PKCS #11: Cryptographic Token Interface Standard

An RSA Laboratories Technical Note
Version 1.0
April 28, 1995

RSA Laboratories
100 Marine Parkway
Redwood City, CA 94065 USA
(415) 595-7703
(415) 595-4126 (fax)
E-Mail: rsa-labs@rsa.com

Foreword

As public-key cryptography begins to see wide application and acceptance one thing is increasingly clear: If it is going to be as effective as the underlying technology allows it to be, there must be interoperable standards. Even though vendors may agree on the basic public-key techniques, compatibility between implementations is by no means guaranteed. Interoperability requires strict adherence to an agreed-upon standard format for transferred data.

Towards that goal, RSA Laboratories has developed, in cooperation with representatives of industry, academia and government, a family of standards called Public-Key Cryptography Standards, or PKCS for short.

PKCS is offered by RSA Laboratories to developers of computer systems employing public-key technology. It is RSA Laboratories' intention to improve and refine the standards in conjunction with computer system developers, with the goal of producing standards that most if not all developers adopt.

The role of RSA Laboratories in the standards-making process is four-fold:

1. Publish carefully written documents describing the standards.

2. Solicit opinions and advice from developers and users on useful or necessary changes and extensions.

3. Publish revised standards when appropriate.

4. Provide implementation guides and/or reference implementations.

During the process of PKCS development, RSA Laboratories retains final authority on each document, though input from reviewers is clearly influential. However, RSA Laboratories' goal is to accelerate the development of formal standards, not to compete with such work. Thus, when a PKCS document is accepted as a base document for a formal standard, RSA Laboratories relinquishes its "ownership" of the document, giving way to the open standards development process. RSA Laboratories may continue to develop related documents, of course, under the terms described above.

The PKCS family currently includes the following documents:

PKCS #1: RSA Encryption Standard. Version 1.5, November 1993.

PKCS #3: Diffie-Hellman Key-Agreement Standard. Version 1.4, November 1993.

PKCS #5: Password-Based Encryption Standard. Version 1.5, November 1993.

PKCS #6: Extended-Certificate Syntax Standard. Version 1.5, November 1993.

PKCS #7: Cryptographic Message Syntax Standard. Version 1.5, November 1993.

PKCS #8: Private-Key Information Syntax Standard. Version 1.2, November 1993.

PKCS #9: Selected Attribute Types. Version 1.1, November 1993.

PKCS #10: Certification Request Syntax Standard. Version 1.0, November 1993.

PKCS #11: Cryptographic Token Interface Standard. Version 1.0, April 1995.

PKCS documents are available by electronic mail to `<pkcs@rsa.com>`, or via anonymous ftp to `ftp.rsa.com` in the `pub/pkcs` directory. There is also a electronic mailing list for discussion of PKCS issues, `<pkcs-users@rsa.com>`; to join the list, send a request to `<pkcs-users-request@rsa.com>`.

Comments on the PKCS documents, requests to register extensions to the standards, and suggestions for additional standards are welcomed. Address correspondence to: PKCS Editor, RSA Laboratories, 100 Marine Parkway, Suite 500, Redwood City, CA 94065; 415/595-7703; fax: 415/595-4126; E-mail: `<pkcs-editor@rsa.com>`.

Acknowledgements

It is difficult to enumerate all the people whose contributions brought PKCS #11 to fruition, and any list will doubtless be incomplete. However, RSA Laboratories would particularly like to acknowledge those who provided feedback on the draft documents:

Richard Ankney, Fischer International

Ashar Aziz, Sun Microsystems Inc.

Ali Bahreman, Bellcore

Frank Balluffi, Bankers Trust

Sara Bitan, RadGaurd LTD.

Eric Blossom, COMSEC Partners

John C. Brainard, Security Dynamics

Liudvikas Bukys, University of Rochester

Steve Burnett, RSA Data Security, Inc.

Victor Chang, RSA Data Security, Inc.

Bruno Couillard, EMCON Ltd.

Greg Dunn, Telequip Corp.

Steve Dussé, RSA Data Security, Inc.

Alan Eldridge, IRIS Associates

Mark H. Etzel, AT&T Bell Laboratories

Bill Fox, National Semiconductor

Hazem Hassan, Datakey, Inc.

Thomas C. Jones, ViaCrypt

John Kennedy, Cylink

Larry Kilgallen, LJK Software

Kevin Kingdon, Novell

Scott Lindsay, Mobius

Roland Lockhart, BNR

Hoa Ly, RSA Data Security, Inc.

Thi Nguyen, Secure Communications Inc.

Denis Pinkas, Bull

Jim Press, ICL

P. Rajaram, Bellcore

William Rohland, Datakey, Inc.

Andrew Ryan, SmartDiskette Limited

Paul Schlyter, AU System

Wolfgang Schneider, GMD

Stephen Seal, CRYPTOCard

Don Stephenson, Sun Microsystems, Inc.

Bruno Struif, GMD

Mandan M. Valluri, National Semiconductor

Charlie Watt, SecureWare

David E. Wood, DataKey, Inc.

Richard R.D. Young, BNR

Arthur Zachai, representing Telequip Corp.

Neal Ziring, NSA

RSA Laboratories is grateful to those who assisted in other ways, and apologizes for any omissions. In all, the mailing list for PKCS #11 consisted of more than 130 people, and as the document was available on RSA’s ftp server, it is likely that that may others saw it as well.

PKCS #11's document editor was Aram Pérez of International Computer Services, under contract to RSA Laboratories; the project coordinator was Burt Kaliski of RSA Laboratories.

Table of Contents

List of Figures

List of Tables

1. Scope

This standard specifies an application programming interface (API), called "Cryptoki," to devices which hold cryptographic information and perform cryptographic functions. Cryptoki, pronounced "crypto-key" and short for "cryptographic token interface," follows a simple object-based approach, addressing 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."

This document specifies the data types and functions available to an application requiring cryptographic services using the ANSI C programming language. These data types and functions will be provided as a C header file by the supplier of a Cryptoki library. A separate document provides a generic, programming language independent Cryptoki interface. Additional documents will provide bindings between Cryptoki and other programming languages.

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. How Cryptoki provides this isolation is beyond the scope of this document, though some conventions for the support of multiple types of device will be addressed in a separate document.

The set of cryptographic mechanisms (algorithms) supported in this version is somewhat limited; but new mechanisms can easily be added without changing the general interface. It is expected that additional mechanisms will be published from time to time in separate documents. It is also possible for token vendors to define their own mechanisms (although for interoperability, registration through the PKCS process is preferable).

Cryptoki is intended for cryptographic devices associated with a single user, so some features that would be included in a general-purpose interface are omitted. For example, Cryptoki does not have a means of distinguishing multiple "users." The focus is on a single user's keys and perhaps a small number of public-key certificates related to them. Moreover, the emphasis is on cryptography. While the device may perform useful non-cryptographic functions, such functions are left to other interfaces.

2. References

ANSI C ANSI/ISO. *ANSI/ISO 9899-1990: American National Standard for Programming Languages -- C.* 1990.

ANSI X9.9 ANSI. *American National Standard X9.9: Financial Institution Message Authentication Code.* 1982.

ANSI X9.17 ANSI. *American National Standard X9.17: Financial Institution Key Management (Wholesale).* 1985.

ANSI X9.31 Accredited Standards Committee X9. *Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry: Part 1: The RSA Signature Algorithm.* Working draft, March 7, 1993.

ANSI X9.42 Accredited Standards Committee X9. *Public Key Cryptography for the Financial Services Industry: Management of Symmetric Algorithm Keys Using Diffie-Hellman.* Working draft, September 21, 1994.

CDPD Ameritech Mobile Communications et al. *Cellular Digital Packet Data System Specifications: Part 406: Airlink Security.* 1993.

FIPS PUB 46–2 National Institute of Standards and Technology (formerly National Bureau of Standards). *FIPS PUB 46-2: Data Encryption Standard.* December 30, 1993.

FIPS PUB 74 National Institute of Standards and Technology (formerly National Bureau of Standards). *FIPS PUB 74: Guidelines for Implementing and Using the NBS Data Encryption Standard.* April 1, 1981.

FIPS PUB 81 National Institute of Standards and Technology (formerly National Bureau of Standards). *FIPS PUB 81: DES Modes of Operation.* December 1980.

FIPS PUB 113 National Institute of Standards and Technology (formerly National Bureau of Standards). *FIPS PUB 113: Computer Data Authentication.* May 30, 1985.

FIPS PUB 180 National Institute of Standards and Technology. *FIPS PUB 180: Secure Hash Standard (SHS).* May 11, 1993. In May 1994, NIST announced a weakness in the Secure Hash Standard defined in FIPS 180; a revised version is expected to be issued as FIPS 180-1.

FIPS PUB 186 National Institute of Standards and Technology. *FIPS PUB 186: Digital Signature Standard.* May 19, 1994.

GCS-API X/Open Company Ltd. *Generic Cryptographic Service API (GCS-API), Base - Draft 2.* February 14, 1995.

ISO 7816-1 ISO. *International Standard 7816-1: Identification Cards — Integrated Circuit(s) with Contacts — Part 1: Physical Characteristics.* 1987.

ISO 7816-4 ISO. *Identification Cards — Integrated Circuit(s) with Contacts — Part 4: Inter-industry Commands for Interchange.* Committee draft, 1993.

ISO/IEC 9796	ISO/IEC. *International Standard 9796: Digital Signature Scheme Giving Message Recovery.* July 1991.
PCMCIA	Personal Computer Memory Card International Association. *PC Card Standard.* Release 2.1, July 1993.
PKCS #1	RSA Laboratories. *RSA Encryption Standard.* Version 1.5, November 1993.
PKCS #3	RSA Laboratories. *Diffie-Hellman Key-Agreement Standard.* Version 1.4, November 1993.
PKCS #7	RSA Laboratories. *Cryptographic Message Syntax Standard.* Version 1.5, November 1993.
RFC 1319	B. Kaliski. *RFC 1319: The MD2 Message-Digest Algorithm.* RSA Laboratories, April 1992.
RFC 1321	R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
RFC 1421	J. Linn. *RFC 1421: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures.* IAB IRTF PSRG, IETF PEM WG, February 1993.
RFC 1423	D. Balenson. *RFC 1423: Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers.* TIS and IAB IRTF PSRG, IETF PEM WG, February 1993.
RFC 1508	J. Linn. *RFC 1508: Generic Security Services Application Programming Interface.* Geer Zolot Associates, September 1993.
RFC 1509	J. Wray. *RFC 1509: Generic Security Services API: C-bindings.* Digital Equipment Corporation, September 1993.
X.208	ITU-T (formerly CCITT). *Recommendation X.208: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* 1988.
X.209	ITU-T (formerly CCITT). *Recommendation X.209: Specification of Abstract Syntax Notation One (ASN.1).* 1988.
X.500	ITU-T (formerly CCITT). *Recommendation X.500: The Directory – Overview of Concepts and Services.* 1988.
X.509	ITU-T (formerly CCITT). *Recommendation X.509: The Directory – Authentication Framework.* 1993. (Proposed extensions to X.509 are given in *ISO/IEC 9594-8 PDAM 1: Information Technology – Open Systems Interconnection – The Directory: Authentication Framework – Amendment 1: Certificate Extensions. 1994.*)

3. Definitions

For the purposes of this standard, the following definitions apply.

API	Application programming interface.
Application	Any computer program that calls the Cryptoki interface.
ASN.1	Abstract Syntax Notation One, as defined in X.208.
Attribute	A characteristic of an object.
BER	Basic Encoding Rules, as defined in X.209.
CBC	Cipher Block Chaining mode, as defined in FIPS PUB 81.
Certificate	A signed message binding a subject name and a public key.
Cryptographic Device	A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software only or a process on a server.
Cryptoki	The Cryptographic Token Interface defined in this standard.
Cryptoki library	A library that implements the functions specified in this standard.
DES	Data Encryption Standard, as defined in FIPS PUB 46-2.
DSA	Digital Signature Algorithm, as defined in FIPS PUB 186.
ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
MAC	Message Authentication Code, as defined in ANSI X9.9.
MD2	RSA Data Security, Inc.'s MD2 message-digest algorithm, as defined in RFC 1319.
MD5	RSA Data Security, Inc.'s MD5 message-digest algorithm, as defined in RFC 1321.
Mechanism	A process for implementing a cryptographic operation.
Object	An item that is stored on a token; may be data, a certificate, or a key.
PIN	Personal Identification Number.
RSA	The RSA public-key cryptosystem, as defined in PKCS #1.
RC2	RSA Data Security’s proprietary RC2 symmetric block cipher.

RC4	RSA Data Security's proprietary RC4 symmetric stream cipher.
Reader	The means by which information is exchanged with a device.
Session	A logical connection between an application and a token.
SHA	Secure Hash Algorithm, as defined in FIPS PUB 180.
Slot	A logical reader that potentially contains a token.
Subject Name	The X.500 distinguished name of the entity to which a key is assigned.
SO	A Security Officer user.
Token	The logical view of a cryptographic device defined by Cryptoki.
User	The person using an application that interfaces to Cryptoki.

4. Symbols and abbreviations

The following symbols are used in this standard:

Table 4-1, Symbols

Symbol	Definition
N/A	Not applicable
R/O	Read-only
R/W	Read/write

The following prefixes are used in this standard:

Table 4-2, Prefixes

Prefix	Description
C_	Function
CK_	Data type
CKA_	Attribute
CKC_	Certificate type
CKF_	Bit flag
CKK_	Key type
CKM_	Mechanism type
CKN_	Notification
CKO_	Object class
CKS_	Session state
CKR_	Return value
CKU_	User type
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pus	a pointer to a CK_USHORT
ul	a CK_ULONG
us	a CK_USHORT

In Cryptoki, a flag is a boolean flag that can be TRUE or FALSE. A zero value means the flag is FALSE, and a non-zero value means the flag is TRUE. Cryptoki defines these labels if they are not already defined.

```
#ifndef FALSE
#define FALSE 0
#endif
```

```
#ifndef TRUE
#define TRUE (!FALSE)
#endif
```

Cryptoki is based on ANSI C types and defines the following data types:

```
/* an unsigned 8-bit value */
typedef unsigned char CK_BYTE;

/* an unsigned 8-bit character */
typedef CK_BYTE CK_CHAR;

/* a BYTE-sized Boolean flag */
typedef CK_BYTE CK_BBOOL;

/* an unsigned value, at least 16 bits long */
typedef unsigned short int CK_USHORT;

/* an unsigned value, at least 32 bits long */
typedef unsigned long int CK_ULONG;

/* at least 32 bits, each bit is a Boolean flag */
typedef CK_ULONG CK_FLAGS;
```

Cryptoki also uses pointers to these data types which are implementation dependent. These pointers are:

```
CK_BYTE_PTR    /* Pointer to a CK_BYTE */
CK_CHAR_PTR    /* Pointer to a CK_CHAR */
CK_USHORT_PTR  /* Pointer to a CK_USHORT */
CK_VOID_PTR    /* Pointer to a void */

NULL_PTR       /* a NULL pointer */
```

It follows that many of the data and pointer types will vary somewhat from one environment to another (e.g., a CK_ULONG will sometimes be 32 bits, and sometimes perhaps 64 bits). However, these details should not affect the application, assuming it is compiled with a Cryptoki header file consistent with the Cryptoki library to which the application is linked.

All numbers and values expressed in this document are decimal, unless they are preceded by “0x”, in which case they are hexadecimal values.

The **CK_CHAR** data type holds characters from the following table, taken from ANSI C:

Table 4-3, Character Set

Category	**Characters**
Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Numbers	0 1 2 3 4 5 6 7 8 9
Graphic characters	! “ # % & ‘ ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
Blank character	‘ ‘

5. General overview

Portable computing devices such as smart cards, PCMCIA cards, and smart diskettes are ideal tools for implementing public-key cryptography, as they provide a way to store the private-key component of a public-key/private-key pair securely, under the control of a single user. With such a device, a cryptographic application, rather than performing cryptographic operations itself, programs the device to perform the operations, with sensitive information such as private keys never being revealed. As more applications are developed for public-key cryptography, a standard programming interface for the these devices becomes increasingly valuable. This standard addresses this need.

5.1 Design goals

Cryptoki was intended from the beginning as an interface between applications and all kinds of portable cryptographic devices, such as those based on smart cards, PCMCIA cards, and smart diskettes. There are already standards (de facto or official) for interfacing to these devices at some level. For instance, the mechanical characteristics and electrical connections are well-defined, as are the methods for supplying commands and receiving results. (See, for example, ISO 7816, or the PCMCIA specifications.)

What remained to be defined were particular commands for performing cryptography. It would not be enough simply to define command sets for each kind of device, as that would not solve the general problem of an *application* interface independent of the device. To do so is still a long-term goal, and would certainly contribute to interoperability. The primary goal of Cryptoki was a lower-level programming interface that abstracts the details of the devices, and presents to the application a common model of the cryptographic device, called a “cryptographic token” (or simply “token”).

A secondary goal was resource sharing. As desktop multi-tasking operating systems become more popular, a single device should be shared between more than one application. In addition, an application should be able to interface to more than one device at a given time.

It is not the goal of Cryptoki to be a generic interface to cryptographic operations or security services, although one certainly could build such operations and services with the functions that Cryptoki provides. Thus, Cryptoki is intended to complement, not compete with such emerging and evolving interfaces as “Generic Security Services Application Programming Interface” (RFC’s 1508 and 1509) and “Generic Cryptographic Service API” (GCS-API) from X/Open.

5.2 General model

Cryptoki's general model is illustrated in the following figure. The model begins with one or more applications that need to perform certain cryptographic operations, and ends with a cryptographic device, on which some or all of the operations are actually performed. A user may be associated with an application.

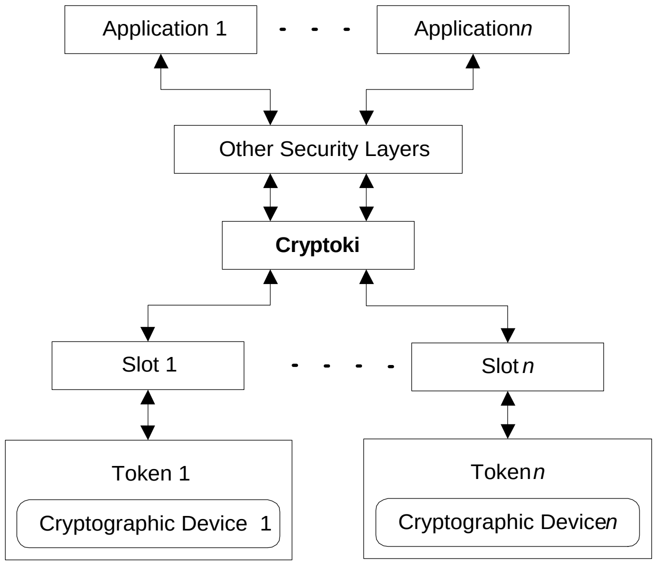

Figure 5-1, General Model

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of "slots". Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is "present in the slot" (typically) when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots and applications can connect to all those tokens.

A cryptographic device can perform some cryptographic operations, following a certain command set; these commands are typically passed through standard device drivers, for instance PCMCIA card services or socket services. Cryptoki makes the cryptographic device look logically like every other device, regardless of the implementation technology. Thus the application need not interface directly to the device drivers (or even know which ones are involved); Cryptoki hides these details. Indeed, the "device" may be implemented entirely in software, for instance as a process running on a server; no hardware is necessary.

Cryptoki would likely be implemented as a library supporting the functions in the interface, and applications would be linked to the library. An application may be linked to Cryptoki directly, or Cryptoki could be a so-called "shared" library (or dynamic link library), in which case the application would link the library dynamically. Shared libraries are fairly straightforward in operating systems such as Microsoft Windows™, OS/2™, and can be achieved, without too much difficulty, in Unix™ and DOS systems.

The dynamic approach would certainly have advantages as new libraries are made available, but from a security perspective, there are some drawbacks. In particular, if the library is easily replaced, then there is the possibility that an attacker can substitute a rogue library that intercepts a user's PIN. From a security perspective, direct linking would probably be better. However, whether the linking is direct or dynamic, the programming interface between the application and Cryptoki remains the same.

The kinds of devices and capabilities supported will depend on the particular Cryptoki library. This standard only specifies the interface to the library, not its features. In particular, not all libraries will support all the mechanisms (algorithms) defined in this interface (since not all tokens are expected to support all the mechanisms), and libraries will likely support only a subset of all the kinds of cryptographic devices that are available. (The more kinds, the better, of course, and it is anticipated that libraries will be developed supporting multiple kinds of token, not just those from a single vendor.) It is expected that as applications are developed that interface to Cryptoki, standard library and token "profiles" will emerge.

5.3 Logical view of a token

Cryptoki's logical view of a token is a device that stores objects and can perform cryptographic functions. Cryptoki defines three classes of object: Data, Certificates, and Keys. A data object is defined by an application. A certificate object stores a public-key certificate. A key object stores an encryption key. The encryption key be may a public key (RSA, DSA or Diffie-Hellman), a private key (RSA, DSA or Diffie-Hellman) or a secret key (RC2, RC4, DES, etc.). This view is illustrated in the following figure. The key types given are those supported for this version of Cryptoki; other key types may well be added in future versions.

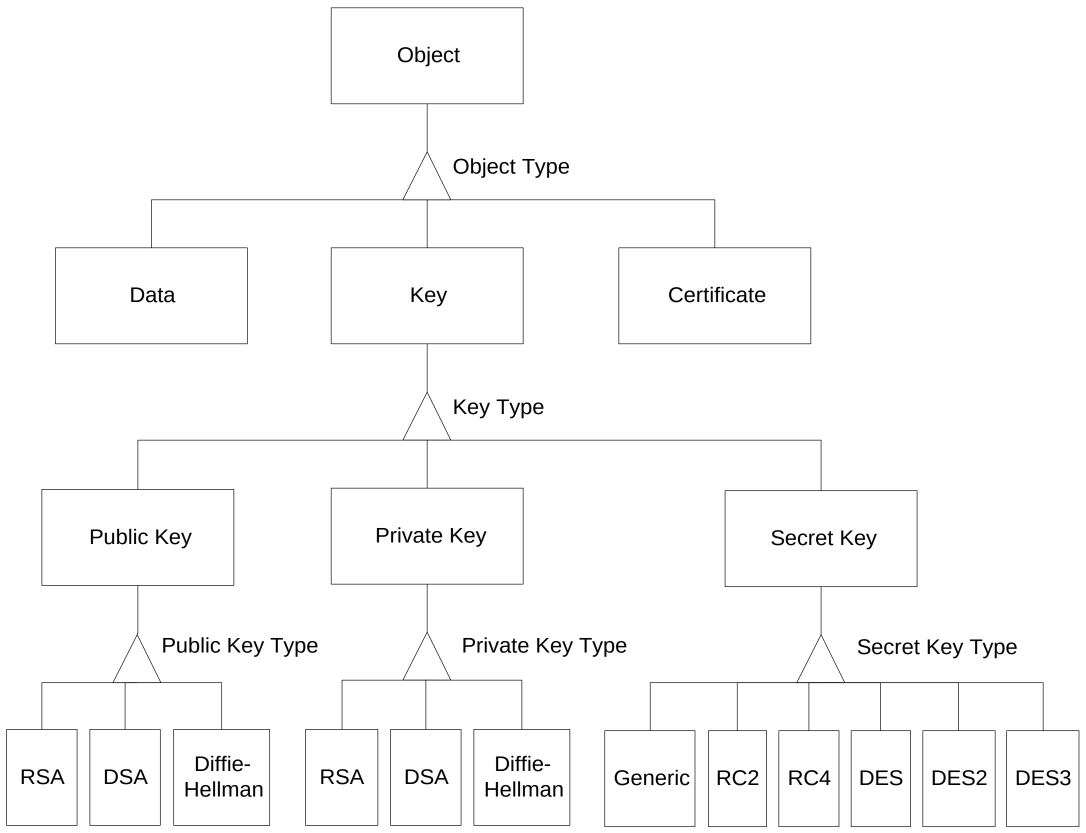

Figure 5-2, Object Hierarchy

Objects are also classified according to their lifetime and visibility. "Token objects" are visible to all applications connected to the token, and remain in the token after the "session" or connection between an application and the token is closed. "Session objects" are visible only to the application that creates them, and are destroyed automatically when the session is closed.

Further classification defines access requirements. "Public objects" are visible to all applications that have a session with the token. "Private objects" are visible to an application only after a user has been authenticated to the token by a PIN.

A token can create and destroy objects, manipulate them, and search for them. It can also perform cryptographic functions on objects. It is possible for the token to perform the cryptographic operations in parallel with the application, assuming the underlying device has its own processor. In addition, a token may have an internal random number generator.

It is important to distinguish between the logical view of a token and the actual implementation, because not all cryptographic devices will have this concept of "objects," or be able to perform every kind of cryptographic function. Many devices will simply have fixed storage places for keys of a fixed algorithm, and be able to do a limited set of operations. Cryptoki's role is to translate this into the logical view, mapping attributes to fixed storage elements and so on. Not all Cryptoki libraries and tokens need to support every object type. It is expected that standard "profiles" will be developed, specifying sets of algorithms to be supported.

"Attributes" are characteristics that distinguish an instance of an object. In Cryptoki, there are general attributes, such as whether the object is private or public. There are also attributes particular to an object, such as a modulus or exponent for RSA keys.

5.4 Users

This version of Cryptoki recognizes two token user types. One type is a Security Officer (SO). The other type is the normal user. Both types of user must be authenticated with a PIN to the token before any access to private objects is allowed. Some tokens may require that a user be authenticated before any cryptographic function can be performed on the token, whether or not it involves private objects. The role of the SO is to initialize a token and to set the normal user's PIN, and possibly manipulate some public objects. A normal user cannot log in until the SO has set the user's PIN.

Other than the support for two types of user, Cryptoki does not address the relationship between the SO and a community of users. In particular, the SO and the User may be the same person or may be different, but such matters are outside the scope of this standard.

With respect to PINs, Cryptoki assumes only that they are variable-length character strings from the set in Table 4-3. Any translation to the device's requirements is left to the Cryptoki library. The following items are beyond the scope of Cryptoki:

- Any padding of the PIN.

- How the PINs are generated (by the user, by the application, or some other means).

Future version of Cryptoki will address other means of authentication, such as biometrics and PIN entry via a PIN pad attached to the device or its reader.

5.5 Sessions

Cryptoki requires that an application "open a session" with a token before the application has access to the token's objects and functions. The session provides the logical connection between the application and the token. A session can be a read/write (R/W) session or a read-only (R/O) session. Read/write and read-only refer to the access to token objects, not to session objects. In both session types, an application can create, read, write and destroy session objects, and read token objects. However, only in a read/write session can an application create, write and destroy token objects.

After a session is opened, the application has access to the token's "public" objects. To gain access to the token's "private" objects, a user must log in and be authenticated.

Cryptoki supports multiple sessions on multiple tokens. An application may have one or more sessions with one or more tokens. A token may have multiple sessions with one or more applications. Some tokens may allow only one read/write session at any given time.

An open session can be in one of several states. The session state determines allowable access to objects and functions that can be performed on them. The session states are described in the next two sections.

5.5.1 Read-only session states

A read-only session can be in one of two states, as illustrated in the following figure. When the session is opened, it is in the "R/O Public Session" state. Only the normal user may open a read-only session.

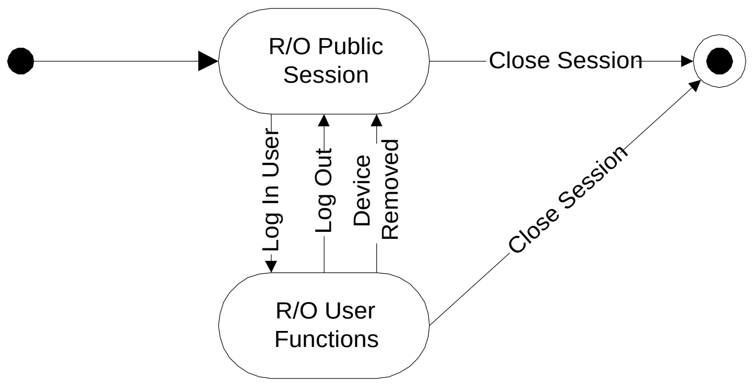

Figure 5-3, Read-Only Session States

The following table describes the session states:

Table 5-1, Read-Only Session States

State	Description
R/O Public Session	The application has opened a read-only session. The application has read-only access to public objects on the token.
R/O User Functions	The normal user has been authenticated to the token. The application has read-only access to public and private objects on the token.

5.5.2 Read/write session states

A read/write session can be in one of three states, as illustrated in the following figure. When the session is opened, it is in the "R/W Public Session" state.

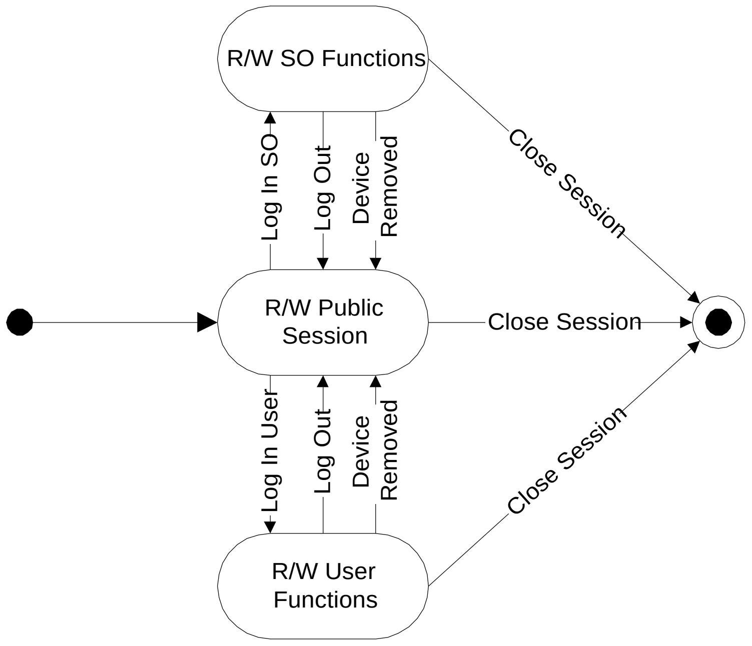

Figure 5-4, Read/Write Session States

The following table describes the session states:

Table 5-2, Read/Write Session States

State	**Description**
R/W Public Session	The application has opened a read/write session. The application has read/write access to public objects on the token.
R/W SO Functions	The Security Officer has been authenticated to the token. The application has read/write access only to public objects on the token, not to private objects. The SO can set the normal user's PIN.
R/W User Functions	The normal user has been authenticated to the token. The application has read/write access to public and private objects on the token.

5.5.3 Session events

Session events cause the session state to change. The following table describes the events.

Table 5-3, Session Events

Event	**Occurs when...**
Log In SO	the SO is authenticated to the token.
Log In User	the normal user is authenticated to the token.
Log Out	the application logs out the current user.
Close Session	the application closes the session or an application closes all sessions.
Device Removed	the device underlying the token has been removed from its slot.

Note that when the device is removed, the user is automatically logged out. However, the session remains open. If the device is reinserted, the application can log in the user again without opening a new session.

5.6 Function overview

The Cryptoki API consists of a number of functions, spanning slot and token management through object management, as well as cryptographic functions. These functions are presented in the following table.

Table 5-4, Summary of Cryptoki Functions

Category	Function	Description
General purpose	C_Initialize	initializes Cryptoki
	C_GetInfo	obtains general information about Cryptoki
Slot and token management	C_GetSlotList	obtains a list of slots in the system
	C_GetSlotInfo	obtains information about a particular slot
	C_GetTokenInfo	obtains information about a particular token
	C_GetMechansimList	obtains a list of mechanisms supported by a token
	C_GetMechanismInfo	obtains information about a particular mechanism
	C_InitToken	initializes a token
	C_InitPIN	initializes the normal user's PIN
	C_SetPIN	modifies the PIN of the current user
Session management	C_OpenSession	opens a connection or "session" between an application and a particular token
	C_CloseSession	closes a session
	C_CloseAllSessions	closes all sessions with a token
	C_GetSessionInfo	obtains information about the session
	C_Login	logs into a token
	C_Logout	logs out from a token
Object management	C_CreateObject	creates an object
	C_CopyObject	creates a copy of an object
	C_DestroyObject	destroys an object
	C_GetObjectSize	obtains the size of an object in bytes
	C_GetAttributeValue	obtains an attribute value of an object
	C_SetAttributeValue	modifies an attribute value of an object
	C_FindObjectsInit	initializes an object search operation
	C_FindObjects	continues an object search operation
Encryption and decryption	C_EncryptInit	initializes an encryption operation
	C_Encrypt	encrypts single-part data
	C_EncryptUpdate	continues a multiple-part encryption operation
	C_EncryptFinal	finishes a multiple-part encryption operation
	C_DecryptInit	initializes a decryption operation
	C_Decrypt	decrypts single-part encrypted data
	C_DecryptUpdate	continues a multiple-part decryption operation
	C_DecryptFinal	finishes a multiple-part decryption operation
Message digesting	C_DigestInit	initializes a message-digesting operation
	C_Digest	digests single-part data
	C_DigestUpdate	continues a multiple-part digesting operation
	C_DigestFinal	finishes a multiple-part digesting operation

Category	**Function**	**Description**
Signature and verification	C_SignInit	initializes a signature operation
	C_Sign	signs single-part data
	C_SignUpdate	continues a multiple-part signature operation
	C_SignFinal	finishes a multiple-part signature operation
	C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
	C_SignRecover	signs single-part data, where the data can be recovered from the signature
	C_VerifyInit	initializes a verification operation
	C_Verify	verifies a signature on single-part data
	C_VerifyUpdate	continues a multiple-part verification operation
	C_VerifyFinal	finishes a multiple-part verification operation
	C_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature
	C_VerifyRecover	verifies a signature on single-part data, where the data is recovered from the signature
Key management	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair
	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
Random number generation	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Function management	C_GetFunctionStatus	obtains updated status of a function running in parallel with the application
	C_CancelFunction	cancels a function running in parallel with the application
Callbacks	Notify	processes notifications from Cryptoki

Functions in the “Encryption and decryption,” “Message digesting,” “Signature and verification,” and “Key management” categories may run in parallel with the application if the token has the capability and the session is opened in this mode.

6. Security considerations

As an interface to cryptographic devices, Cryptoki provides a basis for security in a computer or communications system. Two of the particular features of the interface that facilitate such security are the following:

1. Access to private objects on the token, and possibly to cryptographic functions, requires a PIN. Thus possessing the cryptographic device that implements the token is not sufficient; the PIN is also needed.

2. Maximum protection is given to objects marked "sensitive"—they cannot be read from the token, nor exported through the cryptographic functions (though they can be used as keys).

It is expected that access to private and sensitive object by means other than Cryptoki (e.g., other programming interfaces, or reverse engineering of the device) would be difficult.

If a device does not have a tamper-proof environment or protected memory in which to store private and sensitive objects, the device may encrypt the objects with a master key which is perhaps derived from the user's PIN. The particular mechanism for protecting private objects is left to the device implementation, however.

Based on these features it should be possible to design applications in such a way that the token can provide adequate security for the objects the applications manage.

Of course, cryptography is only one element of security, and the token is only one component in a system. While the token itself may be secure, one must also consider the security of the operating system by which the application interfaces to it, especially since the PIN is passed through the operating system. It is easy for a rogue application on the operating system to obtain the PIN; it is also possible that other devices monitoring communication lines to the cryptographic device can obtain the PIN. Rogue applications and devices may also change the commands sent to the cryptographic device to obtain other services than what the application requested.

It is important to be sure that the system is secure against such attack. Cryptoki may well play a role here, for instance if a token is involved in the "booting up" of the system.

It is important to note that none of the attacks just described can compromise objects marked "sensitive," since the "sensitive" attribute cannot be changed once set. However, during key generation, before a private key is marked "sensitive," a copy of the private key could be obtained by the rogue application, so it is important to generate keys in a more trusted environment, than the environment in which one performs normal operations.

An application may also want to be sure that the token is "legitimate" in some sense (for a variety of reasons, including export restrictions). This is outside the scope of the present standard, but it can be achieved by distributing the token with a built-in, certified public/private-key pair, by which the token can prove its identity. The certificate would be signed by an authority (presumably the one indicating that the token is "legitimate"), whose public key is known to the application. The application would verify the certificate, and challenge the token to prove its identity by signing a time-varying message with its built-in private key.

Once a normal user has been authenticated to the token, Cryptoki does not restrict which cryptographic operations the user may perform. The user may perform any operation supported by the token.

7. Data types

Cryptoki's data types are described in following subsections, organized into categories, based on the kind of information they represent.

7.1 General information

Cryptoki represents general information with the following types.

♦ CK_VERSION

CK_VERSION is a structure that describes the version of Cryptoki. It is defined as follows:

```
typedef struct CK_VERSION {
   CK_BYTE major;
   CK_BYTE minor;
} CK_VERSION;
```

The fields of the structure have the following meanings:

major	major version number, the integer portion of the version
minor	minor version number, the hundredths portion of the version

For version 1.0, *major* = 1 and *minor* = 0. For version 2.1, *major* = 2 and *minor* = 10. Minor revisions of the standard are always upwardly compatible within the same major version number.

♦ CK_INFO

CK_INFO provides general information about Cryptoki. It is defined as follows:

```
typedef struct CK_INFO {
   CK_VERSION version;
   CK_CHAR manufacturerID[32];
   CK_FLAGS flags;
} CK_INFO;
```

The fields of the structure have the following meanings:

version	Cryptoki interface version number, for compatibility with future revisions of this interface
manufacturerID	ID of the Cryptoki library manufacturer; must be padded with the blank character (' ')
flags	bit flags reserved for future versions; must be zero for this version

♦ CK_INFO_PTR

CK_INFO_PTR points to a CK_INFO structure. It is implementation dependent.

♦ CK_NOTIFICATION

CK_NOTIFICATION enumerates the types of notifications that Cryptoki provides to an application. It is defined as follows:

```
typedef enum CK_NOTIFICATION {
   CKN_SURRENDER,
   CKN_COMPLETE,
   CKN_DEVICE_REMOVED
} CK_NOTIFICATION;
```

The notifications have the following meanings:

CKN_SURRENDER	Cryptoki is surrendering the execution of a function so that the application may perform other operations. After performing such operations, the application should indicate to Cryptoki whether to continue or cancel the function.
CKN_COMPLETE	A function running in parallel has completed.
CKN_DEVICE_REMOVED	Cryptoki detected that the device underlying the token has been removed from the reader (assuming the token has the capability)

7.2 Slot and token types

Cryptoki represents slot and token information with the following types.

♦ CK_SLOT_ID

CK_SLOT_ID is a Cryptoki assigned value that identifies a slot. It is defined as follows:

```
typedef CK_ULONG CK_SLOT_ID;
```

A **CK_SLOT_ID** is returned by **C_GetSlotList**.

♦ CK_SLOT_ID_PTR

CK_SLOT_ID_PTR points to a CK_SLOT_ID. It is implementation dependent.

♦ CK_SLOT_INFO

CK_SLOT_INFO provides information about a slot. It is defined as follows:

```
typedef struct CK_SLOT_INFO {
   CK_CHAR slotDescription[64];
   CK_CHAR manufacturerID[32];
   CK_FLAGS flags;
} CK_SLOT_INFO;
```

The fields of the structure have the following meanings:

slotDescription	character-string description of the slot (the type of interface between the device and the computer); must be padded with the blank character (' ')
manufacturerID	ID of the "slot" manufacturer; must be padded with the blank character (' ')
flags	bits flags that provide capabilities of the slot.

The following table defines the flags.

Table 7-1, Slot Information Flags

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x0001	TRUE if a token is present in the slot (e.g., a device is in the reader)
CKF_REMOVABLE_DEVICE	0x0002	TRUE if the reader supports removable devices
CKF_HW_SLOT	0x0004	TRUE if the slot is a hardware slot as opposed to a software slot implementing a "soft token"

♦ CK_SLOT_INFO_PTR

CK_SLOT_INFO_PTR points to a CK_SLOT_INFO structure. It is implementation dependent.

♦ CK_TOKEN_INFO

CK_TOKEN_INFO provides information about a token. It is defined as follows:

```
typedef struct CK_TOKEN_INFO {
   CK_CHAR label[32];
   CK_CHAR manufacturerID[32];
   CK_CHAR model[16];
   CK_CHAR serialNumber[16];
   CK_FLAGS flags;
   CK_USHORT usMaxSessionCount;
   CK_USHORT usSessionCount;
   CK_USHORT usMaxRwSessionCount;
   CK_USHORT usRwSessionCount;
   CK_USHORT usMaxPinLen;
```

```
      CK_USHORT usMinPinLen;
      CK_ULONG ulTotalPublicMemory;
      CK_ULONG ulFreePublicMemory;
      CK_ULONG ulTotalPrivateMemory;
      CK_ULONG ulFreePrivateMemory;
   } CK_TOKEN_INFO;
```

The fields of the structure have the following meanings:

label	application defined label, assigned during token initialization; must be padded with the blank character (' ')
manufacturerID	ID of the device manufacturer; must be padded with the blank character (' ')
model	model of the device; must be padded with the blank character (' ')
serialNumber	character-string serial number of the device; must be padded with the blank character (' ')
flags	bit flags indicating capabilities and status of the device as defined below
usMaxSessionCount	maximum number of sessions that can be opened with the token at one time
usSessionCount	number of sessions that are currently open with the token
usMaxRwSessionCount	maximum number of read/write sessions that can be opened with the token at one time
usRwSessionCount	number of read/write sessions that are currently open with the token
usMaxPinLen	maximum length in bytes of the PIN
usMinPinLen	minimum length in bytes of the PIN
ulTotalPublicMemory	the total amount of memory in bytes occupied by public objects
ulFreePublicMemory	the amount of free (unused) memory in bytes for public objects
ulTotalPrivateMemory	the total amount of memory in bytes occupied by private objects
ulFreePrivateMemory	the amount of free (unused) memory in bytes for private objects

The *flags* parameter is defined as follows:

Table 7-2, Token Information Flags

Bit Flag	**Mask**	**Meaning**
CKF_RNG	0x0001	TRUE if the token has its own random number generator
CKF_WRITE_PROTECTED	0x0002	TRUE if the token is write-protected
CKF_LOGIN_REQUIRED	0x0004	TRUE if a user must be logged in to perform cryptographic functions
CKF_USER_PIN_INITIALIZED	0x0008	TRUE if the normal user's PIN has been initialized
CKF_EXCLUSIVE_EXISTS	0x0010	TRUE if an exclusive session exists

♦ CK_TOKEN_INFO_PTR

CK_TOKEN_INFO_PTR points to a CK_TOKEN_INFO structure. It is implementation dependent.

7.3 Session types

Cryptoki represents session information with the following types.

♦ CK_SESSION_HANDLE

CK_SESSION_HANDLE is a Cryptoki-assigned value that identifies a session. It is defined as follows:

```
    typedef CK_ULONG CK_SESSION_HANDLE;
```

♦ CK_SESSION_HANDLE_PTR

CK_SESSION_HANDLE_PTR points to a CK_SESSION_HANDLE. It is implementation dependent.

♦ CK_USER_TYPE

CK_USER_TYPE enumerates the types of Cryptoki users described in Section 5.4. It is defined as follows:

```
typedef enum CK_USER_TYPE {
   CKU_SO,    /* Security Officer */
   CKU_USER   /* Normal user */
} CK_USER_TYPE;
```

♦ CK_STATE

CK_STATE enumerates the session states decribed in Sections 5.5.1 and 5.5.2. It is defined as follows:

```
typedef enum CK_STATE {
   CKS_RW_PUBLIC_SESSION,
   CKS_RW_USER_FUNCTIONS,
   CKS_RO_PUBLIC_SESSION,
   CKS_RO_SO_FUNCTIONS,
   CKS_RO_USER_FUNCTIONS
} CK_STATE;
```

♦ CK_SESSION_INFO

CK_SESSION_INFO provides information about a session. It is defined as follows:

```
typedef struct CK_SESSION_INFO {
   CK_SLOT_ID slotID;
   CK_STATE state;
   CK_FLAGS flags;
   CK_USHORT usDeviceError;
} CK_SESSION_INFO;
```

The fields of the structure have the following meanings:

slotID	ID of the slot that interfaces with the token
state	the state of the session
flags	bit flags that define the type of session; the flags are defined below
usDeviceError	an error code defined by the cryptographic device. Used for errors not covered by Cryptoki.

The *flags* are defined in the following table.

Table 7-3, Session Information Flags

Bit Flag	Mask	Meaning
CKF_EXCLUSIVE_SESSION	0x0001	TRUE if the session is exclusive; FALSE if the session is shared
CKF_RW_SESSION	0x0002	TRUE if the session is read/write; FALSE if the session is read-only
CKF_SERIAL_SESSION	0x0004	TRUE if cryptographic functions are performed in serial with the application; FALSE if the functions may be performed in parallel with the application

♦ CK_SESSION_INFO_PTR

CK_SESSION_INFO_PTR points to a CK_SESSION_INFO structure. It is implementation dependent.

7.4 Object types

Cryptoki represents object information with the following types.

♦ CK_OBJECT_HANDLE

CK_OBJECT_HANDLE is a token-specific identifier for an object. It is defined as follows:

```
    typedef CK_ULONG CK_OBJECT_HANDLE;
```

The handle is assigned by Cryptoki when an object is created. The handle for an object is unique among all objects in the token at a given time, and remains constant until the object is destroyed.

Cryptoki considers an object handle valid if and only if the object exists and is accessible to the application. In particular, object handles for private objects are valid if only if a user is logged in.

♦ CK_OBJECT_HANDLE_PTR

CK_OBJECT_HANDLE_PTR points to a CK_OBJECT_HANDLE. It is implementation dependent.

♦ CK_OBJECT_CLASS

CK_OBJECT_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes. It is defined as follows:

```
    typedef CK_USHORT CK_OBJECT_CLASS;
```

For this version of Cryptoki, the following classed of objects are defined:

```
    #define CKO_DATA            0x0000
    #define CKO_CERTIFICATE     0x0001
    #define CKO_PUBLIC_KEY      0x0002
```

```
#define CKO_PRIVATE_KEY     0x0003
#define CKO_SECRET_KEY      0x0004
#define CKO_VENDOR_DEFINED  0x8000
```

Object classes **CKO_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.

♦ CK_OBJECT_CLASS_PTR

CK_OBJECT_CLASS_PTR points to a CK_OBJECT_CLASS structure. It is implementation dependent.

♦ CK_KEY_TYPE

CK_KEY_TYPE is a value that identifies a key type. It is defined as follows:

```
typedef CK_USHORT CK_KEY_TYPE;
```

For this version of Cryptoki, the following key types are defined:

```
#define CKK_RSA             0x0000
#define CKK_DSA             0x0001
#define CKK_DH              0x0002
#define CKK_GENERIC_SECRET  0x0010
#define CKK_RC2             0x0011
#define CKK_RC4             0x0012
#define CKK_DES             0x0013
#define CKK_DES2            0x0014
#define CKK_DES3            0x0015
#define CKK_VENDOR_DEFINED  0x8000
```

Key types **CKK_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their key types through the PKCS process.

♦ CK_CERTIFICATE_TYPE

CK_CERTIFICATE_TYPE is a value that identifies a certificate type. It is defined as follows:

```
typedef CK_USHORT CK_CERTIFICATE_TYPE;
```

For this version of Cryptoki, the following certificate types are defined:

```
#define CKC_X_509           0x0000
#define CKC_VENDOR_DEFINED  0x8000
```

Certificate types **CKC_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their certificate types through the PKCS process.

♦ CK_ATTRIBUTE_TYPE

CK_ATTRIBUTE_TYPE is a value that identifies an attribute type. It is defined as follows:

```
typedef CK_USHORT CK_ATTRIBUTE_TYPE;
```

For this version of Cryptoki, the following attribute types are defined:

```
#define CKA_CLASS               0x0000
#define CKA_TOKEN               0x0001
#define CKA_PRIVATE             0x0002
#define CKA_LABEL               0x0003
#define CKA_APPLICATION         0x0010
#define CKA_VALUE               0x0011
#define CKA_CERTIFICATE_TYPE    0x0080
#define CKA_ISSUER              0x0081
#define CKA_SERIAL_NUMBER       0x0082
#define CKA_KEY_TYPE            0x0100
#define CKA_SUBJECT             0x0101
#define CKA_ID                  0x0102
#define CKA_SENSITIVE           0x0103
#define CKA_ENCRYPT             0x0104
#define CKA_DECRYPT             0x0105
#define CKA_WRAP                0x0106
#define CKA_UNWRAP              0x0107
#define CKA_SIGN                0x0108
#define CKA_SIGN_RECOVER        0x0109
#define CKA_VERIFY              0x010A
#define CKA_VERIFY_RECOVER      0x010B
#define CKA_DERIVE              0x010C
#define CKA_MODULUS             0x0120
#define CKA_MODULUS_BITS        0x0121
#define CKA_PUBLIC_EXPONENT     0x0122
#define CKA_PRIVATE_EXPONENT    0x0123
#define CKA_PRIME_1             0x0124
#define CKA_PRIME_2             0x0125
#define CKA_EXPONENT_1          0x0126
#define CKA_EXPONENT_2          0x0127
#define CKA_COEFFICIENT         0x0128
#define CKA_PRIME               0x0130
#define CKA_SUBPRIME            0x0131
#define CKA_BASE                0x0132
#define CKA_VALUE_BITS          0x0160
#define CKA_VALUE_LEN           0x0161
#define CKA_VENDOR_DEFINED      0x8000
```

Section 8 defines the attributes for each object class. Attribute types **CKA_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their attribute types through the PKCS process.

♦ CK_ATTRIBUTE

CK_ATTRIBUTE is a structure that includes the type, length and value of an attribute. It is defined as follows:

```
typedef struct CK_ATTRIBUTE {
   CK_ATTRIBUTE_TYPE type;
   CK_VOID_PTR pValue;
   CK_USHORT usValueLen;
} CK_ATTRIBUTE;
```

The fields of the structure have the following meanings:

type	the attribute type
pValue	pointer to the value of the attribute
usValueLen	length in bytes of the value

If an attribute has no value, then *pValue* = NULL_PTR, and *usValueLen* = 0. An array of **CK_ATTRIBUTE**s is called a "template" and is used for creating, manipulating and searching for objects. Note that *pValue* is an "void" pointer, facilitating the passing of arbitrary values. Both the application and Cryptoki library must ensure that the pointer can be safely cast to the expected type (e.g., without word-alignment errors).

♦ CK_ATTRIBUTE_PTR

CK_ATTRIBUTE_PTR points to a CK_ATTRIBUTE structure. It is implementation dependent.

♦ CK_DATE

CK_DATE is a structure that defines a date. It is defined as follows:

```
typedef struct CK_DATE{
   CK_CHAR year[4];
   CK_CHAR month[2];
   CK_CHAR day[2];
} CK_DATE;
```

The fields of the structure have the following meanings:

year	the year ("1900" - "9999")
month	the month ("01" - "12")
day	the day ("01" - "31")

The fields hold numeric characters from the character set in Table 4-3, not the literal byte values.

7.5 Mechanisms

A mechanism specifies how a certain process is to be performed. Cryptoki supports the following types for describing mechanisms. Section 10 provides a complete description of the mechanisms and their relation to the functions.

♦ CK_MECHANISM_TYPE

CK_MECHANISM_TYPE is a value that identifies a mechanism type. It is defined as follows:

```
typedef CK_USHORT CK_MECHANISM_TYPE;
```

For this version of Cryptoki, the following mechanism types are defined:

```
#define CKM_RSA_PKCS_KEY_PAIR_GEN  0x0000
#define CKM_RSA_PKCS               0x0001
#define CKM_RSA_9796               0x0002
#define CMK_RSA_X_509              0x0003
#define CKM_DSA_KEY_PAIR_GEN       0x0010
#define CKM_DSA                    0x0011
#define CKM_DH_PKCS_KEY_PAIR_GEN   0x0020
#define CKM_DH_PKCS_DERIVE         0x0021
#define CKM_RC2_KEY_GEN            0x0100
#define CKM_RC2_ECB                0x0101
#define CKM_RC2_CBC                0x0102
#define CKM_RC2_MAC                0x0103
#define CKM_RC4_KEY_GEN            0x0110
#define CKM_RC4                    0x0111
#define CKM_DES_KEY_GEN            0x0120
#define CKM_DES_ECB                0x0121
#define CKM_DES_CBC                0x0122
#define CKM_DES_MAC                0x0123
#define CKM_DES2_KEY_GEN           0x0130
#define CKM_DES3_KEY_GEN           0x0131
#define CKM_DES3_ECB               0x0132
#define CKM_DES3_CBC               0x0133
#define CKM_DES3_MAC               0x0134
#define CKM_MD2                    0x0200
#define CKM_MD5                    0x0210
#define CKM_SHA_1                  0x0220
#define CKM_VENDOR_DEFINED         0x8000
```

Mechanism types **CKM_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their mechanism types through the PKCS process.

♦ CK_MECHANISM_TYPE_PTR

CK_MECHANISM_TYPE_PTR points to a CK_MECHANISM_TYPE structure. It is implementation dependent.

♦ CK_MECHANISM

CK_MECHANISM is a structure that specifies a particular mechanism. It is defined as follows:

```
typedef struct CK_MECHANISM {
   CK_MECHANISM_TYPE mechanism;
   CK_VOID_PTR pParameter;
   CK_USHORT usParameterLen;
} CK_MECHANISM;
```

The fields of the structure have the following meanings:

mechanism	the type of mechanism
pParameter	pointer to the parameter if required by the mechanism
usParameterLen	length in bytes of the parameter

Note that *pParameter* is an "void" pointer, facilitating the passing of arbitrary values. Both the application and Cryptoki library must ensure that the pointer can be safely cast to the expected type (e.g., without word-alignment errors).

♦ CK_MECHANISM_PTR

CK_MECHANISM_PTR points to a CK_MECHANISM structure. It is implementation dependent.

♦ CK_MECHANISM_INFO

CK_MECHANISM_INFO is a structure that provides information about a particular mechanism. It is defined as follows:

```
typedef struct CK_MECHANISM_INFO {
   CK_ULONG ulMinKeySize;
   CK_ULONG ulMaxKeySize;
   CK_FLAGS flags;
} CK_MECHANISM_INFO;
```

The fields of the structure have the following meanings:

ulMinKeySize	the minimum size of the key for the mechanism
ulMaxKeySize	the maximum size of the key for the mechanism
flags	bit flags specifying mechanism capabilities

The *flags* are defined as follows.

Table 7-4, Mechanism Information FLags

Bit Flag	Mask	Meaning
CKF_HW	0x0001	TRUE if the mechanism is performed by the device; FALSE if the mechanism is performed in software
CKF_EXTENSION	0x8000	TRUE if an extension to the flags; FALSE if no extensions. Must be FALSE for this version.

♦ CK_MECHANISM_INFO_PTR

CK_MECHANISM_INFO_PTR points to a CK_MECHANISM_INFO structure. It is implementation dependent.

♦ CK_RC2_CBC_PARAMS

CK_RC2_CBC_PARAMS is a structure that provides the parameters to the CKM_RC2_CBC mechanism. It is defined as follows:

```
typedef struct CK_RC2_CBC_PARAMS {
   CK_USHORT usEffectiveBits;
   CK_BYTE iv[8];
} CK_RC2_CBC;
```

The fields of the structure have the following meanings:

usEffectiveBits	the effective number of bits in the RC2 search space, must be between 1 and 1024
iv	the initialization vector for cipher block chaining mode

7.6 Functions

Cryptoki represents information about functions with the following data types.

♦ CK_ENTRY

CK_ENTRY is an entry (or function) into Cryptoki. It is implementation dependent.

♦ CK_RV

CK_RV is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
typedef CK_USHORT CK_RV;
```

For this version of Cryptoki, the following return values are defined:

```
#define CKR_OK                              0x0000
#define CKR_CANCEL                          0x0001
#define CKR_HOST_MEMORY                     0x0002
#define CKR_SLOT_ID_INVALID                 0x0003
#define CKR_FLAGS_INVALID                   0x0004
#define CKR_ATTRIBUTE_READ_ONLY             0x0010
#define CKR_ATTRIBUTE_SENSITIVE             0x0011
#define CKR_ATTRIBUTE_TYPE_INVALID          0x0012
#define CKR_ATTRIBUTE_VALUE_INVALID         0x0013
#define CKR_DATA_INVALID                    0x0020
#define CKR_DATA_LEN_RANGE                  0x0021
#define CKR_DEVICE_ERROR                    0x0030
#define CKR_DEVICE_MEMORY                   0x0031
#define CKR_DEVICE_REMOVED                  0x0032
#define CKR_ENCRYPTED_DATA_INVALID          0x0040
#define CKR_ENCRYPTED_DATA_LEN_RANGE        0x0041
#define CKR_FUNCTION_CANCELED               0x0050
#define CKR_FUNCTION_NOT_PARALLEL           0x0051
#define CKR_FUNCTION_PARALLEL               0x0052
#define CKR_KEY_HANDLE_INVALID              0x0060
#define CKR_KEY_SENSITIVE                   0x0061
#define CKR_KEY_SIZE_RANGE                  0x0062
#define CKR_KEY_TYPE_INCONSISTENT           0x0063
#define CKR_MECHANISM_INVALID               0x0070
#define CKR_MECHANISM_PARAM_INVALID         0x0071
#define CKR_OBJECT_CLASS_INCONSISTENT       0x0080
```

```
    #define CKR_OBJECT_CLASS_INVALID              0x0081
    #define CKR_OBJECT_HANDLE_INVALID             0x0082
    #define CKR_OPERATION_ACTIVE                  0x0090
    #define CKR_OPERATION_NOT_INITIALIZED         0x0091
    #define CKR_PIN_INCORRECT                     0x00A0
    #define CKR_PIN_INVALID                       0x00A1
    #define CKR_PIN_LEN_RANGE                     0x00A2
    #define CKR_SESSION_CLOSED                    0x00B0
    #define CKR_SESSION_COUNT                     0x00B1
    #define CKR_SESSION_EXCLUSIVE_EXISTS          0x00B2
    #define CKR_SESSION_HANDLE_INVALID            0x00B3
    #define CKR_SESSION_PARALLEL_NOT_SUPPORTED    0x00B4
    #define CKR_SESSION_READ_ONLY                 0x00B5
    #define CKR_SIGNATURE_INVALID                 0x00C0
    #define CKR_SIGNATURE_LEN_RANGE               0x00C1
    #define CKR_TEMPLATE_INCOMPLETE               0x00D0
    #define CKR_TEMPLATE_INCONSISTENT             0x00D1
    #define CKR_TOKEN_NOT_PRESENT                 0x00E0
    #define CKR_TOKEN_NOT_RECOGNIZED              0x00E1
    #define CKR_TOKEN_WRITE_PROTECTED             0x00E2
    #define CKR_UNWRAPPING_KEY_HANDLE_INVALID     0x00F0
    #define CKR_UNWRAPPING_KEY_SIZE_RANGE         0x00F1
    #define CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT  0x00F2
    #define CKR_USER_ALREADY_LOGGED_IN            0x0100
    #define CKR_USER_NOT_LOGGED_IN                0x0101
    #define CKR_USER_PIN_NOT_INITIALIZED          0x0102
    #define CKR_USER_TYPE_INVALID                 0x0103
    #define CKR_WRAPPED_KEY_INVALID               0x0110
    #define CKR_WRAPPED_KEY_LEN_RANGE             0x0112
    #define CKR_WRAPPING_KEY_HANDLE_INVALID       0x0113
    #define CKR_WRAPPING_KEY_SIZE_RANGE           0x0114
    #define CKR_WRAPPING_KEY_TYPE_INCONSISTENT    0x0115
    #define CKR_VENDOR_DEFINED                    0x8000
```

Section 9 defines the meanings of each **CK_RV** value. Return values **CKR_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their return through the PKCS process.

8. Objects

Cryptoki recognizes a number of classes of objects, as defined in the CK_OBJECT_CLASS data type. Objects consist of a set of attributes, each of which has a given value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and the attributes they support.

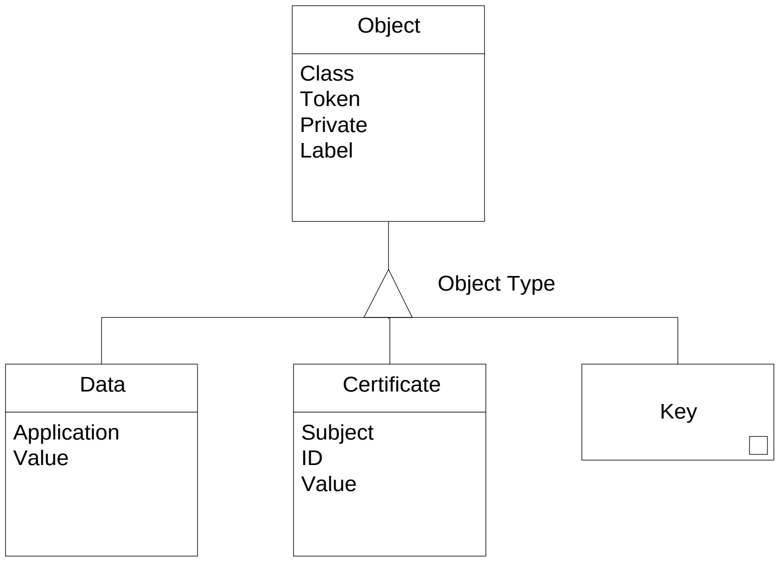

Figure 8-1, Cryptoki Object Hierarchy

The following figure illustrates the details of the key objects.

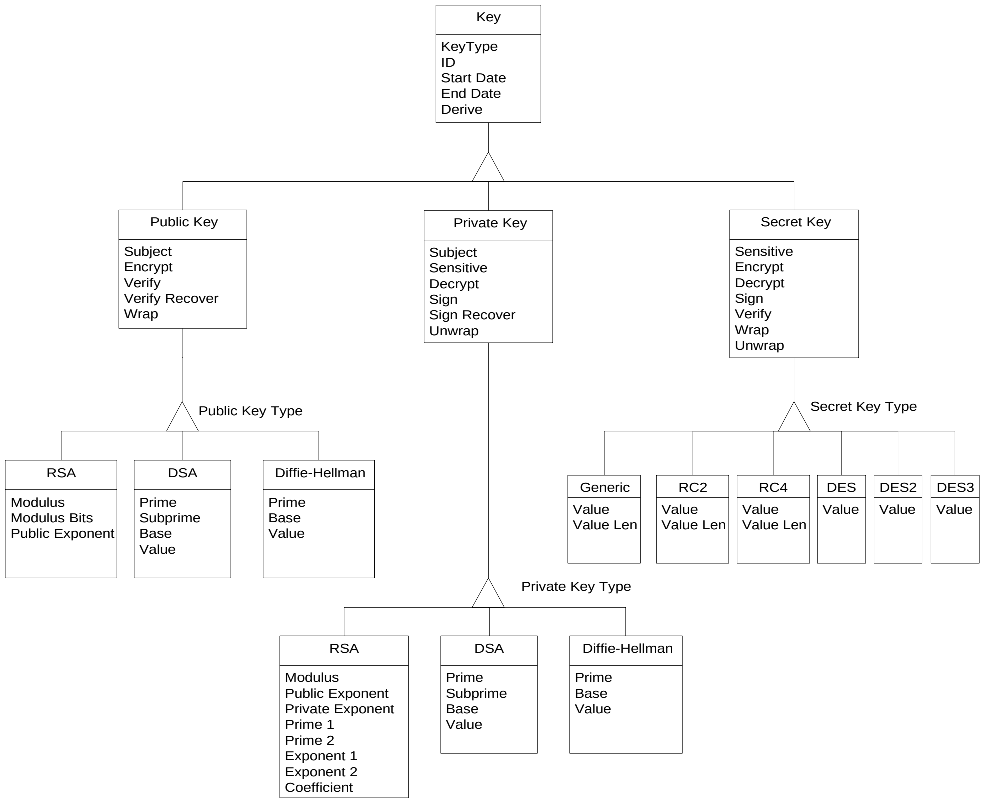

Figure 8-2, Key Object Detail

Cryptoki provides functions for creating and destroying objects, and for obtaining and modifying the values of attributes. Some of the cryptographic functions (e.g., key generation) also create objects to hold their results.

Objects are always "well-formed" in Cryptoki—that is, an object always contains required attributes, and the attributes are always consistent with one another, from the time the object is created. (This is in contrast with object-based paradigms where an object has no attributes other than perhaps a class when it is created, and is "uninitialized" for some time. In Cryptoki, objects are always initialized.)

To ensure that the required attributes are defined, the functions that create objects take a "template" as an argument, where the template specifies initial attribute values. The template can also provide input to cryptographic functions that create objects (e.g., it can specify a key size). Cryptographic functions that create objects may also contribute some of the initial attribute values (see Section 8 for details). In any case, all the attributes supported by an object class that do not have default values must be specified when an object is created, either in the template, or by the function.

Tables in this section define attributes in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (e.g., CK_OBJECT_CLASS). Attributes may also take the following types:

Byte array	an arbitrary string (array) of **CK_BYTE**s
Big integer	a string of **CK_BYTE**s representing an integer of arbitrary size, most significant byte first, without a sign bit (e.g., the integer 32768 is represented as the byte string 0x80 0x00)

Local string	a string of **CK_CHAR**s (see Table 4-3)

8.1 Common attributes

The following table defines the attributes common to all objects.

Table 8-1, Common Object Attributes

Attribute	**Data Type**	**Meaning**
CKA_CLASS[1]	CK_OBJECT_CLASS	Object class (type)
CKA_TOKEN	CK_BBOOL	TRUE if object is a token object (vs. session object) (default FALSE)
CKA_PRIVATE	CK_BBOOL	TRUE if object is a private object (vs. public object) (default FALSE)
CKA_LABEL	Local string	Description of the object (default empty)

[1]Must be specified when object is created

Only the **CKA_LABEL** attribute may be modified after the object is created. (The **CKA_TOKEN** and **CKA_PRIVATE** attributes can be changed in the process of copying an object.)

When the **CKA_PRIVATE** attribute is TRUE, a user may not access the object until the user has been authenticated to the token.

The **CKA_LABEL** attribute is intended to assist users in browsing.

Additional attributes for each object type are described in the following sections. Note that only attributes visible to applications are listed. Objects may well carry other information, useful to a token, which is not visible to the application.

8.2 Data objects

Data objects (object class **CKO_DATA**) hold information defined by an application. Other than providing access to a data objects, Cryptoki does not attach any special meaning to a data object. The following table lists the attributes supported by data objects, in addition to the common attributes listed in Table 8-1.

Table 8-2, Data Object Attributes

Attribute	**Data type**	**Meaning**
CKA_APPLICATION	Local string	Description of the application that manages the object (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)

Both of these attributes may be modified after the object is created.

The **CKA_APPLICATION** attribute provides a means for applications to distinguish among the objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has access to a data object, however.

The following is a sample template for creating a data object:

```
CK_OBJECT_CLASS class = CKO_DATA;
CK_CHAR label[] = “A data object”;
CK_CHAR application[] = “An application”;
CK_BYTE data[] = “Sample data”;
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_APPLICATION, application, sizeof(application)},
   {CKA_VALUE, data, sizeof(data)}
};
```

8.3 Certificate objects

Certificate objects (object class **CKO_CERTIFICATE**) hold public-key certificates. Other than providing access to a certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes listed in Table 8-1.

Table 8-3, Common Certificate Object Attributes

Attribute	Data type	Meaning
CKA_CERTIFICATE_TYPE[1]	CK_CERTIFICATE_TYPE	Type of certificate

[1]Must be specified when the object is created.

8.3.1 X.509 certificate objects

X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes listed in Table 8-1 and Table 8-3.

Table 8-4, X.509 Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[1]	Byte array	DER encoding of the certificate subject name
CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
CKA_ISSUER	Byte array	DER encoding of the certificate issuer name (default empty)
CKA_SERIAL_NUMBER	Byte array	DER encoding of the certificate serial number (default empty)
CKA_VALUE[1]	Byte array	BER encoding of the certificate

[1]Must be specified when the object is created.

Only the **CKA_ID**, **CKA_ISSUER**, and **CKA_SERIAL_NUMBER** attributes may be modified after the object is created.

The **CKA_ID** attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same **CKA_ID** value without introducing any ambiguity.)

It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). But Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

The **CKA_ISSUER** and **CKA_SERIAL_NUMBER** attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the proposed version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the **CKA_ID** value be identical to the key identifier in such a certificate extension.

The following is a sample template for creating a certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_CHAR label[] = “A certificate object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_VALUE, certificate, sizeof(certificate)},
};
```

8.4 Key objects

Key objects hold encryption keys, which can be public keys, private keys, or secret keys. The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes listed in Table 8-1.

Table 8-5, Common Key Attributes

Attribute	**Data Type**	**Meaning**
CKA_KEY_TYPE[1]	CK_KEY_TYPE	Type of key
CKA_ID	Byte array	Key identifier for key (default empty)
CKA_START_DATE	CK_DATE	Start date for the key (default empty)
CKA_END_DATE	CK_DATE	End date for the key (default empty)
CKA_DERIVE	CK_BBOOL	TRUE if key supports key derivation (default FALSE)

[1]Must be specified when the object is created.

All of these attributes except **CKA_KEY_TYPE** may be modified after the object is created.

Note that the start and end dates are for reference only; Cryptoki does not attach any special meaning to them. In particular, it does not restrict usage of a key according to the dates; this is up to the application.

The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys, this is for multiple keys held by the same subject; the key identifier for a public key and its corresponding private key should be the same. The key identifier should also be the same as for the corresponding certificate. Cryptoki does not enforce this association, however. (See Section 8.3 for further commentary.)

In the case of secret keys, the meaning of the **CKA_ID** attribute is up to the application.

8.5 Public key objects

Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. This version of Cryptoki recognizes three types of public keys: RSA, DSA, and Diffie-Hellman. The following table defines the attributes common to all public keys, in addition to the common attributes listed in Table 8-1 and Table 8-5.

Table 8-6, Common Public Key Attributes

Attribute	**Data type**	**Meaning**
CKA_SUBJECT	Byte array	DER encoding of the key subject name
CKA_ENCRYPT	CK_BBOOL	TRUE if key supports encryption[1]
CKA_VERIFY	CK_BBOOL	TRUE if key supports verification[1]
CKA_VERIFY_RECOVER	CK_BBOOL	TRUE if key supports verification where the data is recovered from the signature[1]
CKA_WRAP	CK_BBOOL	TRUE if key supports wrapping[1]

[1] Default is up to the token, based on what mechanisms it supports; the application can specify an explicit value in the template, and Cryptoki may reject it if no compatible mechanism is supported.

All of these attributes may be modified after the object is created.

It is intended in the interests of interoperability that the subject name and key identifier for a public key will be the same as those for the corresponding certificate and private key. (However, it is not required that the certificate and private key also be stored on the token.)

8.5.1 RSA public key objects

RSA public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_RSA**) hold RSA public keys. The following table defines the RSA public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-6.

Table 8-7, RSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS[1]	Big integer	Modulus *n*
CKA_MODULUS_BITS[2]	CK_USHORT	Length in bits of modulus *n*
CKA_PUBLIC_EXPONENT[1]	Big integer	Public exponent *e*

[1]Must be specified when the object is created. [2]Specify this attribute only in a template for generating a key of this type.

None of these attributes may be modified after the object is created.

Depending on the token, there may be limits on the length of key components. See PKCS #1 for more information on RSA keys.

The following is a sample template for creating an RSA public key object:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_CHAR label[] = “An RSA public key object”;
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_WRAP, &true, 1},
   {CKA_ENCRYPT, &true, 1},
   {CKA_MODULUS, modulus, sizeof(modulus)},
   {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)},
};
```

8.5.2 DSA public key objects

DSA public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_DSA**) hold DSA public keys. The following table defines the DSA public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-6.

Table 8-8, DSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1]	Big integer	Base *g*
CKA_VALUE[1]	Big integer	Public value *y*

[1]Must be specified when the object is created.

None of these attributes may be modified after the object is created.

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "DSA parameters." See FIPS PUB 186 for more information on DSA keys.

The following is a sample template for creating an DSA public key object:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_CHAR label[] = "A DSA public key object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
};
```

8.5.3 Diffie-Hellman public key objects

Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_DH**) hold Diffie-Hellman public keys. The following table defines the RSA public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-6.

Table 8-9, Diffie-Hellman Public Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_PRIME[1]	Big integer	Prime *p*
CKA_BASE[1]	Big integer	Base *g*
CKA_VALUE[1]	Big integer	Public value *y*

[1]Must be specified when object is created.

None of these attributes may be modified after the object is created.

The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the "Diffie-Hellman parameters." Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

The following is a sample template for creating a Diffie-Hellman public key object:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_CHAR label[] = "A Diffie-Hellman public key object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BBOOL true = TRUE;
```

```
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
};
```

8.6 Private key objects

Private key objects (object class **CKO_PUBLIC_KEY**) hold private keys. This version of Cryptoki recognizes three types of private key: RSA, DSA, and Diffie-Hellman. The following table defines the attributes common to all private keys, in addition to the common attributes listed in Table 8-1 and Table 8-5.

Table 8-10, Common Private Key Attributes

Attribute	**Data type**	**Meaning**
CKA_SUBJECT	Byte array	DER encoding of certificate subject name (default empty)
CKA_SENSITIVE	CK_BBOOL	TRUE if object is sensitive[1]
CKA_DECRYPT	CK_BBOOL	TRUE if key supports decryption[1]
CKA_SIGN	CK_BBOOL	TRUE if key supports signatures where the signature is an appendix to the data[1]
CKA_SIGN_RECOVER	CK_BBOOL	TRUE if key supports signatures where the data can be recovered from the signature[1]
CKA_UNWRAP	CK_BBOOL	TRUE if key supports unwrapping[1]

[1] Default is up to the token, based on what mechanisms it supports; the application can specify an explicit value in the template, and Cryptoki may reject it if no compatible mechanism is supported.

All of these attributes may be modified after the object is created. However, the **CKA_SENSITIVE** attribute may only be set to TRUE.

It is intended in the interests of interoperability that the subject name and key identifier for a private key will be the same as those for the corresponding certificate and public key. (However, it is not required that the certificate and public key also be stored on the token.)

If the **CKA_SENSITIVE** attribute is TRUE, then certain attributes of the private key cannot be revealed outside the token. Also, the private key cannot be wrapped if the **CKA_SENSITIVE** attribute is TRUE, since it could potentially be recovered outside the token if the unwrapping key is known outside. The attribute table for the each type of private key specifies which attributes are not revealed.

8.6.1 RSA private key objects

RSA private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_RSA**) hold RSA private keys. The following table defines the RSA private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-10.

Table 8-11, RSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS[1]	Big integer	Modulus *n*
CKA_PUBLIC_EXPONENT[1]	Big integer	Public exponent *e*
CKA_PRIVATE_EXPONENT[1,2]	Big integer	Private exponent *d*
CKA_PRIME_1[1,2]	Big integer	Prime *p*
CKA_PRIME_2[1,2]	Big integer	Prime *q*
CKA_EXPONENT_1[1,2]	Big integer	Private exponent *d* modulo *p*-1
CKA_EXPONENT_2[1,2]	Big integer	Private exponent *d* modulo *q*-1
CKA_COEFFICIENT[1,2]	Big integer	CRT coefficient q^{-1} mod *p*

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE.

None of these attributes may be modified after the object is created.

Depending on the token, there may be limits on the length of the key components. See PKCS #1 for more information on RSA keys.

The following is a sample template for creating an RSA private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_CHAR label[] = “An RSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE modulus[] = {...};
CK_BYTE publicExponent[] = {...};
CK_BYTE privateExponent[] = {...};
CK_BYTE prime1[] = {...};
CK_BYTE prime2[] = {...};
CK_BYTE exponent1[] = {...};
CK_BYTE exponent2[] = {...};
CK_BYTE coefficient[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, 1},
   {CKA_DECRYPT, &true, 1},
   {CKA_SIGN, &true, 1},
   {CKA_MODULUS, modulus, sizeof(modulus)},
   {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
   {CKA_PRIVATE_EXPONENT, privateExponent, sizeof(privateExponent)},
   {CKA_PRIME_1, prime1, sizeof(prime1)},
   {CKA_PRIME_2, prime2, sizeof(prime2)},
   {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
   {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
   {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
};
```

8.6.2 DSA private key objects

DSA private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_DSA**) hold DSA private keys. The following table defines the DSA private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-10.

Table 8-12, DSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1]	Big integer	Base *g*
CKA_VALUE[1,2]	Big integer	Private value *x*

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE.

None of these attributes may be modified after the object is created.

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "DSA parameters." See FIPS PUB 186 for more information on DSA keys.

The following is a sample template for creating a DSA private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_CHAR label[] = "A DSA private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, 1},
   {CKA_SIGN, &true, 1},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.6.3 Diffie-Hellman private key objects

Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_DH**) hold Diffie-Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-10.

Table 8-13, Diffie-Hellman Private Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_PRIME[1]	Big integer	Prime *p*
CKA_BASE[1]	Big integer	Base *g*
CKA_VALUE[1,2]	Big integer	Private value *x*
CKA_VALUE_BITS[3]	CK_USHORT	Length in bits of private value *x*

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE. [3]Specify this attribute only in a template for generating a key of this type.

None of these attributes may be modified after the object is created.

The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the "Diffie-Hellman parameters." Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

The following is a sample template for creating a Diffie-Hellman private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_CHAR label[] = "A Diffie-Hellman private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, 1},
   {CKA_DERIVE, &true, 1},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7 Secret key objects

Secret key objects (object class **CKO_SECRET_KEY**) hold secret keys. This version of Cryptoki recognizes six types of secret key: generic, RC2, RC4, DES, DES2, and DES3. The following table defines the attributes common to all secret keys, in addition to the common attributes listed in Table 8-1 and Table 8-5.

Table 8-14, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE	CK_BBOOL	TRUE if object is sensitive (default FALSE)
CKA_ENCRYPT	CK_BBOOL	TRUE if key supports encryption[1]
CKA_DECRYPT	CK_BBOOL	TRUE if key supports decryption[1]
CKA_SIGN	CK_BBOOL	TRUE if key supports signatures (i.e., authentication codes) where the signature is an appendix to the data[1]
CKA_VERIFY	CK_BBOOL	TRUE if key supports verification (i.e., of authentication codes) where the signature is an appendix to the data[1]
CKA_WRAP	CK_BBOOL	TRUE if key supports wrapping[1]
CKA_UNWRAP	CK_BBOOL	TRUE if key supports unwrapping[1]

[1] Default is up to the token, based on what mechanisms it supports; the application can specify an explicit value in the template, and Cryptoki may reject it if no compatible mechanism is supported.

All of these attributes may be modified after the object is created. However, the **CKA_SENSITIVE** attribute may only be set to TRUE.

If the **CKA_SENSITIVE** attribute is TRUE, then certain attributes of the secret key cannot be revealed outside the token. Also, the secret key cannot be wrapped if the **CKA_SENSITIVE** attribute is TRUE, since it could potentially be recovered outside the token if the unwrapping key is known outside. The attribute table for the each type of secret key specifies which attributes are not revealed.

8.7.1 Generic secret key objects

Generic secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_GENERIC_SECRET**) hold generic secret keys. This keys do not support encryption, decryption, signatures or verification; however, other keys can be derived from them. The following table defines the generic secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-15, Generic Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,2]	Byte array	Key value (arbitrary length)
CKA_VALUE_LEN[3]	CK_USHORT	Length in bytes of key value

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE. [3]Specify this attribute only in a template for unwrapping or deriving a key of this type.

None of these attributes may be modified after the object is created.

The following is a sample template for creating a generic secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
CK_CHAR label[] = "A generic secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_DERIVE, &true, 1},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.2 RC2 secret key objects

RC2 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_RC2**) hold RC2 keys. The following table defines the RC2 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-16, RC2 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,2]	Byte array	Key value (1 to 128 bytes)
CKA_VALUE_LEN[3]	CK_USHORT	Length in bytes of key value

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE. [3]Specify this attribute only in a template for generating, unwrapping or deriving a key of this type.

None of these attributes may be modified after the object is created.

The following is a sample template for creating an RC2 secret key object:

```
   CK_OBJECT_CLASS class = CKO_SECRET_KEY;
   CK_KEY_TYPE keyType = CKK_RC2;
   CK_CHAR label[] = “An RC2 secret key object”;
   CK_BYTE value[] = {...};
   CK_BBOOL true = TRUE;
   CK_ATTRIBUTE template[] = {
      {CKA_CLASS, &class, sizeof(class)},
      {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
      {CKA_TOKEN, &true, 1},
      {CKA_LABEL, label, sizeof(label)},
      {CKA_ENCRYPT, &true, 1},
      {CKA_VALUE, value, sizeof(value)}
   };
```

8.7.3 RC4 secret key objects

RC4 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_RC4**) hold RC4 keys. The following table defines the RC4 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-17, RC4 Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,2]	Byte array	Key value (1 to 256 bytes)
CKA_VALUE_LEN[3]	CK_USHORT	Length in bytes of key value

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE. [3]Specify this attribute only in a template for generating, unwrapping or deriving a key of this type.

None of these attributes may be modified after the object is created.

The following is a sample template for creating an RC4 secret key object:

```
   CK_OBJECT_CLASS class = CKO_SECRET_KEY;
   CK_KEY_TYPE keyType = CKK_RC4;
   CK_CHAR label[] = “An RC4 secret key object”;
   CK_BYTE value[] = {...};
   CK_BBOOL true = TRUE;
   CK_ATTRIBUTE template[] = {
      {CKA_CLASS, &class, sizeof(class)},
      {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
      {CKA_TOKEN, &true, 1},
      {CKA_LABEL, label, sizeof(label)},
      {CKA_ENCRYPT, &true, 1},
      {CKA_VALUE, value, sizeof(value)}
   };
```

8.7.4 DES secret key objects

DES secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_DES**) hold single-length DES keys. The following table defines the DES secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-18, DES Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,2]	Byte array	Key value (always 8 bytes long)

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE.

None of these attributes may be modified after the object is created.

The following is a sample template for creating a DES secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_CHAR label[] = "A DES secret key object";
CK_BYTE value[8] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, 1},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.5 DES2 secret key objects

DES2 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_DES2**) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-19, DES2 Secret Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_VALUE[1,2]	Byte array	Key value (always 16 bytes long)

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE.

None of these attributes may be modified after the object is created.

The following is a sample template for creating a double-length DES secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES2;
CK_CHAR label[] = "A DES2 secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, 1},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.6 DES3 secret key objects

DES3 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_DES3**) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-5 and Table 8-14.

Table 8-20, DES3 Secret Key Object Attributes

Attribute	**Data type**	**Meaning**
CKA_VALUE[1,2]	Byte array	Key value (always 24 bytes long)

[1]Must be specified when object is created. [2]Cannot be revealed when **CKA_SENSITIVE** attribute is TRUE.

None of these attributes may be modified after the object is created.

The following is a sample template for creating a triple-length DES secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES3;
CK_CHAR label[] = “A DES3 secret key object”;
CK_BYTE value[24] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, 1},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, 1},
   {CKA_VALUE, value, sizeof(value)}
};
```

9. Functions

Cryptoki's functions are organized into the following categories:

- general purpose
- slot and token management
- session management
- object management
- encryption and decryption
- message digesting
- signature and verification
- key management
- function management
- callback

Each function returns a **CK_RV** value. The following table lists each function return value (in alphabetical order):

Table 9-1, Return Values

Return Value	**Meaning**
CKR_ATTRIBUTE_READ_ONLY	attribute cannot be modified
CKR_ATTRIBUTE_SENSITIVE	attribute is sensitive and cannot be revealed
CKR_ATTRIBUTE_TYPE_INVALID	attribute type is invalid
CKR_ATTRIBUTE_VALUE_INVALID	attribute value is invalid
CKR_CANCEL	function should be canceled
CKR_DATA_INVALID	data is invalid
CKR_DATA_LEN_RANGE	data length is out of range
CKR_DEVICE_ERROR	device error
CKR_DEVICE_MEMORY	not enough memory on device
CKR_DEVICE_REMOVED	device has been removed
CKR_ENCRYPTED_DATA_INVALID	encrypted data is invalid
CKR_ENCRYPTED_DATA_LEN_RANGE	encrypted data length is out of range
CKR_FLAGS_INVALID	flags are invalid
CKR_FUNCTION_CANCELED	function has been canceled
CKR_FUNCTION_NOT_PARALLEL	no function is executing in parallel

Return Value	**Meaning**
CKR_FUNCTION_PARALLEL	function is executing in parallel
CKR_HOST_MEMORY	not enough memory on host
CKR_KEY_HANDLE_INVALID	key handle is invalid
CKR_KEY_SENSITIVE	key is sensitive and cannot be revealed
CKR_KEY_SIZE_RANGE	key size is out of range
CKR_KEY_TYPE_INCONSISTENT	key type is inconsistent with mechanism
CKR_MECHANISM_INVALID	mechanism is invalid
CKR_MECHANISM_PARAM_INVALID	mechanism parameter is invalid
CKR_OBJECT_CLASS_INCONSISTENT	object class is inconsistent with mechanism
CKR_OBJECT_CLASS_INVALID	object class is invalid
CKR_OBJECT_HANDLE_INVALID	object handle is invalid
CKR_OK	function has completed successfully
CKR_OPERATION_ACTIVE	another operation is already active
CKR_OPERATION_NOT_INITIALIZED	operation has not been initialized
CKR_PIN_INCORRECT	PIN is incorrect
CKR_PIN_INVALID	new PIN contains invalid characters
CKR_PIN_LEN_RANGE	new PIN length is out of range (assuming token specifies range)
CKR_SESSION_CLOSED	session has been closed
CKR_SESSION_COUNT	session limits have been reached
CKR_SESSION_EXCLUSIVE_EXISTS	an exclusive session already exists
CKR_SESSION_HANDLE_INVALID	session handle is invalid
CKR_SESSION_PARALLEL_NOT_SUPPORTED	parallel execution is not supported
CKR_SESSION_READ_ONLY	session is read-only
CKR_SIGNATURE_INVALID	signature is invalid
CKR_SIGNATURE_LEN_RANGE	signature length is out of range
CKR_SLOT_ID_INVALID	slot ID is invalid
CKR_TEMPLATE_INCOMPLETE	template is incomplete
CKR_TEMPLATE_INCONSISTENT	template is inconsistent
CKR_TOKEN_NOT_PRESENT	slot does not contain a token
CKR_TOKEN_NOT_RECOGNIZED	the token was not recognized
CKR_TOKEN_WRITE_PROTECTED	token is write-protected
CKR_UNWRAPPING_KEY_HANDLE_INVALID	unwrapping key handle is invalid
CKR_UNWRAPPING_KEY_SIZE_RANGE	unwrapping key size is out of range
CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT	unwrapping key type is inconsistent with mechanism
CKR_USER_ALREADY_LOGGED_IN	a user is already logged in
CKR_USER_NOT_LOGGED_IN	a user is not logged in
CKR_USER_PIN_NOT_INITIALIZED	the user's PIN has not been intialized
CKR_USER_TYPE_INVALID	user type is invalid
CKR_WRAPPED_KEY_INVALID	wrapped key is invalid

Return Value	Meaning
CKR_WRAPPED_KEY_LEN_RANGE	wrapped key length is out of range
CKR_WRAPPING_KEY_HANDLE_INVALID	wrapping key handle is invalid
CKR_WRAPPING_KEY_SIZE_RANGE	wrapping key size is out of range
CKR_WRAPPING_KEY_TYPE_INCONSISTENT	wrapping key type is inconsistent with mechanism

9.1 General purpose

Cryptoki provides the following general purpose functions.

♦ C_Initialize

```
CK_RV CK_ENTRY C_Initialize(
     CK_VOID_PTR pReserved
);
```

C_Initialize initializes the Cryptoki library. **C_Initialize** should be the first call made by an application. This function is implementation defined; Cryptoki may, for example, initialize its internal memory buffers, or any other resources it may require. The *pReserved* parameter is reserved for future versions. For this version, it should be set to NULL_PTR.

Return values: CKR_OK, CKR_HOST_MEMORY

Example:

```
CK_RV rv;

rv = C_Initialize(NULL_PTR);
```

♦ C_GetInfo

```
CK_RV CK_ENTRY C_GetInfo(
     CK_INFO_PTR pInfo
);
```

C_GetInfo returns general information about Cryptoki. *pInfo* points to the location that receives the information.

Return values: CKR_OK, CKR_HOST_MEMORY

Example:

```
CK_INFO info;
CK_RV rv;

rv = C_GetInfo(&info);
if( rv == CKR_OK ){
   if( info.version.major == 1 ){
      .
      .
      .
   }
}
```

9.2 Slot and token management

Cryptoki provides the following functions for slot and token management.

♦ C_GetSlotList

```
CK_RV CK_ENTRY C_GetSlotList(
     CK_BBOOL tokenPresent,
     CK_SLOT_ID_PTR pSlotList,
     CK_USHORT_PTR pusCount
);
```

C_GetSlotList obtains a list of slots in the system. *tokenPresent* indicates whether the list includes only those slots with a token present (TRUE), or all slots (FALSE); *pSlotList* points to the location that receives the list (array) of slot IDs; and *pusCount* points to the location that receives the number of slots.

The application should call this function twice. The first time, *pSlotList* should be NULL_PTR. In this case, Cryptoki only returns the number of slots. The second time, *pSlotList* should point to a location large enough to receive the list of slots. If there are no slot IDs to return, the location that *pusCount* points to receives 0.

Return values: CKR_OK, CKR_HOST_MEMORY

Example:

```
   CK_SLOT_ID_PTR pSlotList;
   CK_USHORT usCount;
   CK_RV rv;

   rv = C_GetSlotList(FALSE, NULL_PTR, &usCount);
   if( (rv == CKR_OK) && (usCount > 0) ){
      pSlotList = (CK_SLOT_ID_PTR) malloc(usCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList(FALSE, pSlotList, &usCount);
      if( rv == CKR_OK ){
         .
         .
         .
      }
      free(pSlotList);
   }
```

♦ C_GetSlotInfo

```
CK_RV CK_ENTRY C_GetSlotInfo(
     CK_SLOT_ID slotID,
     CK_SLOT_INFO_PTR pInfo
);
```

C_GetSlotInfo obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo* points to the location that receives the slot information.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_HOST_MEMORY

Example:

```
CK_SLOT_ID_PTR pSlotList;
CK_USHORT usCount;
CK_SLOT_INFO info;
CK_RV rv;

rv = C_GetSlotList(FALSE, NULL_PTR, &usCount);
if( (rv == CKR_OK) && (usCount > 0) ){
   pSlotList = (CK_SLOT_ID_PTR) malloc(usCount * sizeof(CK_SLOT_ID));
   rv = C_GetSlotList(FALSE, pSlotList, &usCount);
   if( rv == CKR_OK ){
      rv = C_GetSlotInfo(pSlotList[0], &info);
      .
      .
      .
   }
   free(pSlotList);
}
```

♦ C_GetTokenInfo

```
CK_RV CK_ENTRY C_GetTokenInfo(
     CK_SLOT_ID slotID,
     CK_TOKEN_INFO_PTR pInfo
);
```

C_GetTokenInfo obtains information about a particular token in the system. *slotID* is the ID of the token's slot; *pInfo* points to the location that receives the token information.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_HOST_MEMORY, CKR_TOKEN_NOT_RECONIZED

Example:

```
CK_SLOT_ID_PTR pSlotList;
CK_USHORT usCount;
CK_TOKEN_INFO info;
CK_RV rv;

rv = C_GetSlotList(TRUE, NULL_PTR, &usCount);
if( (rv == CKR_OK) && (usCount > 0) ){
   pSlotList = (CK_SLOT_ID_PTR) malloc(usCount * sizeof(CK_SLOT_ID));
   rv = C_GetSlotList(TRUE, pSlotList, &usCount);
   if( rv == CKR_OK ){
      rv = C_GetTokenInfo(pSlotList[0], &info);
      .
      .
      .
   }
   free(pSlotList);
}
```

♦ C_GetMechanismList

```
CK_RV CK_ENTRY C_GetMechanismList(
     CK_SLOT_ID slotID,
     CK_MECHANISM_TYPE_PTR pMechanismList,
     CK_USHORT_PTR pusCount
);
```

C_GetMechanismList obtains a list of mechanism types supported by a token. *slotID* is the ID of the token's slot; *pMechanismList* points to the location that receives the list (array) of mechanism types; and *pusCount* points to the location that receives the number of mechanisms.

The application should call this function twice. The first time, *pMechanismList* should be NULL_PTR. In this case, Cryptoki only returns the number of mechanisms supported. The second time, *pMechanismList* should point to a location large enough to receive the list of mechanism types.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_HOST_MEMORY

Example:

```
   CK_SLOT_ID slotID;
   CK_MECHANISM_TYPE_PTR pMechanismList;
   CK_USHORT usCount;
   CK_RV rv;

   rv = C_GetMechanismList(slotID, NULL_PTR, &usCount);
   if( (rv == CKR_OK) && (usCount > 0) ){
      pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(usCount *
             sizeof(CK_MECHANISM_TYPE));
      rv = C_GetMechanismList(slotID, pMechanismList, &usCount);
      if( rv == CKR_OK ){
         .
         .
         .
      }
      free(pMechanismList);
   }
```

♦ C_GetMechanismInfo

```
CK_RV CK_ENTRY C_GetMechanismInfo(
     CK_SLOT_ID slotID,
     CK_MECHANISM_TYPE type,
     CK_MECHANISM_INFO_PTR pInfo
);
```

C_GetMechanismInfo obtains information about a particular mechanism possibly supported by a token. *slotID* is the ID of the token's slot; *type* is the type of mechanism; and *pInfo* points to the location that receives the mechanism information.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_HOST_MEMORY

Example:

```
CK_SLOT_ID_PTR pSlotList;
CK_USHORT usCount;
CK_MECHANISM_INFO info;
CK_RV rv;

rv = C_GetSlotList(TRUE, NULL_PTR, &usCount);
if( (rv == CKR_OK) && (usCount > 0) ){
   pSlotList = (CK_SLOT_ID_PTR) malloc(usCount * sizeof(CK_SLOT_ID));
   rv = C_GetSlotList(TRUE, pSlotList, &usCount);
   if( rv == CKR_OK ){
      rv = C_GetMechanismInfo(pSlotList[0], CKM_MD2, &info);
      .
      .
      .
   }
   free(pSlotList);
}
```

♦ C_InitToken

```
CK_RV CK_ENTRY C_InitToken(
     CK_SLOT_ID slotID,
     CK_CHAR_PTR pPin,
     CK_USHORT usPinLen,
     CK_CHAR_PTR pLabel
);
```

C_InitToken initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN; *usPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (must be padded with the blank characters).

When a token is initialized, all objects are destroyed that can be destroyed (i.e., all except for "indestructible" objects such as keys built in to the token). Also, access by the normal user is disabled until the SO sets the normal user's PIN. Depending on the token, some "default" objects may be created, and attributes of some objects may be set to default values.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_ERROR, CKR_PIN_LEN_RANGE, CKR_TOKEN_NOT_RECOGNIZED

Example:

```
CK_SLOT_ID slotID;
CK_CHAR pin[] = {“MyPIN”};
CK_CHAR label[32];
CK_RV rv;

memset(label, ‘ ’, sizeof(label));
memcpy(label, “My first token”, sizeof(“My first token”));
rv = C_InitToken(slotID, pin, sizeof(pin), label);
if( rv == CKR_OK ){
    .
    .
    .
}
```

♦ C_InitPIN

```
CK_RV CK_ENTRY C_InitPIN(
     CK_SESSION_HANDLE hSession,
     CK_CHAR_PTR pPin,
     CK_USHORT usPinLen
);
```

C_InitPIN initializes the normal user’s PIN. *hSession* is the session’s handle; *pPin* points to the normal user’s PIN; and *usPinLen* is the length in bytes of the PIN.

This function can only be called in the “R/W SO Functions” state.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_CLOSED, CKR_USER_NOT_LOGGED_IN, CKR_PIN_LEN_RANGE, CKR_PIN_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_CHAR newPin[]= {“NewPIN”};
CK_RV rv;

rv = C_InitPIN(hSession, newPin, sizeof(newPin));
if( rv == CKR_OK ){
    .
    .
    .
}
```

♦ C_SetPIN

```
CK_RV CK_ENTRY C_SetPIN(
     CK_SESSION_HANDLE hSession,
     CK_CHAR_PTR pOldPin,
     CK_USHORT usOldLen,
     CK_CHAR_PTR pNewPin,
     CK_USHORT usNewLen
);
```

C_SetPIN modifies the PIN of user that is currently logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *usOldLen* is the length of the old PIN; *pNewPin* points to the new PIN; and *usNewLen* is the length of the new PIN.

This function can only be called in the "R/W SO Functions" state or "R/W User Functions" state.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_CLOSED, CKR_USER_NOT_LOGGED_IN, CKR_PIN_INCORRECT, CKR_PIN_LEN_RANGE, CKR_PIN_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_CHAR oldPin[] = {"OldPIN"};
CK_CHAR newPin[] = {"NewPIN"};
CK_RV rv;

rv = C_SetPIN(hSession, oldPin, sizeof(oldPin), newPin, sizeof(newPin));
if( rv == CKR_OK ){
   .
   .
   .
}
```

9.3 Session management

Cryptoki provides the following functions for session management.

A typical application would call **C_OpenSession** after selecting a token and **C_CloseSession** after completing all operations with the token. Only in special cases, such as when other applications connected to a token have failed, would an application call **C_CloseAllSessions**.

An application may have concurrent sessions with more than one token. It is also possible that a token may have concurrent sessions with more than one application.

♦ C_OpenSession

```
CK_RV CK_ENTRY C_OpenSession(
     CK_SLOT_ID slotID,
     CK_FLAGS flags,
     CK_VOID_PTR pApplication,
     CK_RV CK_ENTRY (*Notify)(CK_SESSION_HANDLE hSession,
        CK_NOTIFICATION event, CK_VOID_PTR pApplication),
     CK_SESSION_HANDLE_PTR phSession
);
```

C_OpenSession opens a session between an application and a token. *slotID* is the slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to the notification callback; *Notify* is the address of the notification callback function; and *phSession* points to the location that receives the handle for the new session.

The *flags* parameter consists of the logical-or of zero or more bit flags defined in the **CK_SESSION_INFO** data type. If no bits are set in the *flags* parameter, then the session is opened as a shared, read-only session, with the cryptographic functions performed in parallel with the application (assuming the token has this capability—otherwise functions are performed in serial).

In a parallel session, cryptographic functions may return control to the application before completing (the return value CKR_FUNCTION_PARALLEL indicates this condition). The application may call **C_GetFunctionStatus** to obtain updated status of the function, which will be CKR_FUNCTION_PARALLEL until the function completes, and CKR_OK or another return value indicating an error when the function completes. Alternatively, the application can wait until Cryptoki sends notification that the function has completed through the **Notify** callback. The application may also call **C_CancelFunction** to cancel the function.

If an application calls another function (cryptographic or otherwise) before one that is executing in parallel completes, Cryptoki will wait until the one that is executing completes. Thus an application can run only one function at any given time in a given session. (To achieve parallel execution of multiple functions, the application should open additional sessions.)

Cryptographic functions running in serial with the application may surrender control through the **Notify** callback, so that the application may perform other operations or cancel the function.

Non-cryptographic functions always run in serial with the application, and do not surrender control.

There may be a limit on the number of concurrent sessions with the token, which may depend on whether the session is "read-only" or "read/write." There can only be one exclusive session with a token.

If the token is in "write-protected" (as indicated in the CK_TOKEN_INFO structure), then the session also must be "read-only."

The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not support the callback, it should pass NULL_PTR as the address. The *Notify* callback function is described in Section 0.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_FLAGS_INVALID, CKR_SESSION_COUNT, CKR_SESSION_PARALLEL_NOT_SUPPORTED, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_SLOT_ID slotID;
CK_RV rv;
CK_BYTE application;

CK_RV CK_ENTRY MyNotify(CK_SESSION_HANDLE hSession,
      CK_NOTIFICATION event, CK_VOID_PTR pApplication);

slotID = 1;
rv = C_OpenSession(slotID, CKF_EXCLUSIVE_SESSION, &application, MyNotify,
         &hSession);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_CloseSession

```
CK_RV CK_ENTRY C_CloseSession(
     CK_SESSION_HANDLE hSession
);
```

C_CloseSession closes a session between an application and a token. *hSession* is the session's handle.

When a session is closed, session objects created during the session are destroyed automatically, and if a function is running in parallel with the application, it is canceled.

Depending on the token, when the last session with the token is closed, the token may be "ejected" from its reader, assuming this capability exists.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_SLOT_ID slotID;
```

```
CK_RV rv;
CK_BYTE application;

CK_RV CK_ENTRY MyNotify(CK_SESSION_HANDLE hSession,
      CK_NOTIFICATION event, CK_VOID_PTR pApplication);

slotID = 1;
rv = C_OpenSession(slotID, CKF_EXCLUSIVE_SESSION, &application, MyNotify,
         &hSession);
if( rv == CKR_OK ){
   .
   .
   .
   C_CloseSession(hSession);
}
```

♦ C_CloseAllSessions

```
CK_RV CK_ENTRY C_CloseAllSessions(
     CK_SLOT_ID slotID
);
```

C_CloseAllSessions closes all sessions with a token. *slotID* specifies the token's slot.

This function should only be called when there is no other way to recover control of a token, such as when other applications connected to the token have failed.

Depending on the token, the token may be "ejected" from its reader, assuming this capability exists.

When an application is disconnected from a token in this manner, it receives a CKR_SESSION_CLOSED error on its next call to Cryptoki.

Return values: CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SLOT_ID slotID;
CK_RV rv;

slotID = 1;
rv = C_CloseAllSessions(slotID);
```

♦ C_GetSessionInfo

```
CK_RV CK_ENTRY C_GetSessionInfo(
     CK_SESSION_HANDLE hSession,
     CK_SESSION_INFO_PTR pInfo
);
```

C_GetSessionInfo obtains information about the session. *hSession* is the session's handle; and *pInfo* points to the location that receives the session information.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED

Example:

```
CK_SESSION_HANDLE hSession;
CK_SESSION_INFO info;
CK_RV rv;

rv = C_GetSessionInfo(hSession, &info);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_Login

```
CK_RV CK_ENTRY C_Login(
     CK_SESSION_HANDLE hSession,
     CK_USER_TYPE userType,
     CK_CHAR_PTR pPin,
     CK_USHORT usPinLen
);
```

C_Login logs a user into a token. *hSession* is the session's handle; *userType* is the user type; *pPin* points to the user's PIN; and *usPinLen* is the length of the PIN. Depending on the user type and the current session type, the state will become one of the following: "R/W SO Functions", "R/O SO Functions", "R/W User Functions", or "R/O User Functions".

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_TYPE_INVALID, CKR_PIN_INCORRECT, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_CHAR userPIN[] = {"MyPIN"};
CK_RV rv;

rv = C_Login(hSession, CKU_USER, userPIN, sizeof(userPIN));
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_Logout

```
CK_RV CK_ENTRY C_Logout(
     CK_SESSION_HANDLE hSession
);
```

C_Logout logs a user out from a token. *hSession* is the session's handle. Depending on the current user type and the current session type, the state will become either "R/W Public Session" or "R/O Public Session".

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_USER_NOT_LOGGED_IN, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_CHAR userPIN[] = {"MyPIN"};
CK_RV rv;

rv = C_Login(hSession, CKU_USER, userPIN, sizeof(userPIN));
if( rv == CKR_OK ){
   .
   .
   .
   C_Logout(hSession);
}
```

9.4 Object management

Cryptoki provides the following functions for managing objects. Additional functions for managing key objects are described in Section 9.8.

♦ C_CreateObject

```
CK_RV CK_ENTRY C_CreateObject(
     CK_SESSION_HANDLE hSession,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount,
     CK_OBJECT_HANDLE_PTR phObject
);
```

C_CreateObject creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's template; *usCount* is the number of attributes in the template; and *phObject* points to the location that receives the new object's handle.

Only session object can be created during a read-only session. Only public objects can be created when no user is logged in.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_CLOSED, CKR_OBJECT_CLASS_INVALID, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE
   hData,
   hCertificate,
   hKey;
CK_OBJECT_CLASS
    dataClass = CKO_DATA,
    certificateClass = CKO_CERTIFICATE,
    keyClass = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_CHAR application[] = {“My Application”};
CK_BYTE dataValue[] = {...};
CK_BYTE subject[] = {...};
CK_BYTE id[] = {...};
CK_BYTE certificateValue[] = {...};
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BYTE true = TRUE;
CK_ATTRIBUTE dataTemplate[] = {
   {CKA_CLASS, &dataClass, sizeof(dataClass)},
   {CKA_TOKEN, &true, 1},
   {CKA_APPLICATION, application, sizeof(application)},
   {CKA_VALUE, dataValue, sizeof(dataValue)}
};
CK_ATTRIBUTE certificateTemplate[] = {
   {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
   {CKA_TOKEN, &true, 1},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_VALUE, certificateValue, sizeof(certificateValue)}
};
CK_ATTRIBUTE keyTemplate[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_WRAP, &true, 1},
   {CKA_MODULUS, modulus, sizeof(modulus)},
   {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
 };
CK_RV rv;


/* Create a data object */
rv = C_CreateObject(hSession, &dataTemplate, 4, &hData);
if( rv == CKR_OK ){
   .
   .
   .
}
/* Create a certificate object */
rv = C_CreateObject(hSession, &certificateTemplate, 5, &hCertificate);
```

```
    if( rv == CKR_OK ){
       .
       .
       .
    }
    /* Create a RSA private key object */
    rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
    if( rv == CKR_OK ){
       .
       .
       .
    }
```

♦ C_CopyObject

```
CK_RV CK_ENTRY C_CopyObject(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount,
     CK_OBJECT_HANDLE_PTR phNewObject
);
```

C_CopyObject copies an object, creating a new object for the copy. *hSession* is the session's handle; *hObject* is the object's handle; *pTemplate* points to the template for the new object; *usCount* is the number of attributes in the template; and *phNewObject* points to the location that receives the handle for the copy of the object.

The template may specify new values of any attributes of the object that can ordinarily be modified, and it may also specify new values of the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object).

Only session objects can be created during a read-only session. Only public objects can be created when no user is logged in.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_CLOSED, CKR_OBJECT_HANDLE_INVALID, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hKey;
    CK_OBJECT_HANDLE hNewKey;
    CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
    CK_KEY_TYPE keyType = CKK_DES;
    CK_BYTE id[] = {...};
    CK_BYTE keyValue[] = {...};
    CK_BYTE false = FALSE;
    CK_BYTE true = TRUE;
```

```
CK_ATTRIBUTE keyTemplate[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &false, 1},
   {CKA_ID, id, sizeof(id)},
   {CKA_VALUE, keyValue, sizeof(keyValue)}
};
CK_ATTRIBUTE copyTemplate[] = {
   {CKA_TOKEN, &true, 1}
};
CK_RV rv;

/* Create a DES secret key session object */
rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
if( rv == CKR_OK ){
   /* Create a copy on the token */
   rv = C_CopyObject(hSession, hKey, &copyTemplate, 1, &hNewKey);
   .
   .
   .
}
```

♦ C_DestroyObject

```
CK_RV CK_ENTRY C_DestroyObject(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject
);
```

C_DestroyObject destroys an object. *hSession* is the session's handle; and *hObject* is the object's handle.

Only session objects can be destroyed during a read-only session. Only public objects can be destroyed when no user is logged in.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_CLOSED, CKR_OBJECT_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_OBJECT_CLASS dataClass = CKO_DATA;
CK_CHAR application[] = {"My Application"};
CK_BYTE value[] = {...};
CK_BYTE true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &dataClass, sizeof(dataClass)},
   {CKA_TOKEN, &true, 1},
   {CKA_APPLICATION, application, sizeof(application)},
   {CKA_VALUE, value, sizeof(value)}
};
CK_RV rv;

rv = C_CreateObject(hSession, &template, 4, &hObject);
```

```
    if( rv == CKR_OK ){
       .
       .
       .
       C_DestroyObject(hSession, hObject);
    }
```

♦ C_GetObjectSize

```
CK_RV CK_ENTRY C_GetObjectSize(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject,
     CK_USHORT_PTR pusSize
);
```

C_GetObjectSize gets the size of an object in bytes. *hSession* is the session's handle; *hObject* is the object's handle; and *pusSize* points to the location that receives the size in bytes of the object.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OBJECT_HANDLE_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hObject;
    CK_OBJECT_CLASS dataClass = CKO_DATA;
    CK_CHAR application[] = {"My Application"};
    CK_BYTE dataValue[] = {...};
    CK_BYTE value[] = {...};
    CK_BYTE true = TRUE;
    CK_ATTRIBUTE template[] = {
       {CKA_CLASS, &dataClass, sizeof(dataClass)},
       {CKA_TOKEN, &true, 1},
       {CKA_APPLICATION, application, sizeof(application)},
       {CKA_VALUE, value, sizeof(value)}
    };
    CK_USHORT usSize;
    CK_RV rv;

    rv = C_CreateObject(hSession, &template, 4, &hObject);
    if( rv == CKR_OK ){
       rv = C_GetObjectSize(hSession, hObject, &usSize);
       .
       .
       .
       C_DestroyObject(hSession, hObject);
    }
```

♦ C_GetAttributeValue

```
CK_RV CK_ENTRY C_GetAttributeValue(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount
);
```

C_GetAttributeValue obtains the value of one or more object attributes. *hSession* is the session's handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute values are to be obtained, and receives the attribute values; and *usCount* is the number of attributes in the template.

The application must ensure that the location that receives a attribute value can hold the value. If it does not know the length of the value, then the application should pass NULL_PTR as the *pValue* parameter for the attribute in the template and **C_GetAttributeValue** will only return the length of the value. See Section 8 for more details on attributes.

If the object is marked "sensitive", it may not be possible to obtain the value of the attribute.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OBJECT_HANDLE_INVALID, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_SENSITIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_BYTE_PTR pModulus, pExponent;
CK_ATTRIBUTE template[] = {
    {CKA_MODULUS, NULL_PTR, 0},
    {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
};
CK_RV rv;

rv = C_GetAttributeValue(hSession, hObject, &template, 2);
if( rv == CKR_OK ){
   pModulus = (CK_BYTE_PTR) malloc(template[0].usValueLen);
   template[0].pValue = pModulus;
   pExponent = (CK_BYTE_PTR) malloc(template[1].usValueLen);
   template[1].pValue = pExponent;
   rv = C_GetAttributeValue(hSession, hObject, &template, 2);
   if( rv == CKR_OK ){
      .
      .
      .
   }
   free(pModulus);
   free(pExponent);
}
```

♦ C_SetAttributeValue

```
CK_RV CK_ENTRY C_SetAttributeValue(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount
);
```

C_SetAttributeValue modifies the value of one or more attributes of an object. *hSession* is the session's handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute values are to be modified and their new values; and *usCount* is the number of attributes in the template.

Only session objects can be modified during a read-only session.

Not all attributes can be modified; see Section 8 for more details.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY CKR_SESSION_CLOSED, CKR_OBJECT_HANDLE_INVALID, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_CHAR label[] = {“New label”};
CK_ATTRIBUTE template[] = {
   CKA_LABEL, label, sizeof(label)
};
CK_RV rv;

rv = C_SetAttributeValue(hSession, hObject, &template, 1);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_FindObjectsInit

```
CK_RV CK_ENTRY C_FindObjectsInit(
     CK_SESSION_HANDLE hSession,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount
);
```

C_FindObjectsInit initializes a search for token and session objects that match a template. *hSession* is the session's handle; *pTemplate* points to a search template that specifies the attribute values to match; and *usCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-byte match with all attributes in the template. To find all objects, set *usCount* is 0.

After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain the handles of the objects matching the template. At most one search operation may be active at a given time in a given session.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_FindObjects**.

♦ C_FindObjects

```
CK_RV CK_ENTRY C_FindObjects(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE_PTR phObject,
     CK_USHORT usMaxObjectCount,
     CK_USHORT_PTR pusObjectCount
);
```

C_FindObjects continues a search for token and session objects that match a template, obtaining additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives the list (array) of additional object handles; *usMaxObjectCount* is the maximum number of object handles to be returned; and *pusObjectCount* points to the location that receives the actual number of object handles returned. If there are no more objects matching the template, then the location that *pusObjectCount* points to receives 0.

The search must have been initialized with **C_FindObjectsInit**.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_USHORT usObjectCount;
CK_RV rv;

rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
if( rv == CKR_OK ){
  while (1) {
    rv = C_FindObjects(hSession, &hObject, 1, &usObjectCount);
    if (rv != CKR_OK || usObjectCount == 0)
      break;
    .
    .
    .
  }
}
```

9.5 Encryption and decryption

Cryptoki provides the following functions for encrypting and decrypting data. All these functions run in parallel with the application if the session was opened with the CKF_SERIAL_SESSION flag set to FALSE and the token supports parallel execution.

♦ C_EncryptInit

```
CK_RV CK_ENTRY C_EncryptInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; and *hKey* is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, must be TRUE.

After calling **C_EncryptInit**, the application may call **C_Encrypt** to encrypt data in a single part, or **C_EncryptUpdate** one or more times followed by **C_EncryptFinal** to encrypt data in multiple parts. The encryption operation is "active" until the application calls **C_Encrypt** or **C_EncryptFinal**. To process additional data (in single or multiple parts), the application must call **C_EncryptInit** again. At most one cryptographic operation may be active at a given time in a given session. **C_EncryptInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-2, Encryption Mechanisms

Mechanism	**Key type**
PKCS #1 RSA[1]	RSA public
X.509 (raw) RSA[1]	RSA public
RC2 (ECB and CBC mode)	RC2
RC4	RC4
DES (ECB and CBC mode)	DES
triple-DES (ECB and CBC mode)	double or triple-length DES

[1] Single-part only.

Section 0 provides more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_Encrypt**.

♦ C_Encrypt

```
CK_RV CK_ENTRY C_Encrypt(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_USHORT usDataLen,
     CK_BYTE_PTR pEncryptedData,
     CK_USHORT_PTR pusEncryptedDataLen
);
```

C_Encrypt encrypts single-part data. *hSession* is the session's handle; *pData* points to the data; *usDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the encrypted data; and *pusEncryptedData* points to the location that receives the length in bytes of the encrypted data.

The encryption operation must have been initialized with **C_EncryptInit**.

For constraints on data length, refer to the description of the encryption mechanism.

C_Encrypt is equivalent to a sequence of **C_EncryptUpdate** and **C_EncryptFinal**.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DES_ECB, NULL_PTR, 0
};
CK_BYTE encryptedData[8];
CK_USHORT usEncryptedDataLen;
CK_BYTE data[8];
CK_RV rv;

memset(data, ‘A’, sizeof(data));
rv = C_EncryptInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_Encrypt(hSession, data, sizeof(data), encryptedData,
         &usEncryptedDataLen);
}
```

♦ C_EncryptUpdate

```
CK_RV CK_ENTRY C_EncryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_USHORT usPartLen,
     CK_BYTE_PTR pEncryptedPart,
     CK_USHORT_PTR pusEncryptedPartLen
);
```

C_EncryptUpdate continues a multiple-part encryption operation, processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *usPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the encrypted data part; and *pusEncryptedPart* points to the location that receives the length of the encrypted data part.

The encryption operation must have been initialized with **C_EncryptInit**. This function may be called any number of times in succession.

For constraints on data length, refer to the description of the encryption mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_EncryptFinal.**

♦ C_EncryptFinal

```
CK_RV CK_ENTRY C_EncryptFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pLastEncryptedPart,
     CK_USHORT_PTR pusEncryptedPartLen
);
```

C_EncryptFinal finishes a multiple-part encryption operation. *hSession* is the session's handle; *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any; and *pusLastEncryptedPartLen* points to the location that receives the length of the last encrypted data part.

The encryption operation must have been initialized with **C_EncryptInit**.

For constraints on data length, refer to the description of the encryption mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
    #define BUF_SZ 512
```

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
   CKM_DES_CBC, iv, sizeof(iv)
};
CK_BYTE encryptedData[BUF_SZ];
CK_USHORT usEncryptedDataLen;
CK_BYTE data[2*BUF_SZ];
CK_RV rv;

memset(iv, 0, sizeof(iv));
memset(data, 'A', 2*BUF_SZ);
rv = C_EncryptInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   C_EncryptUpdate(hSession, &data[0], BUF_SZ, encryptedData,
         &usEncryptedDataLen);
   .
   .
   .
   C_EncryptUpdate(hSession, &data[BUF_SZ], BUF_SZ, encryptedData,
         &usEncryptedDataLen);
   .
   .
   .
   C_EncryptFinal(hSession, encryptedData, &usEncryptedDataLen);
}
```

♦ C_DecryptInit

```
CK_RV CK_ENTRY C_DecryptInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_DecryptInit initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to the decryption mechanism; and *hKey* is the handle of the decryption key.

The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption, must be TRUE.

After calling **C_DecryptInit**, the application may call **C_Decrypt** to encrypt data in a single part, or **C_DecryptUpdate** one or more times followed by **C_DecryptFinal** to encrypt data in multiple parts. The decryption operation is "active" until the application calls **C_Decrypt** or **C_DecryptFinal**. To process additional data (in single or multiple parts), the application must call **C_DecryptInit** again. At most one cryptographic operation may be active at a given time in a given session. **C_DecryptInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-3, Decryption Mechanisms

Mechanism	**Key type**
PKCS #1 RSA[1]	RSA public
X.509 (raw) RSA[1]	RSA public
RC2 (ECB and CBC mode)	RC2
RC4	RC4
DES (ECB and CBC mode)	DES
triple-DES (ECB and CBC mode)	double or triple-length DES

[1] Single-part only.

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_Decrypt**.

♦ C_Decrypt

```
CK_RV CK_ENTRY C_Decrypt(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedData,
     CK_USHORT usEncryptedDataLen,
     CK_BYTE_PTR pData,
     CK_USHORT_PTR pusDataLen
);
```

C_Decrypt decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData* points to the encrypted data; *usEncryptedDataLen* is the length of the encrypted data; *pData* points to the location that receives the recovered data; and *pusDataLen* points to the location that receives the length of the recovered data.

The decryption operation must have been initialized with **C_DecryptInit**.

For constraints on data length, refer to the description of the decryption mechanism.

C_Decrypt is equivalent to a sequence of **C_DecryptUpdate** and **C_DecryptFinal**.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_ENCRYPTED_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DES_ECB, NULL_PTR, 0
};
CK_BYTE encryptedData[8];
CK_BYTE data[8];
CK_USHORT usDataLen;
CK_RV rv;

memset(encryptedData, ‘A’, sizeof(encryptedData));
rv = C_DecryptInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_Decrypt(hSession, encryptedData, sizeof(encryptedData), data,
         &usDataLen);
}
```

♦ C_DecryptUpdate

```
CK_RV CK_ENTRY C_DecryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedPart,
     CK_USHORT usEncryptedPartLen,
     CK_BYTE_PTR pPart,
     CK_USHORT_PTR pusPartLen
);
```

C_DecryptUpdate continues a multiple-part decryption operation, processing another encrypted data part. *hSession* is the session’s handle; *pEncryptedPart* points to the encrypted data part; *usEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the recovered data part; and *pusPartLen* points to the location that receives the length of the recovered data part.

The decryption operation must have been initialized with **C_DecryptInit**. This function may be called any number of times in succession.

For constraints on data length, refer to the description of the decryption mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_ENCRYPTED_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_DecryptFinal**.

♦ C_DecryptFinal

```
CK_RV CK_ENTRY C_DecryptFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pLastPart,
     CK_USHORT_PTR usLastPartLen
);
```

C_DecryptFinal finishes a multiple-part decryption operation. *hSession* is the session's handle; *pLastPart* points to the location that receives the last recovered data part, if any; and *pusLastPartLen* points to the location that receives the length of the last recovered data part.

The decryption operation must have been initialized with **C_DecryptInit**.

For constraints on data length, refer to the description of the decryption mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_ENCRYPTED_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
    CKM_DES_CBC, iv, sizeof(iv)
};
CK_BYTE encryptedData[2*BUF_SZ];
CK_BYTE data[BUF_SZ];
CK_USHORT usDataLen;
CK_RV rv;

memset(iv, 0, sizeof(iv));
memset(encryptedData, ‘A’, 2*BUF_SZ);
rv = C_DecryptInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   C_DecryptUpdate(hSession, &encryptedData[0], BUF_SZ, data, &usDataLen);
   .
   .
   .
   C_DecryptUpdate(hSession, &encryptedData[BUF_SZ], BUF_SZ, data,
          &usDataLen);
   .
   .
   .
   C_DecryptFinal(hSession, data, &usDataLen);
}
```

9.6 Message digesting

Cryptoki provides the following functions for digesting data. All these functions run in parallel with the application if the session was opened with the CKF_SERIAL_SESSION flag set to FALSE and the token supports parallel execution.

♦ C_DigestInit

```
CK_RV CK_ENTRY C_DigestInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism
);
```

C_DigestInit initializes a message-digesting operation. *hSession* is the session's handle; and *pMechanism* points to the digesting mechanism.

After calling **C_DigestInit**, the application may call **C_Digest** to digest in a single part, or **C_DigestUpdate** one or more times followed by **C_DigestFinal** to digest data in multiple parts. The message-digesting operation is "active" until the application calls **C_Digest** or **C_DigestFinal**. To process additional data (in single or multiple parts), the application must call **C_DigestInit** again. At most one cryptographic operation may be active at a given time in a given session. **C_DigestInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-4, Digesting Mechanisms

Mechanism
MD2
MD5
SHA-1

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_Digest**.

♦ C_Digest

```
CK_RV CK_ENTRY C_Digest(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_USHORT usDataLen,
     CK_BYTE_PTR pDigest,
     CK_USHORT_PTR pusDigestLen
);
```

C_Digest digests data in a single part. *hSession* is the session's handle, *pData* points to the data; *usDataLen* is the length of the data; *pDigest* points to the location that receives the message digest; and *pusDigestLen* points to the location that receives the length of the message digest.

The digest operation must have been initialized with **C_DigestInit**.

For constraints on data length, refer to the description of the message-digesting mechanism.

C_Digest is equivalent to a sequence of **C_DigestUpdate** and **C_DigestFinal**.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_MECHANISM mechanism = {
   CKM_MD2, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE digest[16];
CK_USHORT usDigestLen;
CK_RV rv;

rv = C_DigestInit(hSession, &mechanism);
if( rv == CKR_OK ){
   rv = C_Digest(hSession, data, sizeof(data), digest, &usDigestLen);
}
```

♦ C_DigestUpdate

```
CK_RV CK_ENTRY C_DigestUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_USHORT usPartLen
);
```

C_DigestUpdate continues a multiple-part message-digesting operation, processing another data part. *hSession* is the session's handle, *pPart* points to the data part; and *usPartLen* is the length of the data part.

The message-digesting operation must have been initialized with **C_DigestInit**. This function may be called any number of times in succession.

For constraints on data length, refer to the description of the message-digesting mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_DigestFinal**.

♦ C_DigestFinal

```
CK_RV CK_ENTRY C_DigestFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pDigest,
     CK_USHORT_PTR pusDigestLen
);
```

C_DigestFinal finishes a multiple-part message-digesting operation, returning the message digest. *hSession* is the session's handle; *pDigest* points to the location that receives the message digest; and *pusDigestLen* points to the location that receives the length of the message digest.

The message-digesting operation must have been initialized with **C_DigestInit**.

For constraints on data length, refer to the description of the message-digesting mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Cryptoki provides the following functions for digesting data.

Example:

```
    CK_SESSION_HANDLE hSession;
    CK_MECHANISM mechanism = {
        CKM_MD2, NULL_PTR, 0
    };
    CK_BYTE data[] = {...};
    CK_BYTE digest[16];
    CK_USHORT usDigestLen;
    CK_RV rv;

    rv = C_DigestInit(hSession, &mechanism);
    if( rv == CKR_OK ){
        rv = C_DigestUpdate(hSession, data, sizeof(data));
        .
        .
        .
        rv = C_DigestFinal(hSession, digest, &usDigestLen);
    }
```

9.7 Signature and verification

Cryptoki provides the following functions for signing data and verifying signatures. (For the purposes of Cryptoki, these operations also encompass data authentication codes.) All these functions run in parallel with the application if the session was opened with the CKF_SERIAL_SESSION flag set to FALSE and the token supports parallel execution.

♦ C_SignInit

```
CK_RV CK_ENTRY C_SignInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_SignInit initializes a signature operation, where the signature is an appendix to the data. *hSession* is the session's handle; *pMechanism* points to the signature mechanism; and *hKey* is the handle of the signature key.

The CKA_SIGN attribute of the signature key, which indicates whether the key supports signatures with appendix, must be TRUE.

After calling **C_SignInit**, the application may call **C_Sign** to sign in a single part, or **C_SignUpdate** one or more times followed by **C_SignFinal** to sign data in multiple parts. The signature operation is "active" until the application calls **C_Sign** or **C_SignFinal**. To process additional data (in single or multiple parts), the application must call **C_SignInit** again. At most one cryptographic operation may be active at a given time in a given session. **C_SignInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-5, Signature Mechanisms

Mechanism	**Key type**
PKCS #1 RSA[1]	RSA private
ISO/IEC 9796 RSA[1]	RSA private
X.509 (raw) RSA[1]	RSA private
DSA[1]	DSA private
RC2-MAC	RC2
DES-MAC	DES
triple-DES-MAC	double-length or triple-length DES

[1] Single-part only.

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_Sign**.

♦ C_Sign

```
CK_RV CK_ENTRY C_Sign(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_USHORT usDataLen,
     CK_BYTE_PTR pSignature,
     CK_USHORT_PTR pusSignatureLen
);
```

C_Sign signs data in a single part, where the signature is an appendix to the data. *hSession* is the session's handle; *pData* points to the data; *usDataLen* is the length of the data; *pSignature* points to the location that receives the signature; and *pusSignatureLen* points to the location that receives the length of the signature.

The signature operation must have been initialized with **C_SignInit**.

For constraints on data length, refer to the description of the signature mechanism.

C_Sign is equivalent to a sequence of **C_SignUpdate** and **C_SignFinal**.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DSA, NULL_PTR, 0
};
CK_BYTE data[20] = {...};
CK_BYTE signature[40];
CK_USHORT usSignatureLen;
CK_RV rv;

rv = C_SignInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_Sign(hSession, data, sizeof(data), signature, &usSignatureLen);
}
```

♦ C_SignUpdate

```
CK_RV CK_ENTRY C_SignUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_USHORT usPartLen
);
```

C_SignUpdate continues a multiple-part signature operation, processing another data part. *hSession* is the session's handle, *pPart* points to the data part; and *usPartLen* is the length of the data part.

The signature operation must have been initialized with **C_SignInit**. This function may be called any number of times in succession.

For constraints on data length, refer to the description of the signature mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_SignFinal**.

♦ C_SignFinal

```
CK_RV CK_ENTRY C_SignFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_USHORT_PTR pusSignatureLen
);
```

C_SignFinal finishes a multiple-part signature operation, returning the signature. *hSession* is the session's handle; *pSignature* points to the location that receives the signature; and *pusSignatureLen* points to the location that receives the length of the signature.

The signature operation must have been initialized with **C_SignInit**.

For constraints on data length, refer to the description of the signature mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE mac[4];
CK_USHORT usMacLen;
CK_RV rv;

rv = C_SignInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_SignUpdate(hSession, data, sizeof(data));
   .
   .
   .
   rv = C_SignFinal(hSession, mac, &usMacLen);
}
```

♦ C_SignRecoverInit

```
CK_RV CK_ENTRY C_SignRecoverInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_SignRecoverInit initializes a signature operation, where the data can be recovered from the signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature mechanism; and *hKey* is the handle of the signature key.

The CKA_SIGN_RECOVER attribute of the signature key, which indicates whether the key supports signatures where the data can be recovered from the signature, must be TRUE.

After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The signature operation is "active" until the application calls **C_SignRecover**. At most one cryptographic operation may be active at a given time in a given session. **C_SignRecoverInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-6, Signature With Recovery Mechanisms

Mechanism	**Key type**
PKCS #1 RSA	RSA private
ISO/IEC 9796 RSA	RSA private
X.509 (raw) RSA	RSA private

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_SignRecover**.

♦ C_SignRecover

```
CK_RV CK_ENTRY C_SignRecover(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_USHORT usDataLen,
     CK_BYTE_PTR pSignature,
     CK_USHORT_PTR pusSignatureLen
);
```

C_SignRecover signs data in a single operation, where the data can be recovered from the signature. *hSession* is the session's handle; *pData* points to the data; *usDataLen* is the length of the data; *pSignature* points to the location that receives the signature; and *pusSignatureLen* points to the location that receives the length of the signature.

The signature operation must have been initialized with **C_SignRecoverInit**.

For constraints on data length, refer to the description of the signature mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hKey;
    CK_MECHANISM mechanism = {
       CKM_RSA_9796, NULL_PTR, 0
    };
    CK_BYTE data[] = {...};
    CK_BYTE signature[128];
    CK_USHORT usSignatureLen;
    CK_RV rv;

    rv = C_SignRecoverInit(hSession, &mechanism, hKey);
    if( rv == CKR_OK ){
       rv = C_SignRecover(hSession, data, sizeof(data), signature,
             &usSignatureLen);
    }
```

♦ C_VerifyInit

```
CK_RV CK_ENTRY C_VerifyInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_VerifyInit initializes a verification operation, where the signature is an appendix to the data. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism; and *hKey* is the handle of the verification key.

The CKA_VERIFY attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, must be TRUE.

After calling **C_VerifyInit**, the application may call **C_Verify** to verify a signature on data in a single part, or **C_VerifyUpdate** one or more times followed by **C_VerifyFinal** to verify a signature on data in multiple parts. The verification operation is "active" until the application calls **C_Verify** or **C_VerifyFinal**. To process additional data (in single or multiple parts), the application must call **C_VerifyInit** again. At most one cryptographic operation may be active at a given time in a given session. **C_VerifyInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-7, Verification Mechanisms

Mechanism	**Key type**
PKCS #1 RSA[1]	RSA public
ISO/IEC 9796 RSA[1]	RSA public
X.509 (raw) RSA[1]	RSA public
DSA[1]	DSA public
RC2-MAC	RC2
DES-MAC	DES
triple-DES-MAC	double-length or triple-length DES

[1] Single-part only.

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_Verify**.

♦ C_Verify

```
CK_RV CK_ENTRY C_Verify(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_USHORT usDataLen,
     CK_BYTE_PTR pSignature,
     CK_USHORT usSignatureLen
);
```

C_Verify verifies a signature in a single-part operation, where the signature is an appendix to the data. *hSession* is the session's handle; *pData* points to the data; *usDataLen* is the length of the data; *pSignature* points to the signature; and *usSignatureLen* is the length of the signature.

The verification operation must have been initialized with **C_VerifyInit**.

For constraints on data length, refer to the description of the verification mechanism.

C_Verify is equivalent to a sequence of **C_VerifyUpdate** and **C_VerifyFinal**.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DSA, NULL_PTR, 0
};
CK_BYTE data[20] = {...};
CK_BYTE signature[40];
CK_RV rv;

rv = C_VerifyInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_Verify(hSession, data, sizeof(data), signature,
          sizeof(signature));
}
```

♦ C_VerifyUpdate

```
CK_RV CK_ENTRY C_VerifyUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_USHORT usPartLen
);
```

C_VerifyUpdate continues a multiple-part verification operation, processing another data part. *hSession* is the session's handle, *pPart* points to the data part; and *usPartLen* is the length of the data part.

The verification operation must have been initialized with **C_VerifyInit**. This function may be called any number of times in succession.

For constraints on data length, refer to the description of the verification mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_VerifyFinal**.

♦ C_VerifyFinal

```
CK_RV CK_ENTRY C_VerifyFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_USHORT usSignatureLen
);
```

C_VerifyFinal finishes a multiple-part verification operation, checking the signature. *hSession* is the session's handle; *pSignature* points to the signature; and *usSignatureLen* is the length of the signature.

The verification operation must have been initialized with **C_VerifyInit**.

For constraints on data length, refer to the description of the verification mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE mac[4];
CK_RV rv;

rv = C_VerifyInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_VerifyUpdate(hSession, data, sizeof(data));
   .
   .
   .
   rv = C_VerifyFinal(hSession, mac, sizeof(mac));
}
```

♦ C_VerifyRecoverInit

```
CK_RV CK_ENTRY C_VerifyRecoverInit(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hKey
);
```

C_VerifyRecoverInit initializes a signature verification operation, where the data is recovered from the signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism; and *hKey* is the handle of the verification key.

The CKA_VERIFY_RECOVER attribute of the verification key, which indicates whether the key supports verification where the data is recovered from the signature, must be TRUE.

After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on data in a single part. The verification operation is "active" until the application calls **C_VerifyRecover**. At most one cryptographic operation may be active at a given time in a given session. **C_VerifyRecoverInit** cannot initialize a new operation if another is already active.

The following mechanisms are supported in this version:

Table 9-8, Verification With Recovery Mechanisms

Mechanism	Key type
PKCS #1 RSA	RSA public
ISO/IEC 9796 RSA	RSA public
X.509 (raw) RSA	RSA public

Section 10 gives more details on the mechanisms.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: See **C_VerifyRecover**.

♦ C_VerifyRecover

```
CK_RV CK_ENTRY C_VerifyRecover(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_USHORT usSignatureLen,
     CK_BYTE_PTR pData,
     CK_USHORT_PTR pusDataLen
);
```

C_VerifyRecover verifies a signature in a single-part operation, where the data is recovered from the signature. *hSession* is the session's handle; *pSignature* points to the signature; *usSignatureLen* is the length of the signature; *pData* points to the location that receives the recovered data; and *pusDataLen* points to the location that receives the length of the recovered data.

The verification operation must have been initialized with **C_VerifyRecoverInit**.

For constraints on data length, refer to the description of the verification mechanism.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_OPERATION_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DATA_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_RSA_9796, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_USHORT usDataLen;
CK_BYTE signature[128];
CK_RV rv;

rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
if( rv == CKR_OK ){
   rv = C_VerifyRecover(hSession, signature, sizeof(signature), data,
         &usDataLen);
}
```

9.8 Key management

Cryptoki provides the following functions for key management. All these functions run in parallel with the application if the session was opened with the CKF_SERIAL_SESSION flag set to FALSE and the token supports parallel execution.

♦ C_GenerateKey

```
CK_RV CK_ENTRY C_GenerateKey(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usCount,
     CK_OBJECT_HANDLE_PTR phKey
);
```

C_GenerateKey generates a secret key, creating a new key object. *hSession* is the session's handle; *pMechanism* points to the key generation mechanism; *pTemplate* points to the template for the new key; *usCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the new key.

The following mechanisms are supported in this version:

Table 9-9, Key Generation Mechanisms

Mechanism	Key type
RC2 key generation	RC2
RC4 key generation	RC4
DES key generation	DES[1]
double-length DES key generation	double-length DES[1]
triple-length DES key generation	triple-length DES[1]

[1] No known "weak" or "semi-weak" DES keys are generated (see FIPS PUB 74).

Section 10 provides more details on the mechanisms and on which attributes the template must specify.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OBJECT_CLASS_INVALID, CKR_OBJECT_CLASS_INCONSISTENT, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
   CKM_DES_KEY_GEN, NULL_PTR, 0
};
CK_RV rv;

rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_GenerateKeyPair

```
CK_RV CK_ENTRY C_GenerateKeyPair(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
     CK_USHORT usPublicKeyAttributeCount,
     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
     CK_USHORT usPrivateKeyAttributeCount,
     CK_OBJECT_HANDLE_PTR phPrivateKey,
     CK_OBJECT_HANDLE_PTR phPublicKey
);
```

C_GenerateKeyPair generates a public-key/private-key pair, creating new key objects. On input, *hSession* is the session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to the template for the public key; *usPublicKeyAttributeCount* is the number of attributes in the public-key template; *pPrivateKeyTemplate* points to the template for the private key; *usPrivateKeyAttributeCount* is the number of attributes in the private-key template; *phPublicKey* points to the location that receives the handle of the new public key; and *phPrivateKey* points to the location that receives the handle of the new private key.

The following mechanisms are supported in this version:

Table 9-10, Key Pair Generation Mechanisms

Mechanism	**Key types**
PKCS #1 RSA key pair generation	RSA public and private
DSA key pair generation	DSA public and private
PKCS #3 Diffie-Hellman key pair generation	DH public and private

Section 10 provides more details on the mechanisms and on which attributes the template must specify.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OBJECT_CLASS_INVALID, CKR_OBJECT_CLASS_INCONSISTENT, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
   CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_USHORT modulusBits = 768;
CK_BYTE publicExponent[] = { 3 };
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
   {CKA_ENCRYPT, &true, 1},
   {CKA_VERIFY, &true, 1},
   {CKA_WRAP, &true, 1},
   {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
   {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
   {CKA_TOKEN, &true, 1},
   {CKA_PRIVATE, &true, 1},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, 1},
   {CKA_DECRYPT, &true, 1},
   {CKA_SIGN, &true, 1},
   {CKA_UNWRAP, &true, 1}
};
CK_RV rv;
```

```
rv = C_GenerateKeyPair(hSession, &mechanism, publicKeyTemplate, 5,
         privateKeyTemplate, 8, &hPublicKey, &hPrivateKey);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_WrapKey

```
CK_RV CK_ENTRY C_WrapKey(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hWrappingKey,
     CK_OBJECT_HANDLE hKey,
     CK_BYTE_PTR pWrappedKey,
     CK_USHORT_PTR pusWrappedKeyLen
);
```

C_WrapKey wraps (i.e., encrypts) a key. *hSession* is the session's handle; *pMechanism* points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and *pusWrappedKeyLen* points to the location that receives the length of the wrapped key.

The CKA_WRAP attribute of the wrapping key, which indicates whether the key supports wrapping, must be TRUE.

The following mechanisms are supported in this version:

Table 9-11, Wrapping Mechanisms

Mechanism	Wrapping key type	Type of key to be wrapped
PKCS #1 RSA	RSA public	RC2, RC4, DES, double or triple-length DES
X.509 (raw) RSA	RSA public	RC2, RC4, DES, double or triple-length DES
RC2 (ECB mode)	RC2	RC2, RC4, DES, double or triple-length DES
DES (ECB mode)	DES	RC2, RC4, DES
triple-DES (ECB mode)	double or triple-length DES	RC2, RC4, DES, double or triple-length DES

Section 10 provides more details on the mechanisms and on which attributes the template must specify.

Return Values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_TYPE_INCONSISTENT, CKR_WRAPPING_KEY_SIZE_RANGE, CKR_KEY_SENSITIVE, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
```

```
CK_OBJECT_HANDLE hWrappingKey, hKey;
CK_MECHANISM mechanism = {
   CKM_DES3_ECB, NULL_PTR, 0
};
CK_BYTE wrappedKey[8];
CK_USHORT usWrappedKeyLen;
CK_RV rv;

rv = C_WrapKey(hSession, &mechanism, hWrappingKey, hKey, wrappedKey,
        &usWrappedKeyLen);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_UnwrapKey

```
CK_RV CK_ENTRY C_UnwrapKey(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hUnwrappingKey,
     CK_BYTE_PTR pWrappedKey,
     CK_USHORT usWrappedKeyLen,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usAttributeCount,
     CK_OBJECT_HANDLE_PTR phKey
);
```

C_UnwrapKey unwraps (i.e. decrypts) a wrapped key, creating a new key object. *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *usWrappedKeyLen* is the length of the wrapped key; *pTemplate* points to the template for the new key; *usAttributeCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the recovered key.

The CKA_UNWRAP attribute of the unwrapping key, which indicates whether the key supports unwrapping, must be TRUE.

The following mechanisms are supported in this version:

Table 9-12, Unwrapping Mechanisms

Mechanism	**Unwrapping key type**	**Recovered key type**
PKCS #1 RSA	RSA private	RC2, RC4, DES, double or triple-length DES
X.509 (raw) RSA	RSA private	RC2, RC4, DES, double or triple-length DES
RC2 (ECB mode)	RC2	RC2, RC4, DES, double or triple-length DES
DES (ECB mode)	DES	RC2, RC4, DES
triple-DES (ECB mode)	double or triple-length DES	RC2, RC4, DES, double or triple-length DES

Section 10 provides more details on the mechanisms and on which attributes the template must specify.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_WRAPPED_KEY_LEN_RANGE, CKR_WRAPPED_KEY_INVALID, CKR_OBJECT_CLASS_INVALID, CKR_OBJECT_CLASS_INCONSISTENT, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hUnwrappingKey, hKey;
CK_MECHANISM mechanism = {
   CKM_DES3_ECB, NULL_PTR, 0
};
CK_BYTE wrappedKey[8] = {...};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_ENCRYPT, &true, 1},
   {CKA_DECRYPT, &true, 1}
};
CK_RV rv;

rv = C_UnwrapKey(hSession, &mechanism, hUnwrappingKey, wrappedKey,
         sizeof(wrappedKey), template, 4, &hKey);
if( rv == CKR_OK ){
   .
   .
   .
}
```

♦ C_DeriveKey

```
CK_RV CK_ENTRY C_DeriveKey(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_OBJECT_HANDLE hBaseKey,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_USHORT usAttributeCount,
     CK_OBJECT_HANDLE_PTR phKey
);
```

C_DeriveKey derives a key from a base key, creating a new key object. *hSession* is the session's handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the handle of the base key; *pTemplate* points to the template for the new key; *usAttributeCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

The following mechanisms are supported in this version:

Table 9-13, Key Derivation Mechanisms

Mechanism	**Base key type**	**Derived key type**
Diffie-Hellman key derivation	DH private	RC2, RC4, DES, double or triple-length DES, or generic

Section 10 provides more details on the mechanisms and on which attributes the template must specify.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY CKR_SESSION_CLOSED, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE, CKR_OBJECT_CLASS_INVALID, CKR_OBJECT_CLASS_INCONSISTENT, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_WRITE_PROTECTED, CKR_OPERATION_ACTIVE, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
CK_MECHANISM keyPairMechanism = {
   CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE publicValue[128];
CK_BYTE otherPublicValue[128];
CK_MECHANISM mechanism = {
   CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)
};
CK_ATTRIBUTE pTemplate[] = {
   CKA_VALUE, &publicValue, sizeof(publicValue)}
};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_BBOOL true = TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_BASE, base, sizeof(base)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
   {CKA_DERIVE, &true, 1}
};
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_ENCRYPT, &true, 1},
   {CKA_DECRYPT, &true, 1}
};
CK_RV rv;

rv = C_GenerateKeyPair(hSession, &keyPairMechanism, publicKeyTemplate, 2,
        privateKeyTemplate, 1, &hPublicKey, &hPrivateKey);
```

```
if( rv == CKR_OK ){
   rv = C_GetAttributeValue(hSession, hPublicKey, &pTemplate, 1);
   if( rv == CKR_OK ){
      .
      /* exchange public values */
      .
      rv = C_DeriveKey(hSession, &mechanism, hPrivateKey, template, 4,
         &hKey);
      if( rv == CKR_OK ){
         .
         .
         .
      }
   }
}
```

9.9 Random number generation

Cryptoki provides the following functions for generating random numbers. All these functions run in parallel with the application if the session was opened with the CKF_SERIAL_SESSION flag set to FALSE and the token supports parallel execution.

♦ C_SeedRandom

```
CK_RV CK_ENTRY C_SeedRandom(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSeed,
     CK_USHORT usSeedLen
);
```

C_SeedRandom mixes additional seed material into the token's random number generator. *hSession* is the session's handle; *pSeed* points to the seed material; and *usSeedLen* is the length in bytes of the seed material.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_OPERATION_ACTIVE, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
CK_SESSION_HANDLE hSession;
CK_BYTE seed[] = {...};
CK_RV rv;

rv = C_SeedRandom(hSession, seed, sizeof(seed));
```

```
    if( rv == CKR_OK ){
        .
        .
        .
    }
```

♦ C_GenerateRandom

```
CK_RV CK_ENTRY C_GenerateRandom(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pRandomData,
     CK_USHORT usRandomLen
);
```

C_GenerateRandom generates random data. *hSession* is the session's handle; *pRandomData* points to the location that receives the random data; and *usRandomLen* is the length in bytes of the random data to be generated.

Return values: CKR_OK, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_HOST_MEMORY, CKR_DEVICE_MEMORY, CKR_OPERATION_ACTIVE, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example:

```
    CK_SESSION_HANDLE hSession;
    CK_BYTE randomData[] = {...};
    CK_RV rv;

    rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
    if( rv == CKR_OK ){
        .
        .
        .
    }
```

9.10 Parallel function management

Cryptoki provides the following functions for managing parallel execution of cryptographic functions.

♦ C_GetFunctionStatus

```
CK_RV CK_ENTRY C_GetFunctionStatus(
     CK_SESSION_HANDLE hSession
);
```

C_GetFunctionStatus obtains an updated status of a function running in parallel with an application. *hSession* is the session's handle.

An application should call this function repeatedly until the return value is no longer CKR_FUNCTION_NOT_PARALLEL.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_FUNCTION_NOT_PARALLEL, CKR_FUNCTION_PARALLEL, CKR_FUNCTION_CANCELED, CKR_HOST_MEMORY, CKR_DEVICE_REMOVED, CKR_DEVICE_ERROR

Example: see **C_CancelFunction**.

♦ C_CancelFunction

```
CK_RV CK_ENTRY C_CancelFunction(
     CK_SESSION_HANDLE hSession
);
```

C_CancelFunction cancels a function running in parallel with an application. *hSession* is the session's handle.

Return values: CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_FUNCTION_NOT_PARALLEL, CKR_HOST_MEMORY

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
   CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_USHORT modulusBits = 768;
CK_BYTE publicExponent[] = {...};
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
   {CKA_ENCRYPT, &true, 1},
   {CKA_VERIFY, &true, 1},
   {CKA_WRAP, &true, 1},
   {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
   {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
   {CKA_TOKEN, &true, 1},
   {CKA_PRIVATE, &true, 1},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, 1},
   {CKA_DECRYPT, &true, 1},
   {CKA_SIGN, &true, 1},
   {CKA_UNWRAP, &true, 1}
};
CK_RV rv;
rv = C_GenerateKeyPair(hSession, &mechanism, publicKeyTemplate, 5,
         privateKeyTemplate, 8, &hPublicKey, &hPrivateKey);
```

```
while ( rv == CKR_FUNCTION_PARALLEL ) {
   /* Check if user want to cancel function */
   if( kbhit() ){
      if( getch() == 27 ){ /* If user hit ESCape key */
         C_CancelFunction(hSession);
         break;
      }
   }
    /* Perform other tasks or delay */
   .
   .
   .
   rv = C_GetFunctionStatus(hSession);
}
```

9.11 Callback function

Cryptoki uses the following callback function to notify the application of certain events.

♦ Notify

```
CK_RV CK_ENTRY Notify(
     CK_SESSION_HANDLE hSession,
     CK_NOTIFICATION event,
     CK_VOID_PTR pApplication
);
```

Notify is an application callback that processes events. *hSession* is the session's handle; *event* is the event; and *pApplication* is an application-defined value (the same as passed to **C_OpenSession**).

When *event* is CKN_SURRENDER, the callback may return CKR_CANCEL to cancel the operation that is currently active. If the callback returns CKR_OK, Cryptoki continues the operation. For other events, the callback should return CKR_OK.

Return values: CKR_OK, CKR_CANCEL.

10. Mechanisms

This section describes the mechanisms that this version of Cryptoki supports for cryptographic operations. The following table summarizes the mechanisms and their uses.

Table 10-1, Mechanisms vs. Functions

	Functions						
Mechanism	**Encrypt & Decrypt**	**Sign & Verify**	**SR & VR**[1]	**Digest**	**Generate**[2]	**Wrap & Unwrap**	**Derive**
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓[3]	✓[3]	✓[3]			✓	
CKM_RSA_9796		✓[3]	✓[3]				
CMK_RSA_X_509	✓[3]	✓[3]	✓[3]			✓	
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA		✓[3]					
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_RC2_KEY_GEN					✓		
CKM_RC2_ECB	✓					✓	
CKM_RC2_CBC	✓						
CKM_RC2_MAC		✓					
CKM_RC4_KEY_GEN					✓		
CKM_RC4	✓						
CKM_DES_KEY_GEN					✓		
CKM_DES_ECB	✓					✓	
CKM_DES_CBC	✓						
CKM_DES_MAC		✓					
CKM_DES2_KEY_GEN					✓		
CKM_DES3_KEY_GEN					✓		
CKM_DES3_ECB	✓					✓	
CKM_DES3_CBC	✓						
CKM_DES3_MAC		✓					
CKM_MD2				✓			
CKM_MD5				✓			
CKM_SHA_1				✓			

[1] SR = SignRecover, VR = VerifyRecover. [2] Generate includes GenerateKey and GenerateKeyPair. [3] Single-part operations only.

10.1 PKCS #1 RSA key pair generation

The PKCS #1 RSA key pair generation mechanism, denoted **CKM_RSA_PKCS_KEY_PAIR_GEN**, is a key generation mechanism based on the RSA public-key cryptosystem as defined in PKCS #1.

It does not have a parameter.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the CKA_MODULUS_BITS and CKA_EXPONENT attributes of the template for the public key.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_MODULUS attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_MODULUS, CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, and CKA_COEFFICIENT attributes to the new private key. Other attributes supported by the RSA public and private key types (specifically the flags indicating which functions the keys support) may also be specified in the templates for the keys or else are assigned default initial values.

Keys generated with this mechanism are compatible with the PKCS #1 RSA, ISO/IEC 9796 RSA, and X.509 (raw) RSA mechanisms.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

10.2 PKCS #1 RSA

The PKCS #1 RSA mechanism, denoted **CKM_RSA_PKCS**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the block formats defined in PKCS #1. It supports single-part encryption and decryption, single-part signatures and verification with and without message recovery, key wrapping, and key unwrapping. This mechanism corresponds only to the part of PKCS #1 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for the `md2withRSAEncryption` and `md5withRSAEncryption` algorithms in PKCS #1.

It does not have a parameter.

This mechanism wraps and unwraps RC2, RC4, DES, double-length DES and triple-length DES keys. For wrapping, the "input" to the encryption operation is the value of the CKA_VALUE attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the CKA_VALUE attribute to the recovered key during unwrapping; other attributes, including the CKA_CLASS attribute, must be specified in the template since the mechanism does preserve the key length.

Constraints on key types and the length of the data are summarized in the following table. For encryption, decryption, signatures and signature verification, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table 10-2, PKCS #1 RSA Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt[1]	RSA public key	≤ k-11	k	block type `02`
C_Decrypt[1]	RSA private key	k	≤ k-11	block type `01`
C_Sign[1]	RSA private key	≤ k-11	k	block type `01`
C_SignRecover	RSA private key	≤ k-11	k	block type `01`
C_Verify[1]	RSA public key	≤ k-11,k (2)	N/A	block type `02`
C_VerifyRecover	RSA public key	k	≤ k-11	block type `02`
C_WrapKey	RSA public key	≤ k-11	k	block type `01`
C_UnwrapKey	RSA private key	k	≤ k-11	block type `01`

[1] Single-part operations only. [2] Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

10.3 ISO/IEC 9796 RSA

The ISO/IEC 9796 RSA mechanism, denoted **CKM_RSA_9796**, is a mechanism for single-part signatures and verification with and without message recovery based on the RSA public-key cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A. This mechanism is compatible with the draft ANSI X9.31 (assuming the length in bits of the X9.31 hash value is a multiple of 8).

This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly, the following transformations are performed:

- data is converted between byte and bit string formats by interpreting the most significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8)

- a signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; is it converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus

It does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. The input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table 10-3, ISO/IEC 9796 RSA Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Sign[1]	RSA private key	$\le\lfloor k/2\rfloor$	k
C_SignRecover[1]	RSA private key	$\le\lfloor k/2\rfloor$	k
C_Verify[1]	RSA public key	$\le\lfloor k/2\rfloor$, k[(2)]	N/A
C_VerifyRecover[1]	RSA public key	k	$\le\lfloor k/2\rfloor$

[1] Single-part operations only. [2] Data length, signature length.

10.4 X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted **CKM_RSA_X_509**, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption, single-part signatures and verification with and without message recovery, key wrapping, and key unwrapping based on the so-called "raw" RSA, as assumed in X.509.

"Raw" RSA as defined here encrypts a byte string by converting it to an integer, most significant byte first, applying "raw" RSA exponentiation, and converting the result to a byte string, most significant byte first. The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

It does not have a parameter.

This mechanism wraps and unwraps RC2, RC4, DES, double-length DES and triple-length DES keys. For wrapping, the "input" to the encryption operation is the value of the CKA_VALUE attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

Constraints on key types and the length of input and output data are summarized in the following table. For encryption, decryption, signatures and signature verification, the input and output data may begin at the same location in memory. In the table, k is the length in bytes of the RSA modulus.

Table 10-4, X.509 (Raw) RSA Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Encrypt[1]	RSA public key	$\le k$	k
C_Decrypt[1]	RSA private key	k	$\le k$
C_Sign[1]	RSA private key	$\le k$	k
C_SignRecover[1]	RSA private key	$\le k$	k
C_Verify[1]	RSA public key	$\le k$,k[(2)]	N/A
C_VerifyRecover[1]	RSA public key	k	$\le k$
C_WrapKey	RSA public key	$\le k$	k
C_UnwrapKey	RSA private key	k	$\le k$

[1] Single-part operations only. [2] Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

10.5 DSA key pair generation

The DSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186.

It does not have a parameter.

The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as specified in the CKA_PRIME, CKA_SUBPRIME, and CKA_BASE attributes of the template for the public key. (Note that this version of Cryptoki does not include a mechanism for generating these DSA parameters.)

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_SUBPRIME, CKA_BASE, and CKA_VALUE attributes to the new private key. Other attributes supported by the DSA public and private key types (specifically the flags indicating which functions the keys support) may also be specified in the templates for the keys or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

10.6 DSA

The DSA mechanism, denoted **CKM_DSA**, is a mechanism for single-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186. (This mechanism corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash value.)

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most significant byte first.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table. The input and output data may begin at the same location in memory.

Table 10-5, DSA Key And Data Length Constraints

Function	**Key type**	**Input length**	**Output length**
C_Sign[1]	DSA private key	20	40
C_Verify[1]	DSA public key	20,40[2]	N/A

[1] Single-part operations only. [2] Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

10.7 PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. (This is analogous to what PKCS #3 calls "phase I.")

It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the CKA_PRIME and CKA_BASE attributes of the template for the public key. If the CKA_VALUE_BITS attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in PKCS #3. (Note that this version of Cryptoki does not include a mechanism for generating a prime and base.)

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, and CKA_VALUE attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of Diffie-Hellman prime sizes, in bits.

10.8 PKCS #3 Diffie-Hellman key derivation

The PKCS #3 Diffie-Hellman key derivation mechanism, denoted **CKM_DH_PKCS_DERIVE**, is a mechanism for key derivation based on Diffie-Hellman key agreement, as defined in PKCS #3. (This is analogous to what PKCS #3 calls "phase II.")

It has a parameter, which is the public value of the other party in the key agreement protocol, represented most significant byte first.

The mechanism derives RC2, RC4, DES, double-length DES, triple-length DES and generic secret keys from the public value of the other party and a Diffie-Hellman private key. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS #3, and truncates the result according to the CKA_CLASS and CKA_KEY_TYPE attributes of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key type must be specified in the template.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of Diffie-Hellman prime sizes, in bits.

10.9 RC2 key generation

The RC2 key generation mechanism, denoted **CKM_RC2_KEY_GEN**, is a key generation mechanism for RSA Data Security's proprietary block cipher RC2.

It does not have a parameter.

The mechanism generates RC2 keys with a particular length in bytes, as specified in the CKA_VALUE_LEN attribute of the template for the key.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the RC2 key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 key sizes, in bits.

10.10 RC2-ECB

RC2-ECB, denoted **CKM_RC2_ECB**, is a mechanism for single- and multiple-part encryption and decryption, key wrapping, and key unwrapping based on RSA Data Security's proprietary block cipher RC2 and electronic codebook mode as defined in FIPS PUB 81.

It has a parameter, a two-byte string that specifies the effective number of bits in the RC2 search space, most significant byte first; the value must be between 1 and 1024.

This mechanism wraps and unwraps RC2, RC4, DES, double-length DES and triple-length DES keys. For wrapping, the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped, padded on the trailing end with up to seven null bytes so that the resulting length is a multiple of eight. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the CKA_CLASS and CKA_KEY_TYPE attributes of the template and, if it has one, and the key type supports it, the CKA_VALUE_LEN attribute of the template. (Truncation is mainly an issue for RC2 and RC4 keys.) The mechanism contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For encryption and decryption, the input and output data (parts) may begin at the same location in memory.

Table 10-6, RC2-ECB Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC2	multiple of 8	same as input length	no final part
C_Decrypt	RC2	multiple of 8	same as input length	no final part
C_WrapKey	RC2	any	input length rounded up to multiple of 8	
C_UnwrapKey	RC2	any	input length rounded up to multiple of 8	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 effective number of bits.

10.11 RC2-CBC

RC2-CBC, denoted **CKM_RC2_CBC**, is a mechanism for single- and multiple-part encryption and decryption, based on RSA Data Security's proprietary block cipher RC2 and cipher block chaining mode as defined in FIPS PUB 81.

It has a parameter, a CK_RC2_CBC_PARAMS structure, where the first field indicates the effective number of bits in the RC2 search space, and the next field is the initialization vector for cipher block chaining mode. The effective number of bits must be between 1 and 1024.

Constraints on key types and the length of data are summarized in the following table. The input and output data (parts) may begin at the same location in memory.

Table 10-7, RC2-CBC Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC2	multiple of 8	same as input length	no final part
C_Decrypt	RC2	multiple of 8	same as input length	no final part

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 effective number of bits.

10.12 RC2-MAC

RC2-MAC, denoted **CKM_RC2_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification, based on RSA Data Security's proprietary block cipher RC2 and data authentication as defined in FIPS PUB 113.

It has a parameter, a two-byte string that specifies the effective number of bits in the RC2 search space, most significant byte first; the value must be between 1 and 1024.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part signing, the data and the signature may begin at the same location in memory.

Table 10-8, RC2-MAC Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	RC2	any	4
C_Verify	RC2	any	4

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 effective number of bits.

10.13 RC4 key generation

The RC4 key generation mechanism, denoted **CKM_RC4_KEY_GEN**, is a key generation mechanism for RSA Data Security's proprietary stream cipher RC4.

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the RC4 key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key; other attributes required by the RC4 key type must be specified in the template.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC4 key sizes, in bits.

10.14 RC4

RC4, denoted **CKM_RC4**, is a mechanism for single- and multiple-part encryption and decryption based on RSA Data Security's proprietary stream cipher RC4.

It does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) The input and output data (parts) may begin at the same location in memory.

Table 10-9, RC4 Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC4	any	same as input length	no final part
C_Decrypt	RC4	any	same as input length	no final part

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure specify the supported range of RC4 key sizes, in bits.

10.15 DES key generation

The DES key generation mechanism, denoted **CKM_DES_KEY_GEN**, is a key generation mechanism for DES as defined in FIPS PUB 46-2.

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.16 DES-ECB

DES-ECB, denoted **CKM_DES_ECB**, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping following DES as defined in FIPS PUB 46-2 and electronic codebook mode as defined in FIPS PUB 81. It does not have a parameter.

This mechanism wraps and unwraps RC2, RC4, and single-length DES keys. For wrapping, the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped, padded on the trailing end with up to seven null bytes so that the resulting length is a multiple of eight. The output data is the same length as the padded input data. It does not wrap the key type, key length or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the CKA_CLASS and CKA_KEY_TYPE attributes of the template and, if it has one, and the key type supports it, the CKA_VALUE_LEN attribute of the template. (Truncation is mainly an issue for RC2 and RC4 keys.) The mechanism contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For encryption and decryption, the input and output data may begin at the same location in memory.

Table 10-10, DES-ECB Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	DES	multiple of 8	same as input length	no final part
C_Decrypt	DES	multiple of 8	same as input length	no final part
C_WrapKey	DES	any	input length rounded up to multiple of 8	
C_UnwrapKey	DES	any	input length rounded up to multiple of 8	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.17 DES-CBC

DES-CBC, denoted **CKM_DES_CBC**, is a mechanism for single- and multiple-part encryption and decryption, following DES as defined in FIPS PUB 46-2 and cipher block chaining mode as defined in FIPS PUB 81.

It has a parameter, an eight-byte initialization vector for cipher block chaining mode.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) The input and output data (parts) may begin at the same location in memory.

Table 10-11, DES-CBC Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	DES	multiple of 8	same as input length	no final part
C_Decrypt	DES	multiple of 8	same as input length	no final part

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.18 DES-MAC

DES-MAC, denoted **CKM_DES_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification, following DES as defined in FIPS PUB 46-2 and data authentication as defined in ANSI X9.9 (binary option) and FIPS PUB 113.

It does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part signing, the data and signature may begin at the same location in memory.

Table 10-12, DES-MAC Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	DES	any	4
C_Verify	DES	any	4

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.19 Double-length DES key generation

The double-length DES key generation mechanism, denoted **CKM_DES2_KEY_GEN**, is a key generation mechanism for double-length DES keys.

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.20 Triple-length DES key generation

The triple-length DES key generation mechanism, denoted **CKM_DES3_KEY_GEN**, is a key generation mechanism for triple-length DES keys.

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE and CKA_VALUE attributes to the new key. Other attributes supported by the DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.21 Triple-DES-ECB

Triple-DES-ECB, denoted **CKM_DES3_ECB**, is a mechanism for single- and multiple-part encryption and decryption, key wrapping, and key unwrapping based on so-called "triple-DES" and electronic codebook mode as defined in FIPS PUB 81.

Triple-DES as defined here follows the "EDE" convention, operating on either double-length or triple-length DES keys. With a double-length DES key, the mechanism encrypts each block with the first DES key, decrypts with the second DES key, then encrypts again with the first DES key. With a triple-length DES keys, the mechanism encrypts each block with the first DES key, decrypts with the second DES key, then encrypts with the third DES key.

It does not have a parameter.

This mechanism wraps and unwraps RC2, RC4, DES, double-length DES and triple-length DES keys. For wrapping, the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped, padded on the trailing end with up to seven null bytes so that the resulting length is a multiple of eight. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the CKA_CLASS and CKA_KEY_TYPE attributes of the template and, if it has one, and the key type supports it, the CKA_VALUE_LEN attribute of the template. (Truncation is mainly an issue for RC2 and RC4 keys.) The mechanism contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For encryption and decryption, the input and output data (parts) may begin at the same location in memory.

Table 10-13, Triple-DES-ECB Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	double-length or triple-length DES	multiple of 8	same as input length	no final part
C_Decrypt	(same as above)	multiple of 8	same as input length	no final part
C_WrapKey	(same as above)	any	input length rounded up to multiple of 8	
C_UnwrapKey	(same as above)	any	input length rounded up to multiple of 8	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.22 Triple-DES-CBC

Triple-DES-CBC, denoted **CKM_DES3_CBC**, is a mechanism for encryption and decryption, based on so-called "triple-DES" and cipher block chaining mode as defined in FIPS PUB 81. It has a parameter, an eight-byte initialization vector for cipher block chaining mode.

Constraints on key types and the length of input and output data are summarized in the following table. The input and output data (parts) may begin at the same location in memory.

Table 10-14, Triple-DES-CBC Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	double-length or triple-length DES	multiple of 8	same as input length	no final part
C_Decrypt	double-length or triple-length DES	multiple of 8	same as input length	no final part

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.23 Triple-DES-MAC

Triple-DES-MAC, denoted **CKM_DES3_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification, based on so-called "triple-DES" and data authentication as defined in FIPS PUB 113.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part signing, the data and the signature may begin at the same location in memory.

Table 10-15, Triple-DES-MAC Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	double-length or triple-length DES	any	4
C_Verify	double-length or triple-length DES	any	4

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.24 MD2

The MD2 mechanism, denoted **CKM_MD2**, is a mechanism for message digesting, following the MD2 message-digest algorithm defined in RFC 1319.

It does not have a parameter.

Constraints on the length of data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part digesting, the data and the digest may begin at the same location in memory.

Table 10-16, MD2 Data Length Constraints

Function	Data length	Digest length
C_Digest	any	16

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.25 MD5

The MD5 mechanism, denoted **CKM_MD5**, is a mechanism for message digesting, following the MD5 message-digest algorithm defined in RFC 1321.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part digesting, the data and the digest may begin at the same location in memory.

Table 10-17, MD5 Data Length Constraints

Function	Data length	Digest length
C_Digest	any	16

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

10.26 SHA-1

The SHA-1 mechanism, denoted **CKM_SHA_1**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180, as subsequently amended by NIST.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. (These constraints apply both to data supplied as a single part, and to each part of multiple-part data.) For single-part digesting, the data and the digest may begin at the same location in memory.

Table 10-18, SHA-1 Data Length Constraints

Function	Input length	Digest length
C_Digest	any	20

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO structure are not used.

Appendix A, Token profiles

This appendix describes "profiles," i.e., sets of mechanisms, which a token should support for various common types of application. It is expected that these sets would be standardized as parts of the various applications, for instance within a list of requirements on the module that provides cryptographic services to the application (which may be a Cryptoki token in some cases). Thus, these profiles are intended for reference only at this point, and are not part of this standard.

The following table summarizes the mechanisms relevant to three common types of application.

Table A-1, Mechanisms vs. profiles

	Application		
Mechanism	**Privacy-Enhanced Mail**	**Government Authentication-only**	**Cellular Digital Packet Data**
CKM_RSA_PKCS_KEY_PAIR_GEN	✓		
CKM_RSA_PKCS	✓		
CKM_RSA_9796			
CMK_RSA_X_509			
CKM_DSA_KEY_PAIR_GEN		✓	
CKM_DSA		✓	
CKM_DH_PKCS_KEY_PAIR_GEN			✓
CKM_DH_PKCS_DERIVE			✓
CKM_RC2_KEY_GEN			
CKM_RC2_ECB			
CKM_RC2_CBC			
CKM_RC2_MAC			
CKM_RC4_KEY_GEN			✓
CKM_RC4			✓
CKM_DES_KEY_GEN	✓		
CKM_DES_ECB	✓		
CKM_DES_CBC	✓		
CKM_DES_MAC			
CKM_DES2_KEY_GEN	✓		
CKM_DES3_KEY_GEN			
CKM_DES3_ECB	✓		
CKM_DES3_CBC			
CKM_DES3_MAC			
CKM_MD2	✓		
CKM_MD5	✓		
CKM_SHA_1		✓	
CKM_SHA_1_DERIVE			

A.1 Privacy-Enhanced Mail

Privacy-Enhanced Mail is a set of protocols and mechanisms providing confidentiality and authentication for Internet electronic mail. Relevant mechanisms include the following (see RFC 1423 for details):

PKCS #1 RSA key pair generation (508–1024 bits)

PKCS #1 RSA (508-1024 bits)

DES key generation

DES-CBC

DES-ECB

double-length DES key generation

triple-DES-ECB

MD2

MD5

Variations on this set are certainly possible. For instance, PEM applications which make use only of asymmetric key management do not need the DES-ECB or triple-DES-ECB mechanisms, or the double-length DES key generation mechanism. Similarly, those which make use only of symmetric key management do not need the PKCS #1 RSA or RSA key pair generation mechanisms.

An "authentication-only" version of PEM with asymmetric key management would not need DES-CBC or DES key generation.

It is also possible to consider "exportable" variants of PEM which replace DES-CBC with RC2-CBC, perhaps limited to 40 bits, and limit the RSA key size to 512 bits.

A.2 Government authentication-only

The U.S. government has standardized on the Digital Signature Algorithm as defined in FIPS PUB 186 for signatures and the Secure Hash Algorithm as defined in FIPS PUB 180 and subsequently amended by NIST for message digesting. The relevant mechanisms include the following:

DSA key generation (512-1024 bits)

DSA (512-1024 bits)

SHA-1

Note that this version of Cryptoki does not have a mechanism for generating DSA parameters.

A.3 Cellular Digital Packet Data

Cellular Digital Packet Data (CDPD) is a set of protocols for wireless communication. The basic set of mechanisms to support CDPD applications includes the following:

Diffie-Hellman key generation (256-1024 bits)

Diffie-Hellman key derivation (256-1024 bits)

RC4 key generation (40-128 bits)

RC4 (40-128 bits)

(The initial CDPD security specification limits the size of the Diffie-Hellman key to 256 bits, but has been recommended that the size be increased to at least 512 bits.)

Note that this version of Cryptoki does not have a mechanism for generating Diffie-Hellman parameters.

Appendix B, Comparison of Cryptoki and Other API's

This appendix compares Cryptoki with the following cryptographic APIs:

- ANSI N13-94 - Guideline X9.TG-12-199X, Using Tessera in Financial Systems: An Application Programing Interface, April 29, 1994
- FIPS PUB XXX - Standard for Cryptographic Service Calls (Draft), April 15, 1994
- X/Open GCS-API - Generic Cryptographic Service API, Draft 2, February 14, 1995

B.1 ANSI N13-1994

This proposed standard defines an API to the Tessera (now known as Fortezza) PCMCIA Crypto Card. It is at a level similar to Cryptoki. The following table lists the ANSI N13-1994 functions with the equivalent Cryptoki functions.

ANSI N13-1994	Equivalent Cryptoki
CI_ChangePIN	C_InitPIN, C_SetPIN
CI_CheckPIN	C_Login
CI_Close	C_CloseSession
CI_Decrypt	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
CI_DeleteCertificate	C_DestroyObject
CI_DeleteKey	C_DestroyObject
CI_Encrypt	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
CI_ExtractX	C_WrapKey
CI_GenerateIV	C_GenerateRandom
CI_GenerateMEK	C_GenerateKey
CI_GenerateRa	C_GenerateRandom
CI_GenerateRandom	C_GenerateRandom
CI_GenerateTEK	C_GenerateKey
CI_GenerateX	C_GenerateKeyPair
CI_GetCertificate	C_FindObjects
CI_Configuration	C_GetTokenInfo
CI_GetHash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal
CI_GetIV	No equivalent
CI_GetPersonalityList	C_FindObjects
CI_GetState	C_GetSessionInfo
CI_GetStatus	C_GetTokenInfo
CI_GetTime	No equivalent
CI_Hash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal

ANSI N13-1994	**Equivalent Cryptoki**
CI_Initialize	C_Initialize
CI_InitializeHash	C_DigestInit
CI_InstallX	C_UnwrapKey
CI_LoadCertificate	C_CreateObject
CI_LoadInitValues	C_SeedRandom
CI_LoadIV	C_EncryptInit, C_DecryptInit
CI_LoadK	C_SignInit
CI_LoadPublicKeyParameters	C_CreateObject
CI_LoadPIN	C_SetPIN
CI_LoadX	C_CreateObject
CI_Open	C_OpenSession
CI_Relay	C_WrapKey
CI_Reset	C_CloseAllSessions
CI_Restore	No equivalent
CI_Save	No equivalent
CI_Select	C_OpenSession
CI_SetKey	C_EncryptInit, C_DecryptInit
CI_SetMode	C_EncryptInit, C_DecryptInit
CI_SetPersonality	C_CreateObject
CI_SetTime	No equivalent
CI_Sign	C_SignInit, C_Sign
CI_Timestamp	No equivalent
CI_Terminate	C_CloseAllSessions
CI_UnwrapKey	C_UnwrapKey
CI_Verify	C_VerifyInit, C_Verify
CI_VerifyTimestamp	No equivalent
CI_WrapKey	C_WrapKey
CI_Zeroize	C_InitToken

B.2 FIPS PUB XXX

This proposed standard defines a set of generic cryptographic service calls for application programs. It is at a level similar to Cryptoki. The following table lists the FIPS PUB XXX functions with the equivalent Cryptoki functions.

FIPS PUB XXX	**Equivalent Cryptoki Functions**
VerifyUser	C_Login
CreateUser	C_InitToken, C_InitPIN
ChangeAuthent	C_SetPIN

FIPS PUB XXX	**Equivalent Cryptoki Functions**
SetUserCommand	No equivalent
ShowUserCommand	No equivalent
DeleteUser	C_InitToken
Logout	C_Logout
Encipher	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
Decipher	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
ComputeDAC	C_SignInit, C_Sign, C_SignUpdate, C_SignFinal
VerifyDAC	C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal
GenRandNum	C_SeedRandom, C_GenerateRandom
GenKey	C_GenerateKey
DeleteKey	C_DestroyObject
LoadKey	C_CreateObject
ShowSecKey	C_FindObjects
ExportKey	C_WrapKey
ImportKey	C_UnwrapKey
XorKeys	No equivalent
SetCount	No equivalent
ReadCount	No equivalent
PubEncipher	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
PubDecipher	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
Hash	C_DigestInit, C_Digest, C_DigestUpdate, C_DigestFinal
PreSign	C_SignInit
SetPubParam	C_GenerateKeyPair
ReadPubParam	C_GetAttributeValue
Sign	C_Sign
VerifySig	C_VerifyInit, C_Verify
GenPubKey	C_GenerateKeyPair
LoadPubKey	C_CreateObject
ShowPubKey	C_FindObjects
RetrvPubKey	C_GetAttributeValue
DeletePubKey	C_DestroyObject
LoadCert	C_CreateObject
RetrvCert	C_GetAttributeValue
PubExportKey	C_WrapKey
PubImportKey	C_UnwrapKey

B.3 GCS-API

This proposed standard defines an API to high-level security services such as authentication of identities and data-origin, non-repudiation, and separation and protection. It is at a higher level than Cryptoki. The following table lists the GCS-API functions with the Cryptoki functions used to implement the functions. Note that full support of GCS-API is left for future versions of Cryptoki.

GCS-API	**Cryptoki implementation**
retrieve_CC	
release_CC	
generate_hash	C_DigestInit, C_Digest
generate_random_number	C_GenerateRandom
generate_checkvalue	C_SignInit, C_Sign, C_SignUpdate, C_SignFinal
verify_checkvalue	C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal
data_encipher	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
data_decipher	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
create_CC	
derive_key	C_DeriveKey
generate_key	C_GenerateKey
store_CC	
delete_CC	
replicate_CC	
export_key	C_WrapKey
import_key	C_UnwrapKey
archive_CC	C_WrapKey
restore_CC	C_UnwrapKey
set_key_state	
generate_key_pattern	
verify_key_pattern	
derive_clear_key	C_DeriveKey
generate_clear_key	C_GenerateKey
load_key_parts	
clear_key_encipher	C_WrapKey
clear_key_decipher	C_UnwrapKey
change_key_context	
load_initial_key	
generate_initial_key	
set_current_master_key	
protect_under_new_master_key	
protect_under_current_master_key	

GCS-API	**Cryptoki implementation**
initialise_random_number_generator	C_SeedRandom
install_algorithm	
de_install_algorithm	
disable_algorithm	
enable_algorithm	
set_defaults	