C function gets DER and sets BER is that DER is a subset of BER. Therefore, a BER reader will be able to understand DER. Unlike BER, a DER encoding is unique. Thus, to avoid any possible confusion, the Cert-C output is DER-encoded data.

Note that BER encoding does not put data into an ASCII string; it is simply a standard way of describing certain abstract objects in binary form. Conversion into BER or DER

is known as BER encoding or DER encoding, while the conversion between binary to ASCII is referred to as Base64 encoding and decoding. This may get confusing, but the word encoding without a BER in front of it generally means binary-to-ASCII. If the encoding is BER encoding or DER encoding, the BER or DER should be explicitly stated.

Character Sets

When using the Cert-C SDK, you often enter information as a string. Cert-C accepts thirteen kinds of strings, listed in the following table:

String Name	Description
VT_BMP_STRING	BMPString is a subtype of UniversalString that has its own unique tag and models the Basic Multilingual Plane (the first 64K-2 cells) of ISO/IEC 10646-1.
VT_GENERAL_STRING	All G and all C sets + SPACE + DELETE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_GRAPHIC_STRING	All G sets + SPACE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_IA5_STRING	Tables 1, 6 + SPACE + DELETE from the ISO International Register of Coded Character Sets to be used with Escape Sequences. It is the entire ASCII character set. In hex, it is all the bytes from 0x20 through 0x7F.
VT_ISO646_STRING	Table 6 + SPACE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_NUMERIC_STRING	The characters 0, 1, through 9 + SPACE.
VT_PRINTABLE_STRING	The characters A, B through Z + a, b through z + 0, 1 through 9 + SPACE + APOSTROPHE + COMMA + (,), +, -, ., /, :, =, ?.
VT_TELETEX_STRING	Tables 6, 87, 102, 103, 106, 107, 126, 144, 150, 153, 156, 164, 165, 168 + SPACE + DELETE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_T61_STRING	Tables 6, 87, 102, 103, 106, 107, 126, 144, 150, 153, 156, 164, 165, 168 + SPACE + DELETE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_UNIVERSAL_STRING	The characters that can appear in the UniversalString type are any of the characters allowed by ISO/IEC 10646-1.
VT_UTF8_STRING	The content of this type conforms to RFC 2279.

String Name	Description
VT_VIDEOTEX_STRING	Tables 1, 13, 72, 73, 87, 89, 102, 108, 126, 128, 129, 144, 150, 153, 164, 165, 168 + SPACE + DELETE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.
VT_VISIBLE_STRING	Table 6 + SPACE from the ISO International Register of Coded Character Sets to be used with Escape Sequences.

Printable String

A printable string is made up of characters that can be printed on any standard printer or computer screen. For instance, the hex byte string `52 53 41 27 73 20 43 65 72 74 2D 43` translates to `RSA's Cert-C`.

`0x52` is the ASCII character `R`, `0x20` is the ASCII character <space>, and so on. However, the list of possible `VT_PRINTABLE_STRING` characters does not correspond exactly to the ASCII character set, or what you can type at the keyboard. `VT_PRINTABLE_STRING` characters consist of the following.

`A - Z`

`a - z`

`0 - 9`

<space> ' ( ) + , - . / : = ?

But suppose you need to use, for example, the hex values `0xCB` or `0x16`. They (along with many others) do not correspond to standard characters you generally find on a keyboard. These are not `VT_PRINTABLE_STRING` characters. If you want to use non-printing characters in your string, use the tag `VT_UTF8_STRING`.

Note: UTF-8 is an alternate way of encoding UNICODE characters using a variable number of bytes per character. It is designed so that `0x00-0x7f` corresponds to the standard ASCII characters. Also, `0x00-0x7f` does not occur inside any multi-byte character. This trait enables many routines that use `0x00` as an end-of-string indicator to work, as well as routines that scan occurrences of standard ASCII characters within a string. For European languages (including English), a typical UTF-8-encoded string is shorter than the standard UNICODE-encoded string, which requires a constant two bytes per character.

Chapter 3

Cert-C Setup

The directions for setting up Cert-C 2.7 are very similar to those of other RSA BSAFE SDK products.

Before you can use Cert-C with your application, you need to install Cert-C and Crypto-C in sibling directories. Then you need to perform some setup tasks, which depend on your development environment. It is possible that you might also need to install some third-party software. This chapter outlines those Cert-C setup requirements.

Cert-C CD-ROM Contents

The Cert-C CD-ROM contains (after decompression and installation) a complete set of the Cert-C 2.7 header and library files that you can compile and link with your products. It also contains source code for the Cert-C service providers so you can modify them to meet any special requirements you might have in your applications. Also included on the CD-ROM are sample programs and the Cert-C documentation set.

The Cert-C CD-ROM contains the following items:

- In the `CertC-27` directory, the `readme-certc.txt`, the `certc_27_releasenotes.pdf`, and the `certc_27_install.pdf` files.
- A `doc` directory that contains the `certc_27_releasenotes.pdf` file and the `thirdpartylicense.pdf` file. The third party licence file contains the complete texts of any license agreements applicable to third-party software in this product. This folder also contains the .pdf files that reflect the latest version of the manuals; the *Basic Developer's Guide* and the *Advanced Developer's Guide*, as well as the `Reference` folder. This folder contains the *API Reference* HTML files. You can access the *API Reference* by opening the `certc_reference.html` file in the `doc` directory.
- An `include` directory that contains all of the necessary header files used to develop Cert-C 2.7 applications.
- A `lib` directory that contains library files compiled with production-quality options for linking into your applications.
- A `provider` directory that contains all of the source code for the Cert-C service providers developed by RSA Security.
- A `rootdb` directory that contains a tool used to assist you in constructing a database of root certificates that can be used to validate certificates used by your applications.
- A `samples` directory that contains numerous samples of how to use various Cert-C features and interfaces.
- A complete copy of Crypto-C for use in developing Cert-C 2.7 applications.

Installing Cert-C

To use Cert-C 2.7, you must have Crypto-C 6.1 installed on your system. If you are upgrading from Cert-C 1.0x, it is not possible to use any versions of Crypto-C prior to 6.1. If you are already using Crypto-C 6.1, you do not need to re-install it. Otherwise, you need to install or upgrade to Crypto-C 6.1 first.

You do not need to uninstall or delete any previous versions of Crypto-C, Cert-C, or BCERT. However, you must install each new product into a separate directory from the old version. If you follow the guidelines given in the *Installation Guide*, old and new versions of Cert-C and Crypto-C will peacefully coexist.

Cert-C and Crypto-C must be installed in sibling directories. For detailed instructions about installing Cert-C, see the *Installation Guide*, `certc_27_install.pdf`, in the `Cert-C2.7` folder of the CD-ROM.

Compatibility with BCERT 1.0x

Backward compatibility with BCERT v1.0x and previous releases of Cert-C is a major feature of Cert-C 2.7. You should be able to recompile applications based upon these products using Cert-C 2.7 with little or no effort. For details on migrating from BCERT to Cert-C, see Appendix B, “BCERT Compatibility” on page 295.

Customizing the UNIX Install Location

RSA Security has centralized all of the information to customize into a single file, named `Makerules.certc_root`, located in the top-level directory. This file is referenced by all of the makefiles to automate building the software libraries, samples, and the `rootdb` utility.

Additionally, platform-specific compilation options are contained in file located in the `make` directory. For example, the file `make/Makerules.hp11.release` contains the build options for the `PLATFORM=hp11` and `BUILDTYPE=release` configuration.

RSA Security used the exact same build structure, contained in these makerules files, to build and test the software contained on the CD-ROM. Once you have a working copy of the Cert-C tree, you must specify the platform and build type that you want to compile. You must edit the `Makerules.certc_root` file and replace all the `REPLACE_ME`

strings to reflect the location and configuration of the software to be built. The following is a prototype version of this information that contains the string REPLACE_ME where your values must be set.

```
# This local copy directory, for example, /home/jeff/Cert-C2.7.
CERTC_ROOT=REPLACE_ME

# See the Basic Developer's Guide for supported platform names,
# for example, hp11.
PLATFORM=REPLACE_ME

# Usually one of "release" or "release_mt".
BUILDTYPE=REPLACE_ME
```

An explanation of these variables is provided in the following paragraphs.

CERTC_ROOT

This variable contains the directory in which your working, writable (not read-only) copy of Cert-C 2.7 resides. In all cases, it is necessary to replace the string REPLACE_ME with the location of this directory. For example, you may have placed your working copy of Cert-C into /home/crystal/CertC-2.7.

PLATFORM

This variable contains the platform string that denotes the platform on (and for) which the application is being developed. A table of these platform strings is provided in Table 3-1, "Compile-Time and Link-Time Strings for Building Applications," on page 53.

BUILDTYPE

This variable contains the string that tells the Makefiles which version of the library you are using to link. If you start developing your own service providers, you may choose to build a debuggable version of your service provider library and link with this library for debugging.

If you are using the libraries that RSA Security provides, and you will neither customize any of this code nor use a debugger to step through the service provider code, then this variable should remain set to release.

UNIX Platform-Specific Build Strings

Table 3-1 contains the PLATFORM strings used to define the correct compile-time and link-time settings for building applications.

Table 3-1 **Compile-Time and Link-Time Strings for Building Applications**

Software Platform	PLATFORM
HP-UX 10.20 (PA-RISC 1.x)	hp1020
HP-UX 11.00 (PA-RISC 2.0.1 32-bit)	hp11
HP-UX 11.00 (PA-RISC 2.0.1 64-bit)	hp64
Solaris 2.6	Solaris26
Solaris 8 and Solaris 9	Solaris28
Red Hat Linux 6.2	RedHat62
Red Hat Linux 7.1 and 8.0	RedHat71
IBM AIX 4.3.3 and AIX 5L 5.2	aix43

If your target platform is not exactly one of these listed here, you should choose the value that is equal to (or lesser than) the one that is appropriate for your platform. For example, if you want to target an application for Red Hat Linux 7.2, you should use the PLATFORM for Red Hat Linux 7.1.

Using the Crypto-C Libraries

Although Cert-C bundles the complete set of Crypto-C libraries, you are licensed to use Crypto-C only in conjunction with Cert-C. Please refer to your RSA Security license agreement for additional information. See the *Installation Guide* (`certc_27_install.pdf`), located in the `Cert-C2.7` folder, for details on how and where to install Crypto-C, and what version of Crypto-C you should use.

Third-Party Source Code

The Cert-C 2.7 software distribution media includes a complete set of the Cert-C sources and libraries that enable customers to customize or enhance their service-provider library, for linking with Cert-C applications. Cert-C also makes use of third-party software. The `thirdpartylicense.pdf` file, located in the `doc` directory, contains the complete texts of any license agreements applicable to third-party software in this product. The following sections list these third-party software distributions and details their redistribution limitations. You are also told where to get the third-party software source if it is not already included in the Cert-C CD-ROM. Installation specifications for the third-party software distributions are also discussed.

CodeBase

Because of source code redistribution restrictions, RSA Security cannot provide you with the source for the database engine used by the Cert-C Default Database service provider in the `provider/db/rsa` directory. Due to dependencies on some of these source files, it is also not possible to compile the sources for the Cert-C LDAP Database service provider (`provider/db/ldap`), although the sources, as well as precompiled objects, for this service provider, are included.

If you want to acquire the source code for the Cert-C Default Database service provider, contact Sequiter Software Inc. at 780-437-2410 or `www.sequiter.com/`.

The CodeBase version that RSA Security licenses and distributes with Cert-C 2.7 in object form is “CodeBase v6.4 for UNIX” and “CodeBase v6.4 for Win32 Platforms.” RSA Security provides both the ‘glue code’ to integrate CodeBase into the Cert-C provider architecture, as well as a Microsoft Studio project file used to build CodeBase.

The following modifications were made to the CodeBase source itself to integrate it with the Cert-C architecture:

All Platforms

- In the file `c4hook.c`, the function named `error4hook()` must be uncommented.
- In the file `d4all.h`, the #define statements for `S4OFF_REPORT` and `S4OFF_TRAN` must be uncommented.

Win32

- In the file `d4all.h`, the #define statements for `S4WIN32` must be uncommented.

UNIX

- In the file `d4all.h`, the #define statements for `S4UNIX` must be uncommented.

You may use the CodeBase software only with the Cert-C product. If you would like to use the CodeBase software for any other purpose, you must license it directly from Sequiter Software Inc.

Makefiles are included in this software distribution. These files enable you to rebuild or modify the contents of the Cert-C service-provider library (`libcertcsp.a`) without these components. Care must be taken to retain the pre-built object files for the `db/ldap` and `db/rsa` providers that were previously described. Retention of these files is necessary to build a complete service-provider library.

LDAP Usage

RSA Security includes the Mozilla organization's LDAP client libraries for the Win32, Solaris, Linux, HP-UX v10.20, and HP-UX v11.00 (32-bit) platforms (Netscape Directory SDK for C v3.0) on the Cert-C CD-ROM. These libraries have been precompiled and are included in the appropriate subdirectory of the `lib` directory for all platforms. Similarly, the include files that go with these libraries are located in the `include/moziLDAP` directory.

These platforms now make use of the software from Mozilla. The Mozilla code is governed by an open source license, which is included in the `thirdpartylicense.pdf` file located in the `doc` directory. You have the option to replace these header files and libraries with others that you choose. However, software substitutions in this area are not supported by RSA Security.

Building and Deploying Cert-C

To build and deploy Cert-C, you must perform these platform-specific instructions.

Win32

To build any of the samples or utilities that RSA Security ships, use the Microsoft Visual Studio workspace files (they have a `.dsw` extension). For the `rootdb` utility, only a project file is provided. However, Visual Studio creates a workspace file for it automatically.

The LDAP client library, `nsldap32v30.dll`, must be contained in the application's library search path. This is often accomplished by installing this DLL into `\<SYSTEM-ROOT>\system32`.

UNIX and GNU-Linux

To build any of the sample or utilities that RSA Security ships, use the `Makefiles` and `Makerules` files. Currently, you cannot build object files or executable files for multiple platforms or multiple configurations in the same source tree. If you compile more than one configuration, then you must have multiple copies of the source trees for these different configurations. Also, we do not recommend you change a build configuration in an already compiled source tree. As object files can reside in many locations while generating a complete build, you must take great care in changing a build configuration.

To choose a configuration to build, you must edit the `Makerules.certc_root` file prior to compilation, see "UNIX Platform-Specific Build Strings" on page 53.

Solaris

Always specify either `-lpthread` when using `cc` or `-lpthread` before `-lc` when using `ld` to link your application. Failure to do so will prevent the random number generator from seeding correctly; this jeopardizes the integrity of random number generation. The standard library's (`libc.a`) `pthread` creation routine seems to fail, whereas the routine in `libpthread.a` works successfully. When these calls fail, information-gathering related to `pthread` timing races is skipped. Because of this, the sources of system randomness are substantially decreased, and limit the randomness of subsequently generated random numbers. This problem occurs when `-lpthread` is

omitted from the cc or ld command line when you link your application. This problem is known to occur in Solaris v2.6, but may be present in other releases.

Sample Programs

The sample programs located in the samples subdirectory are command-line applications that demonstrate some of the aspects of building public-key applications using Cert-C 2.7. They use the Cert-C 2.7 library routines and are provided to Cert-C customers in source form. Modifying source files and building the sample programs is an excellent exercise to start developing with Cert-C.

Building Samples on Win32

The build directory contains the makefiles necessary to build the sample code. To compile the samples, open the samples/make/build/samples.dsw workspace file. Do not open the individual .dsp project files. Most of the samples depend on some common utility functions (found in samples/common) encapsulated in the utils.dsp project. Open samples.dsw to preserve these dependencies.

Note that there are utility files in the samples/common/include and samples/common/source directories that contain definitions to the RSA_* functions used by many of the samples. Therefore, Win32 customers should open the build/samples.dsw MSVC v6.0 workspace file to preserve the dependencies between projects.

The input directory contains input files used as a quick test on many of the sample programs.

Win32 customers can run these tests by using the command-line makefile in build/test.win32. Win32 customers should make sure that the MSVC v6.0 command-line utilities are available. This is usually done by running the vcvars32.bat batch file provided as part of the MSVC v6.0 distribution. (The default installation location of this file is C:\Program Files\Microsoft Visual Studio\VC98\Bin\vcvars32.bat.)

To automatically build and run the samples, Win32 users can go to the build directory and run 'nmake -f test.win32 test'.

Similarly, UNIX customers can build and run the samples by going to the build directory and running 'make -f test.unix test'.

Utility Routines

Cert-C includes several utility routines to help you create and test your applications. See the `samples/common/include` and `samples/common/source` directories for the header files and source files. These utilities encapsulate common functions for your programs and tests. For a list of these utility programs and how they work, see the "Utilities" chapter of the *Advanced Developer's Guide*.

Chapter 4

Getting Started

This chapter outlines Cert-C programmatic information that you should know before you start to develop an application that uses the Cert-C API. First, you learn about how Cert-C represents information using objects. Each Cert-C object is listed and described. Next, you find out how to call the Cert-C API. A model is presented to you to show how Cert-C produces and reads information. You also learn about the Cert-C programming standards, including memory management, the Cert-C context, and how to write cleanup code.

This information will help you understand the examples presented in the following chapters. These examples show you how to create the Cert-C objects, which represent information such as certificates, certificate requests, CRLs, attributes, and extensions. Each example is designed to familiarize you with a real-life programming task, within the certificate management software environment.

Cert-C Objects

This section discusses the use of objects in Cert-C. It lists all the various Cert-C objects and gives a high-level description of what each represents.

Cert-C uses objects to represent information that is to be manipulated by Cert-C; these objects are passed as arguments to Cert-C functions. The Cert-C objects are defined as pointers; they serve as abstractions for various collections of information. Although the details of an object are maintained internally by Cert-C, you can use Cert-C API functions to manipulate the information in the object. For example, without knowing how a certificate object, `CERT_OBJ`, is represented internally in the Cert-C library, you can use Cert-C API functions that operate on the `CERT_OBJ` object to set and get information about the certificate.

You first create an object by calling one of the related functions. Then you can set the object with the desired information, or get information about an object that has already been set by either you or Cert-C. Finally, when the object is no longer needed, you should destroy the object.

To get or set information about an object, you must use a `C_Get*` or `C_Set*` function. These functions enable access to an object's information. You cannot make any assumptions about the format of the data in a Cert-C object.

The following table lists each Cert-C object with a brief description. For detailed information about each object and their related `C_Create*`, `C_Destroy*`, `C_Set*`, and `C_Get*` functions, see the referenced chapter.

Cert-C has the following objects, listed here alphabetically.

Object Name	Description	See
`ATTRIBUTES_OBJ`	Represents extra information about the certificate subject in a certificate request. It is also used as a general mechanism for storing attribute types and values.	Chapter 7
`C_CMS_OBJ`	Represents a CMS message.	*`API Reference`*
`CERT_OBJ`	Represents certificate information.	Chapter 10

Object Name	Description	See
CRL_ENTRIES_OBJ	CRL_ENTRIES_OBJ is the part of the CRL_OBJ object that represents the serial numbers, revocation times, and X.509 v3 CRL Entry extensions for each revoked certificate.	Chapter 14
CRL_OBJ	Represents CRL information.	Chapter 14
EXTENSIONS_OBJ	Represents an X.509 v3 extension set that contains one or more extension entries.	Chapter 15
LIST_OBJ	Represents a collection of abstract data types, including types defined by Cert-C and types defined by your application. For example, LIST_OBJ can contain a list of certificate objects or ITEMs.	Chapter 6
NAME_OBJ	Represents the names of entities involved in a PKI.	Chapter 7
PKCS10_OBJ	Represents certification-request information.	Chapter 8
PKI_CERT_CONF_REQ_OBJ	Represents a confirmation to the CA/RA to accept or reject an issued certificate.	Chapter 9
PKI_CERT_CONF_RESP_OBJ	Represents a confirmation to the client to indicate acceptance of the certificate confirmation request. In the current specification and implementation, supported certificate confirmation response messages do not actually contain any information.	Chapter 9
PKI_CERT_REQ_OBJ	Represents an initialization request or certificate request to a CA/RA to request a certificate.	Chapter 9
PKI_CERT_RESP_OBJ	Represents the initialization response or certification response back to the client.	Chapter 9

Object Name	Description	See
PKI_CERT_TEMPLATE_OBJ	Represents the template that specifies the information that goes into a certificate in the certificate request process.	Chapter 9
PKI_ERROR_MESSAGE_OBJ	Represents PKI messaging error information.	Chapter 9
PKI_KEY_UPDATE_REQ_OBJ	Represents a key update request for a certificate to a CA/RA.	Chapter 9
PKI_KEY_UPDATE_RESP_OBJ	Represents the response back to the client after the client has sent a key update request.	Chapter 9
PKI_MSG_OBJ	Represents certification, key update, certificate revocation, and key archival requests and responses, and any other information that might pass between a certification-requesting application and a CA or RA.	Chapter 9
PKI_REVOKE_REQ_OBJ	Represents a revocation request to a CA/RA to revoke one or more certificates.	Chapter 9
PKI_REVOKE_RESP_OBJ	Represents a response back to the client when the client has sent a revoke certificate request.	Chapter 9
PKI_STATUS_INFO_OBJ	Represents encapsulated provider-specific status and failure information.	Chapter 9

In addition, Cert-C uses the Crypto-C's key object, B_KEY_OBJ. See "Using BSAFE Crypto-C" on page 287, for a quick-start guide to using Crypto-C; it discusses the B_KEY_OBJ.

Calling the Cert-C API

Cert-C API procedures generally follow a predictable sequence:

1. Include the necessary Cert-C header files.
2. Set up a Cert-C context. In this step, you register the desired service providers by calling `C_InitializeCertC`.
3. Perform the desired operations. For example, build an attributes object, sign and verify a certificate, or retrieve a certificate from your database.

   Note: The operations that your application performs depends on the purpose of the application.

   In addition, each object generally goes through the following steps:

   a. Create the object: Instruct Cert-C to allocate space for the object.
   b. Set or Get information: Fill the object with information. This is usually done by loading BER-encoded data or by setting fields individually. Similarly, an application can retrieve the DER-encoded data in the object or retrieve values of individual fields.
   c. Destroy memory: Instruct Cert-C to clear sensitive data and free allocated memory.
4. Clean up: Clear sensitive data, destroy unneeded Cert-C objects, and free allocated memory.

Cert-C Model

In Cert-C, you either produce information or you read information. This is done through objects. The following two sets of steps are intended to give you a rough outline for producing and reading information.

Producing Information

When producing information, you can follow these five general steps. However, there will be situations where these five steps do not fit your programming requirements.

1. Create an object.
2. Enter the information (`C_Add*`, `C_Set*`, or `C_Set*Fields`).
3. Perform the operation (`C_Sign*`).
4. Retrieve the information in DER format (`C_GetDER*`).
5. Destroy the object.

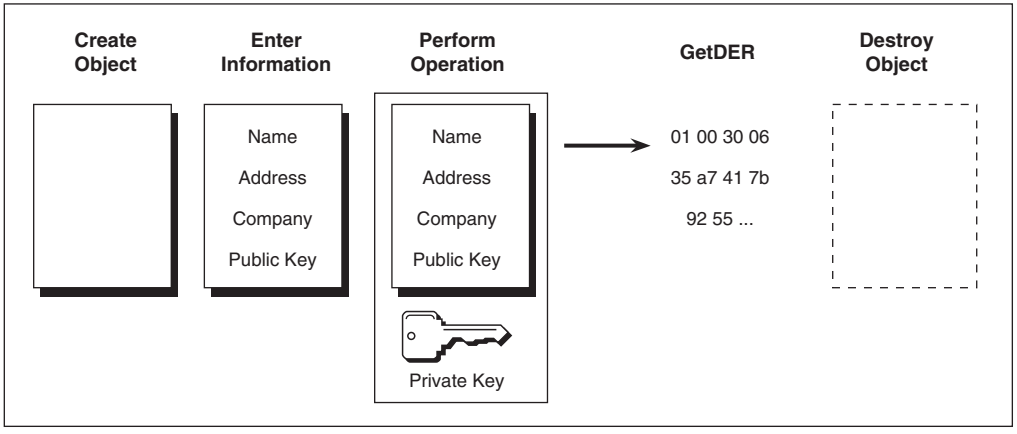


Figure 4-1 **Process to Produce Information**

Reading Information

When reading information, you can follow these five general steps. However, there will be situations where these five steps do not fit your programming requirements.

1. Create an object.

2. Set the object with the information in BER format (`C_Set*BER`).
3. Read the information (`C_Get*`, `C_Get*String`, `C_Get*Fields`).
4. Perform the operation (`C_Verify*`).
5. Destroy the object.

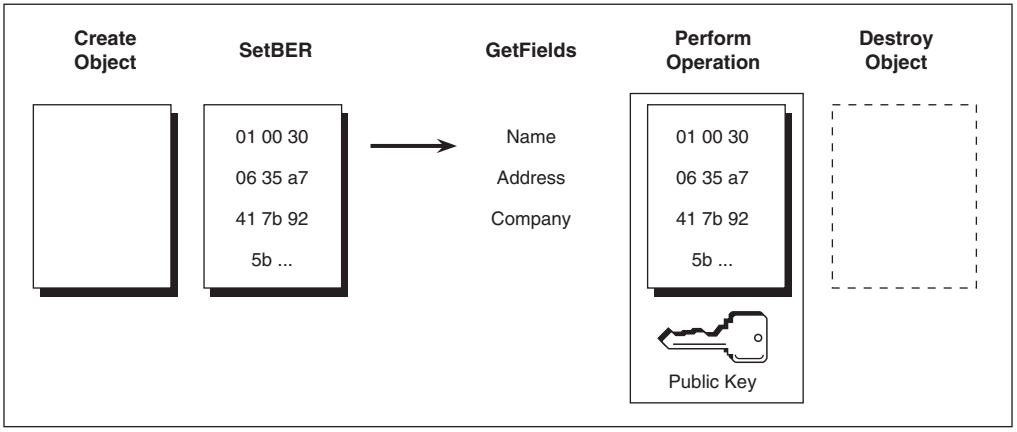


Figure 4-2 **Process to Read Information**

There will be variations on these step-by-step procedures. However, for the most part Cert-C follows one or the other of the five-step processes.

Cert-C Programming Standards

This section presents some programming standards that you should adhere to while writing code using the Cert-C API.

Memory Management

Certain Cert-C functions own and manage particular pieces of memory. You should never clear or tamper with this memory. The memory that you allocate, you must clear. Similarly, the memory that Cert-C allocates, it will clear.

For example, the `C_Get*BER` functions always return memory owned by Cert-C. You must not clear this memory. The `C_Set*BER` function always makes a copy of the BER (if it keeps one), so it may be disposed of at any time by your application.

Cert-C Context

When you call a `C_Create*Object` function to create an object, you pass the function a Cert-C context, `CERTC_CTX`, as the *ctx* parameter. A reference to the `CERTC_CTX` is stored internally in the newly created object, rather than an actual copy of the context. You create an object with a `CERTC_CTX` so you can use the object later to perform subsequent `C_*` calls that require a `CERTC_CTX`; for example, to use a cryptographic service provider to do signing operations. As the context is already contained in the object, you do not need to pass a context to each function that operations on the object. Because a Cert-C context is referenced by an object, you must not prematurely destroy a Cert-C context that is still referenced by a live object.

The attributes, name, and list objects are exceptions to this rule. None of these objects require a `CERTC_CTX`.

Clean Up

Always initiate objects to `NULL_PTR`. This ensures that any clean-up code you write can perform safely and predictably.

Call the `T_memset` function on `*_FIELDS` structures to clear them before you populate their subfields. This ensures forward compatibility. If a new parameter is added to an existing structure, and you call `T_memset`, then you do not need to change your application to set the new parameter to 0 (zero). The `T_memset` statement clears all fields in these structures.

Header Files

To use Cert-C in your application, you must include the `certc.h` header file. The only other header files that you need to include are service-provider header files. Which service-provider header file you need to include depends on which operations you intend to perform in your application. For example, if you intend to do a cryptographic operation, then you need to include the `certcsp.h` header file.

```
#include “certc.h”
#include “certcsp.h”
```

Sample Code Conventions

You will probably make extensive use of the samples included on the Cert-C CD-ROM, when you are writing your application. The following information will help you understand these samples better.

Routine Names

- Routines that start with `C_*` are core Cert-C functions.
- Routines that start with `S_*` are service-provider-specific functions.
- Routines that start with `RSA_*` are utility functions defined in the demo code.

Functions defined in the sample code that you may be particularly interested in begin with `RSA_`. The purpose of the `RSA_` prefix is to highlight those functions and to avoid conflicting function names if you decide to cut and paste this code into your application.

Return Values

All `RSA_` functions, according to standard RSA Security coding practices, return an integer called `status`. If the integer returned is 0 (zero), the function has completed successfully. A non-zero error code indicates an error or abnormal condition.

Cleanup Code

It is a good idea to initialize any object to `NULL_PTR`. If there is an error before an object has the chance to be created, the cleanup code acts on a `NULL_PTR` and does not do any damage.

Cert-C sample coding practices use cleanup functionality to make it easy to break out of a sequence when encountering an error. If you encounter an error, use a `goto` statement to proceed to the necessary cleanup code before you exit. Further code that depends on the offending function call does not execute. However, the code in that cleanup section, such as code that overwrites sensitive memory with zeroes, always executes, whether or not there was an error. This ensures that the routine does not return incomplete results.

Crypto-C API

You can call the Crypto-C API directly. See the *Crypto-C Developer's Guide* for more information about calling the Crypto-C API.

Appendix A, "Using BSAFE Crypto-C" on page 287, presents a quick-start guide to using Crypto-C.

Routine Names

- Routines that start with `B_*` are core Crypto-C functions.
- Structures that start with `B_*` are Crypto-C structures, for example, `B_PKCS11_SESSION`.
- Objects that start with `B_*` are Crypto-C objects; for example, `B_KEY_OBJ`.

Deprecated Functions and Structures

The following is a list of functions that were deprecated in the Cert-C 2.5 release.

Table 4-1 **Functions Deprecated in Cert-C 2.5**

Deprecated Function	Use New Function
C_GeneratePKIProofOfPossession	C_GeneratePKIMsgProofOfPossession
C_GetPKICertRequestFields	For a list of the new C_Get* functions that update the specific fields of a PKI_CERT_REQ_OBJ, see "Get PKI_CERT_REQ_OBJ Functions" on page 150.
C_GetPKICertResponseFields	For a list of the new C_Get* functions that update the specific fields of a PKI_CERT_RESP_OBJ, see "Get PKI_CERT_RESP_OBJ Functions" on page 151.
C_GetPKIMsgFields	For a list of the new C_Get* functions that update specific fields of a PKI_MSG_OBJ, see "Get, Set, or Modify PKI_MSG_OBJ Functions" on page 133.
C_ReadPKICertResponseMsg	C_SetPKIMsgBER
C_RequestPKICert	C_RequestPKIMsg
C_SendPKIMsg	C_SendPKIRequest
C_SetPKICertResponseFields	For a list of the new C_Set* functions that update the specific fields of a PKI_CERT_RESP_OBJ, see "Set or Modify PKI_CERT_RESP_OBJ Functions" on page 151.
C_SetPKICertRequestFields	For a list of the new C_Set* functions that update the specific fields of a PKI_CERT_REQ_OBJ, see "Set or Modify PKI_CERT_REQ_OBJ Functions" on page 149.
C_SetPKIMsgFields	For a list of the new C_Set* functions that update specific fields of a PKI_MSG_OBJ, see "Get, Set, or Modify PKI_MSG_OBJ Functions" on page 133.
C_ValidatePKIProofOfPossession	C_ValidatePKIMsgProofOfPossession
C_WritePKICertRequestMsg	C_GetPKIMsgDER

The following is a list of structures that were deprecated in the Cert-C 2.5 release. The information that was in these deprecated structures is now represented by new PKI

objects.

Table 4-2 **Structures Deprecated in Cert-C 2.5**

Deprecated Structure	Use New Object or Structure
PKI_MSG_FIELDS	PKI_MSG_OBJ
PKI_CERTREQ_FIELDS	PKI_CERT_REQ_OBJ
PKI_CERTRESP_FIELDS	PKI_CERT_RESP_OBJ
PKI_RECIPIENT	PKI_RECIPIENT_INFO

Chapter 5

Cert-C Context and Services

This chapter discusses how to initialize the Cert-C context and discusses other core functionality that hold state variables and track registered service providers. It also discusses functionality to initialize, register, and bind services. Then it discusses the actual services.

Cert-C Handles

Cert-C provides the following handles to hold state variables and track service providers.

- CERTC_CTX
- SERVICE_HANDLER
- SERVICE
- DB_ITERATOR
- STREAM
- EXTENSION_HANDLER
- LIST_OBJ_ENTRY_HANDLER

Using the CERTC_CTX and SERVICE_HANDLER Handles

Cert-C is designed with a context management component to assist applications in specifying and managing the numerous parameters and service providers.

It collects a number of common parameters and state variables together. It manages the Cert-C and service provider initialize and finalize functions. It also tracks the currently registered service providers, manages service-provider register and unregister functions, ordering and grouping of service providers, and binding and unbinding service providers to a SERVICE handle.

The Cert-C context is established when your application calls the C_InitializeCertC function.

```
int C_InitializeCertC (
  SERVICE_HANDLER  *handlers,              /* table of service providers */
  POINTER          *handlerParams,        /* table of handler parameters */
  unsigned int     handlerCount,              /* # of entries in tables */
  CERTC_CTX        *ctx                   /* (out) Cert-C context handle */
);
```

This function allocates the application's CERTC_CTX and initializes the internal fields of the context. It also initializes any service providers passed by the *handlers* parameter, defined as a SERVICE_HANDLER data structure. This data structure provides the Cert-C API with the service-provider information.

At initialization time, your application chooses a service-provider type and a specific

service provider within that type: either one of Cert-C's service providers or a third-party's service provider. Your application registers or dynamically binds the service provider. The service provider's specifications customize the API function calls that interface to your application.

In the case where your application uses a service provider, your application may need to know information about the selected service provider. Also, not all service providers are linked to an API function call.

Note: When performing operations that require a Cert-C context, make sure you use the correct context. For more information about how to use a Cert-C context, see "Cert-C Context" on page 66.

The `C_FinalizeCertC` function unregisters all currently registered service providers and frees all memory associated with the context, and sets the `CERTC_CTX` context handle to `NULL_PTR`.

Use the `C_RegisterService` function to register additional service providers subsequent to Cert-C initialization. This function calls the service provider's initialization function and adds an entry for the service provider in the context's internal list of service providers. The *`order`* input field specifies whether the service provider should be placed before or after other service providers of the same type in the context's internal service table.

Use the `C_UnregisterService` function to unregister a previously registered service provider. The `SERVICE_HANDLER` with the specified type and name is removed from the `CERTC_CTX`, the service provider's finalize function is called, and the memory associated with the context's copy of the service handler is freed.

Cert-C automatically unregisters all currently registered service providers when the Cert-C library is shut down, so the application does not need to call `C_UnregisterService` if the next Cert-C call is `C_FinalizeCertC`.

You must be careful to ensure that the service provider being unregistered is not bound to any `SERVICE` handles. Using a `SERVICE` handle that includes an unregistered service may cause the application to crash.

Initializing the Cert-C Context

In this example, you initialize a Cert-C context and register a service provider at initialization time. It demonstrates the `C_InitializeCertC`, `CERTC_CTX`, and `C_FinalizeCertC` functionality.

You need to link in the `certc` library and possibly other libraries. Make sure the compiler can locate header files. In MSVC v6.0, add the include files by selecting

Project, Settings, C/C++, Preprocessor, and then Additional include directories.

Step 1: Include header files

You must include the necessary header files.

```
#include "certc.h"

/* Include other service-provider-specific header files */
```

Step 2: Set up a Cert-C context

In this step, you create a Cert-C context, CERTC_CTX, and set up the service provider information for each group of service providers you want to use. To do so, you must fill a structure with the service provider's parameters and pass this structure on to an initialization routine.

```
#include “myspheader.h”
#define PROVIDER_COUNT 1

int status = 0;

CERTC_CTX ctx = (CERTC_CTX)NULL_PTR;
SERVICE_HANDLER spTable[PROVIDER_COUNT];
POINTER spParams[PROVIDER_COUNT];
spParams[0] = NULL_PTR;
```

Prepare the service providers that you intend to use.

```
spTable[0].type = SPT_TYPE;                    /* Service Provider Type */
spTable[0].name = "SP Name";            /* Unique null-terminated string */
spTable[0].Initialize = S_InitializeMySP;          /* Initalize function */
```

Step 3: Initialize Cert-C

To initialize Cert-C, call C_InitializeCertC at application startup time. This function registers the service providers and initializes Cert-C internal variables. The second parameter, *spParams*, is an array of pointers to arguments that Cert-C passes to the

corresponding S_Initialize* routine, in the SERVICE_HANDLER array.

```
status = C_InitializeCertC (spTable, spParams, PROVIDER_COUNT, &ctx);
```

Step 4: Perform operations

You can now perform Cert-C functions, such as adding or retrieving a certificate to your database or generating an RSA key pair.

Step 5: Clean up

When you finish performing an operation, you must call C_FinalizeCertC. This function clears sensitive data and frees allocated memory, then finalizes the context.

With objects, what you create, you must destroy. With contexts, what you initialize, you must finalize.

```
C_FinalizeCertC (&ctx);
```

Registering a Service Provider After Cert-C Initialization

This example demonstrates how to register an additional service provider, subsequent to the Cert-C context initialization, by calling the C_RegisterService function. For another example of how to register an additional service provider, see the samples/pkcs7/pkcs11msg.c sample program.

Step 1: Set up service provider

First, you include the service provider's header file. Then you use the SERVICE_HANDLER structure to set up the service-provider initialization information. For more information about the SERVICE_HANDLER structure, see the *API Reference*.

```
typedef struct {
  int   type;                                  /* type of service provider */
  char  *name;                                    /* service provider name */
  int   (*Initialize) (
    CERTC_CTX      ctx,                                    /* Cert-C context */
    POINTER        params,                /* provider-specific parameters */
    SERVICE_FUNCS  *funcs,                    /* (out) provider functions */
    POINTER        *handle                    /* (out) provider handle */
  );
} SERVICE_HANDLER;
```

Finally, you create a pointer to the service provider's initialization parameters. You are registering a new service provider, subsequent to the Cert-C context initialization, so you already have an initialized CERTC_CTX.

```
#include "myspheader.h"

#define SP_NAME “My service provider”
/* Assume a context has already been initialized */

SERVICE_HANDLER serviceHandle =  {SPT_TYPE, SP_NAME, S_InitializeMySP};

POINTER mySPParams = NULL_PTR;
```

Step 2: Register the service provider

Call the C_RegisterService function to register the service provider. This function calls the service provider's initialization function and adds an entry for the service provider in the context's internal list of service providers. For more information about the C_RegisterService function, see the *API Reference*.

```
int C_RegisterService (
  CERTC_CTX        ctx,                          /* Cert-C context handle */
  SERVICE_HANDLER  *handler,               /* service handler to register */
  POINTER          params,           /* service initialization parameters */
  int              order                                 /* first or last */
);
```

You already have an initialized CERTC_CTX. The second parameter, *serviceHandle*, passes an address for the SERVICE_HANDLER that contains the specified service provider's initialization information. The third parameter, *spParams*, is a pointer to an argument that Cert-C passes to the corresponding S_Initialize* routine, in the SERVICE_HANDLER handle. In this example, the service provider is placed after other service providers of the same type in the context's internal service table. To place it first, use SERVICE_ORDER_FIRST.

```
status = C_RegisterService (ctx, &serviceHandle,
                            (POINTER)mySPParams,
                            SERVICE_ORDER_LAST);
if (status != 0)
  goto CLEANUP;
```

Unregistering a Service Provider

This example demonstrates how to unregister a previously registered service provider, by calling the C_UnregisterService function. For more information about the C_UnregisterService function, see the *API Reference*.

```
void C_UnregisterService (
  CERTC_CTX  ctx,                              /* (mod) Cert-C context */
  int        type,                                    /* service type */
  char       *name                              /* service instance name(s) */
);
```

The SERVICE_HANDLER with the specified type and name is removed from the context, the service provider's finalize function is called, and the memory associated with the context's copy of the service handler is freed.

The application must make sure that the service provider being unregistered is not bound to any SERVICE handles. Using a SERVICE handle that includes an unregistered service provider may cause the application to crash.

Note: Cert-C automatically unregisters all currently registered service providers when the Cert-C library is shut down, so the application does not need to call C_UnregisterService if the next Cert-C call is C_FinalizeCertC.

```
C_UnregisterService (ctx, SPT_TYPE, SP_NAME);
```

Using the SERVICE Handle

Use SERVICE as an input parameter for some Cert-C functions. Cert-C functions that target a specific service provider or set of service providers have a SERVICE handle as a parameter. The SERVICE handle is designed to be used for a limited time; that is, to supply a tempory handle to associate selected service providers.

You can bind the SERVICE handle to a single service-provider instance, or you can bind it to a sequence of service-provider instances, all of the same type. All the service providers to be bound to the SERVICE handle must be currently registered with the given Cert-C context. Use the C_BindService and C_BindServices functions to create a SERVICE handle. Once you create a SERVICE handle, you cannot add another service provider to that handle. Instead, you must create a new SERVICE handle and associate all the required service providers with the new SERVICE handle.

Binding a Service

Use the C_BindService function to create a SERVICE handle and bind a single service provider, which is currently registered with the given Cert-C context, to that handle.

```
int C_BindService (
  CERTC_CTX  ctx,                                    /* Cert-C context */
  int        type,                                     /* service type */
  char       *name,                                /* service instance name */
  SERVICE    *service                              /* (out) service pointer */
);
```

For example, to bind the Cert-C In-Memory Database service provider, you must create a SERVICE handle for use with the database API function calls.

```
SERVICE handle = NULL;

status = C_BindService (ctx, SPT_TYPE, SP_NAME, &handle);
if (status != 0)
  goto CLEANUP;
```

Binding More Than One Service

To bind one or more currently registered service providers, of a single type, you create a SERVICE handle, using the C_BindServices function. For more information about the C_BindServices function, see the *API Reference*.

```
int C_BindServices (
  CERTC_CTX     ctx,                                    /* Cert-C context */
  int           type,                                     /* service type */
  char          **names,                             /* service instance names */
  unsigned int  nameCount,              /* # of provider names being bound */
  SERVICE       *service                              /* (out) service pointer */
);
```

Some service-provider types (for example, SPT_DATABASE and SPT_DATABASE2) allow an ordered list of instances to be specified in the service name array. If a NULL_PTR is specified for *names*, all of the service-provider instances of the given type are bound in

registration order.

```
SERVICE       handle = NULL_PTR;
/* Assume the service providers of type SP_TYPE are already */
/* registered with the CERTC_CTX used here. */

status = C_BindServices (ctx, SPT_TYPE, NULL_PTR, 0,
                         &handle);
if (status != 0)
  goto CLEANUP;
```

Unbinding a Service

To unbind a service provider, call the `C_UnbindService` function. This function undoes a previous binding of a service provider to the specified `SERVICE` handle. For more information about the `C_UnbindService` function, see the *API Reference*.

```
int C_UnbindService (
  SERVICE  *service                              /* (mod) service pointer */
);
```

If more than one service provider is bound to a `SERVICE` handle, calling `C_UnbindService` unbinds all of the service providers associated with the specified handle. The function frees any memory allocated by the corresponding `C_BindService` or `C_BindServices` calls.

```
C_UnbindService (&handle);
```

Using the Database Iterator Handle

Use the database iterator handle, `DB_ITERATOR`, to sequentially retrieve records of a particular type from a database or a set of databases. Each of the `C_SelectFirst*` functions initializes the `DB_ITERATOR` handle. Call the `C_FreeIterator` function to set `DB_ITERATOR` to `NULL_PTR`. The `C_SelectFirst*` and `C_SelectNext*` function calls also free `DB_ITERATOR` when they encounter an error or when all of the records of the requested type are retrieved.

For an example of how `DB_ITERATOR` is used, see "Retrieving a Certificate, CRL, or Private Key" on page 212.

Using the STREAM Handle

Use the STREAM handle to represent an open input or output stream when calling Cert-C file stream functions, such as C_ReadStream. For an example of how a STREAM handle is used, see the samples/io/usememio.c sample.

Using the Extension Handler

Use the EXTENSION_HANDLER extensions handler to hold pointers to callback functions for a particular extension type. Cert-C provides a default extension handler for each Cert-C-defined extension type; however, if you override a default extension handler or if you define a new extension type, you must provide the callback functions.

For more information about the EXTENSION_HANDLER, see the *API Reference*. For an example of how it is used, see "User-Defined Extensions" on page 270. For an example of how to register an EXTENSION_HANDLER with a CERTC_CTX, see "Registering a User-Defined Extension" on page 279.

Using the List Object Entry Handler

Use the LIST_OBJ_ENTRY_HANDLER to store any kind of application-defined data in a LIST_OBJ, even though Cert-C has no knowledge of the type of data structure the application requires. The application must set up the AllocAndCopy and Destructor callback functions to handle the type of data structure that it is using. These callback functions must recognize the type of data structure in *value* without being told by the Cert-C function that passes the *value* to the callback. An application can use the AllocAndCopy feature to insert application-defined values into a list object.

The C_AddListObjectEntry and C_InsertListObjectEntry functions use the LIST_OBJ_ENTRY_HANDLER structure as a value for their *handler* input fields.

For more information about the LIST_OBJ_ENTRY_HANDLER, see the *API Reference*. For an example of how to use the LIST_OBJ_ENTRY_HANDLER, see "Creating and Enumerating a List of User-Defined Elements" on page 97.

Other Usage

The PKCS12BagEntryHandler handler is a LIST_OBJ_ENTRY_HANDLER. You can use it to create a LIST_OBJ that contains a list of PKCS12_BAG structures. Use these structures with the C_ReadFromPKCS12 and C_WritetoPKCS12 functions.

The PKI_SP_DATA_HANDLER handler is a LIST_OBJ_ENTRY_HANDLER. Cert-C uses it to allocate and copy service-provider-specific data, and associate it with a PKI message object. Use these structures with the C_SetPKIProviderData function.

Cert-C Services

The Cert-C SDK provides PKI services to your application through its API; this API layer is the primary part of the Cert-C architecture with which your application needs to interface. This API can be categorized into two types of APIs. The first type of API gives your application an interface to the internal Cert-C library, where standard PKI functionality is provided. The second type of API provides additional PKI functionality, interfacing with service providers. This last type of API enables your application to select a Cert-C service provider, a third-party service provider, or a service provider created by you, which provides greater flexibility.

When your application uses a service provider, your application may need to know information about the selected service provider. Also, not all service providers are linked to an API function call. Cert-C is designed with a context management component to assist applications in specifying and managing the numerous parameters and service providers.

Surrender Context

The surrender context, `A_SURRENDER_CTX`, provides a way for you to halt a Cert-C function or to obtain cycles to perform other tasks. It contains a pointer to an application-specific callback function (*`Surrender`*) that Cert-C can use as its surrender function, and a pointer to application-specific information (*`handle`*).

To supply a *`Surrender`* function, you must register a surrender service provider when you initialize the Cert-C context. If you use the Cert-C Text Surrender service provider, it supplies a default *`Surrender`* function. For information about using the Cert-C Text Surrender service provider, see the "Service Provider" section of the *API Reference*.

Registering a Surrender Context

Before calling any other Cert-C function, a typical application initializes the `A_SURRENDER_CTX` value. Each `A_SURRENDER_CTX` value can specify a different *`Surrender`* callback function and a different *`handle`*.

To substitute an application-defined *`Surrender`* function for the default *`Surrender`* function, you can call the `C_GetSurrenderCtx` function, which returns a pointer to the `A_SURRENDER_CTX` structure. (If a text-surrender service provider is not registered, the

C_GetSurrenderCtx function returns a NULL_PTR.)

```
#include "textsurr.h"

A_SURRENDER_CTX *surrCtx = (A_SURRENDER_CTX *)NULL_PTR;

surrCtx = C_GetSurrenderCtx (ctx);
```

Cert-C Service Providers

Cert-C provides the following types of services:

- System
- Text Surrender
- Status Log
- Stream
- Database
- Cryptographic
- Certificate Path Processing
- Certificate Revocation Status
- PKI Certificate Management

For more information about the specifics of each Cert-C service provider, see the "Service Provider" section of the *API Reference*.

System

Use the Cert-C System service provider to manage memory, operate on memory blocks and strings, and obtain the time with platform-specific library functions that are modeled after conventional C library functions. These functions are called directly by Cert-C. The Cert-C System service provider implements an interface between standard system calls required by the Cert-C library and platform-specific system calls. There is only a single service provider for this service, and you do not need to register it.

Text Surrender

Use the Cert-C Text Surrender service provider to surrender control of a Cert-C function to your application. Only one surrender service provider can be registered at

a given time. The Cert-C Text Surrender service provider is the Cert-C implementation of the surrender service provider; it is suitable for use in console, command-line, or other text-mode applications.

Status Log

Use the Cert-C Status Log service provider to append an application's error, warning, or information status message to its status-log file. The Cert-C Status Log service provider is the Cert-C implementation of the status log service provider. If more than one status log service provider is registered, the application's status message is appended to all of the status log service provider's log files.

Stream

Use the Cert-C Stream service provider to implement a stream to read from and write to a file using standard C file input and output functions. The Cert-C Stream service provider is the Cert-C implementation of the stream service provider. It supports a subset of the possible Cert-C supported `IO_*` flag combinations. For more information on the `IO_*` flags, see the `C_OpenStream` function in the *API Reference*.

Database

Use the Cert-C database service providers to store certificates, CRLs, and keys. Cert-C includes the following six Cert-C database service providers:

- Cert-C Default Database service provider

  The Cert-C Default Database service provider provides a persistent local database. Database entries are stored as records in files in the local file system. The database is implemented using an embedded, high-performance database engine suitable for managing small to medium numbers of entries—for example, up to tens of thousands of entries.

- Cert-C In-Memory Database service provider

  The Cert-C In-Memory Database service provider stores entries in list objects that are in memory. The application can supply the lists used by the database, or the database can create temporary lists that are automatically destroyed when the service provider is unregistered. Databases of this type can be useful in caching or in processing lists of certificates, CRLs, private-keys, or PKCS #10 objects returned by other Cert-C functions.

- Cert-C LDAP Database service provider

  The Cert-C LDAP Database service provider retrieves certificates and CRLs from an LDAP repository. You can make an LDAP repository available as a database service provider. Registered database service providers are searched in the order established during the registration of the database service provider.

- Cert-C CryptoAPI Database service provider

  The Cert-C CryptoAPI Database service provider translates Cert-C database function calls into CryptoAPI function calls. This enables the sharing of keys and certificates among applications written to the Cert-C API and applications written to CryptoAPI. This service provider relies upon Microsoft Windows APIs, so it is available only on Microsoft Windows platforms.

- Cert-C SCEP Database service provider

  The Cert-C SCEP Database service provider supports the retrieval of CA and RA certificates, and possibly certificate chains leading to them, from network devices such as routers. RSA Security developed this service provider using the interfaces specified in the Cisco System's *Simple Certificate Enrollment Protocol* specification. This service provider is suitable for network devices that may need to retrieve trusted-root certificates for use with an SCEP PKI service provider when an LDAP server is not available. The Cert-C SCEP Database service provider does not support any SCEP functionality other than CA and RA certificate retrieval. However, see the Cert-C SCEP PKI service provider for other supported SCEP certificate management functionality.

- Cert-C PKCS #11 Database service provider

  The Cert-C PKCS #11 Database service provider implements the database interface to the object handling services of a PKCS #11 v2.x library and token—supporting authenticated read-write access to certificates and private keys. Additional PKCS #11 functionality can be provided through other service providers—for example, cryptographic services can be provided through a cryptographic service provider.

Cryptographic

Use the Cert-C Default Cryptographic service provider to access the RSA BSAFE Crypto-C APIs to provide your application with the necessary functionality to support client cryptographic function calls. You can register or initialize only one cryptographic service provider at a given time.

This service provider supports the Intel Hardware Random Number Generator, the Microsoft CryptoAPI services, and now, direct access to third party PKCS #11

libraries and tokens (and the use of RSA and DSA private keys on those tokens). See the `certc_27_releasenotes.pdf` file for a list of the devices that Cert-C supports. For a list of the cryptograpic operations that Cert-C supports, see the "Service Provider" section of the *API Reference*. Depending on the type of cryptographic service you require, you must initialize the Cert-C Default Cryptographic service provider with a particular initialization function. For more information on the two Cert-C Default Cryptographic service provider initialization functions, see the "Service Provider" section of the *API Reference*.

Certificate Path Processing

Use the Cert-C Certificate Path Processing service provider to implement certificate path processing. This service provider provides certificate path processing according to the profiles outlined in X.509 v1, *RFC 2459*, and *RFC 3280*.

Certificate Revocation Status

Use the Cert-C Certificate Revocation Status service providers to obtain status on a certificate. Cert-C includes the following two Cert-C certificate revocation status service providers:

- Cert-C CRL Revocation Status service provider

  Use the Cert-C CRL Revocation Status service provider to check the validity of a certificate against a set of CRLs.

- Cert-C OCSP Revocation Status service provider

  Use the Cert-C OCSP Revocation Status service provider to check the validity of a certificate without requiring CRLs. This service provider uses the OCSP, and is suitable for client applications that require a method for getting more timely certificate revocation status information than a CRL can usually provide.

PKI Certificate Management

Use the Cert-C PKI service providers to request a certificate, to request a key update, to request a certificate revocation, and to archive a key. Cert-C includes the following four Cert-C PKI service providers:

- Cert-C CRS PKI service provider

  Use the Cert-C CRS PKI service provider to send certificate requests and retrieve certificate responses using a CA that implements the CRS protocol. This service provider implements CRS certificate request and requested certificate pickup functionality according to the *VeriSign CRS Profile Specificatio*n, which is available

directly from VeriSign. Currently, only certificate requests are implemented by this service provider. Other CRS request types, such as certificate revocation and certificate lookup, are not supported.

- Cert-C SCEP PKI service provider

  Use the Cert-C SCEP PKI service provider to send certificate requests and retrieve certificate responses using a CA that implements Cisco Systems' *Simple Certificate Enrollment Protocol* certificate request mechanism. SCEP's primary use is for requesting and retrieving IPSec certificates. This service provider supports only certificate requests. Currently, no other SCEP request types are supported, such as certificate or CRL lookup. These functions may be achieved through the use of the Cert-C Database LDAP service provider. Client bootstrapping functionality—namely, the acquisition of CA root certificates—is implemented by the SCEP database service provider.

- Cert-C CMP PKI service provider

  Use the Cert-C CMP PKI service provider to send certificate requests and retrieve certificate responses using a CA that implements the CMP protocol. This service provider implements the CMP initialization certificate request and response messages (`ir/ip`) and the certificate request and response messages (`cr/cp`) according to the profiles outlined in *RFC 2510* and *RFC 2511* for CMP version 1 messages, and `draft-ietf-pkix-rfc2510bis-06.txt` and `draft-ietf-pkix-rfc2511bis-04.txt` for CMP version 2 messages.

Chapter 6

Using the List Object

List Object

Use the LIST_OBJ object to store and pass a collection of abstract data types, including types defined by Cert-C and types defined by your application. The list object is a generic container for multiple values; the values can be of the same type or of different types. For example:

- Cert-C list objects—Cert-C uses a list object to store an extension's value list; it also uses a list object to store extension values that consist of multiple components, such as the Certificate Policies extension.
- Application-defined list objects—An application can use a list object as a container for any kind of value; each value can even be defined by a different data structure.

Cert-C provides a number of functions that you can use to maintain list objects of common Cert-C data types. For example, it provides a set of functions that you can use to maintain list objects that contain ITEM structures, CERT_OBJ objects, and CRL_OBJ objects; these functions are declared in the certlist.h header file. As another example, there is a set of functions in the cms.h header file to maintain list objects that contain RECIPIENT_INFO structures. Before you create any functions to manage lists of Cert-C objects or structures, check the description of the object or structure to see whether the list management functions are already provided by Cert-C.

List-Object Entry Handler

An application can use a LIST_OBJ as a container for any type of value. It is also possible to use different data structures for each value, provided that the LIST_OBJ_ENTRY_HANDLER is set up correctly.

The application can use the LIST_OBJ_ENTRY_HANDLER to store any kind of user-defined data in the LIST_OBJ, while the library has no knowledge of the required data structure. Since an application knows exactly what kind of data structure is required, it must reflect this requirement by setting up the AllocAndCopy and Destructor callbacks appropriately.

See the *API Reference* entry for the LIST_OBJ_ENTRY_HANDLER structure for a description of the *AllocAndCopy* and *Destructor* callbacks.

The LIST_OBJ_ENTRY_HANDLER.*AllocAndCopy* function is called upon to make a copy of the element that a user wants to add an element to a list object. The C_AddListObjectEntry takes in a pointer to a LIST_OBJ_ENTRY_HANDLER as its fourth argument. This list-object entry handler must correspond to the type of the entry given as the second argument to C_AddListObjectEntry. Routines such as C_AddCertToList or C_AddRecipientToList rely upon LIST_OBJ_ENTRY_HANDLERS defined in the toolkit for those particular objects (C_CertEntryHandler and C_RecipientInfoEntryHandler respectively, for example).

The LIST_OBJ_ENTRY_HANDLER.*Destructor* function is called upon to zeroize sensitive data and free memory allocated by the LIST_OBJ_ENTRY_HANDLER.*AllocAndCopy* function.

List-Object Functions

You must use a Cert-C function to view or modify information in a LIST_OBJ object. For application-defined data types, you must provide a LIST_OBJ_ENTRY_HANDLER. You cannot assume that the LIST_OBJ object points to any specific information. Some examples of the functions that Cert-C provides for adding an entry, deleting an entry, searching for an entry, and so forth, are given in the following tables.

Create, Reset, or Destroy LIST_OBJ Functions

Function	Description
C_CreateListObject	Creates a list object.
C_DestroyListObject	Destroys the list object, freeing all occupied memory.
C_ResetListObject	Destroys all entries in the list object, freeing all memory occupied by these entries.

Set or Modify LIST_OBJ Functions

Function	Description
C_AddCertToList	Adds a copy of a CERT_OBJ to the list object.
C_AddCRLToList	Adds a copy of a CRL_OBJ to the list object.
C_AddItemToList	Adds a copy of an ITEM to the list object.
C_AddPrivateKeyToList	Adds a copy of a private-key object to the list object.
C_AddRecipientToList	Adds a copy of a RECIPIENT_INFO to the list object.
C_AddSignerToList	Adds a copy of a SIGNER_INFO to the list object.
C_AddUniqueCertToList	Adds a copy of a CERT_OBJ only if it is not already in the list.
C_AddUniqueCRLToList	Adds a copy of a CRL_OBJ only if it is not already in the list.
C_AddUniqueItemToList	Adds a copy of an ITEM only if it is not already in the list.
C_AddUniqueRecipientToList	Adds a copy of a RECIPIENT_INFO only if it is not already in the list.
C_AddUniqueSignerToList	Adds a copy of a SIGNER_INFO only if it is not already in the list.
C_AddListObjectEntry	Adds an entry to the list object.
C_InsertListObjectEntry	Inserts an entry to the list object at a given position.
C_DeleteListObjectEntry	Destroys an entry in the list object, freeing all memory occupied by the entry.

Get LIST_OBJ Functions

Function	Description
C_GetListObjectCount	Gets the number of entries in the list object.
C_GetListObjectEntry	Gets a specific entry from the list object.

Creating and Enumerating a List of Objects

The following list object examples create a list of certificate objects and retrieve entries from a list of certificate objects. Other objects like the CERT_OBJ, which are opaque pointers, can be handled similarly. You do not need to define a LIST_OBJ_ENTRY_HANDLER for certificate objects because this handler is already defined in the toolkit. LIST_OBJ_ENTRY_HANDLER is used when you call C_AddCertToList.

Creating a List of Certificates

To add a CERT_OBJ *certObj* into a list, use the C_AddCertToList function.

```
CERT_OBJ certObj;
LIST_OBJ certList = (LIST_OBJ)NULL_PTR;

status = C_CreateListObject (&certList);
if (status != 0)
  goto CLEANUP;

status = C_AddCertToList (certList, certObj, (unsigned int *)NULL_PTR);
if (status != 0)
  goto CLEANUP;
```

Now the *certList* has a copy of the *certObj* and its contents. To free the elements in the list object and dispose of the list object, when it is no longer needed, use the C_DestroyListObject function.

```
C_DestroyListObject (&certList);
```

The call to C_AddCertToList is equivalent to the following:

```
status = C_AddListObjectEntry (certList, (POINTER)certObj,
                               (unsigned int*)NULL_PTR, &C_CertEntryHandler);
```

Enumerating a List of Objects

To enumerate each entry in a list object of certificate objects, you must first find out how many entries there are in the list. To count the entries in the list object, use the C_GetListObjectCount function.

```
unsigned int numEntries = 0;

status = C_GetListObjectCount (certList, &numEntries);
if (status != 0)
  goto CLEANUP;
```

Using the C_GetListObjectEntry function, you can now use the value in *numEntries* to iterate through the elements of the list.

```
POINTER entry = NULL_PTR;

for (i = 0; i < numEntries; i++) {
  status = C_GetListObjectEntry (certs, i, &entry);
  if (status !=0)
    goto CLEANUP;

  DoSomethingToTheCert ((CERT_OBJ)entry);
}
```

Creating and Enumerating a List of Structures

The following list object examples create a list of ITEMs and retrieve entries from a list of ITEMs. C_AddItemToList refers to an internal C_ItemEntryHandler list-object entry handler so you do not need to define a list-object entry handler.

Creating a List of ITEMs

To add an ITEM into a list, use the C_CreateListObject function.

```
ITEM item;
LIST_OBJ itemList = (LIST_OBJ)NULL_PTR;

status = C_CreateListObject (&itemList);
if (status != 0)
  goto CLEANUP;

status = C_AddItemToList (itemList, &item, (unsigned int *)NULL_PTR);
if (status != 0)
  goto CLEANUP;
```

Now the *itemList* has a copy of the ITEM and its contents. To free the elements in the list object and dispose of the list object, when it is no longer needed, use the C_DestroyListObject function.

```
C_DestroyListObject (&itemList);
```

The call to C_AddItemToList is equivalent to the following:

```
status = C_AddListObjectEntry (itemList, (POINTER)&item,
                               (unsigned int *)NULL_PTR,
                               &C_ItemEntryHandler);
```

Note: The difference between this example and the "Create a list of certificates" example is that the CERT_OBJ is already a pointer. The actual structure that lies behind the CERT_OBJ object (defined to be a POINTER) is not visible to the caller, whereas the fields of the ITEM structure are visible to the caller.

Enumerating a List of ITEMs

To enumerate each entry in a list object of ITEMs, you must first find out how many entries there are in the list. To count the entries in the list object, use the C_GetListObjectCount function.

```
unsigned int numEntries = 0;

status = C_GetListObjectCount (itemList, &numEntries);
if (status != 0)
  goto CLEANUP;
```

Using the C_GetListObjectEntry function, you can now use the value in *numEntries* to iterate through the elements of the list.

```
POINTER entry = NULL_PTR;

for (i = 0; i < numEntries; i++) {
  status = C_GetListObjectEntry (itemList, i, &entry);
  if (status != 0)
    goto CLEANUP;

  DoSomethingToTheItem ((ITEM *)entry);
}
```

Creating and Enumerating a List of User-Defined Elements

To create a list of user-defined elements, you must first create your own list-object entry handler. In this example, you create list objects containing structures of type USER_TYPE. Begin by creating your own routines (using the callbacks in the LIST_OBJ_ENTRY_HANDLER as examples) that the list object will use to copy the data in a USER_TYPE structure. You also use these routines to free the data allocated when making the copy of the data.

The function declarations and list object entry handler are as follows:

```
int AllocAndCopyMyData (POINTER *copiedData, POINTER data);
void FreeMyData (POINTER data);

/* Application list object handler */
LIST_OBJ_ENTRY_HANDLER myDataHandler = {
  AllocAndCopyMyData, FreeMyData
};
```

`AllocAndCopyMyData` takes a pointer to *data,* in this example a `USER_TYPE` structure. You allocate a new `USER_TYPE` structure, and the necessary fields in that structure, so that you can make a deep copy of the original data. Then you modify the given pointer so that a pointer to the newly allocated `USER_TYPE` structure is returned.

```
int AllocAndCopyMyData (POINTER *copiedData, POINTER data)
{
  USER_TYPE *newData; /* USER_TYPE is a user-defined data type */

  /* Allocate our new USER_TYPE buffer */
  newData = (USER_TYPE *)T_malloc (sizeof (*newData));
  if (newData = (USER_TYPE *)NULL_PTR)
    return (E_ALLOC);

  /* Copy information from data to newData. CopyMyData is a user-defined
     function, and allocates additional space for fields if needed */
  CopyMyData (newData, (USER_TYPE *)data);

  /* At this point, we have finished a deep copy of the USER_TYPE
     information pointed to by data into newData. Now we set the
     copiedData pointer to point to newData, returning a pointer
     to the newData we allocated for the caller. */
  *copiedData = (POINTER)newData;

  /* Return 0 to indicate success */
  return 0;
}
```

`FreeMyData` completely frees all of the buffers allocated by `AllocAndCopyMyData`,

zeroizing out any buffers that are defined to contain sensitive data.

```
void FreeMyData (POINTER copiedDataPtr)
{
  if (copiedDataPtr == NULL_PTR)
    return;

  /* First, de-allocate all the allocated buffers in copiedDataPtr, */
  /* if any exist. The FreeData function frees the data allocated by */
  /* CopyMyData, which is called by AllocAndCopyMyData. */
  FreeData ((USER_TYPE *)copiedDataPtr);

  /* Now free the actual data buffer itself. */
  T_free (copiedDataPtr);

  return;
}
```

Chapter 7

Using the Name and Attributes Objects

This chapter presents the NAME_OBJ and ATTRIBUTES_OBJ objects and their related APIs. You can use these APIs to create, manipulate, or destroy these objects. This chapter also includes examples on how to use these objects and APIs.

When creating a certificate, you need a way to represent the certificate subject's information. This information is called a distinguished name (DN), as defined in the X.500 standard. A DN should uniquely identify each entity. Cert-C provides the NAME_OBJ to represent a DN. For example, you can use the NAME_OBJ to parse a subject's DN, or it can be used, along with other elements, to create a certificate request.

The X.509 standard defines what information can be contained in a certificate. However, the subject might want to associate additional information with their certificate. Sometimes this information can be included in an extension. For more information about extensions, see chapter 15.

Cert-C provides the ATTRIBUTES_OBJ object to represent and pass extra information about the certificate subject, for example, in a certification request. Usually, this extra information is not allowed in a DN.

The attributes object is a general mechanism for holding attribute types and values. For example, you can use it in a PKCS #10 certificate request to represent information the requestor would like associated with the certificate. The attributes object can also be used in PKCS #10 messages and PKCS #7 Signed-Data messages.

Name Object

Cert-C uses a NAME_OBJ object to represent the names of entities involved in a PKI. A name object contains a DN, as defined in the X.500 standard. A DN should uniquely identify each entity. An X.500-defined DN specifies a path through an X.500-defined directory tree.

The name on an X.509 certificate is actually a DN, which is itself a set of Relative Distinguished Names (RDNs), which in turn is a set of Attribute Value Assertions (AVAs). Finally, an AVA is made up of an attribute type and an attribute value. Figure 7-1 shows the different levels of a DN.

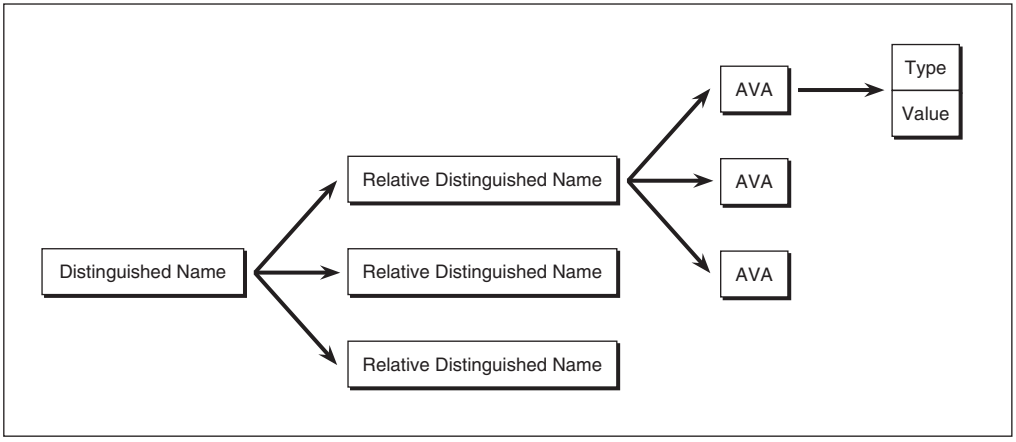


Figure 7-1 **Breakdown of a Distinguished Name**

Each level involves one or more AVAs; the AVA list indicates whether successive AVAs are part of the same level or different levels. There is no significance to the order of AVAs within a level. Furthermore, some environments require that an AVA of any type only appear once within a level. However, Cert-C does not enforce this requirement.

When building a DN, you add information to a name object. Then you add more and more until you have entered all the information you want to be part of the name. Each unit of information you add is a value of a particular type, a type permitted as part of a name. When part of a DN, this value of a particular type is known as an AVA. If this value of a particular type were not part of a DN, it would be known as an attribute.

The DN can be extracted in two forms—as a DER encoding or as a list of AVAs. The

two forms provide equivalent information. The DER encoding is a string of unsigned characters that represents the path; the AVA list contains the AVAs that define each level traversed by the path through the tree.

Name-Object Functions

You must use a Cert-C function to view or modify information in a NAME_OBJ object. You cannot assume that the NAME_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to manipulate a name object are listed in the following tables.

Create, Reset, or Destroy NAME_OBJ Functions

Function	Description
C_CreateNameObject	Creates a new name object.
C_DestroyNameObject	Deletes the name object and de-allocates all memory associated with it.
C_ResetNameObject	Deletes all the name AVA entries in the name object and frees all the associated memory.

Set or Modify NAME_OBJ Functions

Function	Description
C_SetNameBER	Modifies the value of a name to a given BER encoding.

Get NAME_OBJ Functions

Function	Description
C_GetNameDER	Gets a pointer to the DER encoding that represents the value of the name.
C_GetNameString	Gets a NUL-terminated UTF8-string form of the name object, conforming to *RFC 2253*, in the order of least-significant RDN first.

Function	Description
C_GetNameStringReverse	Gets a NUL-terminated UTF8-string form of the name object, conforming to *RFC 2253*, in the order of most-significant RDN first (reverse order to C_GetNameString).
C_IsSubjectSubordinateToIssuer	Checks whether the subject is subordinate to the issuer.

AVA-List Functions

Every DN is comprised of one or more levels and each level can have one or more AVAs. Each AVA has a type, such as AT_ORGANIZATION, and a value, such as the name of the organization. The value also has a tag, which is usually VT_PRINTABLE_STRING; but the value can have a different tag if it can be represented in a form other than a printable string.

A typical application calls C_GetNameAVACount to get the number of AVAs in an AVA list. Next the application calls C_GetNameAVA to obtain each AVA in a name that is being displayed. Then the application calls C_AddNameAVA to construct a new name or add lower levels to an existing name prefix.

Some of the functions that Cert-C provides to access or modify a name object's AVA list are listed in the following table.

Function	Description
C_AddNameAVA	Adds an attribute-value assertion to a name object's AVA list.
C_GetNameAVA	Gets a specific attribute-value assertion from a name object's AVA list.
C_GetNameAVACount	Gets the number of attribute-value assertions in a name object's AVA list.

Attribute Types and Constraints

Cert-C defines a number of attribute types. For some attribute types, Cert-C places some constraints on the corresponding attribute values and their tags. The attribute types and lengths (given as variables that the application can reference), the attribute descriptions, and the attribute value and length constraints are listed in the "Attribute Types and Constraints" section of the *API Reference*.

Creating a Name Object

This example creates a name object and adds DN information to the name object.

You do not need to use the CERTC_CTX context when creating a name object. You can look at the samples/name/name.c sample program and use it to experiment with creating and parsing name objects.

Note: For an example of how to retrieve name information from a NAME_OBJ, see "Retrieving Name-Object Information" on page 216.

Step 1: Create a name object

To create a name object you use the C_CreateNameObject function. For more information on C_CreateNameObject, see the *API Reference*.

```
int C_CreateNameObject (
  NAME_OBJ  *nameObject                          /* (out) New name object */
);
```

Using the C_CreateNameObject function, you declare a variable to be NAME_OBJ and pass its address as the argument. The return value of this routine is a 0 (zero) for success and a non-zero error code when something goes wrong. Any clean-up code always executes, whether an error occurs or not. You should initialize an object to NULL_PTR; if there is an error before an object has the chance to be created, the clean-up code acts on a NULL_PTR and does not do any damage.

```
int status;

NAME_OBJ requestorName = (NAME_OBJ)NULL_PTR;

status = C_CreateNameObject (&requestorName);
if (status != 0)
  goto CLEANUP;
...
CLEANUP;
...
```

Step 2: Enter the name information

Now that you have created a name object, you use C_AddNameAVA to fill it with the

proper information. For more information on C_AddNameAVA, see the *API Reference*.

```
int C_AddNameAVA (
  NAME_OBJ        nameObject,                    /* (in/out) Name object */
  unsigned char  *type,                                /* Attribute type */
  unsigned int    typeLen,                  /* Length of attribute type */
  int             valueTag,                /* Tag for the attribute value */
  unsigned char  *value,                               /* Attribute value */
  unsigned int    valueLen,                /* Length of attribute value */
  int             newLevel,               /* Flag if AVA starts new level */
  unsigned int   *index                             /* (out) Index to AVA */
);
```

The first argument is the name object you just created. The second and third arguments are an attribute type and its length. An AVA consists of a type and a value. The type describes what kind of information this AVA contains. The X.520 standard lists official attributes such as COUNTRY, ORGANIZATION, and COMMON_NAME. You can find some of the most common attributes in the *API Reference*. Cert-C provides OIDs (object identifiers) for these attributes along with their lengths.

The next two arguments refer to a value. The value of the AVA is the information itself. The *valueTag* states which character set is used for the attribute value. For more information on the character sets supported in Cert-C, see "Character Sets" on page 47. Cert-C defines some commonly used attributes and also defines which *valueTag* to use. For more information on these attributes and their value tags, see "Attribute Types and Value Tags" section in the *API Reference*.

The *newLevel* argument is a flag to tell Cert-C when you are beginning a new RDN. Cert-C differentiates between RDNs by starting a new level. You call C_AddNameAVA once for each AVA. When you set the *newLevel* flag to a non-zero value, you are beginning a new level, or RDN. You must add each level in order, beginning at the top. Setting *newLevel* to 0 (zero) means you are still in the same RDN. Figure 7-2 shows

the structure of a DN.

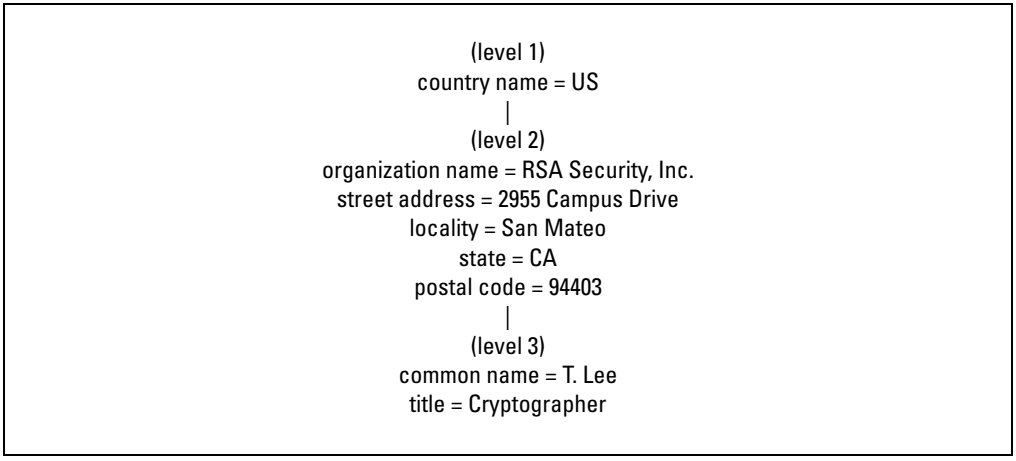


Figure 7-2 **Distinguished Name with Three RDNs**

When you begin a new RDN depends on your organization's structure. Country is an X.500-defined RDN. Other than country, you generally begin a new level when the information in the AVA is different for each certificate requester. For example, Figure 7-2 shows the DN is comprised of the employee's name, the name of the company, and the company's address. If a company has offices throughout the US; then not every employee will have the same address. Therefore, the company name and the address will be on two separate RDNs. However, you may want one single company address on each certificate, regardless of where the employee goes to work. In this case, the company name and address is the same for each employee, so they can make up one RDN.

The last argument is an address of an `unsigned int` variable. Cert-C goes to that address and write an index. Each AVA has an index number. If you want to save that value for reference, pass a pointer to an `unsigned int`. If not, pass a properly cast

NULL_PTR.

```
char *orgName = "RSA Security Inc.";
char *streetAddress = "2955 Campus Dr., Suite 400";
char *locality = "San Mateo";
char *state = "CA";
char *zipCode = "94403";
char commonName[80], title[80], employeeNumber[8];

status = C_AddNameAVA (requestorName, AT_COUNTRY, AT_COUNTRY_LEN,
                       VT_PRINTABLE_STRING, "US", COUNTRY_LEN, 1,
                       (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

status = C_AddNameAVA (requestorName, AT_ORGANIZATION,
                       AT_ORGANIZATION_LEN, VT_PRINTABLE_STRING,
                       (unsigned char *)orgName, T_strlen (orgName),
                       1, (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

status = C_AddNameAVA (requestorName, AT_LOCALITY, AT_LOCALITY_LEN,
                       VT_PRINTABLE_STRING, (unsigned char *)locality,
                       T_strlen (locality), 0, (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

status = C_AddNameAVA (requestorName, AT_STATE, AT_STATE_LEN,
                       VT_PRINTABLE_STRING, (unsigned char *)state,
                       T_strlen (state), 0, (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

status = C_AddNameAVA (requestorName, AT_POSTAL_CODE, AT_POSTAL_CODE_LEN,
                       VT_PRINTABLE_STRING, (unsigned char *)zipCode,
                       T_strlen (zipCode), 0, (unsigned int *)NULL_PTR);
```

```
if (status != 0)
  goto CLEANUP;

puts ("Enter employee name.")
fgets ((char *)commonName, sizeof (commonName), stdin);

status = C_AddNameAVA (requestorName, AT_COMMON_NAME, AT_COMMON_NAME_LEN,
                       VT_PRINTABLE_STRING, (unsigned char *)commonName,
                       T_strlen (commonName), 1, (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

puts ("Enter employee title.")
fgets ((char *)title, sizeof (title), stdin);

status = C_AddNameAVA (requestorName, AT_TITLE, AT_TITLE_LEN,
                       VT_PRINTABLE_STRING, (unsigned char *)title,
                       T_strlen (title), 0, (unsigned int *)NULL_PTR);

if (status != 0)
  goto CLEANUP;

puts ("Enter employee number.")
fgets ((char *)employeeNumber, sizeof (employeeNumber), stdin);
```

Either the certificate requestor or the CA may want additional information about the subject to be added to the certificate; for example, an employee number. However, because an employee number is not an official X.520 attribute, it does not appear in the subject's DN. Cert-C enables you to add additional information about the subject with the attributes object. In this case, you must create a user-defined name attribute to contain the employee number.

For more information on the attributes object and creating user-defined attributes, see "Attributes Object" on page 112.

Step 3: Perform operations

In this example, you do not perform any sign or verify operations.

Step 4: Retrieve the name information in DER format

Now you can retrieve the DER encoding of the name object using the `C_GetNameDER`

function. Once you retrieve the DER encoding, you can send this encoding to anyone who can read BER-encoded information, whether they use Cert-C or not. You can also save this name information in a file or database. You may want to do this to reuse it later to build a certificate request, rather that re-entering the information.

```
int C_GetNameDER (
NAME_OBJ       nameObject,                                  /* Name object */
unsigned char  **der,                          /* (out) DER-encoded name */
unsigned int   *derLen                /* (out) Length of DER-encoded name */
);
```

Using the C_GetNameDER function, you give Cert-C a name object, the address of a pointer, and the address of an unsigned int. Cert-C places at the addresses a pointer to the DER-encoding of the subject name and length. The memory that the pointer to the DER-encoding points to belongs to Cert-C. You do not need to allocate or free that memory. Also, you should not attempt to adjust the data yourself. The information remains unchanged until you call a Cert-C routine that modifies or destroys the name object. To save this information, you must copy it into a file or your own buffer.

```
unsigned char *nameDER;
unsigned int nameDERLen;

status = C_GetNameDER (requestorName, &nameDER, &nameDERLen);
if (status != 0)
  goto CLEANUP;
```

The RSA_WriteDataToFile routine is not a Cert-C routine; it is a demo utility routine. For more information about Cert-C demo utilities, see the "Utilities" chapter in the *Advanced Developer's Guide*. You can use RSA_WriteDataToFile to write binary data to a file.

```
status = RSA_WriteDataToFile
           (nameDER, nameDERLen,
            "Enter name of file to store name object binary");
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the name object

At this stage, you might want to keep and reuse the name object. For example, you will need to use a name object in some of the examples presented in the following

chapters. However, if you no longer need the name object, making sure you have saved any information you need later, then you destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyNameObject (&requestorName);
```

Attributes Object

Cert-C uses an ATTRIBUTES_OBJ object to represent and pass extra information about the certificate subject, for example, in a certification request. Usually, this extra information is not allowed in a DN. The extra information may be attribute types not allowed in a name, user-defined attribute types, or they may also be X.509 certificate extensions.

An attribute object or set is made up of all the extra attributes associated with one entity. Each attribute has an attribute type and one or more values. Some attribute types, such as the time at which a message is signed, can only have one value; other attribute types, such as a postal address, can have multiple values. There is no significance to the order of the different attribute types in an attribute set, or to the order of multiple values for a particular attribute type. Figure 7-3 shows a representation of an attributes object.

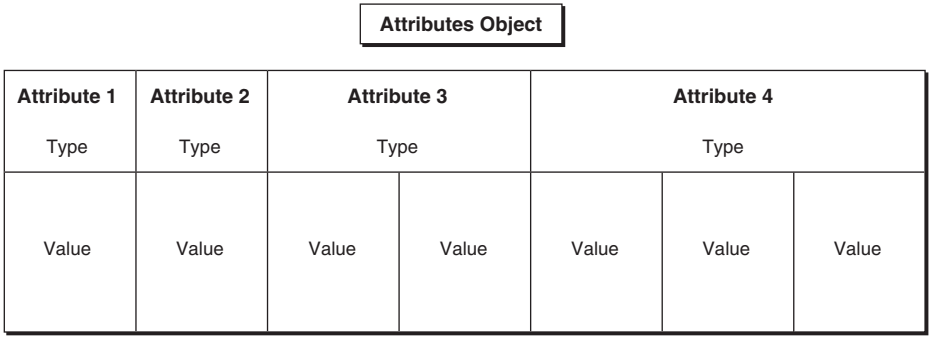


Figure 7-3 **An Attributes Object**

You can extract an attributes set from an attributes object in either of two forms—a DER encoding or a list of attributes. The two forms provide equivalent information. The DER encoding is an unsigned character string that represents the attribute set. The attributes list gives each attribute in the set one at a time.

Attributes-Object Functions

You must use a Cert-C function to view or modify information in an ATTRIBUTES_OBJ. You cannot assume that the ATTRIBUTES_OBJ points to any specific information. Some examples of the functions that Cert-C provides to manipulate an attributes object are

listed in the following table.

Create, Reset, or Destroy ATTRIBUTES_OBJ Functions

Function	Description
C_CreateAttributesObject	Creates a new attributes object.
C_DestroyAttributesObject	Deletes the attributes object and de-allocates all memory associated with it.
C_ResetAttributesObject	Resets an attributes object, returning the attributes object to the state produced by calling the C_CreateAttributesObject function.

Set or Modify ATTRIBUTES_OBJ Functions

Function	Description
C_AddAttributeValueBER	Adds the BER encoding of an attribute value to a specific attribute type in an attributes object's attribute list.
C_AddPostalAddressValue	Adds a postal-address value to a specific postal-address attribute in the attributes object's attribute list.
C_AddStringAttribute	Adds the contents and type tag of a specific string-based attribute to an attributes object's attribute list.
C_DeleteAttributeType	Deletes a specific attribute type and all its values from an attributes object's attributes list.
C_DeleteChallengePasswordAttrib	Deletes the challenge-password attribute in the attributes object's attribute list.
C_DeletePostalAddressAttribute	Deletes the postal-address attribute in the attributes object's attribute list.
C_DeleteSigningTimeAttribute	Deletes the signing-time attribute in the attributes object's attribute list.
C_SetAttributesBER	Sets an attribute object with a set of BER-encoded attribute types and values.

Function	Description
C_SetAttributesNameValueEncoded	Modifies the attributes-object value with a URL-encoded attribute set, and enables the separator characters in the URL encoding to be user-defined.
C_SetAttributesURLEncoded	Modifies the attributes-object value with a URL-encoded attribute set, and uses default values for the separator characters.
C_SetChallengePasswordAttribute	Sets the value of the challenge-password attribute in the attributes object's attribute list.
C_SetSigningTimeAttribute	Sets the value of the signing-time attribute in the attributes object's attribute list.

Get ATTRIBUTES_OBJ Functions

Function	Description
C_GetAttributeType	Gets the type of a specific attribute from an attributes object's attribute list.
C_GetAttributeTypeCount	Gets the number of attributes in an attributes object's attribute list.
C_GetAttributeValueCount	Gets the number of attribute values for a specific attribute type in an attributes object's attribute list.
C_GetAttributeValueDER	Gets the DER encoding of a specific value of a specific attribute in an attributes object's attribute list.
C_GetAttributesDER	Gets a pointer to the DER encoding that represents the attribute set.
C_GetAttributesNameValueEncoded	Gets the URL encoding of the attributes object's value, and enables the separator characters in the URL encoding to be user-defined.
C_GetAttributesURLEncoded	Gets the URL encoding of the attributes object's value, and uses default values for the separator characters.
C_GetChallengePasswordAttribute	Gets the value of the challenge-password attribute in the attributes object's attribute list.

Function	Description
C_GetPostalAddressValue	Gets the value indexed by *valueIndex* in the postal-address attribute in the attributes object's attribute list.
C_GetPostalAddressValueCount	Gets the number of values for the postal-address attribute in the attributes object's attribute list.
C_GetSigningTimeAttribute	Gets the value of the signing-time attribute in the attributes object's attribute list.
C_GetStringAttribute	Gets the contents and type tag of a specific string-based attribute in an attributes object list.

Attribute Types and Constraints

Cert-C defines a number of attribute types. For some attribute types, Cert-C places some constraints on the corresponding attribute values and their tags. The attribute types and lengths (given as variables that the application can reference), the attribute descriptions, and the attribute value and length constraints are listed in the *API Reference*.

Creating an Attributes Object

The attributes object is a general mechanism for holding attribute types and values. For example, you can use it in a PKCS #10 certificate request to represent information the requestor would like associated with the certificate. The attributes object can also be used in PKCS #10 messages and PKCS #7 Signed-Data messages.

The following attributes object example creates an attributes object and adds attribute information to the attributes object.

Note: For an example of how to retrieve attribute information from a ATTRIBUTES_OBJ, see "Retrieving Attributes-Object Information" on page 219.

As with the name object, you do not need to use the CERTC_CTX context when creating an attributes object. You can look at the samples/attrib/attrib.c sample program and use it to experiment with creating and parsing attributes objects.

Step 1: Create an attributes object

To create an attributes object you use the C_CreateAttributesObject function. For

more information on C_CreateAttributesObject, see the *API Reference*.

```
int C_CreateAttributesObject (
  ATTRIBUTES_OBJ  *attributesObj              /* (out) attributes object */
);
```

Using the C_CreateAttributesObject function, you declare a variable to be ATTRIBUTES_OBJ and pass its address as the argument. The return value of this routine is a 0 (zero) if successful and a non-zero error code when something goes wrong. Any clean-up code always executes, whether an error occurs or not. You should initialize an object to NULL_PTR; if there is an error before an object has the chance to be created, the clean-up code acts on a NULL_PTR and does not do any damage.

```
ATTRIBUTES_OBJ extraAttributes = (ATTRIBUTES_OBJ)NULL_PTR;

status = C_CreateAttributesObject (&extraAttributes);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the attributes information

Now that you have created an attributes object, you need to add attribute information to the attributes object. You can choose to use an X.520-defined attribute. However, in this case, you use C_AddStringAttribute to add a user-defined attribute type. The user-defined attribute type will be for the subject's employee number. For more information on C_AddStringAttribute, see the *API Reference*.

```
int C_AddStringAttribute (
  ATTRIBUTES_OBJ  attributesObj,           /* (in/out) attributes object */
  unsigned char   *type,                              /* attribute type */
  unsigned int    typeLen,                  /* length of attribute type */
  int             valueTag,                 /* tag for the string value */
  unsigned char   *value,                              /* string value */
  unsigned int    valueLen                    /* length of string value */
);
```

The first argument is the attributes object that you created. The next two arguments are the attribute's type and length. Because this is not an X.520-defined attribute, you define the attribute's type. In this case, using the *employeeNumber* variable that you declared in the name-object example, you set the employee number attribute type to a string value. Since it is a string of alphanumeric characters, the *valueTag* can be

VT_PRINTABLE_STRING. For more information on the character sets supported in Cert-C, see "Character Sets" on page 47.

```
unsigned char employeeNumberOid[] = "Employee Number";

status = C_AddStringAttribute (extraAttributes, employeeNumberOid,
                               T_strlen (employeeNumberOid),
                               VT_PRINTABLE_STRING, employeeNumber,
                               T_strlen (employeeNumber));
if (status != 0)
  goto CLEANUP;
```

Step 3: Perform operations

In this example, you do not need to perform a sign or verify operation.

Step 4: Retrieve the attributes information in DER format

Now you can retrieve the DER encoding of the attributes object using the C_GetAttributesDER function. Once you retrieve the DER encoding, you can send this encoding to anyone who can read BER-encoded information, whether they use Cert-C or not. You can also save this attributes information in a file or database. You may want to do this to reuse it later to build a certificate request, rather that re-entering the information.

```
int C_GetAttributesDER (
  ATTRIBUTES_OBJ  attributesObj,                    /* Attributes object */
  unsigned char   **der,                 /* (out) DER-encoded attributes */
  unsigned int    *derLen             /* (out) Length of DER-encoded attr */
);
```

Using the C_GetAttributesDER function, you give Cert-C an attributes object, the address of a pointer, and the address of an unsigned int. At the addresses, Cert-C places a pointer to the DER encoding of the attributes and the length of the DER-encoded attributes. The memory that the pointer to the DER-encoding points to belongs to Cert-C. You do not need to allocate or free that memory. Also, you should not attempt to adjust the data yourself. The information remains unchanged until you call a Cert-C routine that modifies or destroys the attributes object. To save this

information, you must copy it into a file or your own buffer.

```
unsigned char *attributesDER;
unsigned int attributesDERLen;

status = C_GetAttributesDER (extraAttributes, &attributesDER,
                             &attributesDERLen);

if (status != 0)
  goto CLEANUP;
```

The RSA_WriteDataToFile routine is not a Cert-C routine; it is a demo utility routine. For more information about Cert-C demo utilities, see the "Utilities" chapter of the *Advanced Developer's Guide*. You can use RSA_WriteDataToFile to write binary data to a file.

```
status = RSA_WriteDataToFile
           (attributesDER, attributesDERLen,
            "Enter name of file to store attributes object binary");
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the attributes object

At this stage, you might want to keep and reuse the attributes object. For example, you will need to use an attributes object in some of the examples presented in the following chapters. However, if you no longer need the attributes object, make sure you have saved any information you need later, then destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyAttributesObject (&extraAttributes);
```

Chapter 8

Creating a Certificate Request

In this chapter, you will learn how to use Cert-C to create a certificate request. Your application can use this feature to create certificate requests suitable for presenting to another application, for example a CA, that fulfills certificate requests.

Cert-C provides two ways for you to create a certificate request. You can create a PKCS #10 certificate request or you can create a PKI certificate request message using one of the Cert-C PKI service providers.

The PKCS #10 format only creates the certificate request; it does not transport the request. You must determine a method of transporting a PKCS #10 certificate request to a CA or RA.

The Cert-C PKI messaging APIs support creating and transporting PKI messages, for example, a certificate request, as specified by the SCEP, CRS, and CMP protocols. For an overview, see “PKI Certificate Request Message” on page 128 or for more detailed information on these types of certificate requests, see chapter 9.

The CA and PKI that your organization uses determines the type of certificate request method that you will use.

PKCS #10 Certificate Request

You can create a PKCS #10 certificate request using the Cert-C PKCS #10 object, `PKCS10_OBJ`, as described in the PKCS #10 standard. The PKCS #10 standard does not provide for transporting or submitting certificate requests to a CA. To see how to create a PKCS #10 certificate request, see "Creating a PKCS #10 Certificate Request" on page 123.

Because the PKCS #10 standard does not provide for the transportation of the certificate request, it is up to you and your CA to determine how to transport the certificate request to the CA. You also need to do this for the certificate response. For example, if a CA does not support a digital method for receiving a certificate request, then the certificate requester creates a PKCS #10 certificate request, signs the certificate request, and obtains the DER encoding of the certificate request. The certificate requester can then send the DER-encoded PKCS #10 certificate request by e-mail, FTP, or even facsimile to the CA. The CA can copy and paste the certificate request into the certificate server application, or enter the information manually. Again, the PKCS #10 certificate-request transportation method is completely up to you and your CA.

Figure 8-1 shows the method to transport a PKCS #10 certificate request to a CA depends on the implementation.

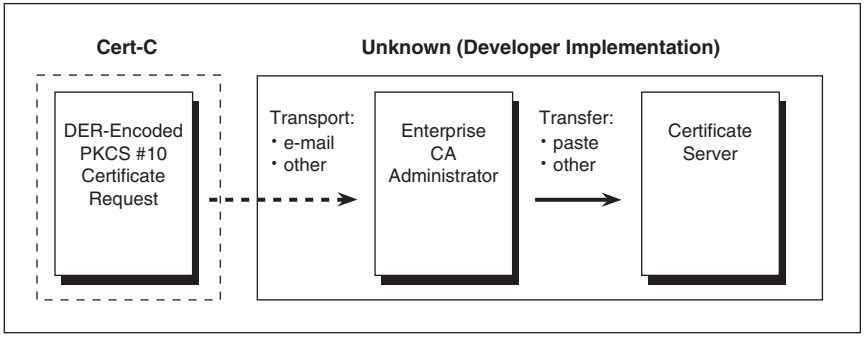


Figure 8-1 **PKCS #10 Certificate Request**

PKCS #10 Object

Cert-C represents certificate-request information with a `PKCS10_OBJ` object. A `PKCS10_OBJ` is used to form the PKCS #10 request binary. It does not cover the

transport of the request or the receipt of the fulfilled request. Cert-C does support certificate-request protocols—for example, CRS, SCEP, and CMP—that include the formation and transport of the request messages. For more information about how to create a certificate request that includes the formation and transport of the request message, see "PKI Certificate Request Message" on page 128. For more information about the request message protocols that Cert-C supports, see the PKI service providers in the "Service Provider" section of the *API Reference*.

PKCS #10-Object Functions

You must use a Cert-C function to view or modify information in a `PKCS10_OBJ` object. You cannot assume that the `PKCS10_OBJ` object points to any specific information. Some examples of the functions that Cert-C provides to generate and manipulate PKCS #10 certification requests are listed in the following table.

Create or Destroy PKCS10_OBJ Functions

Function	Description
`C_CreatePKCS10Object`	Creates a PKCS #10 object.
`C_DestroyPKCS10Object`	Destroys a PKCS #10 object, freeing the memory the certificate-request object occupied.

Set or Modify PKCS10_OBJ Functions

Function	Description
`C_SetPKCS10Fields`	Sets a PKCS #10 object with the values provided in a `PKCS10_FIELDS` structure.
`C_SetPKCS10BER`	Sets the BER encoding of a PKCS #10 object.
`C_SignPKCS10`	Signs a PKCS #10 object.
`C_VerifyPKCS10Signature`	Checks the signature on a PKCS #10 object.

Get PKCS10_OBJ Functions

Function	Description
C_GetPKCS10Fields	Gets the content of the PKCS10_FIELDS structure in a PKCS #10 object.
C_GetPKCS10DER	Gets the DER encoding of a PKCS #10 object.

Creating a PKCS #10 Certificate Request

The following PKCS #10 object example creates a PKCS #10 certificate request. You first create a PKCS #10 object, you add certificate request information to the PKCS #10 object, then you sign the certificate request. Later you can then retrieve the DER-encoded certificate request. You must use the CERTC_CTX context when creating a PKCS #10 object. You can look at the samples/pkcs10/pkcs10.c sample program and use it to experiment with creating and parsing PKCS #10 binary.

Step 1: Create a PKCS #10 certificate request object

To create a PKCS #10 certificate request you need a subject name, possibly some attributes, and the subject's public key. You have already created a name (see "Creating a Name Object" on page 105), and an attribute (see "Creating an Attributes Object" on page 115). See "Key Object" on page 289 to create a public key. Then you build a PKCS #10 certificate request using these objects.

To create a PKCS #10 object you use the C_CreatePKCS10Object function. For more information on C_CreatePKCS10Object, see the *API Reference*.

```
int C_CreatePKCS10Object (
  CERTC_CTX   ctx,                                    /* Cert-C context */
  PKCS10_OBJ  *pkcs10Object          /* (out) PKCS#10 object to be created */
);
```

The C_CreatePKCS10Object function requires a Cert-C context to access a registered cryptographic service provider, and optionally, a registered surrender context. The cryptographic service provider is required for the underlying cryptographic operations; for example, signing the PKCS #10 object. The cryptographic service provider contains the Crypto-C algorithm chooser needed for the underlying cryptographic operations. Because cryptographic operations can take a considerable amount of time, you should also register a surrender context. The surrender context provides a way for you to interrupt lengthy operations or to stop a lengthy operation.

Using the C_CreatePKCS10Object function, you declare a variable to be PKCS10_OBJ and pass its address as the argument. You also pass Cert-C a previously initialized Cert-C context. For more information about initializing a Cert-C context, see "Initializing the Cert-C Context" on page 75. The return value of this routine is a 0 (zero) for success and a non-zero error code when something goes wrong. Any clean-up code always executes, whether an error occurs or not. You should initialize an object to NULL_PTR; if there is an error before an object has the chance to be created,

the clean-up code acts on a NULL_PTR and does not do any damage.

```
CERTC_CTX ctx;
PKCS10_OBJ pkcs10Obj = (PKCS10_OBJ)NULL_PTR;

status = C_CreatePKCS10Object (ctx, &pkcs10Obj);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the PKCS #10 certificate request information

Now that you have created a PKCS #10 object, you need to add certificate-request information to the PKCS #10 object. To fill the PKCS #10 object you use the C_SetPKCS10Fields function. For more information about the C_SetPKCS10Fields function, see the *API Reference*.

```
int C_SetPKCS10Fields (
  PKCS10_OBJ     pkcs10Object,                /* (in/out) PKCS #10 object */
  PKCS10_FIELDS  *pkcs10Fields                          /* PKCS #10 fields */
);
```

The first argument is the PKCS #10 certificate-request object you created. The second argument is a PKCS10_FIELDS structure. This structure holds the information necessary to create a PKCS #10 certification-request message. For more information about PKCS10_FIELDS, see the *API Reference*.

```
typedef struct PKCS10_FIELDS {
  UINT2           version;
  NAME_OBJ        subjectName;
  ITEM            publicKey;
  ATTRIBUTES_OBJ  attribute;
  POINTER         reserved;
} PKCS10_FIELDS;
```

You declare a variable to be a PKCS10_FIELDS structure and fill in the elements with

the information you have already built.

```
PKCS10_FIELDS pkcs10Info;

pkcs10Info.version = CERT_VERSION_1;
pkcs10Info.subjectName = requestorName;
pkcs10Info.publicKey.data = bsafePublicKeyBER->data;
pkcs10Info.publicKey.len = bsafePublicKeyBER->len;
pkcs10Info.attributes = extraAttributes;
pkcs10Info.reserved = NULL_PTR;

status = C_SetPKCS10Fields (pkcs10Obj, &pkcs10Info);
if (status != 0)
  goto CLEANUP;
```

Step 3: Sign the PKCS #10 certificate request

You can now sign the PKCS #10 certificate request. You sign it using the private key associated with the subject's public key in the certificate request. When you send the certificate request to the CA, the CA can use the public key in the certificate request to verify the signature. In this way, the CA can be certain the requestor has access to the private key. Otherwise, anyone can take a public key and present it to the CA and claim to be the owner.

You use the `C_SignPKCS10` function to sign the `PKCS10_OBJ`. For more information about `C_SignPKCS10`, see the *API Reference*.

```
int C_SignPKCS10(
  PKCS10_OBJ  pkcs10Object,                    /* (mod) PKCS#10 object */
  B_KEY_OBJ   subjectPrivateKey,            /* (in) subject's private key */
  int         signAlgorithmID              /* (in) signature algorithm ID */
);
```

The first argument is the `PKCS10_OBJ` that you created. The second argument is a Crypto-C key object, see "Key Object" on page 289. For more information about Crypto-C and its key object, see "Using BSAFE Crypto-C" on page 287 or the *Crypto-C Developer's Guide*. *`signAlgorithmID`* is the signature algorithm identifier. Values for this parameter may be any of the `SA_*` values found in the `certalg.h` header file.

```
status = C_SignPKCS10 (pkcs10Obj, privateKey,
                       SA_SHA1_WITH_RSA_ENCRYPTION);
```

```
if (status != 0)
  goto CLEANUP;
```

Step 4: Retrieve the PKCS #10 certificate request information in DER format

You now have a signed PKCS #10 certificate request object. Ultimately, you want to send the certificate request to a CA in a form that it will understand. To do this you need to use the C_GetPKCS10DER function to retrieve the DER encoding of the certificate-request information.

```
int C_GetPKCS10DER(
  PKCS10_OBJ     pkcs10Object,                    /* (mod) PKCS #10 obj */
  unsigned char  **der,            /* (out) pointer to DER output buffer */
  unsigned int   *derLen            /* (out) length of DER output buffer */
);
```

Using the C_GetPKCS10DER function, you give Cert-C a PKCS #10 object, the address of a pointer and the address of an `unsigned int`. At the addresses, Cert-C places a pointer to the DER encoding of the attributes and the length of the DER-encoded attributes. What the pointer to the DER encoding points to belongs to Cert-C. You do not allocate or free that memory. Also, you should not attempt to adjust the data yourself. The information remains unchanged until you call a Cert-C routine that modifies or destroys the PKCS #10 object. To save this information, you must copy it into a file or your own buffer.

```
unsigned char *pkcs10Der;
unsigned int pkcs10DerLen;

status = C_GetPKCS10DER (pkcs10Obj, &pkcs10Der, &pkcs10DerLen);
if (status != 0)
  goto CLEANUP;
```

The RSA_WriteDataToFile routine is not a Cert-C routine; it is a demo utility routine. For more information about Cert-C demo utilities, see the *Advanced Developer's Guide*.

You can use RSA_WriteDataToFile to write binary data to a file.

```
status = RSA_WriteDataToFile
           (pkcs10Der, pkcs10DerLen,
            "Enter name of file to store PKCS #10 binary");
if (status != 0)
  goto CLEANUP;
```

You can now send the certificate request to the CA.

Step 5: Destroy name, attributes, and PKCS #10 objects

Any object you create you must destroy, making sure you have saved any information you need later. This will free up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing. That is why you should always initialize all objects to NULL_PTR and call the C_Destroy* function later. If there is an error before creating an object, then the C_Destroy* function does not do any damage.

```
CLEANUP:
  C_DestroyNameObject (&requestorName);
  C_DestroyAttributesObject (&extraAttributes);
  C_DestroyPKCS10Object (&pkcs10Obj);
```

PKI Certificate Request Message

You can create an initialization request or a certificate request message using the Cert-C PKI message object, `PKI_MSG_OBJ`, and the PKI certificate request object, `PKI_CERT_REQ_OBJ`. You use the `PKI_MSG_OBJ` and `PKI_CERT_RESP_OBJ` object for the certificate response message.

Cert-C supports the CRS, CMP, and SCEP PKI messaging transport mechanisms. The CRMF message format is also supported for use with the Cert-C CMP PKI service provider. These transport mechanisms are supported through the use of the Cert-C PKI messaging API and the various Cert-C PKI service providers. For more information about the Cert-C PKI service providers, see the "Service Provider" section of the *API Reference*. To create a certificate request and transport it to a CA using CRS, SCEP, or CMP, use the relevant Cert-C PKI service provider and the PKI messaging APIs.

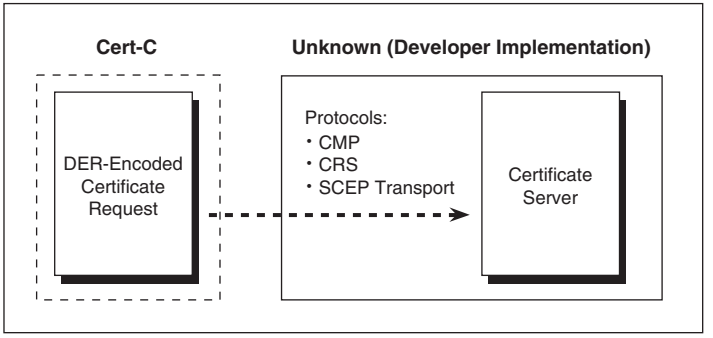


Figure 8-2 **PKI Messaging Certificate Request**

For general information about how to create a PKI request message, see "Creating a PKI Request Message" on page 137. For an example of a PKI certificate request, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

Chapter 9

Creating a PKI Message

When you want to send a request message to a CA or RA, you need a mechanism to communicate with the CA or RA. For example, you might want to request a certificate or revoke a certificate. Cert-C provides transport mechanisms that support the CMP, CRS, and SCEP protocols. These protocols are implemented through the Cert-C PKI service providers. For more information about these service providers, see the "Service Provider" section of the *API Reference*.

You also need a way to represent a request message; in Cert-C, you use the `PKI_MSG_OBJ` object. Whether you choose to use the CMP, CRS, or SCEP protocol, you use the `PKI_MSG_OBJ` object. Cert-C supports PKI request and response messages, of the same type, in the PKI message object.

Cert-C provides request and response objects and APIs to perform the following PKI request and response messages:

- Certificate request and response
- Certificate confirmation request and response
- Key update request and response
- Certificate revocation request and response

When you construct a CMP certification request message, you can also request key archival. This is specified through the *`controls`* field of the `CertReqMessage`, which is in the `PKI_CERT_REQ_OBJ`.

For information about which Cert-C PKI service provider supports each of these PKI

message request and response types, see the "Service Provider" section of the *API Reference*.

PKI Message Object

Cert-C uses a `PKI_MSG_OBJ` object to store and pass PKI message requests and responses that pass between a certification-requesting application and a CA or RA. Cert-C supports certificate requests and responses, certificate confirmation requests and responses, key update requests and responses, and certificate revocation requests and responses.

Cert-C supports the CRS, CMP, and SCEP PKI messaging transport mechanisms. The CRMF message format is also supported for use with the Cert-C CMP PKI service provider. These transport mechanisms are supported through the use of the Cert-C PKI messaging API and the various Cert-C PKI service providers. For more information about the Cert-C PKI service providers, see the "Service Provider" section of the *API Reference*. To create a PKI message request and transport it to a CA using CRS, SCEP, or CMP, use the relevant Cert-C PKI service provider and the PKI messaging APIs

To set a `PKI_MSG_OBJ` with a PKI message type, call `C_SetPKIMsgType` and pass one of the following message types.

- `PKI_MSGTYPE_CERT_REQ`
- `PKI_MSGTYPE_KEY_UPDATE_REQ`
- `PKI_MSGTYPE_REVOKE_REQ`

Similarly, you can call `C_GetPKIMsgType` to determine the type of message in a `PKI_MSG_OBJ`.

Cert-C also supports key archival requests, at the time of the certificate request. To find out which types of PKI request and response messages each Cert-C PKI service provider provides, see the "Service Provider" section of the *API Reference*.

A PKI message object can encapsulate any of the following PKI objects.

PKI Objects

- `PKI_CERT_REQ_OBJ`
- `PKI_CERT_RESP_OBJ`
- `PKI_CERT_CONF_REQ_OBJ`
- `PKI_CERT_CONF_RESP_OBJ`

- PKI_KEY_UPDATE_REQ_OBJ
- PKI_KEY_UPDATE_RESP_OBJ
- PKI_REVOKE_REQ_OBJ
- PKI_REVOKE_RESP_OBJ
- PKI_ERROR_MSG_OBJ

PKI Related Objects

There are two more PKI objects: PKI_CERT_TEMPLATE_OBJ and PKI_STATUS_INFO_OBJ. However, the PKI_MSG_OBJ does not directly encapsulate these objects.

- Both PKI_CERT_REQ_OBJ and PKI_KEY_UPDATE_REQ_OBJ encapsulate the PKI_CERT_TEMPLATE_OBJ object.
- The PKI_CERT_RESP_OBJ, PKI_KEY_UPDATE_RESP_OBJ, PKI_REVOKE_RESP_OBJ, and PKI_CERT_CONF_REQ_OBJ objects encapsulate the PKI_STATUS_INFO_OBJ object.

Deprecated PKI Messaging APIs and Structures

In Cert-C 2.0 you used the C_GetPKIMsgFields and C_SetPKIMsgFields APIs to modify the PKI_MSG_FIELDS structure. This structure represented the fields of a PKI message object. The C_GetPKIMsgFields and C_SetPKIMsgFields API and the PKI_MSG_FIELDS structure were deprecated in Cert-C 2.5. Many other PKI message APIs and structures were deprecated in Cert-C 2.5. The following is a list of those deprecated functions and structures:

Deprecated Functions

- C_GeneratePKIProofOfPossession
- C_GetPKICertRequestFields
- C_GetPKICertResponseFields
- C_GetPKIMsgFields
- C_ReadPKICertResponseMsg
- C_RequestPKICert
- C_SendPKIMsg
- C_SetPKICertResponseFields
- C_SetPKICertRequestFields
- C_SetPKIMsgFields
- C_ValidatePKIProofOfPossession
- C_WritePKICertRequestMsg

Deprecated Structures

- PKI_MSG_FIELDS
- PKI_CERTREQ_FIELDS
- PKI_CERTRESP_FIELDS
- PKI_RECIPIENT

Instead of these deprecated APIs, Cert-C provides new C_Set* and C_Get* APIs that you must call directly on the PKI message object. These APIs modify the internal fields of a PKI message object. Cert-C also provides new C_Set* and C_Get* APIs to modify the internal fields of the new PKI objects, which are encapsulated in the PKI_MSG_OBJ. These new PKI objects and their related APIs are described in detail later in this chapter.

There are three categories of APIs for the PKI message object. The first set of APIs act on the actual PKI_MSG_OBJ; for example, APIs that create, destroy, or reset the PKI_MSG_OBJ. The second set of APIs manipulate the individual PKI message fields in a PKI_MSG_OBJ. The third set of APIs perform operations based on the contents of the PKI_MSG_OBJ; for example, generating or sending a PKI request message to a CA or RA.

PKI Message Object Functions

You must use a Cert-C function to view or modify information in a PKI_MSG_OBJ object. You cannot assume that the PKI_MSG_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI message object are listed in the following table.

Create, Reset, or Destroy PKI_MSG_OBJ Functions

Function	Description
C_AddPKIMsg	Adds a specific type of PKI message to a PKI message object.
C_CreatePKIMsgObject	Creates a new PKI message object and stores the result.
C_DeletePKIMsg	Deletes a PKI message in a PKI message object.
C_DestroyPKIMsgObject	Destroys a PKI message object and frees its associated memory.

Function	Description
C_DestroyPKIProviderData	Destroys the provider-specific data stored in a PKI_MSG_OBJ object by calling the Destroy function specified in the handler provided to the C_SetPKIProviderData function.
C_GetPKIMsg	Gets a PKI message from a PKI message object.
C_GetPKIMsgCount	Gets the number of PKI message objects in a single PKI message object.
C_GetPKIProviderData	Retrieves provider-specific data previously associated with a PKI_MSG_OBJ message, by a call to C_SetPKIProviderData. Typically used by a service provider.
C_ResetPKIMsgObject	Resets a PKI message object to the initial state produced by calling the C_CreatePKIMsgObject function.
C_SetPKIProviderData	Associates service provider-specific data with the PKI_MSG_OBJ object. Typically used by a service provider.

Get, Set, or Modify PKI_MSG_OBJ Functions

Function	Description
C_GetPKIMsgExtraCerts	Gets any extra certificates that are stored in a PKI message object.
C_GetPKIMsgExtraCRLs	Gets any extra CRLs that are stored in a PKI message object.
C_GetPKIMsgFreeText	Gets a list of text strings stored in a PKI message object, which contain context-specific information to accompany the message.
C_GetPKIMsgGeneralInfo	Gets a set of messaging attributes stored in a PKI message object, which are used to convey context-specific information.
C_GetPKIMsgProtectionType	Gets the protection type of an initialized PKI message object.
C_GetPKIMsgRecipient	Gets the recipient information stored in a PKI message object and populates a PKI_RECIPIENT_INFO structure.
C_GetPKIMsgRecipientNonce	Gets the message-recipient nonce stored in a PKI message object.

Function	Description
C_GetPKIMsgSender	Gets the sender information stored in a PKI message object and populates a PKI_SENDER_INFO structure.
C_GetPKIMsgSenderNonce	Gets the message-sender nonce stored in a PKI message object.
C_GetPKIMsgTime	Gets the time when the PKI message was generated, which is stored in a PKI message object.
C_GetPKIMsgTransID	Gets the transaction ID used to associate a request message with its corresponding response message, which is stored in a PKI message object.
C_GetPKIMsgType	Gets the message type of a PKI message object.
C_GetPKIMsgVersion	Gets the value of the protocol version, stored in a PKI message object.
C_SetPKIMsgExtraCerts	Sets extra certificates in the PKI message object. The recipient can use these certificates to build a certificate chain.
C_SetPKIMsgExtraCRLs	Sets extra CRLs in the PKI message object. The recipient can use these certificates to build a certificate chain.
C_SetPKIMsgFreeText	Sets the PKI message object with a list of text strings that contain context-specific information to accompany the PKI message.
C_SetPKIMsgGeneralInfo	Sets the PKI message object with a set of messaging attributes used to convey context-specific information in the PKI message.
C_SetPKIMsgProtectionType	Sets the message protection type of the PKI message.
C_SetPKIMsgRecipient	Sets or initializes the recipient-specific information in a PKI message object.
C_SetPKIMsgRecipientNonce	Sets the message-recipient nonce in a PKI message object.
C_SetPKIMsgSender	Sets or initializes the sender-specific information in a PKI message object.
C_SetPKIMsgSenderNonce	Sets the message-sender nonce in a PKI message object.
C_SetPKIMsgTime	Sets the PKI message generation time in a PKI message object.
C_SetPKIMsgTransID	Sets the transaction ID, used to associate a request message with its corresponding response message, in a PKI message object.

Function	Description
C_SetPKIMsgType	Sets the PKI message type that this PKI message object represents. If the type is changed, the type-specific information is updated.
C_SetPKIMsgVersion	Sets the value of the protocol version in a PKI message object.

Operation PKI_MSG_OBJ Functions

There are two ways to make a PKI request. You can use the C_RequestPKIMsg function (high-level function) or you can use the C_GetPKIMsgDER, C_SendPKIRequest and C_SetPKIMsgBER sequence of functions (low-level functions).

You should use the high-level functions in your application. Use the low-level functions only for debugging purposes and doing batch processing.

For example, the high-level function, C_RequestPKIMsg, generates a request's DER encoding, sends the request's DER to the CA, decodes the response's DER from the CA, and populates the response message object.

Calling the low-level functions, C_GetPKIMsgDER, C_SendPKIRequest and C_SetPKIMsgBER require your application to handle and pass objects or information between the function calls.

The following functions are high-level functions. You should use these functions in your application.

High-Level Function	Description
C_GeneratePKIMsgProofOfPossession	Generates a Proof-Of-Possession (POP) for a specified PKI request message in the PKI message object.
C_RequestPKIMsg	Sends a PKI request message to the specified PKI service provider. Optionally, a follow-on request confirms the first request.
C_ValidatePKIMsgProofOfPossession	Validates a POP for a specified PKI request message in the PKI message object.

The following functions are low-level functions. You can use these functions for

debugging your application or batch processing.

Low-Level Function	Description
C_GetPKIMsgDER	Creates a serialized request message according to the protocol implemented by the specified PKI service provider. This API does not transmit the message to a CA or RA. This function also applies relevant cryptographic protections, such as digital signatures or envelopes, to the message as a part of the serialization process.
C_SendPKIRequest	Sends a serialized PKI request message to a CA or RA and returns an encoded PKI response message, and the status of the PKI request.
C_SetPKIMsgBER	Processes a PKI response message. It validates the cryptographic protections (for example, digital signatures) and sets the serialized response in a PKI message object.

Creating a PKI Request Message

This example demonstrates the general steps involved in creating a PKI request message. To send a PKI request message, you must register a PKI service provider. For more information about the various Cert-C PKI service providers, see the "Service Provider" section of the *API Reference*. For demonstration purposes, this example considers a PKI certificate request using the Cert-C CMP PKI service provider.

For specific PKI request message examples, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*. To generate the key update request information, see the "CMP Key Update Request Example" example or to generate the certificate revocation request information, see the "CMP Certificate Revocation Example" example.

Before you can create a PKI certificate request, a key update request, or a certificate revocation request, you must generate the PKI request information. For example, a name object and a key object.

In this example, you create a PKI request message. When you create a PKI request message, you encapsulate a PKI request object in a PKI message object, `PKI_MSG_OBJ`. The following is a list of the PKI request objects:

- `PKI_CERT_REQ_OBJ`
- `PKI_CERT_CONF_REQ_OBJ`
- `PKI_KEY_UPDATE_REQ_OBJ`
- `PKI_REVOKE_REQ_OBJ`

You create a PKI message object and a PKI request object, then you set the PKI message object with the PKI request object information. You send the PKI request message to a CA or RA. The CA sends a PKI response message to you. You must examine this response and take the appropriate action for the certificate-request protocol that you used. If you find the response is not valid, then evaluate the error message.

For a certificate request, if you use the CMP protocol v1 (*RFC 2510*, CMP1), the `C_RequestPKIMsg` function does not return a response message in the *`response`* output parameter. If you use the CMP protocol v2, (`draft-ietf-pkix-rfc2510bis-06.txt`, CMP 2), the `C_RequestPKIMsg` function returns a response message in the *`response`* output parameter. If an error occurs, `C_RequestPKIMsg` returns a non-zero return code. You should check the return code and the value of response to make sure the request is successful.

Step 1: Register a PKI service provider with a Cert-C context

Initialize Cert-C and register a PKI service provider with the CERTC_CTX. You can do this by calling either C_InitializeCertC or C_RegisterService. The SERVICE_HANDLER structure contains service-provider information and the service-provider initialization function. In this example, you use the Cert-C CMP PKI service provider. The service-provider initialization routine is S_InitializeCMP and the service-provider-specific parameters are stored in a PKI_CMP_SP_INIT_PARAMS structure. To see how to initialize and register a service provider, see "Initializing the Cert-C Context" on page 75.

For more information about C_InitializeCertC and SERVICE_HANDLER, see the *API Reference*.

```
#include "cmp.h"

#define SP_COUNT 1
#define CMP_PROVIDER_NAME "RSA CMP provider"

  CERTC_CTX ctx = NULL;
  SERVICE_HANDLER spTable[SP_COUNT];
  POINTER spParams[SP_COUNT];

  PKI_CMP_SP_INIT_PARAMS cmpInitParams;

  spTable[0].type = SPT_PKI;
  spTable[0].name = CMP_PROVIDER_NAME;
  spTable[0].Initialize = S_InitializeCMP;

  T_memset ((POINTER)&cmpInitParams, 0, sizeof (cmpInitParams));
```

Set the Cert-C CMP PKI service provider's PKI_CMP_SP_INIT_PARAMS.*initChoice* field to PKI_CMP_INIT_METHOD_STRUCT.

Set the PKI service provider's PKI_CMP_SP_INIT_PARAMS.*method.initStruct.profile* field to one of the service provider's profile identifiers. In this example, you set the *profile* field to PKI_CMP_PROFILE_KCA6.

You must also set a TRANSPORT_INFO structure with the information required to locate the CA you wish to use. The PKI_CMP_SP_INIT_PARAMS.*method.initStruct.transport*

field is a TRANSPORT_INFO structure.

```
/* This is the only option currently available */
cmpInitParams.initChoice = PKI_CMP_INIT_METHOD_STRUCT;

cmpInitParams.method.initStruct.profile = PKI_CMP_PROFILE_KCA6;

spParams[0] = (POINTER)&cmpInitParams;

status = C_InitializeCertC (spTable, spParams, SP_COUNT, &ctx);
if (status != 0)
  goto CLEANUP;
```

Step 2: Create and set a PKI message object

In this step, you create a PKI message object and set it with the appropriate information.

Step 2a: Create a PKI message object

Create a PKI message object, PKI_MSG_OBJ, using the C_CreatePKIMsgObject function.

```
PKI_MSG_OBJ pkiMsgObj = (PKI_MSG_OBJ)NULL_PTR;

status = C_CreatePKIMsgObject (ctx, &pkiMsgObj);
if (status != 0)
  goto CLEANUP;
```

Step 2b: Set the PKI message type

Set the PKI message object with the type of PKI message you want to create. You call C_SetPKIMsgType to set one of the following PKI request message types:

- PKI_MSGTYPE_CERT_REQ
- PKI_MSGTYPE_KEY_UPDATE_REQ
- PKI_MSGTYPE_REVOKE_REQ
- PKI_MSGTYPE_CERT_CONF_REQ

In this example, you set the PKI message type to PKI_MSGTYPE_CERT_REQ.

```
status = C_SetPKIMsgType (pkiMsgObj, PKI_MSGTYPE_CERT_REQ);
if (status != 0)
  goto CLEANUP;
```

Step 2c: Set the message protection type

Set the type of protection you want to use to cryptographically protect the PKI message. The protection-type information is stored in a PKI_PROTECT_INFO structure. This protection information is required when you call the C_RequestPKIMsg function.

To set the protection-type information, call the C_SetPKIMsgProtectionType function and pass one of the following protection algorithms:

- PKI_MSG_PROTECTION_NONE
- PKI_MSG_PROTECTION_SIGN
- PKI_MSG_PROTECTION_ENVELOPE
- PKI_MSG_PROTECTION_SIGN_THEN_ENVELOPE
- PKI_MSG_PROTECTION_ENVELOPE_THEN_SIGN
- PKI_MSG_PROTECTION_PBM

If you want to use a shared secret to protect the PKI message, PKI_PROTECT_INFO.*secret* should point to an ITEM that specifies the shared secret. The protection algorithms determine whether a shared secret or public/private key is needed.

If the PKI message-protection algorithms use a public or private key, the PKI_PROTECT_INFO.*protectionCtx* field should specify the certification-path context. It supplies both a source for any certificates required and a source for the private key, if one is needed.

For more information about setting the protection information, see the C_SetPKIMsgProtectionType function and the PKI_PROTECT_INFO structure in the *API Reference*.

Step 2d: Set the sender information

Set the PKI_SENDER_INFO structure with the message sender information. In this example, you set PKI_ENTITY_ID to the PKI_ENTITY_GENERALNAME_KEYID flag to supply the sender's general name and optional sender key identifier. Once you have filled the PKI_SENDER_INFO structure, call the C_SetPKIMsgSender function to set the information into the PKI message object.

The PKI_SENDER_INFO structure has a large number of fields (and these have subfields). Some of these fields are optional, depending on the service provider. For information on which fields are required by a specific service provider, see the "Service Provider" section of the *API Reference*.

If you have not initialized the PKI_SENDER_INFO structure with a prior call to the C_GetPKIMsgSender function, then you should call T_memset before calling the C_SetPKIMsgSender function, to ensure that all unused fields are initialized to 0 (zero).

For more information about setting message-sender information, see the `C_SetPKIMsgSender` function and the `PKI_SENDER_INFO` structure in the *API Reference*.

Step 2e: Set optional PKI message fields

At this point, you have the option to set more PKI message fields. These are optional fields. For a list of the functions to set these fields, see "Get, Set, or Modify PKI_MSG_OBJ Functions" on page 133.

You now have a PKI message object into which you can place a PKI request message object.

Step 3: Create and set a PKI request object

In this step, you create a PKI request object and set it with the appropriate information.

Step 3a: Create a PKI request object

You must choose which type of PKI request object that you want to create, and set (encapsulate) in the PKI message object. However, the PKI request object must match the PKI message type you choose in "Step 2b: Set the PKI message type" on page 139. Table 9-1 shows the different types of request objects you can create and the function you call to create the request object:

Table 9-1 **PKI Request Objects and Their Create Functions**

Request Type	Object	Create Function
Certificate request	`PKI_CERT_REQ_OBJ`	`C_CreatePKICertReqObject`
Key update request	`PKI_KEY_UPDATE_REQ_OBJ`	`C_CreatePKIKeyUpdateReqObject`
Certificate revocation request	`PKI_REVOKE_REQ_OBJ`	`C_CreatePKIRevokeReqObject`

In this example, you create a certificate-request object, `PKI_CERT_REQ_OBJ`. To create a certificate-request object, call the `C_CreatePKICertReqObject` function.

```
PKI_CERT_REQ_OBJ certReqObj = (PKI_CERT_REQ_OBJ)NULL_PTR;

status = C_CreatePKICertReqObject (ctx, &certReqObj);
if (status != 0)
  goto CLEANUP;
```

Step 3b: Set the PKI request object

What you set in the PKI request object is specific to the type of PKI request object you created in "Step 3a: Create a PKI request object" on page 141. For more information on the functions that set the different PKI request objects, see the appropriate PKI request object section in this chapter. See the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide* for examples that set specific PKI request objects.

Note: At this point, if you are creating a certificate request, you should decide whether you want to do key archival or not. Key archival options are specified in the *controls* field of the PKI_CERT_REQ_OBJ. For more information about how to do key archival, see "CMP Key Archival Request Example" in the *Advanced Developer's Guide* or see the samples/cmp/cmpreq.c sample program.

In this example, you created a certificate-request object. For an initialization or certificate request, a key-update request, or a certificate-revocation request, you must also create and set a certificate-template object. To create a certificate-template object, call the C_CreatePKICertTemplateObject function.

```
PKI_CERT_TEMPLATE_OBJ certTemplateObj = (PKI_CERT_TEMPLATE_OBJ)NULL_PTR;

status = C_CreatePKICertTemplateObject (ctx, &certTemplateObj);
if (status != 0)
  goto CLEANUP;
```

You must also set the fields of the PKI_CERT_TEMPLATE_OBJ. To set the PKI_CERT_TEMPLATE_OBJ, see "Certificate-Template Object" on page 162 for a list of the C_Set* functions.

Alternatively, you could transfer certificate information from a CERT_OBJ to a PKI_CERT_TEMPLATE_OBJ. To transfer the certificate information, you call the C_GetPKICertTemplateFromCertObject function.

When you have created and set the PKI_CERT_TEMPLATE_OBJ object, you call the C_SetPKICertReqCertTemplate function to set the PKI_CERT_REQ_OBJ with the PKI_CERT_TEMPLATE_OBJ object.

```
status = C_SetPKICertReqCertTemplate (certReqObj, certTemplateObj);
if (status != 0)
  goto CLEANUP;
```

Step 4: Add PKI request object to the PKI message object

Call the C_AddPKIMsg function to add the PKI request information to the PKI_MSG_OBJ. In this example, you add the PKI_CERT_REQ_OBJ to the PKI_MSG_OBJ.

You must add the PKI request object to the PKI message object before you generate or supply the POP value. When you call C_AddPKIMsg, the function returns the index in the PKI_MSG_OBJ for the added request. You need this index to call the C_GeneratePKIMsgProofOfPossession function.

```
unsigned int reqIndex;

status = C_AddPKIMsg (pkiMsgObj, (POINTER)certReqObject, &reqIndex);
if (status != 0)
  goto CLEANUP;
```

You can use C_GetPKIMsgCount and C_GetPKIMsg to access the request in the PKI message object, and make modifications to the request object.

Step 5 (optional): Generate proof-of-possession

If you create a certificate request or a key update request, you can call the C_GeneratePKIMsgProofOfPossession function to generate a POP value. This value may be required by some CAs.

A POP is a calculated value that can prove the entity that created the request has control over the private key that corresponds to the public key for which a certificate is requested.

You must also supply a PKI_POP_GEN_INFO structure that is populated with the appropriate values to indicate the desired method for POP.

For more information about the C_GeneratePKIMsgProofOfPossession function and the PKI_POP_GEN_INFO structure, see the *API Reference*.

```
status = C_GeneratePKIMsgProofOfPossession (ctx, CMP_PROVIDER_NAME,
                                            pkiMsgObj, reqIndex,
                                            privateKey,
                                            &popGenInfo);
if (status != 0)
  goto CLEANUP;
```

Step 6: Send the PKI message

Before you send the PKI request message, you need to prepare to receive the response message. You need to create another PKI_MSG_OBJ, using the C_CreatePKIMsgObj function.

```
PKI_MSG_OBJ pkiMsgRespObj = (PKI_MSG_OBJ)NULL_PTR;

status = C_CreatePKIMsgObject (ctx, &pkiMsgRespObj);
if (status != 0)
  goto CLEANUP;
```

You also need to create a database SERVICE handle. The C_RequestPKIMsg function uses this SERVICE handle to place any certificates, CRLs, or keys, which are returned, into a database. For information about how to create a database SERVICE handle, see "Binding a Service" on page 80. In "Step 2c: Set the message protection type" on page 140, you already set a PKI_PROTECT_INFO *protectInfo* with the appropriate message-protection information.

```
int C_RequestPKIMsg (
  CERTC_CTX         ctx,           /* (in)  Cert-C context                */
  char              *pki,          /* (in)  PKI protocol handler name     */
  PKI_MSG_OBJ       pkiRequest,    /* (in)  Request object                */
  PKI_PROTECT_INFO  *pProtectInfo, /* (in)  Protection/integrity info     */
  SERVICE           db,            /* (out) Where to put keys, certs, crls */
  PKI_MSG_OBJ       response);     /* (out) Response message object       */
```

You can now call the C_RequestPKIMsg function to send the PKI message. The C_RequestPKIMsg function generates the request's DER, sends the DER-encoded request to the CA, decodes the DER-encoded response from the CA, and populates the PKI response-message object, which was created by the call to C_CreatePKIMsgObject. For more information about the C_RequestPKIMsg function, see the *API Reference*.

```
status = C_RequestPKIMsg (ctx, CMP_PROVIDER_NAME, pkiMsgObj, &protectInfo,
                          db, pkiMsgRespObj);
```

Step 7: Process the PKI message response

Call the C_GetPKIMsgType function to determine which type of PKI response message was returned. You received either a valid PKI response message type or an error

message. The following are the possible types of responses:

- PKI_MSGTYPE_CERT_RESP
- PKI_MSGTYPE_REVOKE_RESP
- PKI_MSGTYPE_KEY_UPDATE_RESP
- PKI_MSGTYPE_CERT_CONF_REQ
- PKI_MSGTYPE_ERROR_MSG

In this example, you call the C_GetPKIMsgType function to determine if you received a PKI_MSGTYPE_CERT_RESP or a PKI_MSGTYPE_ERROR_MSG.

```
unsigned int msgType;

status = C_GetPKIMsgType (pkiMsgRespObj, &msgType);
if (status != 0)
  goto CLEANUP;
```

If you receive a valid PKI response-message type, go to "Step 7a: Process a valid PKI response" on page 145. If you receive a PKI error message go to "Step 7b: Process a PKI error message" on page 147.

Step 7a: Process a valid PKI response

Call the C_GetPKIMsgCount function on the PKI_MSG_OBJ to determine how many response messages need to be processed. For each response message, call the C_GetPKIMsg function to extract the PKI response object.

In this example, you call C_GetPKIMsg to extract a PKI_MSGTYPE_CERT_RESP object.

```
unsigned int i, msgCount, certReqStatus;
PKI_CERT_RESP_OBJ certRespObj;
PKI_STATUS_INFO_OBJ certRespStatusObj;

status = C_GetPKIMsgCount (pkiMsgRespObj, &msgCount);
if (status != 0)
  goto CLEANUP;

for (i = 0; i < msgCount; i++) {
  status = C_GetPKIMsg (pkiMsgRespObj, (POINTER *)&certRespObj, i);
  if (status != 0)
    goto CLEANUP;
```

Call the C_GetPKICertRespStatus or C_GetPKIRevokeRespStatus function to extract the

PKI_STATUS_INFO_OBJ from the response message. This object belongs to Cert-C; therefore, you must not destroy this object.

```
status = C_GetPKICertRespStatus (pkiMsgRespObj, &certRespStatusObj);
if (status != 0)
  goto CLEANUP;
```

Call the C_GetPKIStatus function to get the PKI status information from a PKI_STATUS_INFO_OBJ. You do this to see if the request message was successful.

For more functions that can extract information from the PKI_STATUS_INFO_OBJ, see the "PKI Status-Information Object" on page 165, or the *API Reference*.

```
status = C_GetPKIStatus ((POINTER)certRespStatusObj, &certReqStatus);
if (status != 0)
  goto CLEANUP;
```

After you call the C_GetPKIStatus function, you need to examine the request status to see if the request was granted or not. In this example, you examine *certReqStatus* to see if the request was granted or not. The following is a list of the Cert-C supported request status options.

- PKI_STATUS_GRANTED
- PKI_STATUS_GRANTED_MODS
- PKI_STATUS_REJECTED
- PKI_STATUS_WAITING
- PKI_STATUS_WARNING_REVOCATION
- PKI_STATUS_REVOCATION
- PKI_STATUS_WARNING_KEY_UPDATE

If the request status is PKI_STATUS_REJECTED, call C_GetPKIFailInfo on the response's PKI_STATUS_INFO_OBJ object for more information.

For a certificate request, if the request status is PKI_STATUS_GRANTED or PKI_STATUS_GRANTED_MODS, call C_GetPKICertRespRequestedCert to get a pointer to the CERT_OBJ that contains the newly-issued certificate.

```
CERT_OBJ newCert;

status = C_GetPKICertRespRequestedCert (certRespObj, &newCert);
if (status != 0)
  goto CLEANUP;
```

For a revocation request, if the request status is PKI_STATUS_GRANTED or PKI_STATUS_GRANTED_MODS, call C_GetPKIRevokeRespCRLs to output the list of CRLs returned by the CA or call C_GetPKIRevokeRespCertID to output the list of revoked certificates. Both functions output in the form of a PKI_CERT_IDENTIFIER structure.

For a CMP protocol certificate response or a CMP protocol key update response, if the certificate is acceptable, then you need to create and send a certificate confirmation request message. For more information about the confirmation request and response objects, see "PKI Certificate-Confirmation Request Object Functions" on page 152 and "PKI Certificate-Confirmation Response Object Functions" on page 154. Or, for more information about how to create a confirmation request and process a confirmation response, see "Step 7b: Create a PKI Message for a confirmation message" and "Step 7f: Process the confirmation response" of the "CMP Certificate Request Example," in the *Advanced Developer's Guide*. A certificate revocation response message does not require a confirmation request.

Step 7b: Process a PKI error message

Call the C_GetPKIMsgCount function on the PKI_MSG_OBJ to find out how many error messages were returned. For each error message, call the C_GetPKIMsg function to extract the PKI_ERROR_MSG_OBJs. You can now call C_GetPKIStatus, C_GetPKIFailInfo, C_GetPKIStatusString, or C_GetPKIFailInfoAux on each PKI_ERROR_MSG_OBJ to retrieve more information about the errors.

For more information about these functions, which handle a PKI response error, see the *API Reference*.

Step 8: Destroy objects and clean up

Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing. That is why you should always initialize all objects to NULL_PTR and call the C_Destroy* function later. If there is an error before creating an object, then the C_Destroy* function does not do any damage. Next you

call C_FinalizeCertC to free allocated memory and zeroize sensitive data.

```
C_DestroyPKICertTemplateObject (&certTemplateObj);
C_DestroyPKICertConfReqObject (&certConfReqObj);
C_DestroyPKICertReqObject (&certReqObj);
C_DestroyPKIMsgObject (&pkiMsgConfRespObj);
C_DestroyPKIMsgObject (&pkiMsgConfReqObj);
C_DestroyPKIMsgObject (&pkiMsgRespObj);
C_DestroyPKIMsgObject (&pkiMsgObj);
C_FinalizeCertC (&ctx);
```

PKI Certificate-Request Object

Use the PKI_CERT_REQ_OBJ object to send an initialized request or certificate request to the CA or RA to request a certificate.

You can create an initialization-request or a certificate-request message using the Cert-C PKI message object, PKI_MSG_OBJ, and the PKI certificate-request object, PKI_CERT_REQ_OBJ. You encapsulate the PKI certificate request in the PKI message object. Later, you use the PKI_MSG_OBJ and PKI_CERT_RESP_OBJ object for the certificate-response message.

For general information about how to create a PKI request message, see "Creating a PKI Request Message" on page 137. For an example of a PKI certificate request, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Certificate-Request Object Functions

You must use a Cert-C function to view or modify information in a PKI_CERT_REQ_OBJ object. You cannot assume that the PKI_CERT_REQ_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-request object are listed in the following tables.

Create or Destroy PKI_CERT_REQ_OBJ Functions

Function	Description
C_CreatePKICertReqObject	Creates and initializes a PKI certificate-request object.
C_DestroyPKICertReqObject	Destroys the PKI certificate-request object and frees any memory associated with it.

Set or Modify PKI_CERT_REQ_OBJ Functions

Function	Description
C_SetPKICertReqCertTemplate	Sets the certificate-template object.
C_SetPKICertReqControls	Sets the controls which are attributes affecting certificate issuance.

Function	Description
C_SetPKICertReqID	Sets the certificate-request ID to match request and response.
C_SetPKICertReqPOPType	Sets the POP type.
C_SetPKICertReqRegInfo	Sets *regInfo*, the supplementary information.

Get PKI_CERT_REQ_OBJ Functions

Function	Description
C_GetPKICertReqCertTemplate	Gets the certificate-template object.
C_GetPKICertReqControls	Gets the value of controls that are attributes affecting certificate issuance.
C_GetPKICertReqID	Gets the value of certificate-request ID to match request and response.
C_GetPKICertReqPOPType	Gets the value of POP type.
C_GetPKICertReqRegInfo	Gets the supplementary information *regInfo*.

PKI Certificate-Response Object

Use the PKI_CERT_RESP_OBJ object to parse the initialization response or certification response received from the certificate server.

For an example that demonstrates a PKI certificate response, see the *Advanced Developer's Guide*.

PKI Certificate-Response Object Functions

You must use a Cert-C function to view or modify information in a PKI_CERT_RESP_OBJ object. You cannot assume that the PKI_CERT_RESP_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate request object are listed in the following tables.

Create or Destroy PKI_CERT_RESP_OBJ Functions

Function	Description
C_CreatePKICertRespObject	Creates and initializes a PKI certificate-response object.
C_DestroyPKICertRespObject	Destroys the PKI certificate-response object and frees any memory associated with it.

Set or Modify PKI_CERT_RESP_OBJ Functions

These functions are usually called by the service provider as it parses the incoming PKI message.

Function	Description
C_SetPKICertRespCACerts	Sets the list of CA CERT_OBJs.
C_SetPKICertRespCertReqID	Sets the ID to match request and response.
C_SetPKICertRespRegInfo	Sets *regInfo*, the supplementary information.
C_SetPKICertRespRequestedCert	Sets the requested certificate.
C_SetPKICertRespRequestedPrivateKey	Sets the requested private key.
C_SetPKICertRespStatus	Sets the certification status for the response.

Get PKI_CERT_RESP_OBJ Functions

These functions are usually called by your application to extract information populated by the service provider.

Function	Description
C_GetPKICertRespCACerts	Gets the list of CA CERT_OBJs.
C_GetPKICertRespCertReqID	Gets the ID to match request and response.
C_GetPKICertRespRegInfo	Gets *regInfo*, the supplementary information.
C_GetPKICertRespRequestedCert	Gets the requested certificate.
C_GetPKICertRespRequestedPrivateKey	Gets the requested private key.
C_GetPKICertRespStatus	Gets the certification status.

PKI Certificate-Confirmation Request Object

Use the PKI_CERT_CONF_REQ_OBJ object to send a confirmation to a CA or RA to accept or reject a certificate.

For an example that demonstrates a PKI certificate confirmation-request object, see "Step 7b: Create a PKI Message for a confirmation message" of the "CMP Certificate Request Example," in the *Advanced Developer's Guide*.

PKI Certificate-Confirmation Request Object Functions

You must use a Cert-C function to view or modify information in a PKI_CERT_CONF_REQ_OBJ object. You cannot assume that the PKI_CERT_CONF_REQ_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-confirmation request object are listed in the following tables.

Create or Destroy PKI_CERT_CONF_REQ_OBJ Functions

Function	Description
C_CreatePKICertConfReqObject	Creates a certificate-confirmation request object.
C_DestroyPKICertConfReqObject	Destroys the PKI certificate-confirmation request object and frees any memory associated with it.

Set or Modify PKI_CERT_CONF_REQ_OBJ Functions

Function	Description
C_SetPKICertConfReqCert	Sets the certificate to confirm in a certificate-confirmation request object.
C_SetPKICertConfReqCertReqId	Sets the certificate-request ID of a certificate-confirmation request object.

Function	Description
C_SetPKICertConfReqConfirmStatus	Sets the confirmation status of a certificate-confirmation request object.
C_SetPKICertConfReqStatus	Sets the status information of a certificate-confirmation request object.

Get PKI_CERT_CONF_REQ_OBJ Functions

Function	Description
C_GetPKICertConfReqCert	Gets the certificate needed to confirm from a certificate-confirmation request object.
C_GetPKICertConfReqCertReqId	Gets the certificate-request ID from a certificate-confirmation request object.
C_GetPKICertConfReqConfirmStatus	Gets the confirmation status from a certificate-confirmation request object.
C_GetPKICertConfReqStatus	Gets status info from a certificate-confirmation request object.

PKI Certificate-Confirmation Response Object

Use the PKI_CERT_CONF_RESP_OBJ to parse the confirmation response from the certificate server. For the current Cert-C version, all Cert-C PKI service provider supported certificate-confirmation response messages do not actually contain any information. This is why the certificate-confirmation response object only has create and destroy methods.

For an example that demonstrates a PKI certificate-confirmation response object, see "Step 7f: Process the confirmation response" of the "CMP Certificate Request Example," in the *Advanced Developer's Guide*.

PKI Certificate-Confirmation Response Object Functions

You must use a Cert-C function to view or modify information in a PKI_CERT_CONF_RESP_OBJ object. You cannot assume that the PKI_CERT_CONF_RESP_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-confirmation response object are listed in the following table.

Create or Destroy PKI_CERT_CONF_RESP_OBJ Functions

Function	Description
C_CreatePKICertConfRespObject	Creates a certificate-confirmation response object.
C_DestroyPKICertConfRespObject	Destroys the PKI certificate-confirmation response object and frees any memory associated with it.

PKI Key-Update Request Object

Use the PKI_KEY_UPDATE_REQ_OBJ to send a key-update request for a certificate to the CA/RA.

PKI_KEY_UPDATE_REQ_OBJ is defined as a PKI_CERT_REQ_OBJ object in the pkikumsg.h header file. The C_Get*, C_Set*, or modify APIs that apply to PKI_CERT_REQ_OBJ also apply to PKI_KEY_UPDATE_REQ_OBJ.

For an example that demonstrates a PKI key-update request object, see "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Key-Update Request Object Functions

You must use a Cert-C function to view or modify information in a PKI_KEY_UPDATE_REQ_OBJ object. You cannot assume that the PKI_KEY_UPDATE_REQ_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI key-update request object are listed in the following table.

Create or Destroy PKI_KEY_UPDATE_REQ_OBJ Functions

Function	Description
C_CreatePKIKeyUpdateReqObject	Creates and initializes a PKI key-update request object.
C_DestroyPKIKeyUpdateReqObject	Destroys the PKI key-update request object and frees any memory associated with it.

Get, Set, or Modify PKI_KEY_UPDATE_REQ_OBJ Functions

The C_Get*, C_Set*, or modify APIs that apply to PKI_CERT_REQ_OBJ also apply to PKI_KEY_UPDATE_REQ_OBJ. For a list of these API's, see "PKI Certificate-Request Object Functions" on page 149.

PKI Key-Update Response Object

Use the PKI_KEY_UPDATE_RESP_OBJ object to parse the key-update response received from the certificate server.

PKI_KEY_UPDATE_RESP_OBJ is defined as a PKI_CERT_RESP_OBJ object in the pkikumsg.h header file. The C_Get*, C_Set*, or modify APIs that apply to PKI_CERT_RESP_OBJ also apply to PKI_KEY_UPDATE_RESP_OBJ.

For an example that demonstrates a PKI key-update response object, see the *Advanced Developer's Guide*.

PKI Key-Update Response Object Functions

You must use a Cert-C function to view or modify information in a PKI_KEY_UPDATE_RESP_OBJ object. You cannot assume that the PKI_KEY_UPDATE_RESP_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI key-update response object are listed in the following table.

Create or Destroy PKI_KEY_UPDATE_RESP_OBJ Functions

Function	Description
C_CreatePKIKeyUpdateRespObject	Creates and initializes a PKI key-update response object.
C_DestroyPKIKeyUpdateRespObject	Destroys the PKI key-update response object and frees any memory associated with it.

Get, Set, or Modify PKI_KEY_UPDATE_RESP_OBJ Functions

The C_Get*, C_Set*, or modify APIs that apply to PKI_CERT_RESP_OBJ also apply to PKI_KEY_UPDATE_RESP_OBJ. For a list of these APIs, see "PKI Certificate-Response Object Functions" on page 150.

PKI Revocation Request Object

Use the PKI_REVOKE_REQ_OBJ to send a certificate-revocation request to the CA/RA to revoke one or more certificates.

For an example that demonstrates a PKI certificate-revocation request object, see the "PKI Transaction Samples and Examples" chapter of the *Advanced Developer's Guide*.

PKI Revocation Request Object Functions

You must use a Cert-C function to view or modify information in a PKI_REVOKE_REQ_OBJ object. You cannot assume that the PKI_REVOKE_REQ_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-revocation request object are listed in the following table.

Create or Destroy PKI_REVOKE_REQ_OBJ Functions

Function	Description
C_CreatePKIRevokeReqObject	Creates and initializes a PKI certificate-revocation request object.
C_DestroyPKIRevokeReqObject	Destroys the PKI certificate-revocation request object and frees any memory associated with it.

Set or Modify PKI_REVOKE_REQ_OBJ Functions

Function	Description
C_SetPKIRevokeReqBadSinceDate	Sets the date when the requested certificate is invalid to the PKI certificate-revocation request object.
C_SetPKIRevokeReqExtensions	Sets the value of the CRL entry extensions in the PKI certificate-revocation request object.
C_SetPKIRevokeReqRevocationReason	Sets the revocation reason in the PKI certificate-revocation request object.
C_SetPKIRevokeReqRevokeCert	Sets the details of the certificate to be revoked in the PKI certificate-revocation request object.

Get PKI_REVOKE_REQ_OBJ Functions

Function	Description
C_GetPKIRevokeReqBadSinceDate	Gets the date when the requested certificate is invalid from the PKI certificate-revocation request object.
C_GetPKIRevokeReqExtensions	Gets the *crlEntry* extensions from the PKI certificate-revocation request object. This is a read-only value.
C_GetPKIRevokeReqRevocationReason	Gets the value of the revocation reason in the PKI certificate-revocation request object.
C_GetPKIRevokeReqRevokeCert	Gets the information related to the certificate that is being revoked from the PKI certificate-revocation request object into a certificate object.

PKI Revocation Response Object

Use the PKI_REVOKE_RESP_OBJ object to parse the certificate-revocation response received from the certificate server.

For an example that demonstrates a PKI certificate-revocation response object, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Revocation Response Object Functions

You must use a Cert-C function to view or modify information in a PKI_REVOKE_RESP_OBJ object. You cannot assume that the PKI_REVOKE_RESP_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-revocation response object are listed in the following table.

Create or Destroy PKI_REVOKE_RESP_OBJ Functions

Function	Description
C_CreatePKIRevokeRespObject	Creates and initializes the PKI certificate-revocation response object.
C_DestroyPKIRevokeRespObject	Destroys the PKI certificate-revocation response object and frees any memory associated with it.

Set or Modify PKI_REVOKE_RESP_OBJ Functions

Function	Description
C_SetPKIRevokeRespCertID	Sets the values of the PKI_CERT_IDENTIFIER structure in the PKI certificate-revocation response object.
C_SetPKIRevokeRespCRLs	Sets the value of the list of CRLs into the PKI certificate-revocation response object.
C_SetPKIRevokeRespStatus	Sets the value of the PKI_STATUS_INFO_OBJ in the PKI certificate-revocation response object.

Get PKI_REVOKE_RESP_OBJ Functions

Function	Description
C_GetPKIRevokeRespCertID	Gets the value of the PKI_CERT_IDENTIFIER structure field(s) in the PKI certificate-revocation response object.
C_GetPKIRevokeRespCRLs	Gets the LIST_OBJ of *crls* from the PKI certificate-revocation response object.
C_GetPKIRevokeRespStatus	Gets the value of the PKI_STATUS_INFO_OBJ object from the PKI certificate-revocation response object.

PKI Error-Message Object

Use the PKI_ERROR_MESSAGE_OBJ to convey error information for a PKI message.

For an example that demonstrates a PKI error message object, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Error-Message Object Functions

You must use a Cert-C function to view or modify information in a PKI_ERROR_MESSAGE_OBJ object. You cannot assume that the PKI_ERROR_MESSAGE_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI error-message object are listed in the following tables.

Create or Destroy PKI_ERROR_MESSAGE_OBJ Functions

Function	Description
C_CreatePKIErrorMsgObject	Creates and initializes a PKI error-message object.
C_DestroyPKIErrorMsgObject	Destroys the PKI error-message object and frees any memory associated with it.

Set or Modify PKI_ERROR_MESSAGE_OBJ Functions

Function	Description
C_SetPKIFailInfo	Sets additional information about failure cases in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_SetPKIFailInfoAux	Sets the PKI service-provider-specific failure code in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_SetPKIFailInfoAuxString	Sets a list of service-provider-specific failure strings in a PKI_ERROR_MSG_OBJ.
C_SetPKIStatus	Sets the overall PKI status in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_SetPKIStatusString	Sets a list of NUL-terminated text strings, which represent the status value, in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ. This text is displayed to a user.

Get PKI_ERROR_MESSAGE_OBJ Function

Function	Description
C_GetPKIFailInfo	Gets additional information about failure cases from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIFailInfoAux	Gets the PKI service-provider-specific failure code from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIFailInfoAuxString	Gets a list of service-provider-specific failure strings from a PKI_ERROR_MSG_OBJ.
C_GetPKIStatus	Gets the overall PKI status from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIStatusString	Gets a list of NUL-terminated text strings, which represent the status value, from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ. This text is displayed to a user.

Certificate-Template Object

Use the PKI_CERT_TEMPLATE_OBJ to represent the template that specifies the information that goes into a certificate in the certificate request process. It is different from the CERT_OBJ; all the fields in the PKI_CERT_TEMPLATE_OBJ object are optional.

The PKI_MSG_OBJ does not directly encapsulate the PKI_CERT_TEMPLATE_OBJ object. PKI_CERT_TEMPLATE_OBJ can be encapsulated by either the PKI_CERT_REQ_OBJ or PKI_KEY_UPDATE_REQ_OBJ object, which in turn is encapsulated by the PKI_MSG_OBJ object.

For an example that demonstrates a PKI certificate-template object, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Certificate-Template Object Functions

You must use a Cert-C function to view or modify information in a PKI_CERT_TEMPLATE_OBJ object. You cannot assume that the PKI_CERT_TEMPLATE_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI certificate-template object are listed in the following tables.

Create or Destroy PKI_CERT_TEMPLATE_OBJ Functions

Function	Description
C_CreatePKICertTemplateObject	Creates and initializes a PKI certificate-template object.
C_DestroyPKICertTemplateObject	Destroys the PKI certificate-template object and frees any memory associated with it.

Set or Modify PKI_CERT_TEMPLATE_OBJ Functions

Function	Description
C_SetCertTemplateExtensions	Sets the certificate extensions object that contains X.509 v3 extensions for the certificate.
C_SetCertTemplateIssuerName	Sets the issuer name that contains the name of the issuer that signed the certificate.
C_SetCertTemplateIssuerUniqueID	Sets the issuer unique ID that contains the certificate's issuer's unique identifier.
C_SetCertTemplatePublicKey	Sets the certificate's DER-encoded public key.
C_SetCertTemplateSerialNumber	Sets the certificate's serial number.
C_SetCertTemplateSignatureAlgorithm	Sets the certificate's signature algorithm.
C_SetCertTemplateSubjectName	Sets the certificate's subject name.
C_SetCertTemplateSubjectUniqueID	Sets the subject unique ID that contains the certificate subject's unique identification.
C_SetCertTemplateValidityEnd	Sets the validity end time of the certificate.
C_SetCertTemplateValidityStart	Sets the validity start time of the certificate.
C_SetCertTemplateVersion	Sets the certificate's version number.

Get PKI_CERT_TEMPLATE_OBJ Functions

Function	Description
C_GetCertTemplateExtensions	Gets the certificate's extensions object that contains X.509 v3 extensions for the certificate.
C_GetCertTemplateIssuerName	Gets the issuer name that contains the name of the issuer that signed the certificate.
C_GetCertTemplateIssuerUniqueID	Gets the issuer unique ID that contains the certificate's issuer's unique identifier.
C_GetCertTemplatePublicKey	Gets the certificate's DER-encoded public key.
C_GetCertTemplateSerialNumber	Gets the certificate's serial number.

Function	Description
C_GetCertTemplateSignatureAlgorithm	Gets the signature algorithm that indicates the algorithm used to create the certificate signature.
C_GetCertTemplateSubjectName	Gets the certificate's subject name.
C_GetCertTemplateSubjectUniqueID	Gets the subject unique ID that contains the certificate subject's unique identification.
C_GetCertTemplateValidityEnd	Gets the validity end time of the certificate.
C_GetCertTemplateValidityStart	Gets the validity start time of the certificate.
C_GetCertTemplateVersion	Gets the certificate's version number.
C_GetPKICertTemplateFromCertObject	Gets information from a certificate object to populate a certificate-template object.

PKI Status-Information Object

Use the PKI_STATUS_INFO_OBJ to represent provider-specific status and failure information.

The PKI_MSG_OBJ does not directly encapsulate the PKI_STATUS_INFO_OBJ object. PKI_STATUS_INFO_OBJ can be encapsulated by the PKI_CERT_RESP_OBJ, PKI_KEY_UPDATE_RESP_OBJ, PKI_REVOKE_RESP_OBJ, or PKI_CERT_CONF_REQ_OBJ objects, which in turn is encapsulated by the PKI_MSG_OBJ object.

For an example that demonstrates a PKI status-information object, see the "PKI Transaction Samples and Examples" chapter in the *Advanced Developer's Guide*.

PKI Status-Information Object Functions

You must use a Cert-C function to view or modify information in a PKI_STATUS_INFO_OBJ object. You cannot assume that the PKI_STATUS_INFO_OBJ object points to any specific information. Some examples of the functions that Cert-C provides to work with a PKI status-information object are listed in the following tables.

Create or Destroy PKI_STATUS_INFO_OBJ Functions

Function	Description
C_CreatePKIStatusInfoObject	Creates and initializes a PKI status-information object.
C_DestroyPKIStatusInfoObject	Destroys the PKI status-information object and frees any memory associated with it.

Set or Modify PKI_STATUS_INFO_OBJ Functions

The following C_Set* APIs are also used with the PKI error object.

Function	Description
C_SetPKIFailInfo	Sets additional information about failure cases in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_SetPKIFailInfoAux	Sets the PKI service-provider-specific failure code in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.

Function	Description
C_SetPKIStatus	Sets the overall PKI status in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_SetPKIStatusString	Sets a list of NUL-terminated text strings, which represent the status value, in a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ. This text is displayed to a user.

Get PKI_STATUS_INFO_OBJ Function

The following C_Get* APIs are also used with the PKI error object.

Function	Description
C_GetPKIFailInfo	Gets additional information about failure cases from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIFailInfoAux	Gets the PKI service-provider-specific failure code from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIStatus	Gets the overall PKI status from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ.
C_GetPKIStatusString	Gets a list of NUL-terminated text strings, which represent the status value, from a PKI_STATUS_INFO_OBJ or a PKI_ERROR_MSG_OBJ. This text is displayed to a user.

Chapter 10

Creating an X.509 Certificate

Cert-C represents X.509 certificate information with a `CERT_OBJ` object. You use this object to store and pass information about a particular certificate. This chapter presents the `CERT_OBJ` object and its related APIs. Also included are several examples that demonstrate how you can use the `CERT_OBJ` and its APIs.

Certificate Object

Cert-C represents certificate information with a CERT_OBJ object. Use the CERT_OBJ object to store and pass information about a particular certificate.

A certificate's version can be CERT_VERSION_1, CERT_VERSION_2, or CERT_VERSION_3.

- If a certificate's version is CERT_VERSION_2, then it can contain an *issuerUniqueID* and a *subjectUniqueID*.
- If a certificate's version is CERT_VERSION_3, then it can also include an *extensionsObject*, which represents X.509 v3 certificate extensions.

Certificate-Object Functions

You must use a Cert-C function to view or modify information in a CERT_OBJ object. You cannot assume that the CERT_OBJ points to any specific information. Some examples of the functions that Cert-C provides to manipulate certificates and check certificate signatures are listed in the following table.

Create, Use, or Destroy CERT_OBJ Functions

Function	Description
C_BuildCertPath	Constructs a path from a given certificate to a trusted certificate or CRL.
C_CreateCertObject	Creates a certificate object.
C_DestroyCertObject	Destroys a certificate object, freeing the memory that the certificate object occupied.
C_SignCert	Signs a certificate object.
C_VerifyCertSignature	Verifies a CERT_OBJ's signature.

Set CERT_OBJ Functions

Function	Description
C_SetCertBER	Sets the BER encoding of a certificate object.
C_SetCertFields	Sets a certificate object to the values provided in a CERT_FIELDS structure.
C_SetCertInnerBER	Sets the BER encoding of the inner portion of a certificate object.

Get CERT_OBJ Functions

Function	Description
C_GetCertDER	Gets the DER encoding of a certificate object.
C_GetCertFields	Gets the content of the CERT_FIELDS structure in the certificate object.
C_GetCertInnerDER	Gets the DER encoding of the inner portion of a certificate object.

Creating a Certificate Object

This example shows you how to create a CERT_OBJ object and set it with a binary BER encoding of an X.509 certificate. You can also set a CERT_OBJ object with the data from a CERT_FIELDS structure.

Step 1: Create a CERT_OBJ

Call the C_CreateCertObject function to create a CERT_OBJ and to allocate the needed memory. For more information about the C_CreateCertObject function, see the *API Reference*.

```
int C_CreateCertObject (
  CERT_OBJ   *certObj,                          /* (out) Certificate object */
  CERTC_CTX  ctx);                                   /* (in) Cert-C context */
```

The C_CreateCertObject function takes a CERTC_CTX as a parameter. When you create a CERT_OBJ object, it contains a reference to the given CERTC_CTX. The Cert-C context is

needed when you use this certificate object with a service that requires a CERTC_CTX. For example, a CERTC_CTX is needed for an initialized pseudorandom number generator, a surrender context, or an extension handler.

In this example, assume that you have an initialized CERTC_CTX, *ctx*.

```
CERT_OBJ certObj = (CERT_OBJ)NULL_PTR;

status = C_CreateCertObject (&certObj, ctx);
if (status != 0)
  goto CLEANUP;
```

Step 2: Set the certificate information

There are two ways to set information in a CERT_OBJ object. If you have a binary BER-encoded X.509 Certificate, see step 2a. Alternatively, you can supply the information in a CERT_FIELDS structure, see step 2b.

If you have Base64-encoded data that you wish to convert to binary, see the samples/b64/b64.c sample program.

Step 2a: Set the CERT_OBJ with the BER-encoded X.509 certificate information

Set the CERT_OBJ object with the BER-encoded X.509 certificate information, using the C_SetCertBER function. For more information about the C_SetCertBER function, see the *API Reference*.

```
int C_SetCertBER (
  CERT_OBJ       certObj,                  /* (in/out) Certificate object */
  unsigned char  *ber,                        /* BER-encoded certificate */
  unsigned int   berLen             /* Length of BER-encoded certificate */
);
```

Call the C_SetCertBER function and pass it an ITEM, *certBER*, that contains the certificate binary.

```
status = C_SetCertBER (certObj, certBER.data, certBER.len);
if (status != 0)
  goto CLEANUP;
```

At this point, *certObj* contains the X.509 Certificate information.

Step 2b: Set the CERT_OBJ with the CERT_FIELDS Information

First, you need to prepare a CERT_FIELDS structure. For more information about the CERT_FIELDS structure, see the *API Reference*.

```
typedef struct CERT_FIELDS {
  UINT2       version;
  ITEM        serialNumber;
  int         signatureAlgorithm;
  NAME_OBJ    issuerName;
  struct {
    UINT4 start;
    UINT4 end;
  } validity;
  NAME_OBJ    subjectName;
  ITEM        publicKey;
  BIT_STRING  issuerUniqueID;      /* v2 and v3 only. Set the data field */
                                    /* to NULL_PTR, len to 0 if omittied */
  BIT_STRING  subjectUniqueID;     /* v2 and v3 only. Set the data field */
                                      /* to NULL_PTR, len to 0 if omitted */
  EXTENSIONS_OBJ certExtensions;                        /* v3 only. Set to */
                                              /* (EXTENSIONS_OBJ)NULL_PTR */
                                                          /*  if omitted */
  POINTER reserved;                        /* Reserves for future expansion */
} CERT_FIELDS;
```

You need to create a NAME_OBJ object for the issuer name and another one for the subject name. For more information on creating a NAME_OBJ, see "Creating a Name Object" on page 105.

You also need to create an EXTENSIONS_OBJ object to contain any certificate extensions. For more information on creating an EXTENSIONS_OBJ, see "Creating an Extensions Object" on page 257.

Instead of creating the NAME_OBJ and EXTENSIONS_OBJ, you can call the C_GetCertFields function on the blank CERT_OBJ, which you created in step 1. The C_GetCertFields function retrieves the already created NAME_OBJ objects and EXTENSIONS_OBJ object.

Populate the remaining CERT_FIELDS fields with the desired values. For more information about the remaining CERT_FIELDS fields, see the *API Reference*.

Call the C_SetCertFields function to set a copy of the CERT_FIELDS information into

the CERT_OBJ object. For more information about the C_SetCertFields function, see the *API Reference*.

```
int C_SetCertFields (
  CERT_OBJ     certObj,                    /* (in/out) Certificate object */
  CERT_FIELDS  *certFields                          /* Certificate fields */
);
```

After calling this function, the value in *certFields* becomes the actual value of *certObj*.

```
CERT_FIELDS certFields;

status = C_SetCertFields (certObj, *certFields);
if (status != 0)
  goto CLEANUP;
```

You now sign the certificate with the issuer's private key, using the C_SignCert function. For more information about the C_SignCert function, see the *API Reference*.

```
int C_SignCert (
  CERT_OBJ   certObj,                       /* (mod) Certificate object */
  B_KEY_OBJ  privateKey                                /* Signing key */
  . . .
);
```

The B_KEY_OBJ can be a key object that points to a private key on a hardware device. For more information about the B_KEY_OBJ object, see Appendix A. If you are going to use a hardware device, make sure you use the CERTC_CTX associated with the necessary service providers, when you call C_CreateCertObject to create the CERT_OBJ. For more information, see "Cert-C PKCS #11 Database Service Provider" and "Cert-C CryptoAPI Database Service Provider" in the "Service Provider" section of the *API Reference*. For this example, assume that you already have a B_KEY_OBJ *caPrivateKey* that contains the issuer's private key.

```
status = C_SignCert (certObj, caPrivateKey);
if (status != 0)
  goto CLEANUP;
```

Step 3: Destroy the CERT_OBJ object

You no longer need the CERT_OBJ object, so you should destroy it now. Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing. That is why you should always initialize all objects to NULL_PTR and call the C_Destroy* function later. If there is an error before creating an object, then the C_Destroy* function does not do any damage.

```
CLEANUP:
  C_DestroyCertObject (&certObj);
```

Fulfilling the PKCS #10 Certificate Request

In this example, you assume the role of a CA. In this capacity, you receive the DER-encoding of a certificate request.

To build a certificate you must get the certificate information out of the certificate request and into a certificate. When you receive a certificate request, you must also check whether the requestor has included some extra attributes. Since that information will not be part of the certificate, you need to capture that data separately. An attribute may include v3 extensions. For more information about building extensions, see "Creating an Extensions Object" on page 257.

You create and initialize a certificate object. Next, you create a PKCS #10 object and fill it with the BER-encoded certificate request. You verify the certificate request's signature and extract the `PKCS10_FIELDS` information. Then you fill the `CERT_FIELDS` structure with information from the `PKCS10_FIELDS` structure, and with other information. You set the `CERT_OBJ` with the updated `CERT_FIELDS` information. Next, you take the certificate object you just created and sign it. Then, you get the DER encoding of the certificate, which can be sent to the certificate requestor or stored in a file. Finally, you can destroy the PKCS #10 object.

Step 1: Create a certificate and a PKCS #10 object

You need to create a certificate object and a PKCS #10 object. You have already created an attributes object (see "Creating an Attributes Object" on page 115). For more information about `C_CreateCertObject` and `C_CreatePKCS10Object`, see the *API Reference*.

Step 1a: Create a certificate object

Using the `C_CreateCertObject` function, you declare a variable to be `CERT_OBJ` and pass its address as the argument. In the second argument, you pass Cert-C a previously initialized Cert-C context. A properly cast `NULL_PTR` is not an option for the this argument.

```
int C_CreateCertObject (
  CERT_OBJ   *certObj,                              /* (out) Cert object */
  CERTC_CTX  ctx                                       /* Cert-C context */
);
```

For more information about initializing a Cert-C context, see "Initializing the Cert-C

Context" on page 75.

```
CERT_OBJ newCertificateObj = (CERT_OBJ)NULL_PTR;

status = C_CreateCertObject (&newCertificateObj, ctx);
if (status != 0)
  goto CLEANUP;
```

Step 1b: Create a PKCS #10 object

In "Creating a PKCS #10 Certificate Request" on page 123, as the certificate requestor, you created a DER-encoded PKCS #10 certificate request. Now, as the CA, you create and initialize a `PKCS10_OBJ` with that information. Using the `C_CreatePKCS10Object` function, you create a `PKCS10_OBJ` and pass it the same initialized *`ctx`*. For more information about `C_CreatePKCS10Object`, see the *API Reference*.

```
PKCS10_OBJ pkcs10Obj = (PKCS10_OBJ)NULL_PTR;

status = C_CreatePKCS10Object (ctx, &pkcs10Obj);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the PKCS #10 certificate information

Using the `C_SetPKCS10BER` function, you set the `PKCS10_OBJ` with the BER-encoded PKCS #10 certificate request. For more information about the `C_SetPKCS10BER` function, see the *API Reference*.

```
int C_SetPKCS10BER(
  PKCS10_OBJ     pkcs10Object,                    /* (mod) PKCS#10 object */
  unsigned char  *ber,                /* (in) Input buffer containing BER */
  unsigned int   berLen                          /* (in) Input buffer length */
);
```

The first argument is the `PKCS10_OBJ` you just created. The second argument points to the BER encoding of a PKCS #10 object. In this example, assume you have an `ITEM` *`pkcs10BER`* that contains the BER-encoded PKCS #10 binary.

```
ITEM pkcs10BER;
```

```
status = C_SetPKCS10BER (pkcs10Obj, pkcs10BER.data, pkcs10BER.len);
if (status != 0)
  goto CLEANUP;
```

You now have a PKCS10_OBJ that contains the BER-encoded certificate request.

Step 3: Verify the PKCS #10 signature, set a CERT_FIELDS, and sign the X.509 certificate

In this step, you perform the operations to verify the PKCS #10 signature, retrieve the PKCS10_FIELDS information, set the CERT_FIELDS structure with the PKCS #10 information, and sign the X.509 certificate.

Step 3a: Verify the PKCS #10 signature

You must prove the entity that generated and sent the PKCS #10 certificate request to you actually has access to the private key that corresponds to the public key contained in the certificate request. The certificate request was signed by the requestor using the subject's private key, so you can verify the signature using the public key contained in the PKCS #10 certificate request.

Using the C_VerifyPKCS10Signature function, you verify the signature. For more information about C_VerifyPKCS10Signature, see the *API Reference*.

```
status = C_VerifyPKCS10Signature (pkcs10Obj);
if (status != 0)
  goto CLEANUP;
```

Step 3b: Retrieve the PKCS10_FIELDS information

Next, you initialize a PKCS10_FIELDS structure and get the individual PKCS #10 fields from the PKCS #10 object. Using the C_GetPKCS10Fields function, you get the PKCS #10 information. For more information about C_GetPKCS10Field, see the *API Reference*.

```
PKCS10_FIELDS pkcs10Fields;

status = C_GetPKCS10Fields (pkcs10Obj, &pkcs10Fields);
if (status != 0)
  goto CLEANUP;
```

Step 3c: Fill the certificate object's CERT_FIELDS structure

Now, you need to fill the CERT_OBJ's CERT_FIELDS structure from the PKCS10_FIELDS structure. You also sets the CERT_OBJ's CERT_FIELDS structure with other information. For more information about CERT_FIELDS, see the *API Reference*.

```
typedef struct CERT_FIELDS {
  UINT2           version;
  ITEM            serialNumber;
  int             signatureAlgorithm;
  NAME_OBJ        issuerName;
  struct {
          UINT4  start;
          UINT4  end;
  } validity;
  NAME_OBJ        subjectName;
  ITEM            publicKey;
  BIT_STRING      issuerUniqueID;                /* Version 2 and 3 only */
  BIT_STRING      subjectUniqueID;               /* Version 2 and 3 only */
  EXTENSIONS_OBJ  certExtensions;                      /* Version 3 only */
  POINTER         reserved;                                  /* Reserved */
} CERT_FIELDS;
```

Using the C_GetCertFields function, you first get the blank CERT_FIELDS structure from the new CERT_OBJ, which you just created.

```
CERT_FIELDS newCertInfo;

status = C_GetCertFields (newCertificateObj, &newCertInfo);
if (status != 0)
  goto CLEANUP;
```

You specify *version* to be CERT_VERSION_3 and *signatureAlgorithm* to be SA_SHA1_WITH_RSA_ENCRYPTION. You will use this algorithm later to sign the certificate.

```
newCertInfo.version = CERT_VERSION_3;
newCertInfo.signatureAlgorithm = SA_SHA1_WITH_RSA_ENCRYPTION;
```

You need to specify the new certificate's serial number. As a CA, assume you have a unique serial number available. You specify *serialNumber* to be this unique serial

number.

```
unsigned char newSerialNumber[4] = {
  1, 0, 0, 1
};

newCertInfo.serialNumber.data = newSerialNumber;
newCertInfo.serialNumber.len = sizeof (newSerialNumber);
```

You need to set the issuer name for the new certificate. As a CA, assume you have the BER-encoded issuer name.

The `GetCAInfoFromStorage` routine is not a Cert-C routine. It is a placeholder for a routine that obtains the CA's BER-encoded X.500 Name. You write this routine to best fit your application.

```
ITEM caNameBERFromStorage = {NULL, 0};

status = GetCAInfoFromStorage (&caNameBERFromStorage.data,
                               &caNameBERFromStorage.len);
if (status != 0)
  goto CLEANUP;

status = C_SetNameBER (newCertInfo.issuerName, caNameFromStorage.data,
                       caNameFromStorage.len);
if (status != 0)
  goto CLEANUP;
```

You extract the subject name from the PKCS #10 certificate request and use it for the subject name in the new certificate. Using `C_GetNameDER`, you extract the subject name from *`pkcs10Fields`*, then using `C_SetNameBER`, you set *`newCertInfo`*'s subject name to the subject name extracted from *`pkcs10Fields`*. For more information about `C_GetNameDER` and `C_SetNameBER`, see the *API Reference*.

```
ITEM subjectName = {NULL, 0};

status = C_GetNameDER (pkcs10Fields.subjectName, &subjectName.data,
                       &subjectName.len);
if (status != 0)
  goto CLEANUP;
```

```
status = C_SetNameBER (newCertInfo.subjectName, subjectName.data,
                       subjectName.len);
if (status != 0)
  goto CLEANUP;
```

You set the validity start time to the current time and the expiration time to be a year from now.

```
T_time (&newCertInfo.validity.start);
newCertInfo.validity.end = certFields.validity.start +
  ((3600 * 24 * 365) - 1);
```

Finally, you set the new certificate's *publicKey* field to the public key contained in *pkcs10Fields.publicKey*.

```
newCertInfo.publicKey.data = pkcs10Fields.publicKey.data;
newCertInfo.publicKey.len = pkcs10Fields.publicKey.len;
```

At this stage, you have the option to initialize the *newCertInfo.certExtensions* EXTENSIONS_OBJ. You created this EXTENSIONS_OBJ when you called the C_GetCertFields function. For more information about how to do this, see "Reading Extensions in an Attributes Object" on page 267.

Step 3d: Set CERT_OBJ with the new CERT_FIELDS information

You have just modified the CERT_FIELDS structure from *newCertificateObj*. Now, you must call C_SetCertFields to reconcile the internal state of the CERT_OBJ with the changes made to the CERT_FIELDS structure. The CERT_OBJ is not properly initialized with the new data and cannot be used in any operations, until you do this step.

```
status = C_SetCertFields (newCertificateObj, &newCertInfo);
if (status != 0)
  goto CLEANUP;
```

With these steps completed, you can go ahead and build the certificate.

Step 3e: Sign the X.509 certificate

You will now sign the certificate object. To sign the certificate, you use the C_SignCert

function. For more information about C_SignCert, see the *API Reference*.

```
int C_SignCert (
  CERT_OBJ   certObj,                          /* (mod) Certificate object */
  B_KEY_OBJ  privateKey                                  /* Signing key */
);
```

The B_KEY_OBJ can be a key object that points to a private key on a hardware device. For more information about the B_KEY_OBJ object, see Appendix A. If you are going to use a hardware device, then when you call C_CreateCertObject to create the CERT_OBJ, make sure you use the CERTC_CTX associated with the necessary service providers. For more information, see "Cert-C PKCS #11 Database Service Provider" and "Cert-C CryptoAPI Database Service Provider" in the "Service Provider" section of the *API Reference*. For this example, assume that you already have a B_KEY_OBJ *caPrivateKey* that contains the issuer's private key.

```
status = C_SignCert (newCertificateObj, caPrivateKey);
if (status != 0)
  goto CLEANUP;
```

Step 4: Retrieve the X.509 certificate in DER format

You now have a signed certificate object, not an X.509 certificate. You need to get the certificate information out of the CERT_OBJ and into a format other applications can recognize, such as the DER-encoded format. To get the certificate information out of the CERT_OBJ, you use the C_GetCertDER function.

For more information about C_GetCertDER, see the *API Reference*.

```
int C_GetCertDER (
  CERT_OBJ       certObj,                          /* Certificate object */
  unsigned char  **der,                         /* (out) DER-encoded cert */
  unsigned int   *derLen             /* (out) Length of DER-encoded cert */
);
```

Cert-C returns an address to where you can go to find the certificate DER, not the certificate information itself. You should copy the certificate information into a database or a file.

```
ITEM certDer = {NULL, 0};
```

```
status = C_GetCertDER (newCertificate, &certDer.data, &certDer.len);
if (status != 0)
  goto CLEANUP;
```

The RSA_WriteDataToFile routine is not a Cert-C routine; it is a demo utility routine. For more information about Cert-C demo utilities, see the "Utilities" chapter in the *Advanced Developer's Guide*. You can use RSA_WriteDataToFile to write binary data to a file.

```
status = RSA_WriteDataToFile (certDer.data, certDer.len,
                              "Enter name of file to store cert binary");
if (status != 0)
  goto CLEANUP;
```

You can now send the signed certificate to the certificate requestor.

Step 5: Destroy the PKCS #10 certificate object

You no longer need the PKCS10_OBJ, so you should destroy it now. Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing. That is why you should always initialize all objects to NULL_PTR and call the C_Destroy* function later. If there is an error before creating an object, then the C_Destroy* function does not do any damage.

```
CLEANUP:
  C_DestroyPKCS10Object (&pkcs10Obj);
```

Manipulating Certificate Information

In the previous examples in this chapter, you created a certificate object. In this example, assume that you have received a certificate from someone, from a CA, or perhaps directly from the certificate holder. You want to read the information on the certificate, verify the CA's signature on the certificate, and extract the public key.

The steps `ObtainCAPublicKey` and `C_VerifyCertSignature` can be replaced by one call to `C_BuildCertPath` to validate a certificate chain. For more information, see "Validating a Certificate Path" on page 193. Most applications do not call the `C_VerifyCertSignature` or `C_ValidateCert` functions directly; instead, they call `C_BuildCertPath` to validate a certificate chain.

Step 1: Create a certificate object

In this example, assume you have received a certificate in the BER-encoded format and placed it into a buffer. You have named the buffer *`certBER`*, and it is *`certBERLen`* bytes long.

```
unsigned char *certBER;
unsigned int certBERLen;
```

First, you need to create a certificate object. For more information about `C_CreateCertObject`, see the *API Reference*.

```
int C_CreateCertObject (
  CERT_OBJ   *certObj,                          /* (out) Cert object */
  CERTC_CTX  ctx                                   /* Cert-C context */
);
```

Using the `C_CreateCertObject` function, you declare a variable to be `CERT_OBJ` and pass its address as the argument. In the second argument, you pass Cert-C a previously initialized Cert-C context. A properly cast `NULL_PTR` is not an option for the this argument. For more information about initializing a Cert-C context, see "Initializing the Cert-C Context" on page 75.

```
CERT_OBJ certObject = (CERT_OBJ)NULL_PTR;
B_KEY_OBJ caPublicKeyObject = (B_KEY_OBJ)NULL_PTR;
B_KEY_OBJ certPublicKeyObject = (B_KEY_OBJ)NULL_PTR;
CERT_FIELDS certFields;
```

```
status = C_CreateCertObject (&certObject, ctx);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the certificate information

Using the C_SetCertBER function, you set the CERT_OBJ with the BER-encoded certificate. For more information about the C_SetCertBER function, see the *API Reference*.

```
int C_SetCertBER (
  CERT_OBJ       certObj,                  /* (in/out) Certificate object */
  unsigned char  *ber,                         /* BER-encoded certificate */
  unsigned int   berLen              /* Length of BER-encoded certificate */
);
```

The first argument is the CERT_OBJ you just created. The second argument points to the BER encoding of a certificate object. Assume that you have an ITEM *certBER* that contains the BER-encoded certificate binary.

```
status = C_SetCertBER (certObject, certBER, certBERLen);
if (status != 0)
  goto CLEANUP;
```

You now have a CERT_OBJ that contains the BER-encoded certificate information.

Step 3: Read the certificate information

Next, you want to get the certificate information into a readable format. You initialize a CERT_FIELDS structure and fill it with the individual certificate fields from the certificate object. Using the C_GetCertFields function, you get the certificate fields. For more information about C_GetCertFields and CERT_FIELDS, see the *API Reference*.

```
int C_GetCertFields (
  CERT_OBJ     certObj,                              /* Certificate object */
  CERT_FIELDS  *certFields          /* (out) Certificate fields structure */
);
```

Using the C_GetCertFields function, you give Cert-C a certificate object and the address of a CERT_FIELDS structure. Cert-C places the certificate information at the

address. The memory that the pointer to the certificate information points to belongs to Cert-C. You do not need to allocate or free that memory. Also, you should not attempt to adjust the data yourself. The information remains unchanged until you call a Cert-C routine that modifies or destroys the certificate object. To save this information, you must copy it into a file or your own buffer.

```
CERT_FIELDS certFields;

status = C_GetCertFields (certObject, &certFields);
if (status != 0)
  goto CLEANUP;
```

You can now perform your own routines to extract specific certificate information. The `DisplayCertInfo` and `ObtainCAPublicKey` routines are placeholders for code that you create to extract certificate information.

`DisplayCertInfo` should extract and display the certificate information in a readable format, while `ObtainCAPublicKey` should get the certificate issuer's public key and store it in a Crypto-C key object. For more information about creating a key object, see "Using BSAFE Crypto-C" on page 287. A more comprehensive description on using Crypto-C is available in the *Crypto-C Developer's Guide*. The issuer's public key is then used to verify the CA's signature on the certificate.

```
status = DisplayCertInfo (&certFields);
if (status != 0)
  goto CLEANUP;

status = ObtainCAPublicKey (&certFields, &caPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Step 4 Verify the certificate's signature and extract the public key

In this step, you verify the certificate's signature and extract the certificate's public key; then you can perform an operation with the extracted public key.

Step 4a: Verify the certificate's signature

You must verify that the certificate sent to you does in fact belong to the certificate subject and was issued by a trusted CA. The certificate was signed by the issuer using the issuer's private key. So, using the `C_VerifyCertSignature` function, you can verify the signature by using the issuer's public key. The `ObtainCAPublicKey` routine

in step 3 found the issuer's certificate; it then obtained the issuer's public key and placed it in a Crypto-C key object. You use this public key to verify the CA's signature on the certificate. For more information about `C_VerifyCertSignature`, see the *API Reference*.

Note: The variable parameter format makes this function backward-compatible with BCERT v1.0.

```
int C_VerifyCertSignature (
  CERT_OBJ certObj,                                    /* Certificate object */
  B_KEY_OBJ publicKey                                    /* Verification key */
  . . .
);
```

To verify the CA's signature on the certificate, you call the `C_VerifyCertSignature` function. The first argument is the certificate object, which contains the certificate that was sent to you. The second argument is a Crypto-C key object that contains the issuer's public key.

```
status = C_VerifyCertSignature (certObject, caPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

You have verified the CA's signature on the certificate, so now you can safely use the certificate.

Step 4b: Extract the certificate's public key

Before you can do anything with the certificate, you need to extract the certificate's public key. The `ExtractCertPublicKey` routine is a placeholder for code that you will create to extract the certificate's public key. You write this routine to best fit your application.

In this routine, you build a public-key object from the certificate subject's public key. To do this, you take the BER-encoding of the certificate subject's public key, which is the associated `CERT_FIELDS`'s *`publicKey`* field, and create a key object. For more information about creating a key object, see "Using BSAFE Crypto-C" on page 287. A more comprehensive description on using Crypto-C is in the *Crypto-C Developer's*

Guide.

```
status = ExtractCertPublicKey (&certFields, &certPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Step 4c: Perform an operation using the certificate's public key

Now that you have the certificate subject's public key, you can perform operations that require a public key. For example, you can create a digital envelope or verify another signature. The `UsePublicKey` routine is a place holder for code you create that uses a public key. You write this routine to best fit your application.

```
status = UsePublicKey (certPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the certificate and key objects

Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing. That is why you should always initialize all objects to `NULL_PTR` and call the `C_Destroy*` function later. If there is an error before creating an object, then the `C_Destroy*` function does not do any damage.

```
CLEANUP:
  C_DestroyCertObject (&certObject);
  B_DestroyKeyObject (&certPublicKeyObject);
  B_DestroyKeyObject (&caPublicKeyObject);
```

Chapter 11

Verifying Certificates and CRLs

Cert-C provides APIs to verify that a certificate or CRL is valid. With these APIs you construct a certificate path from the starting object (certificate or CRL), which you want to verify, to a trusted certificate. This trusted certificate can be a trusted-root certificate (self-signed) or it can be a certificate that you explicitly state is to be trusted.

The certificates in the certificate path must chain together by matching the subject-issuer names. If extensions are available, then the certificates must chain together by matching the *subjectKeyIdentifier* and *authorityKeyIdentifier* values.

Once a certificate path is constructed, you can check that each object (certificate or CRL) is valid at the specified validity time; in other words, it has not expired. You might also want to include certificate revocation checking to verify that none of the certificates in the path have been revoked. Cert-C provides APIs to verify if a certificate has been revoked using CRLs or the OCSP standard.

There are APIs to verify name constraints, basic CA constraints, policy and policy-mapping, and that key usage is correct.

Cert-C also provides low-level APIs to verify a signature on an individual certificate or CRL.

Trusted Root

When you receive a certificate, it is already signed by a CA. To verify the certificate's authenticity, you verify the CA's signature. To do that, you need the CA's public key. However you get the CA's public key, you want it certified so that you can be sure the key you use to verify the certificate is genuine.

The best way to verify a public key is with a certificate. So you will want a copy of the CA's certificate. However, you need to check who signed the CA's certificate. If it is another CA, then you check that CA's certificate too. You can stop checking certificates when you get to a trusted root.

The PKI system relies on a trusted authority to certify CAs. The trusted root issues certificates to CAs. You need to obtain this trusted authority's public key outside the usual certificate hierarchy. This public key is the trusted-root key. You get a copy of this key and verify that you indeed have the right copy by checking it against a number of other independent sources. Maybe this trusted-root key is published in a trade journal or you verify by telephone.

Certificate Chaining

Once you have the trusted-root key, you should save it in a protected format, possibly password-based encrypted. You are not hiding the key, you are making sure no one can replace your copy of that key with another.

Now when you want to verify a CA's certificate, you do so by checking the signature on the CA's certificate using the trusted root's public key.

This is known as certificate chaining. You start with a user's certificate, and verify its validity using the CA's public key, which you extracted from the CA's certificate, the validity of which you verified using the trusted root's public key.

In chaining, there may be a number of levels of CAs between the user and the CA.

Figure 11-1 shows an example of a certificate chain. Patrick can trust CA #1's certificate because it was signed by the trusted root. He can then trust CA #2's certificate because it was signed by CA #1. And finally, he can trust Maria's certificate because it was signed by CA #2.

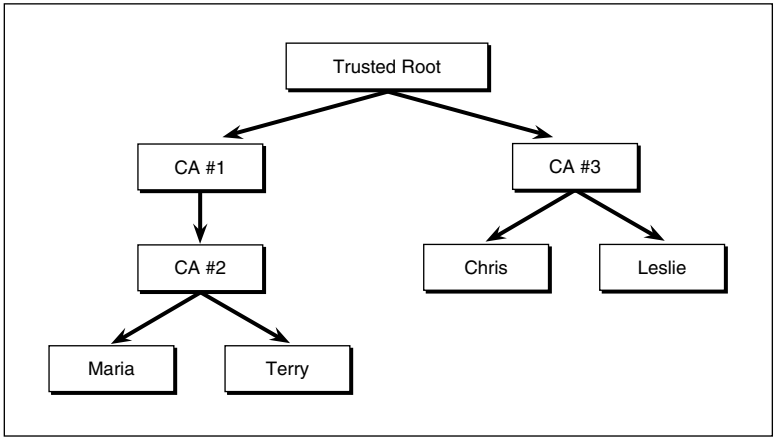


Figure 11-1 **A Certificate Chain**

Verify a Certificate or CRL Functions

Some examples of the functions that Cert-C provides to verify a certificate or CRL are listed in the following tables.

High-Level Functions

These high-level functions are usually used to build a certificate path and check the revocation status of a given certificate.

High-Level Function	Description
C_BuildCertPath	Validates a certificate or CRL by determining whether or not a valid path exists from the object to a certificate in the trusted store. This function calls the C_CheckCertRevocation function to check the revocation status of the certificates in the path. You can set the CERT_PATH_CTX.*pathOptions* field to include the PF_IGNORE_REVOCATION flag; this flag causes path processing not to check the revocation status of each certificate.
C_CheckCertRevocation	Determines the revocation status of a given certificate.

Low-Level Functions

These low-level functions are used to implement service-provider functionality for the high-level functions.

Low-Level Function	Description
C_GetNextCertInPath	Returns a set of certificates (given a certificate or CRL), which could have signed the given certificate or CRL. Usually, your application does not use this API directly. Instead, the application uses C_BuildCertPath.
C_ValidateCert	Validates the information present in a certificate, given a certificate and a public key. You can use this API to implement path processing; however, it does not do any certificate chaining. Usually, your application does not use this API directly. Instead, the application uses C_BuildCertPath.

Low-Level Function	Description
C_VerifyCertSignature	Verifies the signature on a specific certificate, given a certificate and a public key.
C_VerifyCRLSignature	Verifies the signature on a specific CRL, given a CRL and a public key.

You use the C_BuildCertPath and C_CheckCertRevocation APIs most commonly when implementing certificate validation.

Use C_BuildCertPath to build a certificate path to validate either a certificate or a CRL. You can request the appropriate level of checking during path construction. Cert-C supports policy mapping through the PA_PKIX2 flag in the CERT_PATH_CTX.*pathAlgorithm* field.

C_CheckCertRevocation determines the revocation status of the specified certificate. It returns the certificate's revocation status and related information. This function uses either the CRL or the OCSP revocation-status protocol, depending on the certificate revocation status service provider you register when you initialize the Cert-C context.

You can look at the samples/validate/validate.c sample program for an example of how these two APIs are used. There are also other relevant samples in the samples/validate directory.

You almost never call the C_GetNextCertInPath and C_ValidateCert APIs directly in your application. You should only need to call these APIs when you want to construct a routine that performs what the C_BuildCertPath API is defined to do. For an example of how these APIs are implemented, you can look at the BuildCertPath routine in the provider/path/pkix/pkixpath.c provider source code, which is an implementation of the CERT_PATH_FUNCS.*BuildCertPath* callback in the Cert-C Certificate-Path Processing service provider source code. BuildCertPath uses the C_GetNextCertInPath and C_ValidateCert API calls.

The only purpose of C_VerifyCertSignature and C_VerifyCRLSignature is to verify an individual signature. You almost never call these APIs directly in your application.

Service Providers

You need to use a certificate-path processing, a database, and a certificate-revocation status service provider to implement certificate or CRL validation. Cert-C provides these service providers. For more information about the Cert-C service providers, see the *API Reference*.

- The Cert-C Certificate-Path Processing service provider code is called to construct a certificate path. It retrieves certificates and CRLs from a local database or a remote server using the HTTP or LDAP protocol.
- A database `SERVICE` is required in the `CERT_PATH_CTX` for the path-processing code to obtain the needed certificates and CRLs.
- The Cert-C Certificate-Revocation Status service provider code is called to determine a certificate's revocation status.

Validating a Certificate Path

In this example, you build a certificate path to verify a certificate or a CRL. You configure the certificate-path processing context to define how to build and check the certificate path. You also check the revocation status of a certificate.

You register and initialize a database service provider and a certificate-path processing service provider to build a certificate path to verify a certificate or a CRL. You register and initialize a certificate-revocation status service provider to check the revocation status of a particular certificate. In both cases, you create a certificate-path processing context to define the certificate check policy.

Step 1: Register service providers with the Cert-C context

Initialize and register the required service providers with the Cert-C context. You can do this by calling either `C_InitializeCertC` or `C_RegisterService`. In this example, you use the `C_InitializeCertC` function to initialize the Cert-C In-Memory Database, the Cert-C Certificate-Path Processing, and the Cert-C Certificate-Revocation Status service providers. The `SERVICE_HANDLER` structure contains service-provider information and the service-provider initialization function. To see how to initialize and register a service provider, see "Registering a Service Provider After Cert-C Initialization" on page 77.

For more information `C_InitializeCertC` and `SERVICE_HANDLER`, see the *API Reference*.

```
#define SP_COUNT 4
#define CACHE_DB_NAME "Cache IM Database"

CERTC_CTX ctx = NULL;

SERVICE_HANDLER spTable[SP_COUNT] = {;
  {SPT_DATABASE, "Sample IM Database", S_InitializeMemoryDB},
  {SPT_DATABASE, CACHE_DB_NAME, S_InitializeMemoryDB},
  {SPT_CERT_PATH, "Cert Path Processing Provider", S_InitializePKIXPath},
  {SPT_CERT_STATUS, "Cert Revocation Status Provider", S_InitializeCRLStatus}
}
```

```
POINTER spParams[SP_COUNT] = {0};

CRL_STATUS_INIT_PARAMS crlStatParams = {0};

crlStatParams.dbName = CACHE_DB_NAME;

/* This is the parameter for S_InitializeCRLStatus */
spParams[3] = (POINTER)&crlStatParams;

status = C_InitializeCertC (spTable, spParams, SP_COUNT, &ctx);
if (status !=0)
  goto CLEANUP;
```

For more information about the role of the database specified in the CRL_STATUS_INIT_PARAMS.*dbName* field, see the "Cert-C CRL Revocation Status Service Provider" section of the *API Reference*.

Step 2: Prepare CERT_PATH_CTX

Use the CERT_PATH_CTX structure to hold the information necessary for path validation; for example, a set of trusted certificates.

```
typedef struct {
  int       pathAlgorithm;                  /* Path-processing algorithm */
  UINT4     pathOptions;                   /* Modify base path algorithm */
  LIST_OBJ  trustedCerts;                         /* "root" certificates */
  LIST_OBJ  policies;                             /* Acceptable policies */
  UINT4     validationTime;            /* Path must be valid at this time */
  SERVICE   database;                  /* Database(s) for path processing */
} CERT_PATH_CTX;
```

You set the path-processing algorithm, *pathAlgorithm*, according to the standard you want to follow. If you want path processing according to section 6 of *RFC 2459*, set *pathAlgorithm* to PA_PKIX. Or, if you want path processing according to *RFC 3280*, set *pathAlgorithm* to PA_PKIX2. If you set PA_PKIX2 and *pathOptions*.PA_IGNORE_POLICY, then path processing is similar to setting PA_PKIX. In this example, set *pathAlgorithm* to PA_PKIX.

Set the *pathOptions* parameter to 0 (zero). *pathOptions* is a UINT4 value that contains a set of 1-bit flags that you can use to modify the basic certificate-path-processing algorithm. The flags are used to turn off certain checks in certificate-path building or validation. For more information about setting path options, see CERT_PATH_CTX in the

API Reference. If you set *pathAlgorithm*.PA_PKIX2 and *pathOptions*.PA_IGNORE_POLICY, then path processing is similar to setting PA_PKIX.

For this example, assume that you have a LIST_OBJ called *trustedRoots* that contains a collection of one or more CERT_OBJs whose public keys are trusted by the application. These CERT_OBJs represent the trusted certificate store. Valid certification paths end in one of these certificates. See "Creating and Enumerating a List of Objects" on page 94 for an example of how to create a LIST_OBJ.

policies is a LIST_OBJ that contains a set of initial policy identifiers. These identify one or more certificate policies that are acceptable for processing certification paths. Each entry in the list is of type ITEM, where the item value is the OID. If you accept any policy, then set *policies* to (LIST_OBJ)ANY_POLICY. ANY_POLICY is a constant that is defined in certpath.h file.

When you select ANY_POLICY then C_BuildCertPath's *policyInfoList* field returns every unique policy ID (including the anyPolicy OID from the root node when *pathAlgorithm* is PA_PKIX2) found in the tree. When you specify that only certain specific policies are acceptable, by including those policies in CERT_PATH_CTX.*policies*, then *policyInfoList* contains only those policies (and all qualifiers) which are common to both the input list of policies and the developed policy tree. The path is valid for these policies, however, it is possible that *policyInfoList* might not contain the original number of acceptable policies in CERT_PATH_CTX.*policies*.

Set the *validationTime* parameter to 0 (zero) or to PF_VALIDATION_TIME_NOW to indicate that the validation time should be the time at which the certification-path operation is performed.

For this example, assume that you have a SERVICE called *db* that is bound to a database instance. You will obtain certificates and CRLs from this database to construct a certificate path. For an example of how to create a SERVICE handle, see "Using the SERVICE Handle" on page 79.

For more information about CERT_PATH_CTX, see the *API Reference*.

```
CERT_PATH_CTX pathCtx;

pathCtx.pathAlgorithm = PA_PKIX;
pathCtx.pathOptions = 0;
pathCtx.trustedCerts = trustedRoots;
pathCtx.policies = ANY_POLICY;
pathCtx.validationTime = 0;
pathCtx.database = db;
```

Step 3: Verify a certificate path exists and check certificate revocation status

To build a certificate path to verify a certificate or CRL, perform step 3a. In this step, you will also check each certificate's revocation status as you previously set *pathOptions* to 0 (zero). To just check the revocation status of a particular certificate, perform step 3b.

Step 3a: Verify a certificate path exists

Use C_BuildCertPath to validate a certificate or a CRL. The object to validate can be either a CERT_OBJ or a CRL_OBJ. The C_BuildCertPath function constructs a path from *objToValidate* to one of the trusted certificates in the certificate path-processing context (CERT_PATH_CTX.*trustedCerts* field).

```
int C_BuildCertPath (
  CERTC_CTX      ctx,                          /* Cert-C context handle */
  CERT_PATH_CTX  *pathCtx,                    /* path-processing context */
  POINTER        startObject,               /* starting point for the path */
  LIST_OBJ       certPath,                 /* (in/out) resulting cert path */
  LIST_OBJ       crlList,              /* (in/out) CRLs to verify the path */
  LIST_OBJ       crlCerts,                /* (in/out) certs to verify CRLs */
  LIST_OBJ       policyInfoList             /* (in/out) cert policy list */
);
```

Pass a Cert-C context and a pointer to the path-processing context.

For this example, assume that you have an already created CERT_OBJ or CRL_OBJ, and that the object has been initialized with data. *objToValidate* points to this object. To see how to create and set a CERT_OBJ, see "Creating a Certificate Object" on page 169. To see how to create and set a CRL_OBJ, see "Creating a CRL Object" on page 231.

To validate a certification path you pass NULL_PTR values for the *certPath*, *crlList*, *crlCerts* and *policyInfoList* parameters.

You already set the path algorithm to PA_PKIX. This implements path processing according to section 6 of *RFC 2459*. In this example, if path processing is not successful using PA_PKIX, then the PA_X509_V1 algorithm is used to try to build a certificate path. PA_X509_V1 is a simple path-processing algorithm that only uses the v1 fields of a certificate. It performs basic issuer name/subject name matching and signature

verification.

```
status = C_BuildCertPath (ctx, &pathCtx, (POINTER)objToValidate,
                          (LIST_OBJ)NULL_PTR, (LIST_OBJ)NULL_PTR,
                          (LIST_OBJ)NULL_PTR, (LIST_OBJ)NULL_PTR);

if (status != 0) {
  RSA_PrintMessage ("C_BuildCertPath returned 0x%04x ", status);
  RSA_PrintMessage ("when using PA_PKIX.\nTrying PA_X509_V1...\n");

  pathCtx.pathAlgorithm = PA_X509_V1;

  status = C_BuildCertPath (ctx, &pathCtx, (POINTER)objToValidate,
                            (LIST_OBJ)NULL_PTR, (LIST_OBJ)NULL_PTR,
                            (LIST_OBJ)NULL_PTR, (LIST_OBJ)NULL_PTR);

  if (status != 0)
    goto CLEANUP;
}
```

Step 3b: Check revocation status

Use `C_CheckCertRevocation` to check the revocation status of a particular certificate. This function uses either the CRL or the OCSP revocation-status protocol, depending on the certificate-revocation status service provider you register when you initialize the Cert-C context. In this example, you chose to use the Cert-C Certificate Revocation Status service provider in step 1.

```
int C_CheckCertRevocation (
  CERTC_CTX        ctx,                                  /* Cert-C context */
  CERT_PATH_CTX    *pathCtx,                  /* Path-processing context */
  CERT_OBJ         cert,                     /* Certificate to be checked */
  CERT_REVOCATION  *revocation      /* (in/out) Revocation status of cert */
);
```

`C_CheckCertRevocation` returns the certificate's revocation status and related

information in a CERT_REVOCATION data structure.

```
typedef struct {
  int      status;                                  /* Certificate status */
  int      evidenceType;                              /* Type of evidence */
  POINTER  evidence;                                /* Evidence of status */
} CERT_REVOCATION;
```

Again, assume that you have an already created CERT_OBJ object called *objToValidate*, and that you have initialized the object with data. To see how to create and set a CERT_OBJ, see "Creating a Certificate Object" on page 169.

Pass a Cert-C context and a pointer to the path-processing context.

Pass a pointer to an allocated and initialized to zero CERT_REVOCATION structure. The caller is responsible for validating the revocation evidence that is returned (for example, validating the CRL signature and certification path).

```
CERT_REVOCATION revocationInfo = {0};

status = C_CheckCertRevocation (ctx, &pathCtx, objToValidate,
                                &revocationInfo);

if (status != 0)
    goto CLEANUP;

if (revocationInfo.status == CERT_REVOKED) {
  RSA_PrintMessage ("Cert has been revoked!\n");
  status = E_NOT_VALIDATED;
  goto CLEANUP;

} else if (revocationInfo.status == CERT_NOT_REVOKED) {
  RSA_PrintMessage ("Cert has not been revoked!\n");

} else if (revocationInfo.status == CERT_REVOCATION_UNKNOWN) {
  RSA_PrintMessage ("Insufficient information to determine ");
  RSA_PrintMessage ("revocation status.\n");
  /* At this point, the application should decide if this is a fatal error
     or not.  Here, we do not treat it as a fatal condition. */
```

```
} else {
  RSA_PrintMessage ("Invalid CERT_REVOCATION.status value!\n");
  status = E_INVALID_PARAMETER;
  goto CLEANUP;
}
```

Step 4: Clean up

The CERT_REVOCATION structure is populated by the C_CheckCertRevocation call. You are responsible for freeing the revocation evidence; call either C_DestroyCRLEvidence or C_DestroyOCSPEvidence. For more information about how to clean up the data stored in CERT_REVOCATION, see the *API Reference*.

```
if (revocationInfo.evidenceType == CRE_CRL)
  C_DestroyCRLEvidence ((CRL_EVIDENCE **)&revocationInfo.evidence);
else if (revocationInfo.evidenceType == CRE_OCSP)
  C_DestroyOCSPEvidence ((OCSP_EVIDENCE **)&revocationInfo.evidence);

C_FinalizeCertC (&ctx);
```

Verifying a Signature

You must prove the entity that generated and sent a certificate or CRL to you actually has access to the private key that corresponds to the public key contained in the certificate request. The certificate request was signed by the requestor using the subject's private key, so you can verify the signature using the public key contained in the certificate object or the CRL object.

Verifying a Signature on a Certificate

In this example, a certificate was signed by the issuer using the issuer's private key. You need to obtain the certificate issuer's public key; then you can verify the signature on the certificate.

Using the `C_VerifyCertSignature` function, you verify the signature. For more information about `C_VerifyCertSignature`, see the *API Reference*.

```
status = C_VerifyCertSignature (certObj, certPublicKeyObj);
if (status != 0)
  goto CLEANUP;
```

Verifying a Signature on a CRL

In this example, a CRL was signed by the issuer using the issuer's private key. You need to obtain the CRL issuer's public key; then you can verify the signature on the CRL.

Using the `C_VerifyCRLSignature` function, you verify the signature. For more information about `C_VerifyCRLSignature`, see the *API Reference*.

```
status = C_VerifyCRLSignature (crlObject, caPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Chapter 12

Storing and Retrieving Certificates, CRLs, and Private Keys

Storing private keys in a secure mechanism is an intrinsic part of any PKI-enabled system. A certificate can be a public structure; however, the certificate owner alone must possess the private key. Possession of the private key proves that you are the owner of the certificate. Therefore, private keys must be stored securely. It is important that you choose an appropriate storage and protection method for your application. You also need to choose an appropriate user validation method to access the private key.

When you request a certificate from a CA, you need to store the certificate in a certificate database. Later you can retrieve the certificate when you want to perform an operation that requires a certificate; for example, to self-sign the certificate.

You might need to retrieve certificates to validate a certificate path. When you originally received the certificate to be verified, you might also have received all or some of the certificates that you need to verify it. At the time of receipt, you store these certificate in a certificate database.

Cert-C provides APIs and database service providers that you can use to store and retrieve certificates, CRLs, and private keys. The details of the storage mechanisms provided in Cert-C are service-provider-specific.

Cert-C Database APIs

Most of the Cert-C database API calls require the use of a SERVICE handle. The SERVICE handle provides a convenient method to specify a subset of the service providers, registered with a Cert-C context, which are operated on by a particular API call. This saves you from constantly registering and unregistering service-provider instances.

Some of the functions that Cert-C provides to store and retrieve certificates, CRLs, and private keys are listed in the following tables:

Database Service and Iterator Functions

Function	Description
C_BindService	Binds a single currently registered service provider (with a given CERTC_CTX) to a SERVICE handle. A SERVICE handle can be used as a parameter to Cert-C functions that use a specific service provider or a set of service providers.
C_BindServices	Binds one or more currently registered service providers, of a single type and with a given CERTC_CTX, to a SERVICE handle. A service handle is required for some Cert-C API functions that can be directed to a particular service provider or to a set of service providers.
C_UnbindService	Unbinds a previous binding of service providers to the specified handle. It also performs the necessary clean-up for a SERVICE handle created with C_BindService or C_BindServices.
C_FreeIterator	Frees a database iterator previously created by one of the C_SelectFirst* functions. C_FreeIterator can be called to free an iterator before retrieving all of the records of a particular type.

Store Certificate Functions

Function	Description
C_InsertCert	Inserts a certificate into the database or databases that are bound to the SERVICE handle.
C_InsertCertList	Inserts a list of certificates into the database or databases that are bound to the SERVICE handle.

Store CRL Functions

Function	Description
C_InsertCRL	Inserts a CRL into the database or databases that are bound to the SERVICE handle.
C_InsertCRLList	Inserts a list of CRLs into the database or databases that are bound to the SERVICE handle.

Store Private Key Functions

Function	Description
C_InsertPrivateKey	Inserts a private key into the database or databases bound to the SERVICE handle. This API does not insert the certificate into the database bound to the SERVICE handle. And, it does not check to ensure that the private key and the certificate correspond to each other. You can do this checking or it can be done by service-provider implementation.
C_InsertPrivateKeyBySPKI	Inserts a private key into the database or databases bound to the SERVICE handle. The private key is identified by the corresponding subject-public-key identifier. This API does not check to ensure that the private key and the public key correspond to each other. You can do this checking or it can be done by service-provider implementation.

Retrieve Certificate Functions

Function	Description
C_SelectCertByAttributes	Retrieves one or more certificates, identified by the specified attributes (based on the name-value pairs in the ATTRIBUTES_OBJ) and base subject name, from the database or databases bound to the SERVICE handle. C_SelectCertByAttributes then adds a copy of the certificate to the certificate list.
C_SelectCertByExtensions	Retrieves one or more certificates, identified by the specified extensions and base subject name, from the database or databases bound to the SERVICE handle. Use C_CompareExtensions to compare extensions. C_SelectCertByExtensions then adds a copy of the certificate to the certificate list.
C_SelectCertByIssuerSerial	Retrieves the certificate, identified by the specified issuer name and serial number, from the database or databases bound to the SERVICE handle. C_SelectCertByIssuerSerial then adds a copy of the certificate to the certificate list.
C_SelectCertBySubject	Retrieves one or more certificates, identified by the specified subject name, from the database or databases bound to the SERVICE handle. C_SelectCertBySubject then adds a copy of the certificate to the certificate list.
C_SelectFirstCert	Retrieves the first certificate from the database or databases bound to the SERVICE handle, and adds a copy of the certificate to the certificate list. Use this API to begin an enumeration of the certificates in a particular database SERVICE handle. This routine initializes a DB_ITERATOR, which can be used with subsequent calls to C_SelectNextCert.
C_SelectNextCert	Retrieves the next certificate from the database or databases bound to the iterator, and adds a copy of the certificate to the certificate list. Use this API to continue the enumeration that began with the call to C_SelectFirstCert.

Retrieve CRL Functions

Function	Description
C_SelectCRLByIssuerTime	Retrieves a CRL, identified by the specified issuer name and time (whose last update time is the closest to the given time without exceeding the given time), from the database or databases bound to the SERVICE handle. C_SelectCRLByIssuerTime then adds a copy of the matching CRL to the CRL list.
C_SelectFirstCRL	Retrieves the first CRL from the database or databases bound to the SERVICE handle, and adds a copy of the CRL to the CRL list. Use this API to begin an enumeration of the CRLs in a particular database SERVICE handle. This routine initializes a DB_ITERATOR, which can be used with subsequent calls to C_SelectNextCRL.
C_SelectNextCRL	Retrieves the next CRL from the database or databases bound to the iterator, and adds a copy of the CRL to the CRL list. Use this API to continue the enumeration which began with the call to C_SelectFirstCRL.

Retrieve Private Key Functions

Function	Description
C_SelectFirstPrivateKey	Retrieves the first private key from the database or databases bound to the SERVICE handle, and adds a copy to the B_KEY_OBJ object. Use this API to begin an enumeration of the private keys in a particular database SERVICE handle. This routine initializes a DB_ITERATOR, which can be used with subsequent calls to C_SelectNextPrivateKey.
C_SelectNextPrivateKey	Retrieves the next private key from the database or databases bound to the iterator, and adds a copy to the B_KEY_OBJ object. Use this API to continue the enumeration, which began with the call to C_SelectFirstPrivateKey.

Function	Description
C_SelectPrivateKeyByCert	Retrieves the private key, identified by the specified certificate, from the database or databases bound to the SERVICE handle.
C_SelectPrivateKeyBySPKI	Retrieves the private key, identified by the specified subject's public-key identifier, from the database or databases bound to the SERVICE handle.

Cert-C Database Service Providers

Cert-C provides various database service providers to help you store and retrieve certificates, CRLs, and private keys. You use the Cert-C APIs to access the various database service providers' functions.

There are six Cert-C database service providers; the Cert-C Default Database service provider, the Cert-C In-Memory Database service provider, the Cert-C LDAP Database service provider, the Cert-C CryptoAPI Database service provider, the Cert-C SCEP Database service provider, and the Cert-C PKCS #11 Database service provider.

The database service providers are initialized by implementing the callbacks in the `DB_FUNCS` structure. For more information about `DB_FUNCS`, see the *API Reference*. The source code for the Cert-C database service providers is included in the Cert-C standard toolkit distribution. The "Service Provider" section of the *API Reference* details the implementations of the callbacks for each service provider.

Cert-C Default Database service provider

You can use this database service provider to store and retrieve certificates, CRLs, and private keys. It provides a persistent local database. Database entries are stored as records in files in the local file system. This database service provider is suitable for managing a small-to-medium number of entries; for example, up to tens of thousands of entries. Private keys are protected using standard PKCS #5 v2.0 password-based encryption. You provide the password through the `DEFAULT_DB_PARAMS` initialization parameters. You must also register a cryptographic service provider with this database service provider when you want to store or retrieve private keys.

Cert-C In-Memory Database service provider

You can use this database service provider to store and retrieve certificates, CRLs, and private keys. It stores entries in list objects that are in-memory. This database service provider is useful in caching or in processing lists of certificates, private keys, or PKCS #10 objects. This service provider does not encrypt private keys.

Cert-C LDAP Database service provider

You can use this database service provider to retrieve certificates and CRLs from an LDAP repository. You can make an LDAP repository available as a database service provider. The LDAP is a read-only source; it does not implement any write functions.

Cert-C CryptoAPI Database service provider

You can use this database service provider to store and retrieve certificates and key pairs. It translates Cert-C database function calls into CryptoAPI function calls. This enables you to share keys and certificates among applications written to the Cert-C API and applications written to CryptoAPI.

Cert-C SCEP Database service provider

You can use this database service provider to retrieve CA and RA certificates, and possibly certificate chains leading to them, from network devices such as routers. It is suitable for network devices that may need to retrieve trusted-root certificates for use with an SCEP PKI service provider when an LDAP server is not available. Once retrieved, these certificates are usually retained in some type of local storage until the device is either re-initialized or redeployed.

Cert-C PKCS #11 Database service provider

You can use this database service provider to store and retrieve certificates and private keys on a token. You must also register a PKCS #11-enabled cryptographic service provider with this database service provider.

For detailed information about the Cert-C database service providers, see the *API Reference*.

Storing and Retrieving Certificates, CRLs, and Private Keys

When storing or retrieving items in a database, you must first select the desired database service provider. For more information, see the "Service Provider" section of the *API Reference*. You should review the service-provider-specific initialization routine `S_Initialize*`, and note the second parameter's format. In most cases, this parameter is called *`params`*. It points to a structure that contains the specified database service provider's initialization parameters. Each database service provider has a distinct structure to pass initialization parameters to the service provider's initialization functions. You are responsible for creating any elements that these structures require.

The following is a list of samples that demonstrate the use of the database APIs:

- `samples/db/rsadbcert.c` and `samples/db/rsadbm.c` demonstrate the Cert-C Default Database service provider.
- `samples/db/imdbcert.c` demonstrates the Cert-C In-Memory Database service provider.
- `samples/db/ldap.c` demonstrates the Cert-C LDAP Database service provider.
- `samples/db/mscapicert.c` and `samples/db/mscapiroots.c` demonstrate the Cert-C CryptoAPI Database service provider (Win32 only).
- `samples/db/scepdb.c` demonstrates the Cert-C SCEP Database service provider.
- `samples/db/pkcs11db.c` demonstrates the Cert-C PKCS #11 Database service provider. (See the release notes for supported platforms.)

The following two examples outline the steps that a program has to go through to make use of the database API calls that store and retrieve certificates, CRLs, and private keys. The first example stores a certificate in a database. The second example follows on from the first example and retrieves a list of certificates from a database.

Storing a Certificate, CRL, or Private Key

In this example, you initialize the Cert-C In-Memory Database service provider. Its initialization routine is S_InitializeMemoryDB. The second parameter, *params*, takes either a NULL_PTR, which allows the service provider to manage the internal LIST_OBJs, or a pointer to a MEMORY_DB_PARAMS structure that contains the LIST_OBJs. Assume that you previously created the LIST_OBJs for use with this service provider. In this example, you set *params* to a NULL_PTR to allow the service provider to manage the internal LIST_OBJs.

Step 1: Register service providers with CERTC_CTX

Initialize Cert-C and register the Cert-C In-Memory Database service provider with the CERTC_CTX. For more information about registering a service provider, see "Initializing the Cert-C Context" on page 75.

```
#define DB_NAME "Sample IM Database"

  CERTC_CTX ctx = NULL;
  SERVICE_HANDLER spTable[1];
  POINTER spParams[1];

  spTable[0].type = SPT_DATABASE;
  spTable[0].name = DB_NAME;
  spTable[0].Initialize = S_InitializeMemoryDB;

  spParams[0] = NULL_PTR;

  status = C_InitializeCertC (spTable, spParams, SP_COUNT, &ctx);
  if (status != 0)
    goto CLEANUP;
```

Step 2: Bind a SERVICE handle

Create a SERVICE handle to associate a subset of the registered database service provider instances. For more information about binding a SERVICE handle, see

"Binding a Service" on page 80

```
SERVICE db = (SERVICE)NULL_PTR;

status = C_BindService (ctx, SPT_DATABASE, dbName, &db);
if (status != 0)
  goto CLEANUP;
```

You can use the SERVICE handle now with any of the database APIs.

Step 3: Insert a certificate

Assume you have a CERT_OBJ that you want to insert into a database. To insert a certificate into a database, you call C_InsertCert and pass the CERT_OBJ to the routine.

```
status = C_InsertCert (db, certObj);
if (status != 0)
  goto CLEANUP;
```

At this point, assuming you have a CRL_OBJ, you could insert a CRL by using C_InsertCRL. Or, assuming you have a B_KEY_OBJ, you could insert a private key by using C_InsertPrivateKey. However, not all database service providers will allow you to do so. For more information about what each database service provider can store, see "Cert-C Database Service Providers" on page 207, or see the "Service Provider" section of the *API Reference*.

Step 4: Clean up

If you no longer need the list objects, making sure you have saved any information you need later, then you can destroy them now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

Next, you call C_UnbindService to unbind the service provider, and C_FinalizeCertC to free allocated memory and zeroize sensitive data.

```
C_DestroyListObject (&certList);
C_UnbindService (&db);
C_FinalizeCertC (&ctx);
```

Retrieving a Certificate, CRL, or Private Key

In this example, you continue from the example "Storing a Certificate, CRL, or Private Key" on page 210. You have already initialized the Cert-C In-Memory Database service provider and created a SERVICE handle. Here you will enumerate the certificates in the databases associated with the SERVICE handle by creating a DB_ITERATOR. Cert-C copies any certificates that are found into a LIST_OBJ.

Step 1: Register service providers with CERTC_CTX

Initialize Cert-C and register the Cert-C In-Memory Database service provider with the CERTC_CTX. You already did this in the example "Storing a Certificate, CRL, or Private Key" on page 210. For more information about registering a service provider, see "Initializing the Cert-C Context" on page 75.

Step 2: Bind a SERVICE handle

Create a SERVICE handle to associate a subset of the registered database service provider instances. You already did this in the example "Storing a Certificate, CRL, or Private Key" on page 210. For more information about binding a SERVICE handle, see "Binding a Service" on page 80.

Step 3: Enumerate the contents of a SERVICE handle

First, you need to create a LIST_OBJ to store the enumerated certificates.

```
LIST_OBJ certList = (LIST_OBJ)NULL_PTR;

status = C_CreateListObject (&certList);
if (status != 0)
  goto CLEANUP;
```

Next, you need to initialize a database iterator to go through a list of certificates in a database. You call the C_SelectFirstCert to create the database iterator. C_SelectFirstCert retrieves the first certificate it finds and stores it in the LIST_OBJ.

At this point, you could retrieve a CRL by using C_SelectFirstCRL. Assuming you have a B_KEY_OBJ, you could also retrieve a private key by using C_SelectFirstPrivateKey. However, not all database service providers allow you to do so. For more information about what each database service provider can retrieve, see "Cert-C Database Service Providers" on page 207, or see the "Service Provider"

section of the *API Reference*.

```
DB_ITERATOR iterator = (DB_ITERATOR)NULL_PTR;

status = C_SelectFirstCert (db, &iterator, certList);
if (iterator == NULL) {
  RSA_PrintMessage ("Database empty.\n");
  status = 0;
}
```

You can now make subsequent calls using the `C_SelectNextCert` routine. `C_SelectNextCert` retrieves the next certificate it finds and stores it in the `LIST_OBJ`. You know that there are no certificates remaining in the database when Cert-C frees any memory associated with the iterator and the iterator is set to `NULL_PTR`. You can dispose of the database iterator before all of the certificates have been retrieved. To dispose of the database iterator, you call `C_FreeIterator`.

As with this point, you could retrieve a CRL by using `C_SelectNextCRL`. Assuming you have a `B_KEY_OBJ`, you could also retrieve a private key by using `C_SelectNextPrivateKey`. However, not all database service providers allows you to do so. For more information about what each database service provider retrieves, see "Cert-C Database Service Providers" on page 207, or see the "Service Provider" section of the *API Reference*.

```
else if (status != 0)
  goto CLEANUP;
else
  for (;;) {
    status = C_SelectNextCert (&iterator, certList);
    if (iterator == NULL) {
      status = 0;
      break;
    }
    else if (status != 0)
      goto CLEANUP;
  }
```

Step 4: Clean up

If you no longer need the list objects, making sure you have saved any information you need later, then you can destroy them now. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing.

You should also free the database iterator; to do this, you call C_FreeIterator. If a C_SelectFirst* or C_SelectNext* function has returned a non-zero status, it is not necessary to call C_FreeIterator. Upon return, the iterator is set to NULL_PTR.

Next, you call C_UnbindService to unbind the service provider, and C_FinalizeCertC to free allocated memory and zeroize sensitive data.

```
C_DestroyListObject (&certList);
C_FreeIterator (&iterator);
C_UnbindService (&db);
C_FinalizeCertC (&ctx);
```

Chapter 13

Retrieving Certificate Information

Throughout Cert-C, you retrieve information. Maybe the information is the BER-encoding of a certificate request or a CRL. It might be string information that you retrieve, concerning a name or serial number. Almost always, this information belongs to Cert-C. That is, Cert-C returns to you a pointer. If you follow that pointer, it leads you to the information inside a Cert-C object. Cert-C allocates space and places the appropriate information there. Cert-C tells you where that space is.

This happens when an argument to Cert-C is a pointer to a pointer, such as `unsigned char **`. You declare a variable to be a pointer and pass the address of that pointer. Cert-C goes to the address you pass and deposit a pointer. Now if you go to where that pointer points, you can find the information you are looking for. This information, though, belongs to a Cert-C object. Subsequent Cert-C calls that alter or destroy the object render that pointer undefined. You do not need to allocate or free this information; Cert-C does that during the `C_Destroy*` operation. Look at the information all you want, but if you need to save it, copy it into a buffer you created or allocated, or to a file.

There are a couple of exceptions to this rule. First, when Cert-C returns an integer value (for example, `int` or `unsigned int`), declare a variable to be the proper integer type and pass its address. Cert-C deposits an integer at that address. That integer belongs to you. Subsequent calls to Cert-C do not alter it, although it may become obsolete.

In this chapter, you work with several examples that retrieve information from a

NAME_OBJ, a ATTRIBUTES_OBJ, and an EXTENSIONS_OBJ. For an example of how to manipulate CERT_OBJ information, see "Manipulating Certificate Information" on page 182.

Retrieving Name-Object Information

Cert-C uses a NAME_OBJ object to represent the names of entities. Cert-C provides APIs for you to look at the information in a name object, in a form you can understand. In this example, you display the information from a name object in a readable form.

Step 1: Create a name object

You have already created a name object in "Creating a Name Object" on page 105.

Step 2: Set the name object with the name information in BER format

You have already performed this step in "Creating a Name Object" on page 105.

Step 3: Read the name information

You now have a CERT_FIELDS structure that contains all the information in the certificate. You can see this information using various C_Get* functions.

You know the name object is made up of a series of AVAs. You must first find out how many AVAs there are in the name object. Using the C_GetNameAVACount function, you get the AVA count. For more information about C_GetNameAVACount, see the *API Reference*.

```
int C_GetNameAVACount (
  NAME_OBJ  nameObject,                                  /* Name object */
  unsigned  int *count                          /* (out) Number of AVAs */
);
```

Using the C_GetNameAVACount function, you obtain the number of AVAs in *nameObject*'s AVA list, storing the result in *count*.

```
unsigned int avaCount;
```

```
if ((status = C_GetNameAVACount
     (newCertInfo.subjectName, &avaCount)) != 0)
  break;
```

Now that you know how many AVAs there are in the name object, you can use the C_GetNameAVA function to get each AVA. For more information about C_GetNameAVA, see the *API Reference*.

```
int C_GetNameAVA (
  NAME_OBJ       nameObject,                              /* Name object */
  unsigned int   index,                         /* Index in the AVA list */
  unsigned char  **type,                         /* (out) Attribute type */
  unsigned int   *typeLen,             /* (out) Length of attribute type */
  int            *valueTag,             /* (out) Tag for attribute value */
  unsigned char  **value,                        /* (out) Attribute value */
  unsigned int   *valueLen,           /* (out) Length of attribute value */
  int            *newLevel    /* (out) Flag if this AVA starts new level */
);
```

The type is going to be an object identifier (OID). Since an OID is simply a sequence of bytes, not a word or phrase, it will not be easy to read. You can check the returned *type* against the list of attribute types in the *API Reference*. Compare *type* to AT_COMMON_NAME for instance, to see if *type* is a common name. If they are equal, then *value* contains the actual name.

This routine returns pointers to the value. If you go to where those pointers point, you will find the information. Remember, the information is inside the name object; it belongs to Cert-C.

```
int valueTag, newLevel;
unsigned int index, typeLen, valueLen;
unsigned char *type, *value;

for (index = 0; index < avaCount; ++index) {
  if ((status = C_GetNameAVA
       (newRequestorInfo.subjectName, index, &type,
        &typeLen, &valueTag, &value, &valueLen,
        &newLevel)) != 0)
    break;
```

```
  if ((status = DisplayAVA
       (index, type, typeLen, valueTag, value, valueLen,
        newLevel)) != 0)
    break;
}
```

The DisplayAVA routine is not a Cert-C routine. It is a placeholder for a routine that reads the information so that you can verify its accuracy. In this way, you can make sure it conforms to your CA's guidelines. You write this routine to best fit your application.

Step 4: Perform operations

In this example, you do not perform any sign or verify operations. There is no signature to verify on a name object. You should have already verified the signature on the entire certificate request.

Step 5: Destroy the name object

At this stage, you might want to keep and reuse the name object. For example, you need to use a name object in some of the examples presented in the following chapters. However, if you no longer need the name object, making sure you have saved any information you need later, then you destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyNameObject (&requestorName);
```

Retrieving Attributes-Object Information

Generally, an attributes object can contain any information; for example, the `SigningTime` attribute, which contains the time when the PKCS #1 message was signed. In this example, Cert-C uses an `ATTRIBUTES_OBJ` object to represent and pass extra information about the certificate subject in a certification request. Cert-C provides APIs for you to look at the information in an attributes object, in a form you can understand. In this example, you display the information from an attributes object in a readable form.

Step 1: Create an attributes object

You have already created an attributes object in "Creating an Attributes Object" on page 115.

Step 2: Set the attributes object with the attributes information in BER format

You have already performed this step in "Creating an Attributes Object" on page 115.

Step 3: Read the attributes information

You now have an attributes object that contains all the extra attributes. You can see this information using various `C_Get*` functions.

Since the attributes in an attributes object are of your own design, Cert-C does not know much about them. Each attribute is of a particular type, but may possess more than one value.

First, you must find out how many attributes there are in the attributes object. Because the attributes object can contain only one attribute of each type, getting the number of attribute types in an attribute is equivalent to finding the number of attributes in the attributes object. Using the `C_GetAttributeTypeCount` function, you get the number of attributes in the attributes object. For more information about `C_GetAttributeTypeCount`, see the *API Reference*.

```
int C_GetAttributeTypeCount (
  ATTRIBUTES_OBJ  attributesObj,                    /* Attributes object */
  unsigned int    *count          /* (out) Number of distinct attributes */
);
```

Using the C_GetAttributeTypeCount function, you obtain the number of attributes in *attributesObj*'s attributes list, and store the result in *attribTypeCount*.

```
unsigned int attribTypeCount;

if ((status = C_GetAttributeTypeCount
     (extraAttributes, &attribTypeCount)) != 0)
  break;
```

Now that you know how many attribute types there are in the attributes object, you can use the C_GetAttributeType function to get each attribute type. For more information about C_GetAttributeType, see the *API Reference*.

```
int C_GetAttributeType (
  ATTRIBUTES_OBJ  attributesObj,                    /* Attributes object */
  unsigned int    index,                   /* Index in the attribute list */
  unsigned char   **type,                         /* (out) Attribute type */
  unsigned int    *typeLen              /* (out) Length of attribute type */
);
```

You now know the different attribute types contained in the attributes object. Next, you need to find out how many values are associated with each type. You can use the C_GetAttributeValueCount function to do this.

```
int C_GetAttributeValueCount (
  ATTRIBUTES_OBJ  attributesObj,                    /* Attributes object */
  unsigned char   *type,                               /* Attribute type */
  unsigned int    typeLen,                  /* Length of attribute type */
  unsigned int    *count                      /* (out) Number of values */
);
```

Finally, using the C_GetStringAttribute function, you can get each value for each

attribute type.

```
int C_GetStringAttribute (
  ATTRIBUTES_OBJ  attributesObj,                    /* attributes object */
  unsigned char   *type,                               /* attribute type */
  unsigned int    typeLen,                    /* length of attribute type */
  unsigned int    valueIndex,              /* index in the list of values */
  int             *valueTag,            /* (out) tag for the string value */
  unsigned char   **value,                          /* (out) string value */
  unsigned int    *valueLen               /* (out) length of string value */
);
```

In this example, you perform the C_GetAttributeType, C_GetAttributeValueCount, and C_GetStringAttribute routines in a nested for loop.

```
unsigned int typeIndex, attribType;
unsigned int valueIndex, valueCount, valueTag, valueLen;
unsigned char *attribType, *value;

for (typeIndex = 0;
     typeIndex < attribTypeCount;
     ++typeIndex) {

  if ((status = C_GetAttributeType
       (extraAttributes, typeIndex, &attribType,
        &attribTypeLen)) != 0)
    break;

  if ((status = C_GetAttributeValueCount
       (extraAttributes, attribType, attribTypeLen,
        &valueCount)) != 0)
    break;

  for (valueIndex = 0;
       valueIndex < valueCount;
       ++valueIndex) {
    if ((status = C_GetStringAttribute
         (extraAttributes, attribType, attribTypeLen,
          valueIndex, &valueTag, &value, &valueLen)) !=0)
      break;
```

```
    if ((status = DisplayAttributeValue
         (attribType, attribTypeLen, valueIndex, valueTag,
          value, valueLen)) != 0)
      break;

  }
  if (status !=0) break;
}
if (status != 0) break;
```

Remember, Cert-C returns a pointer to the value. That pointer points to a location inside the attributes object and belongs to Cert-C. If you want to save this information, then you should copy it.

The `DisplayAttributeValue` routine is not a Cert-C routine. It is a placeholder for a routine that reads the information so that you can verify its accuracy. In this way, you can make sure it conforms to your CA's guidelines. You write this routine to best fit your application.

Step 4: Perform operations

In this example, you do not perform any operations. There is no signature to verify on an attributes object. You should have already verified the signature on the entire certificate request.

Step 5: Destroy the attributes object

At this stage, you might want to keep and reuse the attributes object. For example, you will need to use an attributes object in some of the examples presented in the following chapters. However, if you no longer need the attributes object, making sure you have saved any information you need for later, then you can destroy it now. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyAttributesObject (&extraAttributes);
```

Retrieving Extensions-Object Information

Cert-C uses an EXTENSIONS_OBJ object to represent X.509 v3 extensions. Cert-C provides APIs for you to look at the information in an extensions object, in a form you can understand. In this example, you display the information from an extensions object in a readable form.

You can call the C_GetExtensionCount to find out how many extensions are in the object. Then you call C_GetExtensionTypeByIndex or C_GetExtensionInfo. Call C_GetExtensionValue to get the actual information.

Step 1: Create an extensions object

See how to create an extensions object in "Creating an Extensions Object" on page 257.

Step 2: Set the extensions object with the extensions information in BER format

See how to set the extensions object in "Creating an Extensions Object" on page 257.

Step 3: Read the extension information

You now have an extensions object that contains all the extensions. You can see this information using various C_Get* functions.

An extensions object can contain a number of extensions, so you must first find out how many extensions there are in the extensions object. You use the C_GetExtensionCount function to get the extension count. For more information about C_GetExtensionCount, see the *API Reference*.

```
int C_GetExtensionCount (
  EXTENSIONS_OBJ  extensionsObject,                  /* extensions object */
  unsigned int    *extensionCount          /* (out) extension entry count */
);
```

Using the C_GetExtensionCount function, you get the extension count.

```
unsigned int extenCount = 0;
```

```
status = C_GetExtensionCount (extenObj, &extenCount);
if (status != 0)
  goto CLEANUP;
```

You now know how many extensions there are in the extensions object. Next, you need to get each extension's information. You can do this by making a call to the `C_GetExtensionInfo` function, which gets extension information at a particular index, for each extension in the extensions object. For more information about the `C_GetExtensionInfo` function, see the *API Reference*.

```
int C_GetExtensionInfo (
  EXTENSIONS_OBJ  extensionsObject,                 /* Extensions object */
  unsigned int    index,                           /* Index of extension */
  EXTENSION_INFO  *extensionInfo                /* Extension information */
);
```

Note: The fields returned from this function are read-only. You do not have to create any objects or items before you call this function. Do not call any functions that modify these fields. Do not call any `C_Set*` or `C_Destroy*` functions on these fields.

If an extension is found, Cert-C places the extension information in an `EXTENSION_INFO` structure.

```
typedef struct EXTENSION_INFO {
  unsigned char  *type;                                 /* Extension's OID */
  unsigned int   typeLen;                         /* Extension's OID length */
  unsigned int   criticalFlag;                     /* Extension criticality */
  unsigned int   valueCount;              /* Extension value entries count */
  POINTER        reserved;                       /* Reserved for future use */
} EXTENSION_INFO;
```

Using the `C_GetExtensionInfo` function, you make a call for each extension in the extensions object. If an extension is found, the extension information is placed in an `EXTENSION_INFO` structure. The *`type`* and *`typeLen`* fields of *`extensionInfo`* are set to the type and type-length of the extension. The *`criticalFlag`* field is set to the extension's criticality flag. The *`valueCount`* field is set to the number of values in the extension's value list. The reserved field is set to `NULL_PTR`, and should be ignored.

```
EXTENSION_INFO extenInfo;
unsigned int i;
```

```
for (i = 0; i < extenCount; i++) {
  status = C_GetExtensionInfo (extenObj, i, &extenInfo);
  if (status != 0)
    goto CLEANUP;
```

You can now retrieve each extension value. You can do this using the `C_GetExtensionValue` function. For more information about the `C_GetExtensionValue` function, see the *API Reference*.

```
int C_GetExtensionValue (
  EXTENSIONS_OBJ  extensionsObject,                  /* Extensions object */
  unsigned int    extensionIndex,                      /* Extension index */
  unsigned int    valueIndex,        /* Index into extension's value list */
  POINTER         *value                         /* (out) Extension's value */
);
```

Note: The fields returned from this function are read-only. You do not have to create any objects or items before you call this function. Do not call any functions that modify these fields. Do not call any `C_Set*` or `C_Destroy*` functions on these fields.

Using the `C_GetExtensionValue` function, you get the extension value, referenced by *j* in the extension's value list, for a particular extension type. The target extension is referenced by *i*. This function returns a `POINTER` to a value. You need to cast this pointer to the appropriate data structure that corresponds to the extension type. To cross-reference the extension types to their corresponding Cert-C data structures, see the "Extension Types and Structures" section in the *API Reference*.

```
POINTER extenValue = NULL_PTR;
unsigned int j;

/*  Examine each value for this particular extension. */
for (j = 0; j < extenInfo.valueCount; j++) {
  status = C_GetExtensionValue (extenObj, i, j, &extenValue);
  if (status != 0)
    goto CLEANUP;
}
```

Note: The sample program utilities `samples/common/include/extnhlp.h` and `samples/common/source/extnhlp.c` include routines that you can use to print out the data, pointed to by `C_GetExtensionValue`'s *`*value`*, in a readable

format, for sample purposes. You can also look at the `samples/common/include/extnutil.h` and `samples/common/source/extnutil.c` sample program utilities, which use these routines.

Step 4: Perform operations

In this example, you do not perform any operations. There is no signature to verify on an extensions object. You should have already verified the signature on the entire certificate request.

Step 5: Destroy the extensions object

At this stage, you might want to keep and reuse the extensions object. For example, you need to use an extensions object in some of the examples presented in the following chapters. However, if you no longer need the extensions object, making sure you have saved any information you need later, then you can destroy it now. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyAttributesObject (&extraAttributes);
```

Chapter 14

CRL and CRL Entries

Each certificate has an expiration date. After expiration, the certificate owner should not use the certificate's key pair. Also, you should not use another entity's public key, for example, to envelope or verify (if the signature post-dates the expiration date), if the entity's certificate has expired.

Note: There are exceptions to this rule. For example, if you have an encrypted document that was encrypted using a key pair that was valid when encrypting, but, it is now expired. In this example, you need to use the expired key pair to decrypt the document.

A certificate owner or CA might want to indicate that a particular certificate is no longer valid, even before the expiration date. The key pair might have been compromised, or an employee might have left the company and no longer has the authority to use the certificate, and its related key pair. In this case, you need to revoke the certificate.

It is the CA's responsibility to revoke a certificate it issued. When a CA revokes a certificate, it must make this information known. It does this by compiling a list of certificates that have been revoked and distributing this list. This list is called a CRL. The CRL is not a list of certificates, but rather a list of certificate serial numbers along with the revocation dates.

Cert-C represents CRL information with a `CRL_OBJ` object. The CRL object has, as one of its components, the `CRL_ENTRIES_OBJ`. Through this object, you can add or delete the certificate serial numbers to the CRL. The `CRL_ENTRIES_OBJ` object is created for

you when you create a CRL object.

Cert-C also provides APIs that you can use to perform revocation checking. Your application can call the `C_CheckCertRevocation` function directly, using either CRLs or OCSP, to check the status of a particular certificate. When checking a certificate against a CRL, you will most likely use the `C_CheckCertRevocation` function. To validate a certificate chain, you use the `C_CheckCertRevocation` function with the `C_BuildCertPath`. To learn more about certificate revocation and certificate chaining, see "Certificate Revocation List" on page 39 and "Certificate Chaining" on page 38.

In this chapter, you work through examples that show you how to manipulate `CRL_OBJ` and `CRL_ENTRIES_OBJ` objects directly. You might want to manipulate these objects directly to parse or display CRL contents when writing or modifying certain service providers. Cert-C provides APIs for you to create a CRL object and to set, get, or modify the information in a CRL object. There are also APIs to set, get, or modify a CRL entries object, as well as adding or deleting a CRL entry to or from a CRL object.

CRL Object

Cert-C represents CRL information with a CRL_OBJ object. Use CRL objects to keep track of revoked certificates. A CRL object contains a list of CRL entries; each consists of a certificate serial number and a revocation time. The serial numbers identify the certificates that were revoked by the CRL issuer. In addition, just like certificates, CRLs have an issuer, a validity period, and a signature.

If the CRL version is CRL_VERSION_2, then the CRL_OBJ can also contain an EXTENSIONS_OBJ, which represents X.509 v3 CRL extensions. Each CRL entry can also contain an EXTENSIONS_OBJ, which represents X.509 v3 CRL Entry extensions.

CRL-Object Functions

You must use a Cert-C function to view or modify information in a CRL_OBJ object. You cannot assume that the CRL_OBJ points to any specific information. Some examples of the functions that Cert-C provides to manipulate a CRL object are listed in the following table.

Create, Use, or Destroy CRL_OBJ Functions

Function	Description
C_BuildCertPath	Constructs a certificate path and verifies a certificate or CRL.
C_CheckCertRevocation	Checks a certificate's revocation status.
C_CreateCRLObject	Creates a CRL object.
C_DestroyCRLObject	Destroys a CRL object, freeing the memory that the CRL object occupied.
C_SignCRL	Signs a CRL object.
C_VerifyCRLSignature	Verifies the signature on a CRL object.

Set or Modify CRL_OBJ Functions

Function	Description
C_PrepareUnsignedCRLForIssuer	Prepares an empty, unsigned CRL for an issuer.
C_SetCRLBER	Sets the BER encoding of a CRL object.

Function	Description
C_SetCRLFields	Sets a CRL object with the given information in the given CRL_FIELDS structure.
C_SetCRLInnerBER	Sets a CRL object to the BER encoding of the 'inner' portion of the CRL, which does not contain the signature.

Get CRL_OBJ Functions

Function	Description
C_GetCRLDER	Gets the DER encoding of a CRL object.
C_GetCRLFields	Gets a CRL_FIELDS structure corresponding to the CRL object.
C_GetCRLInnerDER	Gets the DER encoding of the 'inner' portion of the CRL object, which does not contain the signature.

Creating a CRL Object

In this example, you assume the role of a CA. In this capacity, you are responsible for revoking certificates and making certificate-revocation information known. You do this by building a CRL.

When building a CRL, you should follow the five-step process Create, SetFields, Sign, GetDER, and Destroy. There is one variation to this process: step 1 initializes some of the CRL object's values outlined in the `CRL_FIELDS` structure. In step 2, you need to get the structure first, and then set the fields.

You need a `CERT_CTX` context when creating a CRL object. In this example, assume you have a previously initialized `CERT_CTX` *ctx*. You can look at the `samples/crl/crl.c` sample program and use it to experiment with creating and parsing CRL objects.

Step 1: Create a CRL object

To create a CRL object, you use the `C_CreateCRLObject` function. For more information on `C_CreateCRLObject`, see the *API Reference*.

```
int C_CreateCRLObject (
  CRL_OBJ    *crlObj,                                  /* (out) CRL object */
  CERTC_CTX  ctx                                         /* Cert-C context */
);
```

Using the `C_CreateCRLObject` function, you declare a variable to be `CRL_OBJ` and pass its address as the argument. The Cert-C context holds information that Cert-C uses to handle extensions, as well as information about registered service providers. The return value of this routine is a zero (0) if successful and a non-zero error code when something goes wrong. Any clean-up code always executes, whether an error occurs or not. You should initialize an object to `NULL_PTR`; if there is an error before an object has the chance to be created, the clean-up code acts on a `NULL_PTR` and does not do any damage.

```
CRL_OBJ crlObject = (CRL_OBJ)NULL_PTR;

status = C_CreateCRLObject (crlObject, ctx);
if (status != 0)
  goto CLEANUP;
```

When you call the C_CreateCRLObject function, besides creating a CRL object, you also initialize some of the associated CRL_FIELDS structure fields with the CRL object's values.

```
typedef struct CRL_FIELDS {
  UINT2            version;    /* CRL_VERSION_1 (default) or CRL_VERSION_2 */
  int              signatureAlgorithm;
  NAME_OBJ         issuerName;
  UINT4            lastUpdate;
  UINT4            nextUpdate;
  CRL_ENTRIES_OBJ  crlEntries;
  EXTENSIONS_OBJ   crlExtensions;
  POINTER          reserved;
} CRL_FIELDS;
```

Step 2: Retrieve, update, and set CRL information

In this step, you retrieve the CRL information, update the CRL object information, and then set the CRL information in the CRL object.

Step 2a: Retrieve the CRL information

You now need to retrieve the CRL information. This is because the C_CreateCRLObject initializes some values in the associated CRL_FIELDS structure. You need to get the CRL_FIELDS structure before you set it with additional information later in step 2.

Using the C_GetCRLFields function, you get the CRL_FIELDS structure from the CRL_OBJ that you just created. In the first argument, you pass the newly created CRL_OBJ. The second argument is the CRL_FIELDS structure associated with the newly created CRL_OBJ.

```
CRL_FIELDS crlFields;

status = C_GetCRLFields (crlObject, &crlFields);
if (status != 0)
  goto CLEANUP;
```

At this point, if you look at the associated CRL_FIELDS structure, it contains the

following initialized values.

Table 14-1 **The CRL_FIELDS structure after creating a CRL_OBJ**

Field	Value
version	CRL_VERSION_1
signatureAlgorithm	SA_MD5_WITH_RSA_ENCRYPTION
issuerName	Empty NAME_OBJ
lastUpdate	0
nextUpdate	0
crlEntries	Empty CRL_ENTRIES_OBJ
crlExtensions	Empty EXTENSIONS_OBJ
reserved	0

Step 2b: Enter the new CRL information

Now that you have created a CRL object and initialized some of the associated CRL_FIELDS structure's fields, you need to enter the remaining CRL information.

In this example, assume that the CRL updates are one week apart. Each CA decides its own policy for when their CRL is updated.

```
#define ONE_WEEK_IN_SECONDS 0x93a80
```

You need to set the issuer name for the CRL. As a CA, assume you have the BER-encoded issuer name.

The GetCAInfoFromStorage routine is not a Cert-C routine. It is a placeholder for a routine that obtains the CA's BER-encoded X.500 Name. You will write this routine to best fit your application.

```
unsigned char *caNameBERFromStorage = NULL_PTR;
unsigned int caNameBERFromStorageLen;

status = GetCAInfoFromStorage (&caNameBERFromStorage,
                               &caNameBERFromStorageLen);
if (status != 0)
  goto CLEANUP;
```

```
status = C_SetNameBER (crlFields.issuerName, caNameBERFromStorage,
                       caNameBERFromStorageLen);
if (status != 0)
  goto CLEANUP;
```

You set the last update time to be the current time and the next update value to be a week from the last update time.

```
T_time (&(crlFields.lastUpdate));
crlFields.nextUpdate = crlFields.lastUpdate + ONE_WEEK_IN_SECONDS;
```

In this example, you leave the version and signature algorithm as the default. You also do nothing with the CRL_ENTRIES_OBJ and EXTENSIONS_OBJ objects. For more information about modifying a CRL_ENTRIES_OBJ object, see "CRL-Entries Object Functions" on page 241. See chapter 15, "Extensions," for more information about modifying a EXTENSIONS_OBJ object.

Step 2c: Set the CRL information

You have just modified the CRL_FIELDS structure from the *crlObj*. Now, you must call C_SetCRLFields to reconcile the internal state of the CRL_OBJ with the changes made to the CRL_FIELDS structure. The CRL_OBJ is not properly initialized with the new data and cannot be used in any operations, until you do this step.

```
status = C_SetCRLFields (crlObject, &crlFields);
if (status != 0)
  goto CLEANUP;
```

You now have a CRL object filled with all the proper information, although it has no entries. An entry would be a revoked certificate. This may in fact happen in real life. For information on how to add a CRL entry, see "Adding a CRL Entry to a CRL Object" on page 243. Once you have set all the information, you can go ahead and sign the CRL. A CRL is worthless unless it is signed by the issuing CA.

Step 3: Sign the CRL object

To sign the CRL, you use the C_SignCRL function. For more information about

C_SignCRL, see the *API Reference*.

```
int C_SignCRL (
  CRL_OBJ    crlObj,                                  /* (mod) CRL object */
  B_KEY_OBJ  privateKey                                    /* Signing key */
  . . .
);
```

The first argument is the CRL_OBJ that you created. The second argument is a Crypto-C key object, B_KEY_OBJ. It can point to a private key on a hardware device. For more information, see "Cert-C PKCS #11 Database Service Provider" and "Cert-C CryptoAPI Database Service Provider", in the "Service Provider" section of the *API Reference*.

For this example, assume that you already have a B_KEY_OBJ *caPrivateKey* that contains the issuer's private key. For more information about the B_KEY_OBJ object, see Appendix A. The GetCAPrivateKeyObject routine is not a Cert-C routine. It is a placeholder for a routine that obtains the issuer's private key. You write this routine to best fit your application.

```
B_KEY_OBJ caPrivateKey = (B_KEY_OBJ)NULL_PTR;

status = GetCAPrivateKeyObject (&caPrivateKey);
if (status != 0)
  goto CLEANUP;

status = C_SignCRL (crlObject, caPrivateKey);
if (status != 0)
  goto CLEANUP;
```

Step 4: Retrieve the CRL information in DER format

You now have a signed CRL object; you need to communicate this CRL with others who may not use Cert-C. Therefore, you need to get the CRL information into a format everyone can understand. To do this you use the C_GetCRLDER function to retrieve the DER encoding of the certificate-request information. For more

information about the C_GetCRLDER function, see the *API Reference*.

```
int C_GetCRLDER (
  CRL_OBJ        crlObject,                                  /* CRL object */
  unsigned char  **der,                          /* (out) DER-encoded CRL */
  unsigned int   *derLen              /* (out) length of DER-encoded CRL */
);
```

Using the C_GetCRLDER function, you give Cert-C a CRL object, the address of a pointer and the address of an `unsigned int`. At the addresses, Cert-C places a pointer to the DER encoding of the CRL and its length. What the pointer to the DER encoding points to belongs to Cert-C. You do not need to allocate or free that memory. Also, you should not attempt to adjust the data yourself. The information remains unchanged until you call a Cert-C routine that modifies or destroys the attributes object. To save this information, copy it into a file or your own buffer.

```
unsigned char *crlDer;
unsigned int crlDerLen;

status = C_GetCRLDER (crlObject, &crlDer, &crlDerLen);
if (status != 0)
  goto CLEANUP;
```

The SaveCRLDER routine is not a Cert-C routine. It is a placeholder for a routine that obtains the DER-encoded CRL information. You write this routine to best fit your application.

```
status = SaveCRLDER (crlDer, crlDerLen);
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the CRL and key objects

If you no longer need the CRL object, making sure you have saved any information you need later, then you can destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

```
CLEANUP;
  C_DestroyCRLObject (&crlObject);
```

Reading a CRL Object

In this example, you assume the role of a person who has received the DER encoding of a CRL. Before you can use the CRL information, you need to read the CRL information and verify the signature on the CRL.

If you are reading a CRL, you should follow the five-step process Create, SetBER, VerifySignature, GetFields, and Destroy.

You need a `CERT_CTX` context when creating a CRL object. In this example, assume you have a previously initialized `CERT_CTX` *`ctx`*. You can look at the `samples/crl/crl.c` sample program and use it to experiment with creating and parsing CRL objects.

Step 1: Create a CRL object

First, you need to create a CRL object to hold the CRL information. To create a CRL object, you use the `C_CreateCRLObject` function. You have already created a CRL object, see "Creating a CRL Object" on page 231. For more information on `C_CreateCRLObject`, see the *API Reference*.

```
int C_CreateCRLObject (
  CRL_OBJ    *crlObj,                                  /* (out) CRL object */
  CERTC_CTX  ctx                                         /* Cert-C context */
);
```

Step 2: Enter the CRL information

Next, you set the `CRL_OBJ` with the DER-encoded CRL object. To do this you use the `C_SetCRLBER` function. For more information about the `C_SetCRLBER` function, see the *API Reference*.

```
int C_SetCRLBER (
  CRL_OBJ   crlObject,                              /* (in/out) CRL object */
  unsigned  char *ber,                                 /* BER-encoded CRL */
  unsigned  int berLen                          /* Length of BER-encoded CRL */
);
```

The first argument is the `CRL_OBJ` you just created. The second argument points to the BER encoding of the CRL object. The third argument is the length of the CRL information. Assume the BER encoding of the CRL and its length are located at *`crlBER`*

and *crlBERLen*.

```
unsigned char *crlBER;
unsigned int crlBERLen;

status = C_SetCRLBER (crlObject, crlBER, crlBERLen);
if (status != 0)
  goto CLEANUP;
```

Step 3: Read the CRL information

You read a CRL to see if a particular certificate is on the list. To trust the CRL information, you need to know if it is current and that the signature on the CRL is valid. If you know the CA that issued the CRL, you can go ahead and verify the signature on the CRL. If not, the CRL lists the CRL issuer name. In this example, you read the information first before verifying the signature. You get the name of the issuer and use it to retrieve the issuer's public key from storage.

To read the CRL information, you use the `C_GetCRLFields` function. For more information about `C_GetCRLFields`, see the *API Reference*.

```
int C_GetCRLFields (
  CRL_OBJ     crlObject,                                  /* CRL object */
  CRL_FIELDS  *crlFields                    /* (out) CRL_FIELDS structure */
);
```

First, you declare a `CRL_FIELDS` structure as a variable. Then, you pass its address to the `C_GetCRLFields` function, along with a CRL object. Cert-C returns the CRL information in the `CRL_FIELDS` structure. You also declare a variable to be `B_KEY_OBJ`; this key object is used later in this example to store the issuer's public key. For more information about the `B_KEY_OBJ` object, see Appendix A.

```
CRL_FIELDS crlFields;
B_KEY_OBJ caPublicKeyObject = (B_KEY_OBJ)NULL_PTR;

status = C_GetCRLFields (crlObject, &crlFields);
if (status != 0)
  goto CLEANUP;
```

You now have a `CRL_FIELDS` structure with the CRL's information. You still need to actually read the CRL information. To do this you need a routine that displays the

CRL information.

`DisplayCRLInfo` is not a Cert-C routine. It is a placeholder for a routine that reads the CRL information so that you can read the issuer name, the next update time, and the last update time. You write this routine to best fit your application.

```
status = DisplayCRLInfo (&crlFields);
if (status != 0)
  goto CLEANUP;
```

If the next update time has already passed, you might not want to trust this CRL. You can determine if a certificate has been revoked from an out-of-date CRL (unless it is on hold). However, you cannot conclusively determine that a certificate is valid. The amount of trust that you put into a CRL should depend on your application of a certificate. For more information about CRLs, see "Certificate Revocation List" on page 39. For an example of how to check the validity of a certificate, see "Validating a Certificate Path" on page 193. You can use the `C_CheckCertRevocation` function with the Cert-C CRL Revocation Status service provider.

Now that you know the CRL's issuer name, you can use it to retrieve the issuer's public key. The `RecallCAPublicKey` routine is not a Cert-C routine. It is a placeholder for a routine that retrieves the issuer's public key, so that you can verify the signature on the CRL. You write this routine to best fit your application.

In this example, `RecallCAPublicKey` retrieves the issuer's public key from storage; for example, a database of CA public keys. You pass the associated `CRL_FIELDS` structure to the routine, which contains the *`issuerName`*. The routine matches the CA's name with a public key and builds the key object. For more information about creating a key object, see "Using BSAFE Crypto-C" on page 287. A more comprehensive description on using Crypto-C is in the *Crypto-C Developer's Guide*.

```
status = RecallCAPublicKey (&certFields, &caPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Step 4: Verify the CRL signature

The CRL was signed by the issuer using the issuer's private key. So now that you have the CRL issuer's public key, you can verify the signature on the CRL.

Using the `C_VerifyCRLSignature` function, you verify the signature. For more

information about C_VerifyCRLSignature, see the *API Reference*.

```
status = C_VerifyCRLSignature (crlObject, caPublicKeyObject);
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the CRL object

Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

```
CLEANUP;
  C_DestroyCRLObject (&crlObject);
  B_DestroyKeyObject (&caPublicKeyObject);
```

CRL Entries Object

Some CRL functions act on CRL_ENTRIES_OBJ objects instead of on CRL_OBJ objects.

Use the CRL_ENTRIES_OBJ object to access CRL entries information stored in a CRL_OBJ. The CRL_ENTRIES_OBJ is the part of the CRL_OBJ object that actually contains the serial numbers, revocation times, and X.509 v3 CRL Entry extensions for each revoked certificate.

Cert-C does not provide a way to create or destroy a CRL_ENTRIES_OBJ. Instead, it must be accessed through the *crlEntries* field of the CRL_FIELDS data structure. To get a CRL_FIELDS structure for the examination of the CRL_ENTRIES_OBJ, call C_GetCRLFields. To obtain the CRL_ENTRIES_OBJ, access the *crlEntries* field of the CRL_FIELDS structure.

Cert-C provides functions to add and delete entries in the CRL_ENTRIES_OBJ as well as to reset the CRL_ENTRIES_OBJ. After CRL entries are added or deleted, you must call C_SignCRL to make the CRL valid again.

CRL-Entries Object Functions

You must use a Cert-C function to view or modify information in a CRL_ENTRIES_OBJ. When you call one of these functions, you cannot assume that the CRL_ENTRIES_OBJ points to any specific information. Some examples of the functions that Cert-C provides to manipulate a CRL-entries object are listed in the following table.

Set or Modify CRL_ENTRIES_OBJ Functions

Function	Description
C_AddCRLEntry	Adds a CRL entry to a CRL_ENTRIES_OBJ.
C_DeleteCRLEntry	Deletes an entry from a CRL_ENTRIES_OBJ.
C_FindCRLEntryBySerialNumber	Finds a CRL entry given its serial number.
C_ResetCRLEntries	Resets a CRL_ENTRIES_OBJ.

Get CRL_ENTRIES_OBJ Functions

Function	Description
C_GetCRLEntriesCount	Gets the number of entries in a CRL_ENTRIES_OBJ.
C_GetCRLEntry	Gets an entry from a CRL_ENTRIES_OBJ.

The following examples show you how to manipulate the CRL entries object. If you are using high-level APIs in your application—for example, C_CheckCertRevocation—then it is likely that you do not need to manipulate the CRL-entries object directly.

Adding a CRL Entry to a CRL Object

In this example, you add a CRL entry to the CRL object that you created in the example, "Creating a CRL Object" on page 231. You can look at the `samples/crl/crl.c` sample program and use it to experiment with creating and parsing CRL entry objects.

Step 1: Create a CRL-entries object

You do not need to create a CRL-entries object, it is created for you when you create the CRL object. For more information about creating a CRL object, see "Creating a CRL Object" on page 231.

Step 2: Retrieve CRL_FIELD information, enter CRL-entries information, and set the CRL object

In this step, you retrieve CRL object's CRL-fields information, you enter the CRL-entries information, and then you set the CRL object with the new CRL information.

Step 2a: Retrieve the CRL_FIELD information

To add CRL-entry information to a CRL, you need to retrieve the CRL object's `CRL_FIELDS` structure. One of the structure's elements is a `CRL_ENTRIES_OBJ`. This object contains a `CRL_ENTRIES_INFO` structure where the CRL-entries information is stored.

Using the `C_GetCRLFields` function, you get the `CRL_FIELDS` structure from the `CRL_OBJ` object. In the first argument, you pass the `CRL_OBJ`. The second argument is the `CRL_FIELDS` structure associated with the `CRL_OBJ`.

```
CRL_FIELDS crlFields;

status = C_GetCRLFields (crlObject, &crlFields);
if (status != 0)
  goto CLEANUP;
```

Step 2b: Enter the CRL-entries information

To revoke a certificate, you add the certificate's serial number to the CRL. You do this using the `C_AddCRLEntry` function. For more information about `C_AddCRLEntry`, see

the *API Reference*.

```
int C_AddCRLEntry (
  CRL_ENTRIES_OBJ  crlEntriesObject,       /* (in/out) CRL entries object */
  CRL_ENTRY_INFO   *crlEntryInfo,                  /* Data for CRL entry */
  unsigned int     *index                 /* (out) Index of new CRL entry */
);
```

You add a new CRL entry to the CRL-entries object with the value given in *crlEntryInfo*. The index of the new entry is returned in *index*. The data structure for *crlEntryInfo* is CRL_ENTRY_INFO. For more information about CRL_ENTRY_INFO, see the *API Reference*.

```
typedef struct CRL_ENTRY_INFO{
  ITEM            serialNumber;             /* Certificate serial number */
  UINT4           actionTime;             /* Time cert is revoked or held */
  EXTENSIONS_OBJ  crlEntryExtensions;              /* Extensions object */
  POINTER         reserved;                  /* Reserved for future use */
} CRL_ENTRY_INFO;
```

Assume you already have a CRL object named *crlObject*, and that you have retrieved its CRL_FIELDS structure, which contains the CRL_ENTRIES_OBJ, using the C_GetCRLFields function. You also have the serial number of the certificate to be revoked in ITEM *serialNumber*.

You set the CRL_ENTRIES_OBJ CRL_ENTRY_INFO.*serialNumber* value to the value in ITEM. In this example, you set the CRL_ENTRY_INFO.*crlEntryExtensions* object to a properly cast NULL_PTR. For more information about extensions, see "Extensions Object" on page 254. You should also set the *reserved* value to NULL_PTR. The T_time function obtains the current time, and you set CRL_ENTRY_INFO's *actionTime* value to this current time.

```
ITEM serialNumber;
CRL_FIELDS crlFields;
CRL_ENTRY_INFO crlEntryInfo;
unsigned int crlEntryIndex;

crlEntryInfo.serialNumber = serialNumber;
T_time (&(crlEntryInfo.actionTime));
crlEntryInfo.crlEntryExtensions = (EXTENSIONS_OBJ)NULL_PTR;
crlEntryInfo.reserved = NULL_PTR;
```

You now have an updated CRL_ENTRY_INFO structure. You need to add this to the CRL-entries object, using the C_AddCRLEntry function. For more information about the function, see the *API Reference*.

```
int C_AddCRLEntry (
  CRL_ENTRIES_OBJ  crlEntriesObject,      /* (in/out) CRL entries object */
  CRL_ENTRY_INFO   *crlEntryInfo,                 /* Data for CRL entry */
  unsigned int     *index               /* (out) Index of new CRL entry */
);
```

To add the new CRL-entry information to the CRL-entries object, you pass the address of the updated CRL_ENTRY_INFO structure to the C_AddCRLEntry function. Cert-C returns the address to the index of the new entry in *crlEntryIndex*. The index is to keep track of the certificates in the CRL. You should note that the index number is not permanent. If you delete an entry, the indices of entries after the deleted entry are moved up by one.

```
status = C_AddCRLEntry (crlFields.crlEntries, &crlEntryInfo,
                        &crlEntryIndex);
if (status != 0)
  goto CLEANUP;
```

At this point you stop working directly with the CRL-entries object.

Step 2c: Set the CRL object

Since you just changed an element of the CRL object, you now need to call the C_SetCRLFields function to update the CRL object. For more information about the C_SetCRLFields function, see the *API Reference*.

```
int C_SetCRLFields (
  CRL_OBJ     crlObj,                                 /* (in/out) CRL object */
  CRL_FIELDS  *crlFields                                      /* CRL fields */
);
```

Step 3: Sign the CRL object

You have just updated the CRL object so now you need to resign it. To sign the CRL, you use the C_SignCRL function. For more information about signing the CRL object see “Step 3: Sign the CRL object” on page 234.

Step 4: Retrieve the CRL information in DER format

For more information about how to retrieve the CRL information in DER format, see "Step 4: Retrieve the CRL information in DER format" on page 235.

Step 5: Destroy the CRL object

If you no longer need the CRL object, making sure you have saved any information you need later, then you destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

You did not create the CRL-entries object, so you do not need to destroy it.

```
CLEANUP;
  C_DestroyCRLObject (&crlObject);
```

Deleting a CRL Entry from a CRL Object

In this example, you delete a CRL entry. You might want to do this when a revoked certificate has expired, and you want to save space.

Step 1: Create a CRL-entries object

You do not need to create a CRL-entries object, it is created for you when you create the CRL object. For more information about creating a CRL object, see "Creating a CRL Object" on page 231.

Step 2: Retrieve CRL_FIELDS information, delete CRL-entries information, and set the CRL object.

In this step, you retrieve the CRL object's CRL fields information, delete a CRL entry, and then set the CRL object with the update CRL information.

Step 2a: Retrieve the CRL_FIELDS information

To delete a CRL entry from a CRL, you need to retrieve the CRL object's `CRL_FIELDS` structure. One of the structure's elements is a `CRL_ENTRIES_OBJ`. This object contains a `CRL_ENTRIES_INFO` structure where the CRL-entries information is stored.

Using the `C_GetCRLFields` function, you get the `CRL_FIELDS` structure from the `CRL_OBJ` object. In the first argument, you pass the `CRL_OBJ`. The second argument is the `CRL_FIELDS` structure associated with the `CRL_OBJ`.

```
CRL_FIELDS crlFields;

status = C_GetCRLFields (crlObject, &crlFields);
if (status != 0)
  goto CLEANUP;
```

Step 2b: Delete the CRL entries information

To take a certificate off the CRL, you need the index number of the certificate you want to remove. In the example, "Adding a CRL Entry to a CRL Object" on page 243, when you added an entry, Cert-C returned the index number. However, the index number might have changed since you first added the certificate to the CRL. You need to find its current index. You can find it by calling the `C_FindCRLEntryBySerialNumber` function. For more information about the

C_FindCRLEntryBySerialNumber function, see the *API Reference*.

If you do not know the serial number, you can look at all the entries until you find the correct one. The example, "Reading a CRL-Entries Object" on page 250 describes how you can do that.

```
ITEM serialNumber;
unsigned int crlEntryIndex;

status = C_FindCRLEntryBySerialNumber (crlFields.crlEntries,
                                       serialNumber.data,
                                       serialNumber.len, &crlEntryIndex);
if (status != 0)
  goto CLEANUP;
```

To delete the CRL entry, you use the C_DeleteCRLEntry function. For more information about the C_DeleteCRLEntry function, see the *API Reference*.

```
int C_DeleteCRLEntry (
  CRL_ENTRIES_OBJ  crlEntriesObject,      /* (in/out) CRL entries object */
  unsigned int     crlEntryIndex         /* Index of entry to be deleted */
);
```

In this example, you assume the serial number is stored in ITEM *serialNumber*. You delete the CRL entry by passing the CRL-entry object's CRL_ENTRIES_INFO structure and the entries *crlEntryIndex* to the C_DeleteCRLEntry function. Cert-C deletes the entry in the CRL-entries object referenced by *crlEntryIndex*. The entries after *crlEntryIndex* are all shifted back by one.

```
status = C_DeleteCRLEntry (crlFields.crlEntries, crlEntryIndex);
if (status != 0)
  goto CLEANUP;
```

Step 2c: Set the CRL object

Since you just changed an element of the CRL object, you now need to call the C_SetCRLFields function to update the CRL object. For more information about the

C_SetCRLFields function, see the *API Reference*.

```
int C_SetCRLFields (
  CRL_OBJ     crlObj,                                  /* (in/out) CRL object */
  CRL_FIELDS  *crlFields                                       /* CRL fields */
);
```

Step 3: Sign the CRL object

You have just updated the CRL object; now you need to sign it. To sign the CRL, you use the C_SignCRL function. For more information about signing the CRL object see "Step 3: Sign the CRL object" on page 234.

Step 4: Retrieve the CRL information in DER format

For more information about how to retrieve the CRL information in DER format, see "Step 4: Retrieve the CRL information in DER format" on page 235.

Step 5: Destroy the CRL object

If you no longer need the CRL object, making sure you have saved any information you need later, then you destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

You did not create the CRL-entries object, so you do not need to destroy it.

```
CLEANUP;
  C_DestroyCRLObject (&crlObject);
```

Reading a CRL-Entries Object

In this example, you read the CRL entries from a CRL object. You must first create a CRL object, set it with the CRL information, and verify the signature on the CRL. In the example, "Reading a CRL Object" on page 237, you already performed these steps and ended up with a CRL_FIELDS structure. One of the elements in that structure was a CRL-entries object, so you do not need to create a CRL-entries object.

Step 1: Create a CRL object

For information about how to create a CRL object, see "Step 1: Create a CRL object" on page 237.

Step 2: Enter the CRL information

For information about how to enter the CRL information, see "Step 2: Enter the CRL information" on page 237.

Step 3: Read the CRL entries information

At this stage, assume you have already read the CRL and verified its signature. For more information about how to do this, see "Step 3: Read the CRL information" on page 238 and "Step 4: Verify the CRL signature" on page 239.

You now have a CRL_FIELDS structure that contains the CRL_ENTRIES_OBJ. To look at each of the entries in the CRL-entries object, you must first determine how many entries there are in the object. You can find out how many entries there are using the C_GetCRLEntriesCount function. For more information about the C_GetCRLEntriesCount function, see the *API Reference*.

```
int C_GetCRLEntriesCount (
  CRL_ENTRIES_OBJ  crlEntriesObject,              /* CRL entries object */
  unsigned int     *count                  /* (out) Number of entries in */
                                                    /* crlEntriesObject */
);
```

You pass the CRL-entries object to the C_GetCRLEntriesCount function, and Cert-C returns a pointer to *crlEntriesCount*. Cert-C sets *crlEntriesCount* to the number of revocation entries in the object.

Then, you get each one of the entries by index. You can do this using the C_GetCRLEntry function. For more information about the C_GetCRLEntry function, see

the *API Reference*.

You pass the CRL-entries object and an index to the CRL entry to the `C_GetCRLEntry` function. Cert-C gets the entry in the CRL list of *`crlEntriesObject`* at position *`crlEntryIndex`*, and passes an address to a `CRL_ENTRIES_INFO` structure. The index will always begin at 0 (zero) and run incrementally up to the count minus 1. When you delete an entry, the index of each entry after the deleted entry moves up one space.

```
unsigned int crlEntriesCount, crlEntriesIndex;
CRL_FIELDS crlFields;
CRL_ENTRY_INFO crlEntryInfo;

status = C_GetCRLEntriesCount (crlFields.crlEntries, &crlEntriesCount);
if (status != 0)
  goto CLEANUP;

for (crlEntriesIndex = 0; crlEntriesIndex < crlEntriesCount;
    ++crlEntriesIndex) {
  status = C_GetCRLEntry (crlFields.crlEntries, &crlEntryInfo,
                          crlEntryIndex);
  if (status != 0)
    goto CLEANUP;
```

The `DisplayCRLEntryInfo` function is not a Cert-C routine. It is a placeholder for code that you write to display the CRL-entries information in the form you want. You might want to do this to find an index number to use in deleting an entry. You write this code to best suit your application.

```
status = DisplayCRLEntryInfo (crlEntryIndex, crlEntryInfo);
if (status != 0)
  goto CLEANUP;
```

You can also use the `C_FindCRLEntryBySerialNumber` function (if you know the serial number) to determine the index, then call the `C_GetCRLEntry` without looking at each of the entries.

Step 4: Verify the CRL signature

The signature is verified on the CRL object. You already performed this step at the start of "Step 3: Read the CRL entries information" on page 250.

Step 5: Destroy the CRL object

If you no longer need the CRL object, making sure you have saved any information you need later, then you can destroy it now. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing.

You did not create the CRL entries object, so you do not need to destroy it.

```
CLEANUP;
  C_DestroyCRLObject (&crlObject);
```

Chapter 15

Extensions

X.509 v3 Certificate Extensions

Version 3 of the X.509 standard defines additional information that can be allowed on a certificate. These additional pieces of information are called extensions. The standard also allows each individual CA to create attributes not on the list of standard X.509 attributes, or other information and put them into certificates. This type of information is called the user-defined extension.

In the example, “Fulfilling the PKCS #10 Certificate Request” on page 174, you decomposed a certificate request and got a `CERT_FIELDS` out of the resulting certificate object; one of the elements of the structure was a created, but empty, extensions object. It is through this object that you can add certificate extensions.

In “Creating a CRL Object” on page 231 and “Adding a CRL Entry to a CRL Object” on page 243, you created two empty extensions objects after getting the fields from the created CRL and CRL entries objects. It is through those objects that you add CRL extensions.

Extensions Object

Cert-C represents X.509 v3 extensions with an EXTENSIONS_OBJ object. The EXTENSIONS_OBJ represents an extension set that contains one or more extension entries. Each extension entry is represented in two forms: a DER encoding and a list of values. The two forms provide equivalent information. The DER encoding of an extension entry is represented by an unsigned character string. The value list gives each value in the extension entry one at a time.

Each extension entry includes the extension type, its criticality, its handler, and a value list. There is only one extension entry for each extension type that is in the extensions object. The value list for some extension types can only have a single value at a time; the value list for other extension types can have multiple values at the same time.

When you use the C_CreateCRLObject or C_CreateCertObject function, to create a CRL_OBJ or a CERT_OBJ, respectively, Cert-C creates an EXTENSIONS_OBJ internally. You can access it through the *crlExtensions* field of the CRL_FIELDS data structure, the *crlEntryExtensions* field of the CRL_ENTRY_INFO data structure, or the *certExtensions* field of the CERT_FIELDS data structure. You can also create an EXTENSIONS_OBJ explicitly (without creating a certificate or CRL object) by calling the C_CreateExtensionsObject function.

Cert-C supports the following five X.509 v3 extensions-object types: certificate extensions (CERT_EXTENSIONS_OBJ), CRL extensions (CRL_EXTENSIONS_OBJ), CRL entry extensions (CRL_ENTRY_EXTENSIONS_OBJ), OCSP request extensions (OCSP_REQUEST_EXTENSIONS_OBJ), and OCSP single-certificate extensions (OCSP_SINGLE_EXTENSIONS_OBJ), as well as application-defined extensions. All extensions added to an extensions object must be of the same extensions-object type. When you call the C_CreateExtensionsObject function to create an extensions object, you must ensure that the extension information you provide is consistent with the extensions-object type you use. You must provide an extension-object type when you call the following functions: C_SetExtensionBER, C_SetEncodedExtensionValue, and C_SetExtensionsObjectBER. The X.509 v3 extension types are listed and cross-referenced to their corresponding Cert-C data structures in the "Extension Types and Structures" section of the *API Reference*. For additional details, see EXTENSION_TYPE_INFO in the *API Reference*.

Extensions-Object Functions

You must use a Cert-C function to view or modify information in an EXTENSIONS_OBJ. You cannot assume that the EXTENSIONS_OBJ points to any specific information. Some examples of the functions that Cert-C provides to manipulate an extensions object are listed in the following table:

Create, Reset, or Destroy EXTENSIONS_OBJ Functions

Function	Description
C_CreateExtensionsObject	Creates an *extensionsObject* of type *extensionsObjectType*.
C_DestroyExtensionsObject	Destroys the *extensionsObject* by deleting all the extensions and their value lists.
C_ResetExtensionsObject	Resets an extensions object, returning the extensions object to the state produced by calling the C_CreateExtensionsObject function.

Set or Modify EXTENSIONS_OBJ Functions

Function	Description
C_AddExtensionValue	Adds an extension value to an existing extension entry.
C_CompareExtension	Compares two extensions.
C_CompareExtensions	Compares two extensions objects (each representing a set of extensions).
C_CreateExtension	Creates a new extension entry in the extensions object.
C_DeleteExtensionValue	Deletes an extension value.
C_DestroyExtension	Destroys one extension, and its associated value list, as referenced by the extension-entry index.
C_FindExtensionByType	Finds an extension entry in the extensions object using the extension type.
C_RegisterExtensionType	Registers an application-defined extension or overrides the default setting of a supported standard extension.
C_SelectCertByExtensions	Retrieves one or more certificates identified by the specified extensions and base subject name.
C_SetEncodedExtensionValue	Sets the extension.

Function	Description
C_SetExtensionBER	Sets an extension with the given information.
C_SetExtensionsObjectBER	Sets *extensionsObject* with the new extensions.
C_UnregisterExtensionType	Resets or removes a registered extension handler and extension type from *ctx*.

Get EXTENSIONS_OBJ Functions

Function	Description
C_GetAttributeInExtensionsObj	Transfers data from an extensions object to an attributes object.
C_GetEncodedExtensionValue	Gets the encoded form of the extension's value(s).
C_GetExtensionCount	Gets the total number of extension types contained in an extensions object.
C_GetExtensionDER	Gets the DER encoding of an extension. The encoded value consists of the extension type, criticality, and the encoding of the extension's value list.
C_GetExtensionValue	Gets the value in the extension's value list referenced by the given value entry index.
C_GetExtensionsInAttributesObj	Transfers data from an attributes object to an extensions object.
C_GetExtensionInfo	Gets information about an extension, including associated values.
C_GetExtensionsObjectDER	Gets the DER-encoded value of all the extensions in the extensions object.
C_GetExtensionTypeByIndex	Gets the extension type from the extensions object referenced by the index given in *extensionIndex*.
C_GetExtensionTypeInfo	Searches for the extension type in *ctx*, if an extension type is found, then the associated extension's information is copied into *info*.

Creating an Extensions Object

If you want to add an extension that is defined in the X.509 v3 standard, the task is fairly simple. The following example, "Creating an Extensions Object" on page 257, creates an extensions object and adds an certificate-type extension. It also adds the extension entry's value. In this example, you create an extensions object from scratch. However, if you already have an existing (empty) extensions object from creating a certificate object or you want to add an additional extension to an extensions object, you skip step 1 and start at step 2.

You must use the `CERT_CTX` context when creating an extensions object. You can look at the `samples/exten/exten.c` sample program and use it to experiment with creating and parsing extensions objects.

Note: For an example of how to retrieve extension information from a `EXTENSIONS_OBJ`, see "Retrieving Extensions-Object Information" on page 223.

Step 1: Create an extensions object

To create an extensions object you use the `C_CreateExtensionsObject` function. For more information on `C_CreateExtensionsObject`, see the *API Reference*.

```
int C_CreateExtensionsObject (
  EXTENSIONS_OBJ  *extensionsObject,                /* Extensions object */
  unsigned int    extensionsObjectType          /* Extensions object type */
  CERTC_CTX       ctx                                 /* Cert-C context */
);
```

Using the `C_CreateExtensionsObject` function, you declare a variable to be `EXTENSIONS_OBJ` and pass its address as the argument. Next, you must provide the extension's extension-object type. Possible values for *`extensionObjectType`* are; `CERT_EXTENSIONS_OBJ`, `CRL_EXTENSIONS_OBJ`, `CRL_ENTRY_EXTENSIONS_OBJ`, `OCSP_REQUEST_EXTENSIONS_OBJ`, or `OCSP_SINGLE_EXTENSIONS_OBJ`, as well as application-defined extensions.

Note: All extensions added to an extensions object must be of the same extensions-object type.

In this example, you create a certificate-extensions object by passing `CERT_EXTENSIONS_OBJ` as the *`extensionsObjectType`* parameter. You also pass Cert-C a previously initialized Cert-C context. For more information about initializing a Cert-C

context, see "Initializing the Cert-C Context" on page 75.

The return value of this routine is a 0 (zero) for success and a non-zero error code when something goes wrong. Any clean-up code always executes, whether an error occurs or not. You should initialize an object to NULL_PTR, if there is an error before an object has the chance to be created, the clean-up code acts on a NULL_PTR and does not do any damage.

```
EXTENSIONS_OBJ extenObj = (EXTENSIONS_OBJ)NULL_PTR;

status = C_CreateExtensionsObject (&extenObj, CERT_EXTENSIONS_OBJ, ctx);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the extension information

Now that you have an extensions object, you need to set the extensions object with extension information. This information takes two forms, an extension entry and an extension-entry value. Each extension-entry type can have one or more values associated with that type. To locate a particular extension entry value, Cert-C provides two indices. One index, referred to as *extensionIndex* in this example, is the index to the extension type (for example, a basic constraint type). The second index, referred to as *valueIndex* in this example, is an index into the list of values associated with a particular extension type.

To add an extension entry to the extensions object, you use the C_CreateExtension function. For more information about the C_CreateExtension function, see the *API Reference*.

```
int C_CreateExtension (
  EXTENSIONS_OBJ     extensionsObject,      /* (in/out) Extensions object */
  unsigned char      *type,                            /* Extension OID */
  unsigned int       typeLen,                 /* Length of extension OID */
  unsigned int       *extensionIndex,       /* (out) New extension's index */
  int                criticality,      /* Promote extension's criticality */
  EXTENSION_HANDLER  *handler                         /* Extension handler */
);
```

Using the C_CreateExtension function, you set the extension type to one of the ET_* OIDs. In this example, you set it to ET_ISSUER_ALTNAME to indicate Issuer Alternate Name. Then you set the extension-type length to ET_ISSUER_ALTNAME_LEN. The OID is a defined sequence of bytes. For example, the byte sequence 0x55, 0x1D, 0x08 is the

X.509 standard's way of expressing the Issuer Alternate Name extension type. For a list of the Cert-C supported extension types, see the *API Reference*.

Cert-C makes a list of extensions in an extensions object. Each extension has an index number. `C_CreateExtension`'s fourth argument returns a pointer to the index of the new extension entry in the extensions object.

Each extension has a criticality. When a particular extension's criticality is set to critical, the reader of the certificate must understand the extension type. If the reader does not understand the extension type, then the reader should not accept the certificate. The X.509 standard defines the default criticality of an extension. To accept an extensions default criticality, you set the *`criticality`* flag to `NOT_IN_USE`. However, if the extension type's criticality was set by the `C_RegisterExtensionType` function, the registered criticality is used.

To change an extensions default criticality (if the standard allows you to override it), you set the *`criticality`* flag to `CRITICAL` or `NON_CRITICAL`. For a list of the Cert-C-supported extension types' default criticality, see the "Extension Types and Structures" section of the *API Reference*.

In the last argument, you have the option to pass an extension handler to override the default extension handler. In this example, because `ET_ISSUER_ALTNAME` is a standard extension, you use the default handler, so you set *`handler`* to `NULL_PTR`. However, you can use this argument to tell Cert-C to use a separate extension handler for only this instance of this extension. Passing a new handler at this point does not alter the default handler. Also, the new handler is not used for later creations of the same extension type. To do that, you would have to register an extensions handler with the application context using the `C_RegisterExtensionType` function. For an example of how to register an extensions handler, see "Registering a User-Defined Extension" on page 279. Alternatively, you can look at the `samples/bcdemo/source/userextn.c` sample program, which adds a handler for a user-defined extension. You can also look at the `sample/cert/critical.c` sample program to register a default handler.

Note: You use the extension handler to supply routines that define how to transform an extension value from a C structure to the BER-encoded format, and back again. Cert-C provides default handlers for the Cert-C-supported extensions types, (`ET_*` identifiers). For more information on `EXTENSION_HANDLER` or the extension types that Cert-C supports, see the *API Reference*.

```
unsigned int extensionIndex;
```

```
status = C_CreateExtension (certFields.certExtensions, ET_ISSUER_ALTNAME,
                            ET_ISSUER_ALTNAME_LEN, &extensionIndex, 0,
                            (EXTENSION_HANDLER *)NULL_PTR);
if (status != 0)
  goto CLEANUP;
```

Now you can add the extension value.

To add a value to the extension entry, you use the `C_AddExtensionValue` function. You need to specify the index to the extension entry in the extensions object. Cert-C returns an value index in *`valueIndex`*. This value index identifies the location where the value was added in the list of values associated with that particular extension entry. For more information about the `C_AddExtensionValue` function, see the *API Reference*.

```
int C_AddExtensionValue (
  EXTENSIONS_OBJ  extensionsObject,
  unsigned int    index,
  POINTER         value,
  unsigned int    *valueIndex
);
```

Using the `C_AddExtensionValue` function, you specify the extensions object, the index to the extension entry within the extensions object, and the extension value. The second argument, *`extensionIndex`*, is the value Cert-C returned to you when you called `C_CreateExtension`. *`valueIndex`* points to the index of the new extension-entry value in the extension-entry value list.

This function's third argument, *`value`*, takes a `POINTER` type. You need to cast *`value`* to the appropriate data structure that corresponds to the extension type. In this example, you cast *`value`* to a `ALTERNATE_NAME` structure. For more information about the

ALTERNATE_NAME structure, see the *API Reference*.

```
typedef struct ALTERNATE_NAME {
  unsigned int altNameType;
  union {
    OTHER_NAME otherName;                          /* OTHER_NAME structure */
    ITEM rfc822Name;                                    /* IA5String type */
    ITEM dNSName;                                       /* IA5String type */
    OR_ADDRESS x400Address;                       /* OR_ADDRESS structure */
    NAME_OBJ directoryName;            /* Distinguished Name object type */
    EDI_PARTY_NAME ediPartyName;             /* EDI_PARTY_NAME structure */
    ITEM resourceLocator;                               /* IA5String type */
    ITEM ipAddress;                                  /* Octet-string type */
    ITEM registeredID;                            /* Object-identifier type */
  } altName;
} ALTERNATE_NAME;
```

For more information on the extension types that Cert-C supports and to cross-reference the extension types to their corresponding Cert-C data structures, see the *API Reference*.

In this example, you add a Web address through the *resourceLocator* field of the ALTERNATE_NAME structure. Set ALTERNATE_NAME.*altNameType* to one of the possible CN_* values. In this example, you set *altNameType* to CN_RESOURCE_LOCATOR. For a list of the possible CN_* values, see the *API Reference*.

```
unsigned int valueIndex;
char *webAddress = "http:\/\/www.rsa.com";
ITEM webItem;
ALTERNATE_NAME issuerAltName;

webItem.data = (unsigned char *)webAddress;
webItem.len = T_strlen (webAddress);
issuerAltName.altNameType = CN_RESOURCE_LOCATOR;
issuerAltName.altName.resourceLocator = webItem;

status = C_AddExtensionValue (certFields.extensionsObject, extensionIndex,
                              (POINTER)(&issuerAltName), &valueIndex);
if (status != 0)
  goto CLEANUP;
```

Step 3: Perform operations

In this example, you do not perform any sign or verify operations.

Step 4: Retrieve the extension information in DER format

You now have an extensions object that contains an extension entry and the extension entry's value; however, it is in a Cert-C format. You need to get the extension information out of the EXTENSIONS_OBJ and into a format other applications can recognize, such as the DER-encoded format. To get the extension information out of the EXTENSIONS_OBJ, you use the C_GetExtensionsObjectDER function. You can also get the DER encoding of each individual extension using the C_GetExtensionDER function. For more information about C_GetExtensionsObjectDER and C_GetExtensionDER functions, see the *API Reference*.

```
int C_GetExtensionsObjectDER (
  EXTENSIONS_OBJ  extensionsObject,                  /* Extensions object */
  unsigned char   **der,           /* (out) DER-encoded extension entries */
  unsigned int    *derLen                  /* (out) Length of DER entries */
);
```

You pass the extensions object as the first argument. Cert-C returns addresses that contain pointers to the DER-encoded values of all the extensions in *extensionsObject*. For each extension type in the *extensionsObject*, the corresponding GetEncodedValue callback in the handler is called to obtain the encoded extension value.

```
unsigned char *derData;
unsigned int derDataLen;

status = C_GetExtensionsObjectDER (certFields.extensionsObject, &derData,
                                   &derDataLen);
if (status != 0)
  goto CLEANUP;
```

What the pointer to the DER encoding points to, belongs to Cert-C. You do not need to allocate or free up that memory. Also, do not adjust that data yourself. That information remains unchanged until you call a Cert-C routine that modifies or destroys the extensions object. You should copy the extension information into a database or a file.

The RSA_WriteDataToFile routine is not a Cert-C routine; it is a demo utility routine. For more information about Cert-C demo utilities, see the "Utilities" chapter in the

Advanced Developer's Guide. You can use `RSA_WriteDataToFile` to write binary data to a file.

```
status = RSA_WriteDataToFile (extenDer.data, extenDer.len,
                              "Enter name of file to store extension
                              binary");
if (status != 0)
  goto CLEANUP;
```

Step 5: Destroy the extensions object

At this stage, you might want to keep and reuse the extensions object. For example, you need to use an extensions object in the examples presented in the remainder of this chapter. However, if you no longer need the extensions object, making sure you have saved any information you need later, then you destroy it now. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing.

```
CLEANUP:
  C_DestroyExtensionsObject (&extensionsObject);
```

Extensions Information in an Attributes Object

At some time, you may want to pass additional information (in the certificate request) to the CA. To do this, you build an extensions object and fill it with an extension or extensions. Then, you transfer the extensions to an attributes object. The CA gets the attributes object as part of the certificate request and transfers the extensions from the attributes object into an extensions object.

"Putting Extensions in an Attributes Object" on page 264 explains how to create an attributes object that contains extensions information. This type of attribute is defined in PKCS #9 (`pkcs-9-at-extensionRequest`).

"Reading Extensions in an Attributes Object" on page 267 explains how to read an attributes object that contains extensions information.

Putting Extensions in an Attributes Object

In this example, you build an extensions and an attributes objects; then you transfer the extension information to the attributes object. You can then use the attributes object when you create a certificate request.

Step 1: Create an extensions and an attributes object

First, you need to create an extensions and an attributes object. You have already done this is the "Creating an Extensions Object" on page 257 and "Creating an Attributes Object" on page 115.

```
EXTENSIONS_OBJ extensionsObj = (EXTENSIONS_OBJ)NULL_PTR;
ATTRIBUTES_OBJ attributesObj = (ATTRIBUTES_OBJ)NULL_PTR;

status = C_CreateExtensionsObject (&extensionsObj, CERT_EXTENSIONS_OBJ, ctx);
if (status != 0)
  goto CLEANUP;

status = C_CreateAttributesObject (&attributesObj);
if (status != 0)
  goto CLEANUP;
```

Step 2: Enter the extension information

Now that you have created an extensions object, you need to set the extensions object with extension information. In this example, you add a key-usage extension. Recall that to add information to an extensions object, you must create an extension and then add the value. To add an extension entry to the extensions object, you use the `C_CreateExtension` function. To add a value to the extension entry, you use the `C_AddExtensionValue` function. For more information about the `C_CreateExtension` and `C_AddExtensionValue` functions, see the *API Reference*.

Using the `C_CreateExtension` function, you specify the extensions object, the extension type, the extension-type length, criticality, and optionally an extension handler. *`extensionIndex`* points to the index of the new extension entry in the extensions object.

```
unsigned int extensionIndex, valueIndex;
UINT4 keyUsage;

keyUsage = CF_KEY_ENCIPHERMENT;

status = C_CreateExtension (extensionsObj, ET_KEY_USAGE, ET_KEY_USAGE_LEN,
                            &extensionIndex, 0,
                            (EXTENSION_HANDLER *)NULL_PTR)) != 0)
if (status != 0)
  goto CLEANUP;
```

Using the `C_AddExtensionValue` function, you specify the extensions object, the index to the extension entry within the extension object, and the extension value. *`valueIndex`* points to the index of the new extension-entry value in the extension-entry value list.

```
status = C_AddExtensionValue (extensionsObj, extensionIndex,
                              (POINTER)&keyUsage, &valueIndex);
if (status != 0)
  goto CLEANUP;
```

Step 3: Perform operations

In this example, you do not perform any sign or verify operations.

Step 4: Retrieve the extensions information

Now that you have a filled extensions object, you can transfer it into an attributes

object. To do this, you use the C_GetAttributeInExtensionsObj function. For more information about C_GetAttributeInExtensionsObj, see the *API Reference*.

```
int C_GetAttributeInExtensionsObj (
  EXTENSIONS_OBJ  extensionsObject,           /* (in) Extensions object */
  ATTRIBUTES_OBJ  attributesObject        /* (in/out) Attributes object */
);
```

You pass the extensions and attributes objects to the C_GetAttributeInExtensionsObj function. Cert-C transfers the extension information into the attributes object.

```
status = C_GetAttributeInExtensionsObj (extensionsObj, attributesObj);
if (status != 0)
  goto CLEANUP;
```

Now you can create a certificate request using a PKCS10_FIELDS structure that contains the attributes object you just created. The PKCS10_FIELDS.*attribute* field is set to this attributes object. For more information about creating a certificate request, see "Creating a PKCS #10 Certificate Request" on page 123.

You can also get the extension information in a couple of other ways. You can get the DER encoding of the extensions object using the C_GetExtensionsObjectDER function. Or, you can get the DER encoding of the attributes object using the C_GetAttributesDER function. For more information about these functions, see the *API Reference*.

You can build as many extensions as you want; you only need one extensions object and one attributes object. You can also put user-defined attributes into the same attributes object. For example, in "Creating an Attributes Object" on page 115 you added the attribute Employee Number. If you also added Key Usage, then you can use the same attributes object. So you have one attributes object that contains two attributes, one of which is an X.509 v3 extension.

You can also get the DER encoding of the extensions from an attributes object using the C_GetAttributeValueDER function and the type AT_X509_V3.

Step 5: Destroy the extensions and attributes objects

Any object you create you must destroy, making sure you have saved any information you need later. This frees up any memory allocated by Cert-C. If an object is NULL_PTR, then Cert-C does nothing. That is why you should always initialize all

objects to NULL_PTR and call the C_Destroy* function later. If there is an error before creating an object, then the C_Destroy* function does not do any damage.

```
CLEANUP:
  C_DestroyExtensionsObject (&extensionsObj);
  C_DestroyAttributesObject (&attributesObj);
```

Reading Extensions in an Attributes Object

There are two ways to read extensions in an attributes object. The first way is to get the BER encoding of an attributes object. You build an attributes and an extensions object. Then you set the attributes object with the BER encoding of the attributes object. After, you call the C_GetExtensionsInAttributesObj function. This function transfers the extension information from the attributes object to the extensions object. This is the reverse of what you did in the example, "Putting Extensions in an Attributes Object" on page 264.

The second way to read extensions in an attributes object is to read a certificate request. For example, as a CA, you can receive a certificate request that contains an attribute with extension information. In the example, "Fulfilling the PKCS #10 Certificate Request" on page 174, you created a PKCS10_OBJ and CERT_OBJ. You called the C_GetCertFields function to look at the CERT_OBJ's CERT_FIELDS information and you set the CERT_FIELDS's fields with the PKCS #10 request information. At that point you had a created, but unset, extensions object, CERT_FIELDS.*certExtensions*.

In this example, you continue from the end of "Step 3c: Fill the certificate object's CERT_FIELDS structure" on page 177. You copy information from an attributes object to the extensions object and then read the extensions object.

Step 1: Create an attributes and an extensions object

In the example, "Fulfilling the PKCS #10 Certificate Request" on page 174, when you called the C_GetCertFields function you created an extensions object. This function retrieved the certificate object's CERT_FIELDS structure, which gave you access to the extensions object through CERT_FIELDS.*certExtensions*.

Instead of creating an attributes object, you retrieve it from the PKCS #10 certificate request. Using the C_GetPKCS10Fields function, you create and fill a PKCS10_FIELDS structure. For more information about the C_GetPKCS10Fields function and the

PKCS10_FIELDS structure, see the *API Reference*.

```
typedef struct PKCS10_FIELDS {
  UINT2           version;
  NAME_OBJ        subjectName;
  ITEM            publicKey;
  ATTRIBUTES_OBJ  attribute;
  POINTER         reserved;
} PKCS10_FIELDS;
```

The PKCS10_FIELDS structure contains the attributes object in its *attribute* field.

```
PKCS10_FIELDS pkcs10Fields;

status = C_GetPKCS10Fields (pkcs10Obj, &pkcs10Fields);
if (status != 0)
  goto CLEANUP;
```

Step 2: Copy the attributes-object information into the extensions object and read the extensions-object information

In this step, you copy the information in the attributes object into the extensions object; then you read the extensions-object information.

Step 2a: Enter the extensions information

You need to copy the information in the attributes object into the extensions object. You can do this by calling the C_GetExtensionsInAttributesObj function.

```
int C_GetExtensionsInAttributesObj (
  EXTENSIONS_OBJ  extensionsObject,         /* (in/out) Extensions object */
  ATTRIBUTES_OBJ  attributesObject             /* (in) Attributes object */
);
```

You pass both the attributes and extensions object to the C_GetExtensionsInAttributesObj function. Cert-C copies the information in *pkcs10Fields.attributes* to the extensions object. Assume that *pkcs10Fields.attributes* is an ATTRIBUTES_OBJ that contains an extensions attribute and that *newCertInfo.certExtensions* is an EXTENSIONS_OBJ, which has already been

created.

```
status = C_GetExtensionsInAttributesObj (newCertInfo.certExtensions,
                                         pkcs10Fields.attribute);
if (status != 0)
  goto CLEANUP;
```

Now you can read the extensions object.

Step 2b: Read the extensions object

First you need to find out how many extensions there are in the extensions object. You can do this using the `C_GetExtensionCount` function. Then you can call either the `C_GetExtensionTypeByIndex` or the `C_GetExtensionInfo` function. Finally, you call the `C_GetExtensionValue` function to get the actual extension information.

See the example, "Retrieving Extensions-Object Information" on page 223, for a more detailed description on how to retrieve extension information.

Step 3: Perform operations

In this example, you do not perform any sign or verify operations.

Step 4: Destroy all objects

Any object you create you must destroy, making sure you have saved any information you will need later. This frees up any memory allocated by Cert-C. If an object is `NULL_PTR`, then Cert-C does nothing. That is why you should always initialize all objects to `NULL_PTR` and call the `C_Destroy*` function later. If there is an error before creating an object, then the `C_Destroy*` function does not do any damage.

```
CLEANUP:
  C_DestroyExtensionsObject (&extensionsObj);
  C_DestroyAttributesObject (&attributesObj);
```

User-Defined Extensions

With standard X.509 v3 extensions, it is relatively simple to create an extension and add it to a certificate. Cert-C knows how to handle the standard extensions. However, you might want to add information to a certificate that is not supported by the X.509 v3 standard.

In the example, "Creating an Attributes Object" on page 115, you created an attributes object to hold the employee number information. Since employee number is not a standard X.509 attribute, you used the attributes object. But the information was not part of the actual certificate.

There is a way to place non-standard information into a certificate. You can create a user-defined extension. This can be included in the actual certificate.

With the standard extensions, you can create an extension by giving Cert-C the extension's type. Cert-C looks at the type (for example, `ET_ISSUER_ALTNAME`), sees that it is something it recognizes, and knows how to handle the value. However, if you pass Cert-C an extension it does not recognize, then Cert-C does not know how to handle the value you passed. Therefore, you must tell Cert-C how to handle the input data, and you do this by building an extension handler.

Cert-C provides a default extension handler for each Cert-C-defined extension type; however, if you override a default extension handler or if you define a new extension type, you must provide the callback functions. The `EXTENSION_HANDLER` data structure contains pointers to callback functions for a particular extension type. For more information about `EXTENSION_HANDLER`, see the *API Reference*.

```
typedef struct EXTENSION_HANDLER {
  /* Allocate and add new value to the value list */
  int (*AllocAndCopy) (
    POINTER  *newValue,                          /* (out) New copy of value */
    POINTER  value                                    /* Value to be copied */
  );
```

```
  /* Delete value allocated by AllocAndCopy by freeing its storage */
  VALUE_DESTRUCTOR Destructor;
  /* Get value in encoded format */
  int (*GetEncodedValue) (
    LIST_OBJ       valueList,                    /* Values to be encoded */
    unsigned char  **der,                        /* (out) Encoded values */
    unsigned int   *derLen              /* (out) Length of encoded values */
  );

  /* Decode the encoded value into components and save */
  int (*SetEncodedValue) (
    LIST_OBJ               valueList,                 /* Decoded value(s) */
    unsigned char          *ber,            /* BER value(s) to be decoded */
    unsigned int           berLen,         /* Length of BER to be decoded */
    LIST_OBJ_ENTRY_HANDLER *listEntryHandler         /* List entry handler */
  );
} EXTENSION_HANDLER;
```

The following table lists the four callback functions that you must provide for each extension type, and the Cert-C functions that call each callback function:

Table 15-1 **Cert-C Functions that Call the Extension-Handler Callback Functions**

Callback Function	Cert-C Functions that Call the Callback
AllocAndCopy	C_AddExtensionValue
Destructor	C_DeleteExtensionValue
GetEncodedValue	C_GetEncodedExtensionValue
SetEncodedValue	C_SetEncodedExtensionValue
	C_SetExtensionsObjectBER
	C_SetExtensionBER

To override only one callback in a handler, use the C_GetExtensionTypeInfo function to get a copy of the default handler. Overwrite the target callback; then call the C_RegisterExtensionType function to override the default handler.

The four callback functions that you provide help Cert-C to execute the five-step process of building or reading an object. For more information about the five-step process for building an object, see “Producing Information” on page 64. For more information about the five-step process for reading an object, see “Reading Information” on page 64.

Look at the five steps in building an object, when you use an extension handler.

1. Create an object—at this point Cert-C is not dealing with the actual data yet, so it does not need any help yet.
2. Enter the information—in this case it is the extension value. Since the information you enter is something Cert-C does not recognize, it needs help, so you provide `AllocAndCopy`.
3. Perform the operation—but you do not sign an extensions object, so no help is needed.
4. Retrieve the information in DER format—but with the user-defined extension, Cert-C does not know how to DER encode the value. You need to provide `GetEncodedValue`. Cert-C does know how to DER-encode the rest of the information such as the extension type, the criticality, and where to put identifying bytes.
5. Destroy the object—Cert-C needs to destroy all the information in the object and then the object itself. You tell Cert-C how to destroy the extension value with `Destructor`. Cert-C can destroy the rest of the object.

Look at the five steps in reading an object, when you use an extension handler.

1. Create an object—once again, Cert-C does not need any help in creating the object.
2. Set the object with the information in BER format—because Cert-C does not know the way to BER encode the value of your extension, you need to provide `SetEncodedValue`.
3. Read the information—Cert-C simply returns a pointer to the beginning of the value. It can do that without knowing what the value is or how it is formatted, so it does not need help with this task.
4. Perform the operation—but there is no signature to verify, so no help is needed.
5. Destroy the object—Cert-C needs to destroy all the information in the object and then the object itself. You tell Cert-C how to destroy the extension value with `Destructor`. Cert-C can destroy the rest of the object.

Building an Extension Handler

To build an extension handler, you need to write the four extension handler routines; `AllocAndCopy`, `Destructor`, `GetEncodedValue`, and `SetEncodedValue`. In this example, you write the four routines for an employee-number extension type.

First you need to define the form of an employee number. For examples of what an extension-type form looks like, see the `AUTHORITY_KEY_ID` or `KEY_USAGE` data type in

the *API Reference*. In this example, the employee number will be defined as follows:

```
typedef struct {
  unsigned char *empNumber;
  unsigned int empNumberLen;
  unsigned char *acctngCode;
  unsigned int acctngCodeLen;
} EMPLOYEE_NUMBER;
```

The *empNumber* value is the actual employee number. The *acctngCode* value is an accounting code for bookkeeping purposes.

Writing the AllocAndCopy Routine

When you enter an employee number and accounting code, Cert-C will want to copy that information and place it into an extensions object. You need to write an `AllocAndCopy` routine so that Cert-C can do this. The following example code shows how you can write an `AllocAndCopy` routine for an employee number. The `T_malloc` `T_memset`, `T_memcopy`, and `T_free` functions are Cert-C System service provider functions. For more information about these functions, see the "Service Provider" section of the *API Reference*.

Using `T_malloc`, you allocate a block of memory, the size of `EMPLOYEE_NUMBER`, to *newData*. Using `T_memset`, you set *newData*; the first *sizeof* bytes of *newData* are set to `0`.

You set *source* to *data* and *destination* to *newData*. Both *source* and *destination* now point to two different `EMPLOYEE_NUMBER` structures. You allocate a block of memory, the size of *source.empNumber*, to *destination.empNumber*.

You set *destination.empNumberLen* to the same length as *source.empNumberLen*. Then you copy the first *source.empNumberLen* bytes of *source.empNumber* to *destination.empNumber*.

Finally, you do the same for the accounting code, *acctngCode*, as you did for the employee number, *empNumber*. If an error occurs, the `T_free` function frees the *destination* block of memory and sets *newData* to `NULL_PTR`.

```
int EmpNumAllocAndCopy (POINTER *newData, POINTER data)
{
```

```
  int status = 0;
  EMPLOYEE_NUMBER *source, *destination;

  *newData = T_malloc (sizeof (EMPLOYEE_NUMBER));
  if (*newData == NULL_PTR)
    return (E_ALLOC);
  T_memset (*newData, 0, sizeof (EMPLOYEE_NUMBER));

  source = (EMPLOYEE_NUMBER *)data;
  destination = (EMPLOYEE_NUMBER *)(*newData);

  do {
    destination->empNumber =
      T_malloc (source->empNumberLen);
    if (destination->empNumber == NULL_PTR) {
      status = E_ALLOC;
      break;
    }
    destination->empNumberLen = source->empNumberLen;
    T_memcpy (destination->empNumber, source->empNumber,
              source->empNumberLen);

    destination->acctngCode =
      T_malloc (source->acctngCodeLen);
    if (destination->acctngCode == NULL_PTR) {
      status = E_ALLOC;
      break;
    }
    destination->acctngCodeLen = source->acctngCodeLen;
    T_memcpy (destination->acctngCode, source->acctngCode,
              source->acctngCodeLen);
  } while (0);

  if (status !=0) {
    T_free (destination->empNumber);
    T_free (destination->acctngCode);
    T_free (*newData);
    *newData = NULL_PTR;
  }

  return (status);
}
```

Writing the Destructor Routine

You also need to tell Cert-C how to destroy the EMPLOYEE_NUMBER user-defined extension information. To do this, you need to write a Destructor routine; the following example code shows how. The T_free function is a Cert-C System service-provider function. For more information about this function, see the "Service Provider" section of the *API Reference*.

```
void EmpNumDestructor (POINTER data)
{
  T_free (((EMPLOYEE_NUMBER *)data)->empNumber);
  T_free (((EMPLOYEE_NUMBER *)data)->acctngCode);
  T_free (data);
  return;
}
```

Writing the GetEncodedValue Routine

Cert-C is able to get the user-defined extension information in encoded form, except for the actual value. You need to write a GetEncodedValue routine to tell Cert-C how to encode the value.

RFC 2459, contains the following ASN.1 definition for each extension in an extensions object:

```
Extension ::= SEQUENCE {
  extnId      EXTENSION.&id ({ExtensionSet}),
  critical    BOOLEAN DEFAULT FALSE,
  extnValue   OCTET STRING }
               -- contains a DER encoding of a value of type
               -- &ExtnType for the
               -- extension object identified by extnId --
```

The output of the GetEncodedValue routine is placed into the contents of the extnValue OCTET STRING. In this example, you are not DER encoding any data for the user-defined Employee Number extension, so that raw data appears in the OCTET STRING. Otherwise, if you needed to place a more complicated DER-encoded value in the extnValue OCTET STRING, your implementation of GetEncodedValue must construct that DER-encoded value.

In this example, it is not necessary to define the type or the type length, or to determine how to DER or BER encode anything. All you need to do is put the value—the user-defined extension information in the EMPLOYEE_NUMBER structure—into a

series of bytes.

Later, you take this series of bytes and put it back into the regular format. In this example, you put the employee number into the following series of bytes.

```
A||B||C||D (where || means 'concatenate')
```

The values are defined as follows:

Table 15-2 **A||B||C||D Format**

Format	Size	Variable Name	Value
A	2 bytes	*empNumberLen*	The length of the employee number.
B	*empNumberLen* bytes	*empNumber*	The employee number.
C	2 bytes	*acctngCodeLen*	The length of the accounting code.
D	*acctngCodeLen* bytes	*acctngCode*	The accounting code.

To use the `GetEncodedValue` routine, the extension value must be in a `LIST_OBJ`. Cert-C builds this `LIST_OBJ` using the `AllocAndCopy` routine that you wrote for this extension handler and sets it using the `SetEncodedValue` routine, which you also write.

First, you get the extension value to be encoded using the `C_GetListObjectEntry` function. Since the employee-number type in this example takes only a single value, the index into the `LIST_OBJ` is 0 (zero). If the extension type takes multiple values, you need to devise a scheme that encodes them all. `GetEncodedValue` calls the `C_GetListObjectCount` and `C_GetListObjectEntry` functions to extract the extension value(s) to be encoded from the *`valueList`*.

Next, you allocate the space needed to store the encoded value. Using the `GetEncodedValue` routine, you calculate the length of the employee number and the accounting code and store the length in *`derLen`*. `T_malloc` then allocates a block of memory to store the encoded value and saves a pointer to this block in *`der`*. Cert-C frees up this space later, using the `T_free` function, so you must use the `T_malloc` function to allocate the space.

Finally, you copy the values into the A||B||C||D format using the `T_memcpy`

routine.

```
int EmpNumGetEncodedValue
  (LIST_OBJ valueList, unsigned char **der, unsigned int derLen)
{

  int status;
  UNIT2 temp2Bytes;
  unsigned int position;
  EMPLOYEE_NUMBER *value;

  /* Since the employee-number type takes only a single value, we know the
       index into the list object is 0.
  */
  if ((status = C_GetListObjectEntry
       (valueList, 0, (POINTER *)(&value))) !=0)
    return (status);

  /* The value is in the EMPLOYEE_NUMBER format; now you place it into
       the A||B||C||D format.
  */
  *derLen = value->empNumberLen +
            value ->acctngCodeLen + 4;
  *der = T_mallac (*derLen);
  if (*der == NULL_PTR)
    return (E_ALLOC);

  temp2Bytes = (UNIT2)(value->empNumberLen);
  T_memcpy ((*der), (unsigned char *)&temp2Bytes, 2);
  T_memcpy ((*der) +2, value->empNumber, value->empNumberLen);
  position = value->empNumberLen + 2;

  temp2Bytes = (UNIT2)(value->acctngCodeLen);
  T_memcpy ((*der) + position, (unsigned char *)&temp2Bytes, 2);
  T_memcpy ((*der) + position + 2, value->acctngCode,
              value->acctngCodeLen);

  return (0);
}
```

Writing the SetEncodedValue Routine

Cert-C does not know how to convert the extension type's encoded `OCTET STRING` back into an `EMPLOYEE_NUMBER` structure again. (`SetEncodedValue`'s *ber* input

parameter contains the value octets of this extension's OCTET STRING.) You need to write a SetEncodedValue routine to tell Cert-C how to put the encoded data into the proper form. You do not copy the data from the encoded series of bytes into an EMPLOYEE_NUMBER structure. Instead, you simply set the addresses in the EMPLOYEE_NUMBER structure to those positions in the encoded series of bytes. You can copy the lengths and add the value in EMPLOYEE_NUMBER form to the LIST_OBJ, which your AllocAndCopy routine created. You do this by calling the C_AddListObjectEntry function. Cert-C also passes the proper list-object handler to use. Cert-C builds it from the AllocAndCopy and Destructor routines in this extension handler. At this point, Cert-C copies the data and places it into the extensions object.

```
int EmpNumSetEncodedValue
  (LIST_OBJ valueList, unsigned char *ber, unsigned int berLen,
   LIST_OBJ_ENTRY_HANDLER *listEntryHandler)
{

  int status;
  UNIT2 numLen, codeLen;
  unsigned int position;
  EMPLOYEE_NUMBER employeeNumber;

  T_memcpy ((unsigned char *)&numLen, ber, 2);
  employeeNumber.empNumberLen = (unsigned int)numLen;
  employeeNumber.empNumber = ber + 2;
  position = (unsigned int)(numLen + 2);

  T_memcpy ((unsigned char *)&codeLen, ber + position, 2);
  employeeNumber.acctngCodeLen = (unsigned int)codeLen;
  employeeNumber.acctngCode = ber + position + 2;

  status = C_AddListObjectEntry
    (valueList, (POINTER)(&employeeNumber), (unsigned int *(NULL_PTR),
     listEntryHandler);

  return (status);
}
```

Now that you have an extension handler for EMPLOYEE_NUMBER; you can register the EMPLOYEE_NUMBER extension type.

Registering a User-Defined Extension

When you define an extension, you register the extension handler with a Cert-C context. Then, whenever you create an object, you pass this context so that Cert-C can handle that particular extension type when it appears later.

Building a Cert-C Context to Register a User-Defined Extension

To build a Cert-C context, you initialize and register the extension type. For more information about initializing a Cert-C context, see the example, "Initializing the Cert-C Context" on page 75. You are not required to register any special service providers to get the extension handlers; they are there by default.

Within a Cert-C context, you need to register each new extension type. You are going to add an extension by type and value. When you tell Cert-C what the type is, it will also need to know how to handle the value. Cert-C already recognizes the standard types and knows how to handle the values. Cert-C, however, does not recognize your type unless you register it and describe how to handle it.

Registering a User-Defined Extension

You can register as many extension types with a Cert-C context as you want. In this example, you only register one, the employee-number extension type. To register an extension type, you use the `C_RegisterExtensionType` function.

```
int C_RegisterExtensionType (
  CERTC_CTX            ctx,                    /* Cert-C context */
  EXTENSION_TYPE_INFO  *info                 /* Extension definition */
);
```

To register an extension type, you also use the `EXTENSION_TYPE_INFO` structure. All the extension-type information is passed to the Cert-C context through this structure. *`type`* specifies the type of extension you are registering and *`handler`* specifies the extension type's extension handler. You must initialize this structure when you register the

extension type.

```
typedef struct EXTENSION_TYPE_INFO{
  ITEM               type;                              /* Extension's OID */
  unsigned int       criticality;                 /* Extension's criticality */
  unsigned int       overrideCriticality;    /* Allow override criticality */
  unsigned int       overrideHandler;             /* Allow override handler */
  UINT2              authenObjects;             /* Objects that can include */
                                                          /* this extension */
  unsigned int       uniqueValue;                 /* If non-zero, extension */
                                                  /* can have only one value */
                                        /* if 0, can have multiple values */
  EXTENSION_HANDLER  handler;                          /* Extension's handler */
} EXTENSION_TYPE_INFO;
```

In this example, using the `C_RegisterExtensionType` function, you register the `EMPLOYEE_NUMBER` extension type. The first argument is the Cert-C context that you just initialized. The second argument passes all the user-defined extension information that describes the new extension type you are registering.

The first field of the `EXTENSION_TYPE_INFO` structure is an OID. The X.509 standard defines some OID types. For example, the OID for *`issuerAltName`* is the 3-byte sequence `{0x55, 0x1D, 0x08}`. You can set this OID to be whatever you want. It is advisable to make it several bytes to guarantee you avoid collision with the standard one. In this example, you set the OID to "`Employee Number`". That would be the byte sequence `{0x45, 0x6D, 0x70, ...}`.

The *`criticality`* field decides whether the certificate reader should accept or not accept the certificate, if they do not recognize the extension type and value. In this example, you set *`criticality`* to `NON_CRITICAL`.

The *`overrideCriticality`* field allows you to override the *`criticality`* field, or to make sure that the *`criticality`* field never gets changed. To allow future overrides, you set this field to `ALLOW_OVERRIDE_CRITICALITY`. To disallow future overrides, you set this field to 0 (zero). In this example, you set *`overrideCriticality`* to 0 `(zero)`.

The *`overrideHandler`* field allows you to specify an extension handler to override the default-extension handler in *`handler`*. At some time you might want to create an override-extension handler for this extension type. In that case, you would set this field to `ALLOW_OVERRIDE_HANDLER`. However, if you want this extension type's extension handler to be the only extension handler for this type, then set this field to 0 (zero). In this example, you set *`overrideHandler`* to 0 (zero).

The *`authenObjects`* field specifies the type of objects allowed with this extension type.

For a list of the object types you can set this field to, see the *API Reference*. In this example, you set this field to CERT_EXTENSIONS_OBJ.

The *uniqueValue* field specifies whether this extension type can have multiple values or not. For example, the type ET_ISSUER_ALTNAME can have multiple values. One can be the e-mail address, another a resource locator. In this example, you set this flag to 1. For multiple values, set this flag to another non-zero number.

Finally, you set *handler* to the new extension handler that you just created.

```
EXTENSION_HANDLER empNumExtensionHandler;
empNumExtensionHandler.AllocAndCopy = EmpNumAllocAndCopy;
empNumExtensionHandler.Destructor = EmpNumDestructor;
empNumExtensionHandler.GetEncodedValue = EmpNumGetEncodedValue;
empNumExtensionHandler.SetEncodedValue = EmpNumSetEncodedValue;

char *empNumType = "Employee Number";
EXTENSION_TYPE_INFO empNumInfo;

empNumInfo.type.data = (unsigned char *)empNumType;
empNumInfo.type.len = T_strlen (empNumType);
empNumInfo.criticality = NON_CRITICAL;
empNumInfo.overrideCriticality = 0;
empNumInfo.authenObjects = CERT_EXTENSIONS_OBJ;
empNumInfo.uniqueValue = 1;
empNumInfo.handler = empNumExtensionHandler;

status = C_RegisterExtensionType (ctx, &empNumInfo)
if (status != 0)
  goto CLEANUP;
```

Using a User-Defined Extension

Now that you have defined a user-defined extension type, written its extension handler, and initialized and registered it, you need to pass this Cert-C context when you create an object. For example, to use the user-defined extension when you create a certificate object, you do the following:

```
CERT_OBJ newCertObj = (CERT_OBJ)NULL_PTR;

if ((status = C_CreateCertObject (&newCertObj, ctx)) != 0)
  break;
```

Now whenever Cert-C sees the type "Employee Number", it knows what to do. For example, if you call the C_CreateExtension or C_AddExtensionValue functions with this type and a value in the form of an EMPLOYEE_NUMBER structure, Cert-C can perform these functions.

In Cert-C, whenever you create an object, you must also destroy it once it is no longer being used. As with objects, whenever you initialize a Cert-C context you must also finalize it when you no longer need it.

```
void C_FinalizeCertC (CERTC_CTX *ctx);
```

The Unknown Extension

It is possible that you could receive a certificate or CRL with an unknown extension. Cert-C is able to report the extension type and criticality and it can access the value. However, you must pass the proper data structure, into which Cert-C can format the value. Cert-C must know in advance how to properly format the value into that passed data structure. If the extension is a standard type, Cert-C uses a built-in handler. If this is a user-defined type, Cert-C uses the handler passed in through the application context. An unknown extension is an extension that is neither a registered (pre-defined) extension, nor a user-defined extension. If Cert-C does not have a handler for a particular type extension type, an unknown extension type, it saves the extension's value in an `ITEM` structure.

You call `C_GetExtensionCount` to find out how many extensions there are in the object. Then with a for-loop, you call `C_GetExtensionInfo` on each of the extensions. You now know the type of each extension, so you can call `C_GetExtensionValue` with the proper data structure for the value. You would do this in some sort of switch statement. For example, for `ET_ISSUER_ALTNAME`, use the `ALTERNATE_NAME` structure; for `ET_REASON_CODE`, use `unsigned int`, and so on.

If your application did not recognize the extension type, you still call the `C_GetExtensionValue`. The data structure you pass for the value is an `ITEM`. Cert-C uses the default handler that takes values and simply returns a pointer to the sequence of bytes in the encoding and the length.

The Unknown Critical Extension

When Cert-C encounters a certificate with a critical extension that is unknown to Cert-C, the function that attempted to parse the certificate returns the `E_UNKNOWN_CRITICAL_EXTENSION` error code. This is because you do not have an extension handler defined in the Cert-C context for that type of extension.

For Cert-C to be able to parse this extension, you must add a user-defined extension handler, which implements the callbacks to create and parse this extension value, to the Cert-C context.

Cert-C has a default handler that is used to handle unknown non-critical extensions. Unlike the other extension handlers, the default handler simply translates from BER to an `ITEM` structure or from an `ITEM` structure to DER. (The other extension handlers translate from BER to a C structure specific to the type of information contained in the extension value.)

In this example, you receive an unknown critical extension called ET_SOME_ARBITRARY_EXTENSION.

```
/* The OID indicating the extension type */
unsigned char ET_SOME_ARBITRARY_EXTENSION[] = {0xab, 0xcd, 0xef};
unsigned int ET_SOME_ARBITRARY_EXTENSION_LEN = sizeof
  (ET_SOME_ARBITRARY_EXTENSION);
```

You register an extension handler for the unknown critical extension, ET_SOME_ARBITRARY_EXTENSION, using the default-extension handler's routines. Assume that you have a properly initialized Cert-C context, CERTC_CTX *ctx*.

You call the C_GetExtensionTypeInfo function and pass the ET_UNKNOWN_TYPE object identifier as the extension type, and ET_UNKNOWN_TYPE_LEN as the length of the object identifier.

You set an EXTENSION_TYPE_INFO structure with the new extension-type information, which you are about to register. Set *type.data* to the unknown extension type ET_SOME_ARBITRARY_EXTENSION and *type.len* to ET_SOME_ARBITRARY_EXTENSION_LEN. Next you set the *criticality* flag to NON_CRITICAL.

Note: The reason you set the criticality of this unknown extension type to NON_CRITICAL is because you might receive another certificate with this same extension type; however, it might be set as non-critical. Setting it as non-critical now enables Cert-C to parse this extension when it is critical and also when it is non-critical. Otherwise, you application might receive a E_INVALID_CRITICALITY error code.

Using the C_RegisterExtensionType function, you register the unknown extension type. The second argument, *extTypeInfo*, passes all the user-defined extension information that describes the new extension type you are registering.

```
EXTENSION_TYPE_INFO extTypeInfo;

status = C_GetExtensionTypeInfo (ctx, ET_UNKNOWN_TYPE, ET_UNKNOWN_TYPE_LEN,
&extTypeInfo);
if (status != 0)
  goto CLEANUP;

extTypeInfo.type.data = ET_SOME_ARBITRARY_EXTENSION;
extTypeInfo.type.len = ET_SOME_ARBITRARY_EXTENSION_LEN;
extTypeInfo.criticality = NON_CRITICAL;
```

```
status = C_RegisterExtensionType (ctx, &extTypeInfo);
```

Overriding the Extension Handler

You can override any default or user-defined extension handler. For example, you would override an extension handler when you do not want to DER encode information. Instead, you might want to use your own encoding scheme. To override the extension handler, you register the new extension handler with the application context. You can also override a handler when you call the `C_CreateExtension` function. The default handler is replaced only for that extension.

You can also call a default extension handler and replace only one or two components of that handler. To do this, you call the `C_GetExtensionTypeInfo` function and replace the specified components.

There is one exception. When you register a user-defined extension, you have the option of setting a flag that disallows overriding the handler. If you try to override such an extension, it will be unsuccessful.

For an example, see the `samples/cert/saltname.c` sample program. This sample overrides the default handles for the subject alternative name extension.

Appendix A

Using BSAFE Crypto-C

Crypto-C Model

Whatever you do in Crypto-C, you do from an object. To manipulate Crypto-C objects, you generally follow a six-step process.

1. Create
2. Set
3. Initialize
4. Update
5. Final
6. Destroy

In the *Crypto-C API Reference* you can find the functions corresponding to the six steps. For instance, to create, call a `B_Create` routine—either `B_CreateAlgorithmObject` or `B_CreateKeyObject`. To set, call a `B_Set` routine. To initialize, you call a `B_*Init`, where the * indicates the function you want to perform. For example, to generate a key pair, you would call `B_GenerateInit`; for random algorithms, it is `B_RandomInit`. The last three steps are similar.

During the set routine, you will enter an Algorithm Info type (AI) or Key Info type (KI) and special information (if any) the AI or KI needs. An AI or KI is simply the description of what function the Crypto-C object is to perform. A list of AIs and KIs are in the *Crypto-C API Reference*, along with the special information required.

During the initialize routine, you need to pass an algorithm chooser. A chooser is a way for Crypto-C to know which code to link into the executable. This enables you to link in only the Crypto-C functionality you want, ignore the rest, and consequently keep code size down.

For a more comprehensive description of using Crypto-C, see the *Crypto-C Developer's Guide*. For now, the following sections give code examples for some of the Crypto-C functions you will need to use Cert-C.

Key Object

Cert-C uses the Crypto-C key object, B_KEY_OBJ, to store and retrieve key information necessary to generate a certificate request. The certificate binds a name to a public key. You have already created the name (see"Creating a Name Object" on page 105), all you need now is the public key.

Use Crypto-C to generate an RSA public/private key pair. Then get the public key, using B_GetKeyInfo with the key information type KI_RSAPublicBER, and place the result into an ITEM structure. A more comprehensive description on how to use Crypto-C is in the *Crypto-C Developer's Guide*.

```
ITEM *bsafePublicKeyBER;

status = B_GetKeyInfo ((POINTER *)&bsafePublicKeyBER, publicKey,
                       KI_RSAPublicBER);
if (status != 0)
  goto CLEANUP;
```

You pass the address of a pointer to Crypto-C, which then places at that address a pointer. If you go to where that pointer points, you will find an ITEM that contains the BER encoding of the public key. That ITEM contains a pointer to memory owned by Crypto-C; you will not need to allocate or free this memory. When you destroy or modify the Crypto-C key object, the information will disappear. As long as the key object remains intact, you will be able to see the information. If you want to destroy the key object, save the information first in a file or in your own buffer.

Generating an RSA Key Pair

To generate an RSA key pair, you use a variation on the six-step process. There is no Update call. For more information about generating an RSA key pair, see the *Crypto-C Developer's Guide* or the `samples/keypair/keypair.c` sample program.

Step 1: Create

You build two key objects using an algorithm object. Therefore, there are three objects to create.

```
B_ALGORITHM_OBJ rsaKeyPairGenerator =(B_ALGORITHM_OBJ)NULL_PTR;
B_KEY_OBJ publicKey = (B_KEY_OBJ)NULL_PTR;
B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;

status = B_CreateAlgorithmObject (&rsaKeyPairGenerator);
if (status != 0)
  goto CLEANUP;

status = B_CreateKeyObject (&publicKey);
if (status != 0)
  goto CLEANUP;

status = B_CreateKeyObject (&privateKey);
if (status != 0)
  goto CLEANUP;
```

Step 2: Set

You need to set the algorithm object to `AI_RSAKeyGen`. You also need to pass some special information; the key size (measured in modulus bits) and the public exponent. This special information must be in the form of the following structure.

```
typedef struct {
  unsigned int modulusBits;
  ITEM publicExponent;
} A_RSA_KEY_GEN_PARAMS;
```

Most applications us `F4 = 65537 = 0x 01 00 01` as the public exponent. RSA Security recommends using a key size of at least 768 bits.

You do not need to set the key objects; that is what the key-pair generating object

does.

```
A_RSA_KEY_GEN_PARAMS keyGenParams;

unsigned char f4Data[3] = {
  0x01, 0x00, 0x01
}

keyGenParams.modulusBits = 1024;
keyGenParams.publicExponent.data = f4Data;
keyGenParams.publicExponent.len = 3;

status = B_SetAlgorithmInfo (rsaKeyPairGenerator, AI_RSAKeyGen,
                             (POINTER)&keyGenParams);
if (status != 0)
  goto CLEANUP;
```

Step 3: Init

You need an algorithm chooser containing the AM_RSA_KEY_GEN algorithm method. Also, at this point, Crypto-C is still ignoring the surrender context.

```
B_ALGORITHM_METHOD *RSA_KEY_GEN_CHOOSER[] = {
  &AM_RSA_KEY_GEN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

status = B_GenerateInit (rsaKeyPairGenerator, RSA_KEY_GEN_CHOOSER,
                         (A_SURRENDER_CTX *)NULL_PTR);
if (status != 0)
  goto CLEANUP;
```

Step 4: Update

There is no step 4 in generating an RSA key pair.

Step 5: Generate

In this function Crypto-C sets the key objects with proper RSA keys. To do this, Crypto-C needs a random algorithm object. Follow steps 1 through 4, described in the previous section, to build a random object.

This is a very time-consuming operation; at this point, you may find a surrender context useful. It is not required, so you may pass a properly cast NULL_PTR as well, and Crypto-C will never surrender control.

Since B_GenerateKeypair requires a random-algorithm object, be sure that an instance of a Cert-C Default Cryptographic service provider has been registered with the Cert-C context, then call C_GetRandomObject to get a reference to the random algorithm object in the given CERTC_CTX.

```
B_ALGORITHM_OBJ randomAlgorithm;

status = C_GetRandomObject (ctx, &randomAlgorithm);
if (status != 0)
  goto CLEANUP;

status = B_GenerateKeypair (rsaKeyPairGenerator, publicKey, privateKey,
                            randomAlgorithm, (A_SURRENDER_CTX *)NULL_PTR);
if (status != 0)
  goto CLEANUP;
```

Step 6: Destroy

Do not forget to destroy any object you create.

```
B_DestroyAlgorithmObject (&rsaKeyPairGenerator);
B_DestroyKeyObject (&publicKey);
B_DestroyKeyObject (&privateKey);
```

Getting Key Information Out of a Key Object

You will almost certainly want to save your keys. You cannot save a key object, so you must extract the information out of a key object. Do this using `B_GetKeyInfo`.

This routine will take a key object and return to you a pointer to the information in a form you indicate. The easiest way to save key information is in its BER-encoded format. An RSA key is made up of a number of components, modulus and exponent for a public key, and modulus, private exponent, two primes, two prime exponents, and a coefficient for a private key. The BER-encoding of a key combines all this information into one series of bytes.

You tell Crypto-C how you want the key formatted by passing a KI type. Refer to the *Crypto-C API Reference* for a description of all the KIs available. Each entry describes how to use the KI in a call to `B_GetKeyInfo`.

```
ITEM *publicKeyBER, *privateKeyBER;

status = B_GetKeyInfo ((POINTER *)&publicKeyBER, publicKey,
                       KI_RSAPublicBER);
if (status != 0)
  goto CLEANUP;

status = B_GetKeyInfo ((POINTER *)&privateKeyBER, privateKey,
                       KI_PKCS_RSAPrivateBER);
if (status != 0)
  goto CLEANUP;
```

At this point, you have a pointer to the key data, not the key data itself. Crypto-C returned a pointer that points to the location inside the key object where you can go to find the information. Once you alter or destroy the key object, that pointer is no longer valid. You must copy the key data into your own buffer, file or database before destroying the key object.

In addition, the key data is in cleartext. It is not encrypted or protected in any way. That is not of concern with the public key, but the private key must remain private. You must save that key data in protected format. Your method may be a tamper-resistant hardware device. You may want to encrypt the key using a password-based algorithm or export using PKCS #12 (see the *Crypto-C Developer's Guide* for more information on this topic). See the `samples/keypair/keywrap.c` sample program for an example that encrypts the private key with a password-based algorithm.

Setting a Key Object

There are times in Cert-C when you need a key in an object. Immediately after you have generated a key pair, they are in objects, but you cannot save them in object format. That means you must be able to build a key object from the key data. Do this using B_SetKeyInfo.

```
B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;

unsigned char *pbPrivateKeyData = NULL_PTR;
unsigned char *privateKeyDataBER = NULL_PTR;
unsigned int pbPrivateKeyDataLen, privateKeyDataBERLen;

ITEM privateKeyItem;

status = RecallPBKeyData (&pbPrivateKeyData, &pbPrivateKeyDataLen);
if (status != 0)
  goto CLEANUP;

status = PBUnprotectPrivateKey (&privateKeyDataBER, &privateKeyDataBERLen,
                                pbPrivateKeyData, pbPrivateKeyDataLen);
if (status != 0)
  goto CLEANUP;

privateKeyItem.data = privateKeyDataBER;
privateKeyItem.len = privateKeyDataBERLen;

status = B_CreateKeyObject (&privateKey);
if (status != 0)
  goto CLEANUP;

status = B_SetKeyInfo (privateKey, KI_PKCS_RSAPrivateBER,
                       (POINTER)&privateKeyItem);
if (status != 0)
  goto CLEANUP;
```

The RecallPBKeyData and PBUnprotectPrivateKey routines are not Crypto-C calls; they are there as placeholders to indicate that you get your key data out of storage. The data may be stored with password-based protection, in which case you need to decrypt the data.

Appendix B

BCERT Compatibility

This section summarizes compatibility issues to consider when migrating BCERT applications to Cert-C. You can look at the `bcdemo` program as an example of a BCERT application that has been modified to use Cert-C.

BCERT Backward Compatibility

Minimizing the effort required for BCERT customers to migrate to Cert-C was an important criterion when designing Cert-C. Therefore, the Cert-C API is completely backward-compatible with the BCERT API. This means that at a minimum, a BCERT application only needs to be recompiled (with the compiler referencing the set of header files included with Cert-C) and relinked to produce the new executable, which uses Cert-C instead of BCERT. You do not need to make any changes to the source code, because the necessary header file names have been preserved and modified appropriately.

Recompiling and relinking is all you need to do if you want to run your BCERT applications with Crypto-C 6.1 or later. An example of a BCERT application rebuilt to use Cert-C, with no changes of consequence to the source code, is discussed later in this document. See “An Example: bcdemo” on page 304, where the `_BCERT_API_` macro is discussed.

Even though the Cert-C API is compatible with the BCERT API, a BCERT application must be recompiled in order to use Cert-C. This is because in some cases, function definitions had to be changed in order to maintain backward-compatibility. As an example, consider the `C_SignCert` function. In the BCERT API, the function prototype is as follows:

```
int C_SignCert (
  CERT_OBJ          certObj,
  B_KEY_OBJ         privateKey,
  B_ALGORITHM_OBJ   randomObject,
  A_SURRENDER_CTX   *surrenderContext);
```

The last two arguments are not necessary when using the Cert-C model. This is because the random object and surrender context are now services that are registered and accessible by Cert-C in the Cert-C context. In order to support the old BCERT API and the suggested use in the Cert-C model, the `C_SignCert` prototype has been changed in Cert-C as follows:

```
int C_SignCert (
  CERT_OBJ          certObj,
  B_KEY_OBJ         privateKey,
  …);
```

See “API Modifications/Updates” on page 297 for more information regarding other

API changes.

API Modifications/Updates

In this section, we detail portions of the BCERT API that have been changed or deprecated in Cert-C. (The deprecated BCERT APIs can still be used in the current version of Cert-C, but we recommend that no new applications be developed that use deprecated APIs). For more details on the Cert-C APIs, see the *API Reference*.

Data Structures

BCERT_VERSION

This string was used to hold the BCERT version number in a form suitable for printing out (see `samples/bcdemo/source/demo.c`).

APPL_CTX

The use of the `APPL_CTX` has been deprecated. Instead, use the `CERTC_CTX` to store data related to an application's working environment, such as service-provider instances.

CERT_REQUEST_OBJ, CERT_REQUEST_VERSION_1, CERT_REQUEST_VERSION_2, CERT_REQUEST_FIELDS, DEFAULT_CERT_REQUEST_VERSION

These `CERT_REQUEST_*` data structures have been deprecated. We recommend that you use their `PKCS10_*` counterparts to avoid confusion between the higher-level PKI certificate requests and the lower-level PKCS #10 certificate requests.

Recommended replacements for the data structures are shown in the following table.

Note: `CERT_REQUEST_VERSION_2` was not supported in BCERT, but was present in the header file. Therefore, no replacement for `CERT_REQUEST_VERSION_2` is given.

Table A-1 **Deprecated Structures and Their Recommended Replacements**

Deprecated Structure	Recommended Replacement
`CERT_REQUEST_OBJ`	`PKCS10_OBJ`
`CERT_REQUEST_VERSION_1`	`PKCS10_VERSION_1`

Table A-1 **Deprecated Structures and Their Recommended Replacements**

Deprecated Structure	Recommended Replacement
CERT_REQUEST_FIELDS	PKCS10_FIELDS
DEFAULT_CERT_REQUEST_VERSION	DEFAULT_PKCS10_VERSION

ET_POLICY_CONSTRAINTS, ET_POLICY_CONSTRAINTS_LEN, POLICY_CONSTRAINTS

The implementation of the Policy Constraints extension in BCERT was an implementation of a deprecated version of the extension; it is no longer present in the X.509 standard. For information on the updated implementation, see the *API Reference* for information about ET_POLICY_CONSTRAINTS_36, ET_POLICY_CONSTRAINTS_36_LEN, and the POLICY_CONSTRAINTS_36 structure.

Functions

PKCS #10 Certificate Requests

The following BCERT procedures have been deprecated to avoid confusion that may arise because of the general term "CertRequest" in the function name. They are replaced with procedures that contain the term "PKCS10" to differentiate the functions that operate on the lower-level PKCS #10 objects versus the functions that operate on the higher-level PKI Certificate Requests.

```
int C_CreateCertRequestObject (
  CERT_REQUEST_OBJ  *certRequestObject);

void C_DestroyCertRequestObject (
  CERT_REQUEST_OBJ  *certRequestObject);

int C_GetCertRequestFields (
  CERT_REQUEST_OBJ       certRequestObject,
  CERT_REQUEST_FIELDS    *certRequestFields);

int C_SetCertRequestFields (
  CERT_REQUEST_OBJ       certRequestObject,
  CERT_REQUEST_FIELDS    *certRequestFields);

int C_GetCertRequestDER (
  CERT_REQUEST_OBJ       certRequestObject,
  unsigned char          **der,
  unsigned int           *derLen);
```

```
int C_SetCertRequestBER (
  CERT_REQUEST_OBJ       certRequestObject,
  unsigned char          *ber,
  unsigned int           berLen);

int C_SignCertRequest (
  CERT_REQUEST_OBJ       certRequestObject,
  B_KEY_OBJ              subjectPrivateKey,
  B_ALGORITHM_OBJ        randomObject,
  int                    signatureAlgorithm,
  unsigned char          *digest,
  unsigned int           *digestLen,
  unsigned int           maxDigestLen,
  A_SURRENDER_CTX        *surrenderContext);

int C_VerifyCertRequestSignature (
  CERT_REQUEST_OBJ       certRequestObject,
  unsigned char          *digest,
  unsigned int           *digestLen,
  unsigned int           maxDigestLen,
  A_SURRENDER_CTX        *surrenderContext);
```

Instead, use the following procedures, respectively:

```
int C_CreatePKCS10Object(
  CERTC_CTX         ctx,
  PKCS10_OBJ       *pkcs10Object);

void C_DestroyPKCS10Object(
  PKCS10_OBJ       *pkcs10Object);

int C_GetPKCS10Fields(
  PKCS10_OBJ        pkcs10Object,
  PKCS10_FIELDS    *pkcs10Fields);

int C_SetPKCS10Fields(
  PKCS10_OBJ        pkcs10Object,
  PKCS10_FIELDS    *pkcs10Fields);

int C_GetPKCS10DER(
  PKCS10_OBJ        pkcs10Object,
  unsigned char   **der,
  unsigned int     *derLen);
```

```
int C_SetPKCS10BER(
  PKCS10_OBJ        pkcs10Object,
  unsigned char    *ber,
  unsigned int      berLen);

int C_SignPKCS10(
  PKCS10_OBJ        pkcs10Object,
  B_KEY_OBJ         subjectPrivateKey,
  int               signAlgorithmID);

int C_VerifyPKCS10Signature(
  PKCS10_OBJ        pkcs10Object);
```

The following BCERT API has also been deprecated:

```
int C_DecomposePKCSCertRequestBER (
  CERT_OBJ               certObject,
  ATTRIBUTES_OBJ         attributesObject,
  unsigned char          *certRequestBER,
  unsigned int           certRequestBERLen,
  unsigned char          *digest,
  unsigned int           *digestLen,
  A_SURRENDER_CTX        *surrenderContext);
```

A new Cert-C API call to replace the deprecated `C_DecomposePKCSCertRequestBER` has not been implemented. However, the `GetCertObjFromPKCS10BER` routine provided in `samples/bcdemo/source/fulfill.c` can be used as a replacement.

X.509 Certificates and CRLs

The "`Create`" functions for certificates and CRLs have been changed to take a `CERTC_CTX` instead of an `APPL_CTX`. This will compile with both old and new applications because both context definitions resolve to a `POINTER`. Internally, the implementation can detect which type of context was passed in, and operates accordingly, as follows:

- If the context parameter is an `APPL_CTX`, the BCERT 1.0x implementation is used to preserve old-style behavior.
- If the context parameter is a `CERTC_CTX`, the context is copied into the resulting certificate or CRL object, and used as needed for operations on certificates and CRLs, such as signing or verifying a signature.

Note that in the BCERT 1.0x implementation, it was acceptable to pass in a properly cast `NULL_PTR` for the `APPL_CTX` when creating certificate or CRL objects. When

switching over to use the Cert-C API, you must replace (APPL_CTX)NULL_PTR with the new CERTC_CTX that the application is using. Failure to do this will result in an application crash.

The following function declarations have been revised in Cert-C to take a CERTC_CTX, but will also accept a BCERT APPL_CTX for backward compatibility:

```
int C_CreateCertObject (
  CERT_OBJ          *certObj,
  CERTC_CTX          ctx);

int C_CreateCRLObject (
  CRL_OBJ           *certObj,
  CERTC_CTX          ctx);

int C_CreateExtensionsObject (
  EXTENSIONS_OBJ * extensionsObject,
  unsigned int     extensionsObjectType,
  CERTC_CTX        ctx);

int C_GetExtensionTypeInfo (
  CERTC_CTX         ctx,
  unsigned char     *type,
  unsigned int      typeLen,
  EXTENSION_TYPE_INFO *info);

int C_RegisterExtensionType (
  CERTC_CTX         ctx,
  EXTENSION_TYPE_INFO *info);
```

Operations that sign and verify certificates and CRLs are defined with variable parameter lists. Internally, two implementations are supported: the BCERT 1.0x style and the Cert-C style. The BCERT 1.0x style parameter list has a random object, when signing, and a surrender context. Since both of these are included in the CERTC_CTX, they are not needed as parameters for Cert-C-style calls. Internally, the implementation checks the certificate or CRL object. If it has a CERTC_CTX associated with it, the Cert-C-style implementation will be invoked. Otherwise, the BCERT 1.0x implementation will be executed.

The following function definitions are for signing and verifying signatures on

certificates and CRLs:

```
int C_SignCert (
  CERT_OBJ          certObj,
  B_KEY_OBJ         privateKey,
  …
);

int C_VerifyCertSignature (
  CERT_OBJ          certObj,
  B_KEY_OBJ         publicKey,
  …
);

int C_SignCRL (
  CRL_OBJ           crlObj,
  B_KEY_OBJ         privateKey,
  …
);

int C_VerifyCRLSignature (
  CRL_OBJ           crlObj,
  B_KEY_OBJ         privateKey,
  …
);
```

A variable parameter list is used for backward compatibility. Cert-C checks to see if the *certObj/crlObj* has a CERTC_CTX in it, which was placed there when the object was created. If it does not, the implementation knows that this is a BCERT 1.0x style call, and supports that parameter-list style. If there is a CERTC_CTX in the *certObj/crlObj*, the Cert-C style of parameter list is used, getting the random object and private key from the CERTC_CTX in the *certObj/crlObj*.

The two forms of the parameter list supported in the implementation are:

1. *BCERT 1.0x parameter list*: the implementation for this version is unchanged from BCERT 1.02. See the *BCERT 1.0 Library Reference Manual* for function/parameter list description.

```
int C_SignCert (
  CERT_OBJ          certObj
  B_KEY_OBJ         privateKey
  B_ALGORITHM_OBJ   randomObject
  A_SURRENDER_CTX   *surrenderContext);
```

```
int C_VerifyCertSignature (
  CERT_OBJ          certObj
  B_KEY_OBJ         publicKey
  A_SURRENDER_CTX   *surrender);

int C_SignCRL (
  CRL_OBJ           crlObj
  B_KEY_OBJ         privateKey
  B_ALGORITHM_OBJ   randomObject
  A_SURRENDER_CTX   *surrenderContext);

int C_VerifyCRLSignature (
  CRL_OBJ           crlObj
  B_KEY_OBJ         privateKey
  A_SURRENDER_CTX   *surrenderContext);
```

2. *Cert-C style parameter list*: the implementation uses the Cert-C context in the *certObj*/*crlObj* for the random object and the surrender context.

```
int C_SignCert (
  CERT_OBJ          certObj,
  B_KEY_OBJ         privateKey);

int C_VerifyCertSignature (
  CERT_OBJ          certObj,
  B_KEY_OBJ         publicKey);

int C_SignCRL (
  CRL_OBJ           crlObj,
  B_KEY_OBJ         privateKey);

int C_VerifyCRLSignature (
  CRL_OBJ           crlObj,
  B_KEY_OBJ         privateKey);
```

For C_VerifyCertSignature and C_VerifyCRLSignature, both E_PUBLIC_KEY and E_INVALID_SIGNATURE are returned. However, when there is a problem with the public key that is passed in, E_PUBLIC_KEY is now returned instead of E_INVALID_SIGNATURE. For example, when there is an invalid key length or when the key object is not set. Such problems are present when the following Crypto-C errors occur:

- BE_KEY_INFO
- BE_KEY_LEN
- BE_KEY_NOT_SET

- BE_KEY_OBJ
- BE_KEY_OPERATION_UNKNOWN
- BE_WRONG_KEY_INFO

Except for the preceding errors, E_INVALID_SIGNATURE is returned as before.

An Example: bcdemo

The source for the BCERT Command-Line demo program (bcdemo) is located in the samples/bcdemo directory. To examine the sources, build the program, or run the program within Microsoft Visual Studio, you can open the samples/make/build/samples.dsw workspace, which includes the project file for bcdemo.

This program is an example of an application originally written using BCERT that has been migrated to use Cert-C. It is possible to build and run the program using the original BCERT API only, or to build and run the program using the recommended Cert-C replacements for deprecated BCERT data types and function calls.

To build and run the program in its original form, as shipped with BCERT 1.02, be sure that the _BCERT_API_ preprocessor macro is defined. Most compilers have a -D option to enable you to do this. In Microsoft Visual Studio, go to Project | Settings | C/C++ | General | Preprocessor Definitions (making sure that the appropriate project and configuration on the left-hand side is selected) to list the preprocessor macros you want to define. If the _BCERT_API_ macro is not defined, bcdemo is built using the Cert-C API.

When building bcdemo with the preprocessor macro _BCERT_API_ defined, the code to be compiled is exactly the same as the source for bcdemo, which was shipped with BCERT 1.02, with the following exceptions:

With the use of Crypto-C 4.3 or later, the following preprocessor macro also had to be defined for Win32 platforms (other platforms have an analogous value):
RSA_PLATFORM=RSA_I386_486

- To access the T_time function (in stdlibrf), the following macro is also defined in the project settings: RSA_STD_TIME_FUNCS=RSA_ENABLED
- Line 28 of crl.c was changed from CRL_NUMBER to DEMO_CRL_NUMBER to avoid conflicts with the definition in the Cert-C include file, certext.h.
- Type casts were added on lines 280 and 289 of fulfill.c to silence compiler warnings.

- The statement "`return 0;`" has been added to line 59 of `myprint.c` to silence compiler warnings.
- The statement "`UNUSED_ARG (ioContext);`" has been added to line 99 of `myprint.c` to silence compiler warnings.

Be aware that when the _BCERT_API_ switch is turned on, a global random object is initialized with the same hard-coded seed (see `InitializeRandomObject` in `demo.c`) for demonstration purposes only. A BCERT application developer had to properly seed a random object. When the _BCERT_API_ switch is off, the `bcdemo` application uses the method provided by Cert-C to automatically seed a random-algorithm object (in the `CERTC_CTX`) properly.

If the `bcdemo` executable was built with the _BCERT_API_ switch on, the following header will be displayed:

```
BCERT Demo version 1.02
```

Otherwise, if the _BCERT_API_ switch was not turned on during compilation, the following header is instead displayed:

```
Cert-C Demo 1.0
```

The following description of the `bcdemo` program, from a user's perspective, is done with an executable built with the _BCERT_API_ switch turned on. However, the behavior of the `bcdemo` program is the same (from a user's perspective) whether the BCERT API or Cert-C API is used.

Note that the _BCERT_API_ macro is only used as part of the demo code to enable the original BCERT code and the Cert-C updates to coexist in the same file. None of the Cert-C provider code, toolkit code, or header files use this macro.

User's Guide for bcdemo

Introduction

The BCERT Command-Line Demo (bcdemo) is an example application that can generate and fulfill PKCS #10 certificate requests, and add users to certificate revocation lists (CRLs). The bcdemo program also utilizes X.509 v3 extensions in the certificates and CRLs that it generates.

Running the Demo

The bcdemo program is a menu-driven application. It prompts the user for commands and waits for the responses. The various commands and expected responses will be explained later. To start bcdemo, enter the following at the system prompt:

```
> bcdemo
```

You may also run bcdemo in a response-file mode, where you list the responses you would type at the menu prompts in a file so that bcdemo reads from this prepared file. For example, to read commands from a file named test.in, enter the following at the system prompt:

```
> bcdemo < test.in
```

Notice that this uses '<' to redirect test.in as the input to bcdemo.

For demonstration purposes, there is a default issuer "C = US, O = Example Issuer", and a default subject "C = US, O = Example Issuer, CN = Test User 1".

Using bcdemo

When you type bcdemo at the system prompt (make sure you are in the proper

directory), the following top-level menu appears:

```
BCERT Demo version 1.02
-----------------------------------
F - Fulfill PKCS Certificate Request
G - Generate PKCS Certificate Request
R - Revoke Certificate
Q - Quit
-------------------------------------
Enter command :
```

You may enter the uppercase or lowercase letter for the command you want. Only the first character of the command you type is checked. This is true for all of the commands. So, for example, to fulfill a request, you may type "f" or "F".

Each of the commands on this menu are described below.

Generate PKCS Certificate Request

To generate a PKCS certificate request, enter "g" or "G" at the menu. The bcdemo program will prompt you for the key size and the name of the file that contains the certificate request. The bcdemo program does the following:

- Generates an RSA key pair.
- Sets the subject name to a default subject name of "C = US, O = Example Issuer, CN = Test User 1".
- Sets the issuer name to the default issuer name "C = US, O = Example Issuer".
- Sets the public-key information to the BER encoding of the public key just generated.
- Signs the certificate request with the corresponding private key.
- Writes the certificate request to a file.

The following is an example of the “Generate PKCS Certificate Request” command:

Note: User's inputs are printed in bold.

```
BCERT Demo version 1.02
-----------------------------------
F - Fulfill PKCS Certificate Request
G - Generate PKCS Certificate Request
R - Revoke Certificate
Q - Quit
---------------------------------------
Enter command :
g

Please enter a key size between 512 and 2048:
1024

Generating RSA key pair (might take a while!)...
......

Signing the certificate request...

Certificate Request information:

Version: 0
Subject Name:
        C = US
        O = Example Issuer
        CN = Test User 1
Public Key Info:
30 81 9f 30 0d 06 09 2a 86 48 86 f7 0d 01 01 01
05 00 03 81 8d 00 30 81 89 02 81 81 00 d5 6f d7
5a e7 e3 eb 38 77 df 9b 1b 92 71 b6 65 cd b3 30
d6 2a 4a 47 6a d8 18 7b 10 b4 45 2f 6f 7c fc f1
71 e9 5d 7e 14 f7 83 74 5c 12 7e db 90 06 db 68
83 a1 77 88 1b ff 2e 6d cb 25 49 61 68 15 0b 4f
af 7a 7c 38 88 a3 46 f3 1c c9 e5 42 3b b6 35 27
1d 23 cb 3c 0f eb ae 39 2e 25 d0 8f 9e e5 9a 8c
44 a2 44 cc 42 6e 13 12 c6 1a 9f 79 3b e8 4d 69
83 a9 66 27 87 ff 0b 0f 6a 27 c2 7f 39 02 03 01
00 01
Attributes DER:
31 1e 30 1c 06 09 2a 86 48 86 f7 0d 01 09 05 31
0f 17 0d 39 32 30 31 32 32 31 39 31 34 32 35 5a

Please enter a filename to save the certificate request in:
cert.req
```

```
The certificate request has been generated successfully!
```

Fulfill PKCS Certificate Request

To fulfill a PKCS certificate request, enter "f" or "F" at the menu. The bcdemo program prompts you for the name that contains the certificate request and does the following:

- Validates the signature on the certificate request to make sure that the signed public key corresponds to the private key that is used to sign the certificate request.
- Decomposes a certificate request into an unsigned certificate object.
- Sets the issuer name to the default issuer name "C = US, O = Example Issuer".
- Sets the certificate validity to 1 year, valid 30 days from the current time.
- Sets the serial number.
- Adds the following x.509 v3 extensions:
  - User-Defined Extension Type: contains the signing time attribute.
  - Key Usage: allows the certificate to be used for digital signatures, but not certificate signing.
  - Private-Key Usage Period: the private key validity starts at November 6, 1996 9:06:21.001111 UTC time.
- Signs the certificate.
- Writes the fulfilled certificate to a file.

The following is an example of the "Fulfill PKCS Certificate Request" command:

Note: User's inputs are printed in bold

```
BCERT Demo version 1.02
-----------------------------------
F - Fulfill PKCS Certificate Request
G - Generate PKCS Certificate Request
R - Revoke Certificate
Q - Quit
-------------------------------------
Enter command :
f

Please enter the filename for the certificate request:
cert.req
```

```
Validating signature on the certificate request...

Fulfilled Certificate information:
Version: 2
Signature Algorithm: MD5 WITH RSA ENCRYPTION
Serial Number:
20 01 01 01 0b
Subject Name:
        C = US
        O = Example Issuer
        CN = Test User 1
Issuer Name:
        C = US
        O = Example Issuer
Validity Start:
8/31/1996 - 23:39:30
Validity End:
8/30/1997 - 23:39:29
Public Key:
30 81 9f 30 0d 06 09 2a 86 48 86 f7 0d 01 01 01
05 00 03 81 8d 00 30 81 89 02 81 81 00 d5 6f d7
5a e7 e3 eb 38 77 df 9b 1b 92 71 b6 65 cd b3 30
d6 2a 4a 47 6a d8 18 7b 10 b4 45 2f 6f 7c fc f1
71 e9 5d 7e 14 f7 83 74 5c 12 7e db 90 06 db 68
83 a1 77 88 1b ff 2e 6d cb 25 49 61 68 15 0b 4f
af 7a 7c 38 88 a3 46 f3 1c c9 e5 42 3b b6 35 27
1d 23 cb 3c 0f eb ae 39 2e 25 d0 8f 9e e5 9a 8c
44 a2 44 cc 42 6e 13 12 c6 1a 9f 79 3b e8 4d 69
83 a9 66 27 87 ff 0b 0f 6a 27 c2 7f 39 02 03 01
00 01

Certificate Extensions:

Type #1: User Defined Extensions
Criticality: TRUE
Extension Value:
31 1e 30 1c 06 09 2a 86 48 86 f7 0d 01 09 05 31
0f 17 0d 39 36 30 38 32 38 32 33 33 39 33 30 5a

Type #2: Key Usage
Criticality: FALSE
CF_DIGITAL_SIGNATURE: YES
CF_KEY_CERT_SIGN: NO
```

```
Type #3: Private Key Usage Period
Criticality: FALSE
Starting:
        Year: 1996   Month: 11   Day: 6   Hour: 21
        Minute: 6  Second: 27  Micro-second: 1111  Time zone: 0
Ending:
        Year: 1997   Month: 11   Day: 6   Hour: 21
        Minute: 16  Second: 27  Micro-second: 3333  Time zone: 0
Please enter a filename to save the approved certificate in:
cert.apr
The certificate request has been fulfilled!
```

Revoke certificate

To revoke a certificate, enter "r" or "R" at the menu. The bcdemo program prompts you for the name of the file that contains the certificate to be revoked and does the following:

- Reads in the certificate from file.
- Creates a CRL reason code extension for revoking the certificate.
- Adds the certificate's serial number to the issuer's CRL.
- Updates the last and next update period.
- Signs the CRL with the issuer's private key.
- Writes the CRL to a file.

The following is an example of the "Revoke Certificate" command:

Note: User's inputs are printed in bold.

```
BCERT Demo version 1.02
-----------------------------------
F - Fulfill PKCS Certificate Request
G - Generate PKCS Certificate Request
R - Revoke Certificate
Q - Quit
-----------------------------------
Enter command :
r

Please enter the filename for the certificate to revoke:
cert.apr
```

```
Enter reason for revoking the certificate.

        0 - Unspecified Reason
        1 - Key Compromise
        2 - CA Compromise
        3 - Affiliation Change
        4 - Superseded
        5 - Cessation of Operation
        6 - Hold Certificate
        8 - Remove Certificate From CRL

Please select a number:
5

Please enter the filename to save the new CRL in:
cert.crl

The certificate has been revoked!
```

Quit

Enter "q" or "Q" to quit bcdemo.

Programmer's Guide for bcdemo

The C source code files for the BCERT Command-Line Demo (`bcdemo`) are provided with the toolkit in the `samples/bcdemo/include` and `samples/bcdemo/source` directories. The source files are:

crl.c and crl.h

These files contain the utilities to extract certificate information, to add entries along with the revocation reason to an existing CRL, and to sign the CRL.

dchooser.c and dchooser.h

These files contain the algorithm methods for random number generation and RSA key generation.

demo.c and demo.h

These files contain the main function, the menu, and other user prompts `demo.c` uses the standard C library functions such as `printf()`, `fopen()`, etc.

dmenu.c and dmenu.h

These files contain the utilities to display the menu, get menu selections, and route these commands to the corresponding utility for execution.

dtime.c and dtime.h

These files contain routines to convert seconds since 1970 into day, month, year, hour, minute, and second.

dutil.c and dutil.h

These files contain utilities to make name objects for the default issuer and default subject names, to read from and write to files, and to convert error codes to error messages.

exten.c and exten.h

These files contain utilities to create extension objects and add extension values to

these objects.

fulfill.c and fulfill.h

These files contain utilities to decompose information from a certificate request into an unsigned certificate object, add default-supported X.509 v3 extensions, and sign the certificate.

genreq.c and genreq.h

These files contain utilities to generate RSA key pairs, generate certificate requests for a default subject, and sign requests.

inoutcl.c and inoutcl.h

These files contain utilities to read input and write output either from the standard I/O or from files.

myprint.c and myprint.h

These files contain utilities to print the contents of certificate objects and certificate-request objects, and X.509 v3 extensions.

simpleio.c and simpleio.h

These files contain I/O interface routines such as the `IO_CTX Read()` callback.

The routine `SimpleWriteText()` uses the constant `SKIP_CR` or `SKIP_LF`. `SimpleWriteText()` is used to handle output for `"text"` files properly, by converting CR/LF end-of-line delimiters to the local format. For example, on UNIX, define `SKIP_CR=1` in order to remove carriage return characters, leaving only line feed as the delimiter. The default is not to skip CR and LF, leaving them both in as delimiters.

If any of these values need to be changed from the default, a compiler flag can set them. For example:

```
-DSKIP_CR=1
```

surrende.c and surrende.h

These files contain the surrender context, which is used by Cert-C to surrender control

to the application during a lengthy operation such as RSA key-pair generation. A surrender function callback is included, which will output a `'.'` (period) to the screen each time the application gets control from Cert-C.

userextn.c and userextn.h

These files contain a user-defined extension, which may be added to an *`extensionsObject`* of type `CRL_EXTENSIONS_OBJ` or `CRL_EXTENSIONS_ENTRY_OBJ`. Also, they include callbacks for the extension handler. They demonstrate how an application may create its own user-defined extensions and incorporate them into the Cert-C extensions processing engine.

Appendix C

References

The references contained in this appendix may not be exhaustive. Check the RSA Laboratories Web site (`http://www.rsasecurity.com/rsalabs/`) for references to additional standards. For explanations of standards relevant to Cert-C and certification, search the RSA Laboratories' FAQs (`http://www.rsasecurity.com/rsalabs/faq/`).

ITU Recommendations

The ITU-T texts are available only through subscription. See the International Communication Union (ITU) Web site for more information (`http://www.itu.int/`).

Recommendation	http://www.itu.int/itudoc/itu-t/rec/x/x500up/
Recommendation X.509: Information technology - Open Systems Interconnection - The Directory: Authentication framework	x509.html
Recommendation X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation	x680.html
Recommendation X.690 - Information technology - ASN.1 encoding rules - Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)	x690.html

RSA Laboratories provides an overview of ASN.1, which is entitled *A Layman's Guide to a Subset of ASN.1, BER, and DER*. See the RSA Laboratories Web site (`http://www.rsasecurity.com/rsalabs/pkcs/`).

PKCS

Some standards figure prominently in the Cert-C API in that Cert-C APIs are directly devoted to the implementation of these standards, such as PKCS #7, #10, #11, and #12. Some standards rely on other standards in the series. For instance, PKCS #7 relies upon PKCS #1, #5, and #9. The Cert-C API includes support for those standards as well.

For a complete description of the Public-Key Cryptography Standards, see the RSA Laboratories Web site (`http://www.rsasecurity.com/rsalabs/pkcs/`).

Standard	http://www.rsasecurity.com/rsalabs/pkcs/
RSA Encryption Standard	PKCS #1
Diffie-Hellman Key-Agreement Standard	PKCS #3
Password-Based Cryptography Standard	PKCS #5
Cryptographic Message Syntax Standard	PKCS #7
Private-Key Syntax	PKCS #8
Selected Attribute Types Standard	PKCS #9
Certification Request Syntax Standard	PKCS #10
Cryptographic Token Interface Standard	PKCS #11
Personal Information Exchange Syntax Standard	PKCS #12
Elliptic-Curve Cryptography Standard	PKCS #13
Cryptographic Token Information Format Standard	PKCS #15

PKIX

For additional information regarding ongoing Public-Key Infrastructure X.509 and PKIX drafts and standards, see the IETF Working Group, Public-Key Infrastructure (`http://www.ietf.org/html.charters/pkix-charter.html`).

RFCs

RFC Name	http://www.ietf.org/rfc.html
Internet X.509 Public-Key Infrastructure Certificate and CRL Profile	*RFC 3280*
Internet X.509 Public-Key Infrastructure Certificate and CRL Profile	*RFC 2459*
Internet X.509 Public-Key Infrastructure Certificate Management Protocols [CMP]	*RFC 2510*
Internet X.509 Public-Key Infrastructure Certificate Request Message Format [CRMF]	*RFC 2511*
Internet X.509 Public-Key Infrastructure Online Certificate Status Protocol - OCSP [OCSP]	*RFC 2560*

RFC Drafts

RFC Draft Name	http://www.ietf.org/ids.by.wg/pkix.html
Internet X.509 Public-Key Infrastructure Certificate Management Protocols [CMP]	draft-ietf-pkix-rfc2510bis-06
Transport Protocols for CMP	draft-ietf-pkix-cmp-transport-protocols-04
Internet X.509 Public-Key Infrastructure Certificate Request Message Format [CRMF]	draft-ietf-pkix-rfc2511bis-04

UTF-8

The Cert-C API conforms to *RFC 2279* when representing and requiring character strings. UTF-8 has the characteristic of preserving the full US-ASCII range. This provides compatibility with file systems, parsers, and other software that rely on US-ASCII values, but are transparent to other values.

RFC	http://www.ietf.org/rfc.html
UTF-8, a transformation format of ISO 10646	*RFC 2279*

For information on UNICODE standards, see the UNICODE Consortium Web site (`http://www.unicode.org`).

SCEP

The Cisco Systems' *Simple Certificate Enrollment Protocol* (SCEP) specification can be found at `http://www.cisco.com/warp/public/cc/pd/sqsw/tech/scep_wp.htm`. The Cert-C SCEP Database service provider and Cert-C SCEP PKI service provider APIs conform to this standard.

Index

D

E