PKCS #11: Cryptographic Token Interface Standard

An RSA Laboratories Technical Note
Version 2.0
April 15, 1997

RSA Laboratories
100 Marine Parkway, Suite 500
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 sending electronic mail to <pkcs@rsa.com> or via anonymous ftp to ftp.rsa.com in the pub/pkcs directory. There is an electronic mailing list, <pkcs-tng@rsa.com>, for discussion of issues relevant to the "next generation" of the PKCS standards. To subscribe to this list, send e-mail to <majordomo@rsa.com> with the line "subscribe pkcs-tng" in the message body. To unsubscribe, send e-mail to <majordomo@rsa.com> with the line "unsubscribe pkcs-tng" in the message body.

There is also an electronic mailing list, <cryptoki@rsa.com>, specifically for discussion of PKCS #11. To subscribe to this list, send e-mail to <majordomo@rsa.com> with the line "subscribe cryptoki" in the message body. To unsubscribe, send e-mail to <majordomo@rsa.com> with the line "unsubscribe cryptoki" in the message body.

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>.

It would be difficult to enumerate all the people and organizations who helped to produce PKCS #11. RSA Laboratories is grateful to each and every one of them. Especial thanks go to Bruno Couillard of Chrysalis-ITS and John Centafont of NSA for the many hours they spent writing up parts of this document.

For v1.0, 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. For v2.0, Ray Sidney served as document editor and project coordinator.

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, 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, although some conventions for the support of multiple types of device will be addressed here and in a separate document.

A number of cryptographic mechanisms (algorithms) are supported in this version; in addition, new mechanisms can easily be added later without changing the general interface. It is possible 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 the sake of interoperability, registration through the PKCS process is preferable).

Cryptoki v2.0 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 v2.0 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 180-1 — National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard.* April 17, 1995.

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

FORTEZZA CIPG — NSA, Workstation Security Products. *FORTEZZA Cryptologic Interface Programmers Guide, Revision 1.52*. November, 1995.

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 Abstract Syntax Notation One (ASN.1).* 1988.

X.209 — ITU-T (formerly CCITT). *Recommendation X.209: Specification of Basic Encoding Rules for 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.

BATON MISSI's BATON block cipher.

BER Basic Encoding Rules, as defined in X.209.

CAST Nortel's proprietary symmetric block cipher.

CAST3 Nortel's proprietary symmetric block cipher.

CAST5 Nortel's proprietary symmetric block cipher.

CBC Cipher Block Chaining mode, as defined in FIPS PUB 81.

CDMF Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.

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, such as software only, as 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.

ECDSA Elliptic Curve DSA, as in IEEE P1363.

FASTHASH MISSI's FASTHASH message-digesting algorithm.

IDEA Ascom Systec's symmetric block cipher.

JUNIPER MISSI's JUNIPER block cipher.

KEA MISSI's Key Exchange Algorithm.

LYNKS A smart card manufactured by SPYRUS.

MAC Message Authentication Code, as defined in ANSI X9.9.

MAYFLY MISSI's MAYFLY key agreement algorithm.

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.

OAEP Optimal Asymmetric Encryption Padding for RSA.

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.

RC5 RSA Data Security's RC5 symmetric block cipher.

Reader The means by which information is exchanged with a device.

Session A logical connection between an application and a token.

SET The Secure Electronic Transaction protocol.

SHA-1 The (revised) Secure Hash Algorithm, as defined in FIPS PUB 180, as amended by NIST.

Slot A logical reader that potentially contains a token.

SKIPJACK MISSI's SKIPJACK block cipher.

SSL The Secure Sockets Layer 3.0 protocol.

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
h	a handle
ul	a CK_ULONG
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pul	a pointer to a CK_ULONG

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 32 bits long */
typedef unsigned long int CK_ULONG;

/* a signed value, the same size as a CK_ULONG */
typedef long int CK_LONG;

/* 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_ULONG_PTR  /* Pointer to a CK_ULONG */
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 an 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	' '

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 macros, if needed:

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

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


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 these devices becomes increasingly valuable. This standard addresses this need.

5.1 Design goals

Cryptoki was intended from the beginning to be 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. 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.

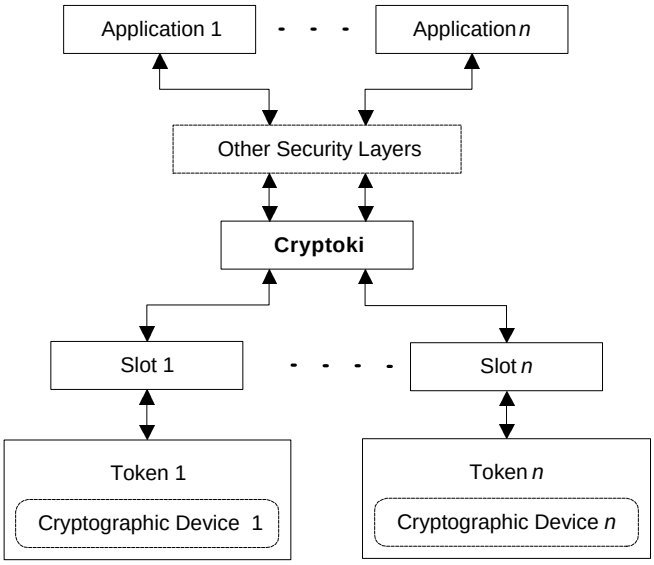

Figure 5-1, General Model

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 tokens in any or all of those slots.

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 special 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 to produce 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, therefore, direct linking is generally preferable. However, whether the linking is direct or dynamic, the programming interface between the application and a Cryptoki library remains the same.

The kinds of devices and capabilities supported will depend on the particular Cryptoki library. This standard specifies only 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, rather than 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 a cryptographic key. The key may be a public key, a private key, or a secret key; each of these types of keys has subtypes for use in specific mechanisms. This view is illustrated in the following figure:

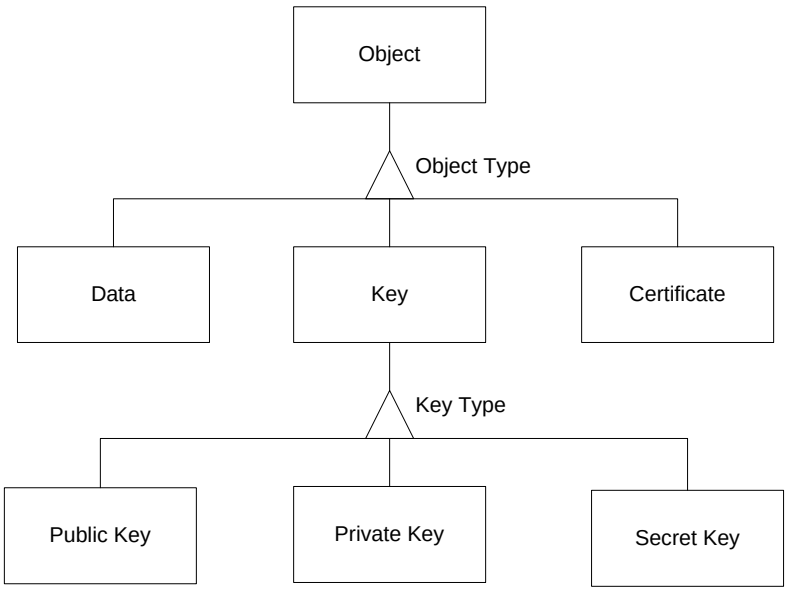

Figure 5-2, Object Hierarchy

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 even after the "sessions" (connections between an application and the token) are closed and the token is removed from its slot. "Session objects" are more temporary: whenever a session is closed by any means, all session objects created by that session are automatically destroyed.

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 or some other token-dependent method (for example, a biometric device).

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

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 a particular type of 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. Only the normal user is allowed access to private objects on the token, and that access is granted only after the normal user has been authenticated. Some tokens may also 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 (or otherwise define how the normal user may be authenticated), and possibly manipulate some public objects. The normal user cannot log in until the SO has set the normal 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 normal user may be the same person or may be different, but such matters are outside the scope of this standard.

With respect to PINs that are entered through an application, Cryptoki assumes only that they are variable-length strings of characters 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 PINs.
- How the PINs are generated (by the user, by the application, or by some other means).

PINs that are entered via some means other than an application (*e.g.*, via a PINpad on the token) are even more abstract. Cryptoki knows how to wait for such a PIN to be entered and used to gain authentication, and little more.

5.5 Sessions

Cryptoki requires that an application open one or more sessions with a token before the application has access to the token's objects and functions. A session provides a 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, modify, and destroy token objects.

All processes or threads of a given application have access to exactly the same sessions and the same session objects. If several applications are running concurrently, it may or may not be the case that they all have access to the same sessions and the same session objects; this is implementation-dependent. Exactly what constitutes an "application" is also implementation-dependent: in some environments, it might be appropriate to consider an application to be a single process; in other environments, that might not be appropriate.

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

When a session is closed, any session objects which were created in that session are destroyed. This holds even for session objects which are "being used" by other sessions.

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

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 Section 5.5.1 and Section 5.5.2.

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 initially opened, it is in either the "R/O Public Session" state (if there are no previously open sessions that are logged in) or the "R/O User Functions" state (if there is already an open session that is logged in). Note that read-only SO sessions do not exist.

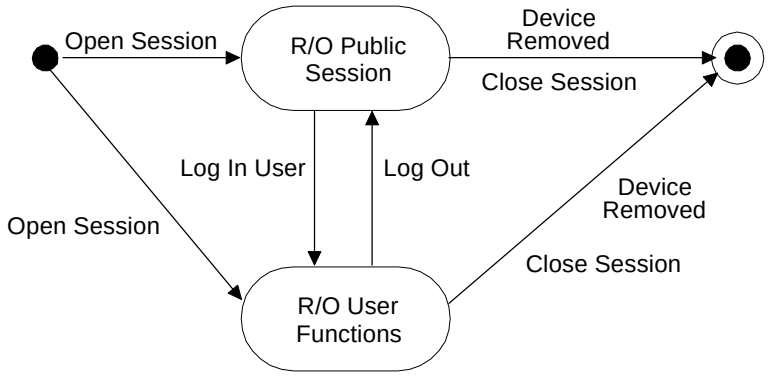

Figure 5-3, Read-Only Session States

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 token objects and read/write access to public session objects.
R/O User Functions	The normal user has been authenticated to the token. The application has read-only access to all token objects (public or private) and read/write access to all session objects (public or private).

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 either the "R/W Public Session" state (if there are no previously open sessions that are logged in), the "R/W User Functions" state (if there is already an open session that the normal user is logged into), or the "R/W SO Functions" state (if there is already an open session that the SO is logged into).

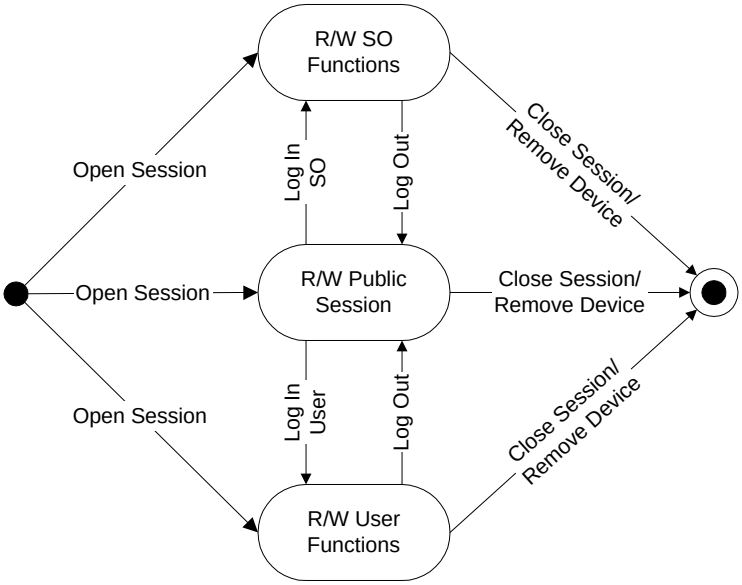

Figure 5-4, Read/Write Session States

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 all public objects.
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 all objects.

5.5.3 Permitted object accesses by sessions

The following table summarizes the kind of access each type of session has to each type of object. A given type of session has either read-only access, read/write access, or no access whatsoever to a given type of object.

Note that creating or deleting an object requires read/write access to it, *e.g.*, a "R/O User Functions" session cannot create a token object.

Table 5-3, Access to Different Types Objects by Different Types of Sessions

Type of object	Type of session				
	R/O Public	R/W Public	R/O User	R/W User	R/W SO
Public session object	R/W	R/W	R/W	R/W	R/W
Private session object			R/W	R/W	
Public token object	R/O	R/W	R/O	R/W	R/W
Private token object			R/O	R/W	

5.5.4 Session events

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

Table 5-4, 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.

When the device is removed, all sessions are automatically logged out. Furthermore, all sessions with the device are closed (this latter behavior is new for v2.0 of Cryptoki)—an application cannot have a session with a token which is not present. In actuality, Cryptoki may not be constantly monitoring whether or not the token is present, and so the token's absence may not be noticed until a Cryptoki function is executed. If the token is re-inserted into the slot before that, Cryptoki may never know that it was missing.

Also new to Cryptoki v2.0 is the fact that all sessions that an application has with a token must have the same login/logout status (*i.e.*, for a given application and token, one of the following holds: all sessions are public sessions; all sessions are SO sessions; or all sessions are user sessions). When an application's session logs in to a token, *all* of that application's sessions with that token become logged in, and when an application's session logs out of a token, *all* of that application's sessions with that token become logged out. Similarly, for example, if an application already has a R/O user session open with a token, and then opens a R/W session with that token, the R/W session is automatically logged in.

This implies that a given application may not simultaneously have SO sessions and user sessions open with a given token. It also implies that if an application has a R/W SO session with a token, then it may not open a R/O session with that token, since R/O SO sessions do not exist. For the same reason, if an application has a R/O session open, then it may not log any other session into the token as the SO.

The above restrictions on the login/logout status of a single application's sessions may also hold for sessions opened by different application. For example, it may be impossible for one application to have a R/O user session open with a token at the same time that another application has a R/W SO session open with the same token. Whether or not this is the case is implementation-dependent (see Section 5.5.7 and Section 5.5.8 for more information).

5.5.5 Session handles and object handles

A session handle is a Cryptoki-assigned value that identifies a session. It is akin to a file handle, and is specified to functions to indicate which session the function should act on. However, a session handle differs from a file handle in that all threads or processes of an application have equal access to all session handles. That is, anything that can be accomplished with a given file handle by one thread or process can also be accomplished with that file handle by any other thread or process belonging to the same application.

Cryptoki also has object handles, which are identifiers used to manipulate objects. Object handles are similar to session handles: all threads or processes of a given application have equal access to objects through object handles. The only exception to this is that R/O sessions only have read-only access to token objects, whereas R/W sessions have read/write access to token objects.

Valid session handles and object handles in Cryptoki always have nonzero values.

5.5.6 Capabilities of sessions

Very roughly speaking, there are three broad types of operations an open session can perform: administrative operations (such as logging in); object management operations (such as destroying an object on the token); and cryptographic operations (such as computing a message digest). In general, a single session can perform only one operation at a time. This is the reason that it may be desirable for a single application to open multiple sessions with a single token. For efficiency's sake, however, a single session can perform the following pairs of operation types simultaneously: message digesting and encryption; decryption and message digesting; signature or MACing and encryption; and decryption and verifying signatures or MACs. Details on performing simultaneous cryptographic operations in one session will be provided in Section 9.13.

5.5.7 Public Cryptoki libraries and private Cryptoki libraries

Cryptoki v2.0 implementations come in two essentially different varieties: "public Cryptoki libraries", in which all applications using a token have access to the same sessions and session objects (this was the only type of Cryptoki library in the Cryptoki v1.0 document), and "private Cryptoki libraries", in which each application has its own private set of sessions and session objects, which no other application can access.

5.5.8 Example of use of sessions

We give here a detailed and lengthy example of how applications can make use of sessions in a private Cryptoki library. Afterwards, we indicate how things would differ if we were making use of a public Cryptoki library, instead. We caution that our example is decidedly *not* meant to

indicate how multiple applications *should* use Cryptoki simultaneously; rather, it is meant to clarify what uses of Cryptoki's sessions and objects and handles are permissible. In other words, instead of demonstrating good technique here, we demonstrate "pushing the envelope".

For our example, we suppose that two applications, **A** and **B**, are using a private Cryptoki library to access a single token. Each application has two processes running: **A** has processes **A1** and **A2**, and **B** has processes **B1** and **B2**.

1. **A1** and **B1** each initialize the Cryptoki library by calling **C_Initialize** (the specifics of Cryptoki functions will be explained in Section 8.7.11). Note that exactly one call to **C_Initialize** should be made for each application (as opposed to one call for every process, for example).

2. **A1** opens a R/W session and receives the session handle 7 for the session. Since this is the first session to be opened for **A**, it is a public session.

3. **A2** opens a R/O session and receives the session handle 4. Since all of **A**'s existing sessions are public sessions, session 4 is also a public session.

4. **A1** attempts to log the SO in to session 7. The attempt fails, because if session 7 becomes an SO session, then session 4 does, as well, and R/O SO sessions do not exist. **A1** receives an error message indicating that the existence of a R/O session has blocked this attempt to log in.

5. **A2** logs the normal user in to session 7. This turns session 7 into a R/W user session, and turns session 4 into a R/O user session. Note that because **A1** and **A2** belong to the same application, they have equal access to all sessions, and therefore, **A2** is able to perform this action.

6. **A2** opens a R/W session and receives the session handle 9. Since all of **A**'s existing sessions are user sessions, session 9 is also a user session.

7. **A1** closes session 9.

8. **B1** attempts to log out session 4. The attempt fails, because **A** and **B** have no access rights to each other's sessions or objects. **B1** receives an error message which indicates that there is no such session handle.

9. **B2** attempts to close session 4. The attempt fails in precisely the same way as **B1**'s attempt to log out session 4 failed.

10. **B1** opens a R/W session and receives the session handle 7. Note that, as far as **B** is concerned, this is the first occurrence of session handle 7. **A**'s session 7 and **B**'s session 7 are completely different sessions.

11. **B1** logs the SO in to [**B**'s] session 7. This turns **B**'s session 7 into a R/W SO session, and has no effect on either of **A**'s sessions.

12. **B2** attempts to open a R/O session. The attempt fails, since **B** already has an SO session open, and R/O SO sessions do not exist. **B1** receives an error message indicating that the existence of an SO session has blocked this attempt to open a R/O session.

13. **A1** uses [**A**'s] session 7 to create a session object **O1** of some sort and receives the object handle 7. Note that a Cryptoki implementation may or may not support separate spaces of handles for sessions and objects.

14. **B1** uses [**B**'s] session 7 to create a token object **O2** of some sort and receives the object handle 7. As with session handles, different applications have no access rights to each other's object handles, and so **B**'s object handle 7 is entirely different from **A**'s object handle 7. Of course, since **B1** is an SO session, it cannot create private objects, and so **O2** must be a public object (if **B1** attempted to create a private object, it would fail).

15. **B2** uses [**B**'s] session 7 to perform some operation to modify the object associated with [**B**'s] object handle 7. This modifies **O2**.

16. **A1** uses [**A**'s] session 4 to perform an object search operation to get a handle for **O2**. The search returns object handle 1. Note that **A**'s object handle 1 and **B**'s object handle 7 now point to the same object.

17. **A1** attempts to use [**A**'s] session 4 to modify the object associated with [**A**'s] object handle 1. The attempt fails, because **A**'s session 4 is a R/O session, and is therefore incapable of modifying **O2**, which is a token object. **A1** receives an error message indicating that the session is a R/O session.

18. **A1** uses [**A**'s] session 7 to modify the object associated with [**A**'s] object handle 1. This time, since **A**'s session 7 is a R/W session, the attempt succeeds in modifying **O2**.

19. **B1** uses [**B**'s] session 7 to perform an object search operation to find **O1**. Since **O1** is a session object belonging to **A**, however, the search does not succeed.

20. **A2** uses [**A**'s] session 4 to perform some operation to modify the object associated with [**A**'s] object handle 7. This operation modifies **O1**.

21. **A2** uses [**A**'s] session 7 to destroy the object associated with [**A**'s] object handle 1. This destroys **O2**.

22. **B1** attempts to perform some operation with the object associated with [**B**'s] object handle 7. The attempt fails, since there is no longer any such object. **B1** receives an error message indicating that its object handle is invalid.

23. **A1** logs out [**A**'s] session 4. This turns **A**'s session 4 into a R/O public session, and turns **A**'s session 7 into a R/W public session.

24. **A1** closes [**A**'s] session 7. This destroys the session object **O1**, which was created by **A**'s session 7.

25. **A2** attempt to use [**A**'s] session 4 to perform some operation with the object associated with [**A**'s] object handle 7. The attempt fails, since there is no longer any such object.

26. **A2** executes a call to **C_CloseAllSessions**. This closes [**A**'s] session 4. At this point, if **A** were to open a new session, the session would not be logged in.

27. **B2** closes [**B**'s] session 7. At this point, if **B** were to open a new session, the session would not be logged in.

28. **A** and **B** each call **C_Finalize** to indicate that they are done with the Cryptoki library.

If **A** and **B** were using a public Cryptoki library, then all processes using the library would have exactly the same access rights to sessions and objects. In other words, in a public library, there is no distinction made between processes belonging to different applications. Anything that can be done by one application's processes with a given session handle can also be done by another application's processes. For example, with a public library, step 8 above would have succeeded. Also, with a public library, step 10 could not return session handle 7, since session handle 7 was already in use.

Furthermore, since public Cryptoki libraries have no notion of which application "owns" a Cryptoki session, all sessions with a given token must have the same login/logout status. Because of this, if one application logs out one of its sessions, all sessions of *all* applications are logged out as well. It is therefore recommended that applications only make a **C_Logout** call under exceptional circumstances. Instead, when an application finishes using a token, it should close all "its" sessions (*i.e.*, all the sessions that it was using) one at a time, and then call **C_Finalize**. Similarly, if an application using a public Cryptoki library calls **C_CloseAllSessions**, all session of *all* applications will be closed, and so an application should not normally execute such a call.

Applications should in general not intentionally attempt to share sessions or session objects with one another, even when they are using a public Cryptoki library (an application may not even know what type of Cryptoki library it is using, of course).

5.6 Function overview

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

Table 5-5, Summary of Cryptoki Functions

Category	Function	Description
General purpose functions	C_Initialize	initializes Cryptoki
	C_Finalize	clean up miscellaneous Cryptoki-associated resources
	C_GetInfo	obtains general information about Cryptoki
	C_GetFunctionList	obtains entry points of Cryptoki library functions
Slot and token management functions	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_GetMechanismList	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

Category	Function	Description
Session management functions	C_OpenSession	opens a connection between an application and a particular token or sets up an application callback for token insertion
	C_CloseSession	closes a session
	C_CloseAllSessions	closes all sessions with a token
	C_GetSessionInfo	obtains information about the session
	C_GetOperationState	obtains the cryptographic operations state of a session
	C_SetOperationState	sets the cryptographic operations state of a session
	C_Login	logs into a token
	C_Logout	logs out from a token
Object management functions	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
	C_FindObjectsFinal	finishes an object search operation
Encryption functions	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
Decryption functions	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_DigestKey	digests a key
	C_DigestFinal	finishes a multiple-part digesting operation

Category	Function	Description
Signing and MACing functions	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
Functions for verifying signatures and MACs	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
Dual-purpose cryptographic functions	C_DigestEncryptUpdate	continues simutaneous multi-part digesting and encryption operations
	C_DecryptDigestUpdate	continues simultaneous multi-part decryption and digesting operations
	C_SignEncryptUpdate	continues simultaneous multi-part signature and encryption operations
	C_DecryptVerifyUpdate	continues simultaneous multi-part decryption and verification operations
Key management functions	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 functions	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Parallel function management functions	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
Callback function		application-supplied function to process notifications from Cryptoki

Functions in the “Encryption functions”, “Decryption functions”, “Message digesting functions”, “Signing and MACing functions”, “Functionre for verifying signatures and MACs”, “Dual-purpose cryptographic functions”, “Key management functions”, and “Random number generation” 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 may not be sufficient to use it; the PIN may also be needed.

2. Additional protection can be given to private keys and secret keys by marking them as "sensitive" or "nonextractable". Sensitive keys cannot be revealed in plaintext off the token, and nonextractable keys cannot be revealed off the token even when encrypted (though they can still be used as keys).

It is expected that access to private, sensitive, or nonextractable 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 may be passed through the operating system. This can make it 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 services other 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, a token may be involved in the "booting up" of the system.

We note that none of the attacks just described can compromise keys marked "sensitive," since a key that is sensitive will always remain sensitive. Similarly, a key that is unextractable cannot be modified to be extractable. However, during key generation, if a private key or secret key is not created as "sensitive" and "unextractable", a copy of the private key could be obtained by a rogue application before these attributes are set. It can therefore be 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 the following subsections, organized into categories, based on the kind of information they represent. The data types for holding parameters for various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves, in Section 10.

7.1 General information

Cryptoki represents general information with the following types:

♦ CK_VERSION

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, an SSL implementation, or the hardware or firmware version of a slot or token. 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 Cryptoki standard are always upwardly compatible within the same major version number.

♦ CK_VERSION_PTR

CK_VERSION_PTR points to a **CK_VERSION** structure. It is implementation-dependent.

♦ CK_INFO

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

```
typedef struct CK_INFO {
    CK_VERSION cryptokiVersion;
    CK_CHAR manufacturerID[32];
    CK_FLAGS flags;
    CK_CHAR libraryDescription[32];
    CK_VERSION libraryVersion;
} CK_INFO;
```


The fields of the structure have the following meanings:

cryptokiVersion	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
libraryDescription	character-string description of the library. Must be padded with the blank character (' ')
libraryVersion	Cryptoki library version number

For libraries written to the Cryptoki v2.0 document, the value of *cryptokiVersion* should be 2.0; the value of *libraryVersion* is the version number of the library software itself.

♦ CK_INFO_PTR

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

♦ CK_NOTIFICATION

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

```
typedef CK_ULONG CK_NOTIFICATION;
```

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

```
#define CKN_SURRENDER 0
#define CKN_COMPLETE 1
#define CKN_DEVICE_REMOVED 2
#define CKN_TOKEN_INSERTION 3
```

The notifications have the following meanings:

CKN_SURRENDER	Cryptoki is surrendering the execution of a function executing in serial so that the application may perform other operations. After performing any desired 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 has detected that the device underlying the token has been removed from the reader. Not all slots/tokens support this notification.

CKN_TOKEN_INSERTION	Cryptoki has detected that the device underlying the token has been inserted into the reader. Not all slots/tokens support this notification.

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_VERSION hardwareVersion;
    CK_VERSION firmwareVersion;
} CK_SLOT_INFO;
```

The fields of the structure have the following meanings:

slotDescription	character-string description of the slot. 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.
hardwareVersion	version number of the slot’s hardware
firmwareVersion	version number of the slot’s firmware

The following table defines the *flags* parameter:

Table 7-1, Slot Information Flags

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x00000001	TRUE if a token is present in the slot (*e.g.*, a device is in the reader)
CKF_REMOVABLE_DEVICE	0x00000002	TRUE if the reader supports removable devices
CKF_HW_SLOT	0x00000004	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_ULONG ulMaxSessionCount;
    CK_ULONG ulSessionCount;
    CK_ULONG ulMaxRwSessionCount;
    CK_ULONG ulRwSessionCount;
    CK_ULONG ulMaxPinLen;
    CK_ULONG ulMinPinLen;
    CK_ULONG ulTotalPublicMemory;
    CK_ULONG ulFreePublicMemory;
    CK_ULONG ulTotalPrivateMemory;
    CK_ULONG ulFreePrivateMe  mory;
    CK_VERSION hardwareVersion;
    CK_VERSION firmwareVersion;
    CK_CHAR utcTime[16];
} 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. Mu st be padded with the blank character (' ')

flags	bit flags indicating capabilities and status of the device as defined below
ulMaxSessionCount	maximum number of sessions that can be opened with the token at one time
ulSessionCount	number of sessions that are currently open with the token
ulMaxRwSessionCount	maximum number of read / write sessions that can be opened with the token at one time
ulRwSessionCount	number of read / write sessions that are currently open with the token
ulMaxPinLen	maximum length in bytes of the PIN
ulMinPinLen	minimum length in bytes of the PIN
ulTotalPublicMemory	the total amount of memory in bytes in which public objects may be stored
ulFreePublicMemory	the amount of free (unused) memory in bytes for public objects
ulTotalPrivateMemory	the total amount of memory in bytes in which private objects may be stored
ulFreePrivateMemory	the amount of free (unused) memory in bytes for private objects
hardwareVersion	version number of hardware
firmwareVersion	version number of firmware
utcTime	current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters). The value of this field only makes sense for tokens equipped with a clock, as indicated in the token information flags (see below)

The following table defines the *flags* parameter:

Table 7-2, Token Information Flags

Bit Flag	Mask	Meaning
CKF_RNG	0x00000001	TRUE if the token has its own random number generator
CKF_WRITE_PROTECTED	0x00000002	TRUE if the token is write-protected
CKF_LOGIN_REQUIRED	0x00000004	TRUE if a user must be logged in to perform cryptographic functions
CKF_USER_PIN_INITIALIZED	0x00000008	TRUE if the normal user's PIN has been initialized
CKF_EXCLUSIVE_EXISTS	0x00000010	TRUE if an exclusive session exists
CKF_RESTORE_KEY_NOT_NEEDED	0x00000020	TRUE if a successful save of a session's cryptographic operations state always contains all keys needed to restore the state of the session
CKF_CLOCK_ON_TOKEN	0x00000040	TRUE if token has its own hardware clock
CKF_SUPPORTS_PARALLEL	0x00000080	TRUE if token supports parallel sessions through this Cryptoki library
CKF_PROTECTED_AUTHENTICATION_PATH	0x00000100	TRUE if token has a "protected authentication path", whereby a user can log in to the token without passing a PIN through the Cryptoki library
CKF_DUAL_CRYPTO_OPERATIONS	0x00000200	TRUE if a single session with the token can perform dual cryptographic operations (see Section 9.13)

Exactly what the **CKF_WRITE_PROTECTED** flag means is not specified in Cryptoki. An application may be unable to perform certain actions on a write-protected token; these actions can include any of the following, among other actions:

- Creating / modifying an object on the token.
- Creating / modifying a token object on the token.
- Changing the SO's PIN.

- Changing the normal user's PIN.

The *ulMaxSessionCount*, *ulSessionCount*, *ulMaxRwSessionCount*, *ulRwSessionCount*, *ulMaxPinLen*, *ulMinPinLen*, *ulTotalPublicMemory*, *ulFreePublicMemory*, *ulTotalPrivateMemory*, and *ulFreePrivateMemory* fields have a quirk in their interpretations. For each of these fields, it is possible for a token to return a special value which means, "I cannot/will not divulge that information." This value is the integer value -1, which (unfortunately) does not fit into a variable of type CK_ULONG. The upshot of all this is that the correct way to interpret (for example) the *ulMaxSessionCount* field is as follows:

```
CK_TOKEN_INFO info;

.
.
.
if ((CK_LONG) info.ulMaxSessionCount == -1) {
    /* Token refuses to give value of ulMaxSessionCount */
    .
    .
    .
} else {
    /* info.ulMaxSessionCount really does contain what it should */
    .
    .
    .
}
```

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

```
typedef CK_ULONG CK_USER_TYPE;
```

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

```
#define CKU_SO 0
#define CKU_USER 1
```

♦ CK_STATE

CK_STATE holds the session state, as decribed in Sections 5.5.1 and 5.5.2. It is defined as follows:

```
typedef CK_ULONG CK_STATE;
```

For this version of Cryptoki, the following session states are defined:

```
#define CKS_RO_PUBLIC_SESSION 0
#define CKS_RO_USER_FUNCTIONS 1
#define CKS_RW_PUBLIC_SESSION 2
#define CKS_RW_USER_FUNCTIONS 3
#define CKS_RW_SO_FUNCTIONS 4
```

♦ 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_ULONG ulDeviceError;
} 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
ulDeviceError	an error code defined by the cryptographic device. Used for errors not covered by Cryptoki.

The following table defines the *flags* parameter:

Table 7-3, Session Information Flags

Bit Flag	Mask	Meaning
CKF_EXCLUSIVE_SESSION	0x00000001	TRUE if the session is exclusive; FALSE if the session is shared
CKF_RW_SESSION	0x00000002	TRUE if the session is read / write; FALSE if the session is read-only
CKF_SERIAL_SESSION	0x00000004	TRUE if cryptographic functions are performed in serial with the application; FALSE if the functions may be performed in parallel with the application
CKF_INSERTION_CALLBACK	0x00000008	this flag is write-only, *i.e.*, is supplied as an argument to a **C_OpenSession** call, but is never set in a session's **CK_SESSION_INFO** structure. It is TRUE if the call is a request for a token insertion callback, instead of being a request to open a session

♦ 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;
```

When an object is created or found on a token, Cryptoki assigns it an object handle for sessions to use to access it. A particular object on a token does not necessarily have a handle which is fixed for the lifetime of the object; however, if a particular session can use a particular handle to access a particular object, then that session will continue to be able to use that handle to acces that object as long as the session continues to exist, the object continues to exist, and the object continues to be accessible to the session.

♦ 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_ULONG CK_OBJECT_CLASS;
```

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

```
#define CKO_DATA            0x00000000
#define CKO_CERTIFICATE     0x00000001
#define CKO_PUBLIC_KEY      0x00000002
#define CKO_PRIVATE_KEY     0x00000003
#define CKO_SECRET_KEY      0x00000004
#define CKO_VENDOR_DEFINED  0x80000000
```

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_ULONG CK_KEY_TYPE;
```

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

```
#define CKK_RSA             0x00000000
#define CKK_DSA             0x00000001
#define CKK_DH              0x00000002
#define CKK_ECDSA           0x00000003
#define CKK_MAYFLY          0x00000004
#define CKK_KEA             0x00000005
#define CKK_GENERIC_SECRET  0x00000010
#define CKK_RC2             0x00000011
#define CKK_RC4             0x00000012
#define CKK_DES             0x00000013
#define CKK_DES2            0x00000014
#define CKK_DES3            0x00000015
#define CKK_CAST            0x00000016
#define CKK_CAST3           0x00000017
#define CKK_CAST5           0x00000018
#define CKK_RC5             0x00000019
#define CKK_IDEA            0x0000001A
#define CKK_SKIPJACK        0x0000001B
#define CKK_BATON           0x0000001C
#define CKK_JUNIPER         0x0000001D
#define CKK_CDMF            0x0000001E
#define CKK_VENDOR_DEFINED  0x80000000
```


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_ULONG CK_CERTIFICATE_TYPE;
```

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

```
#define CKC_X_509           0x00000000
#define CKC_VENDOR_DEFINED  0x80000000
```

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_ULONG CK_ATTRIBUTE_TYPE;
```

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

```
#define CKA_CLASS              0x00000000
#define CKA_TOKEN              0x00000001
#define CKA_PRIVATE            0x00000002
#define CKA_LABEL              0x00000003
#define CKA_APPLICATION        0x00000010
#define CKA_VALUE              0x00000011
#define CKA_CERTIFICATE_TYPE   0x00000080
#define CKA_ISSUER             0x00000081
#define CKA_SERIAL_NUMBER      0x00000082
#define CKA_KEY_TYPE           0x00000100
#define CKA_SUBJECT            0x00000101
#define CKA_ID                 0x00000102
#define CKA_SENSITIVE          0x00000103
#define CKA_ENCRYPT            0x00000104
#define CKA_DECRYPT            0x00000105
#define CKA_WRAP               0x00000106
#define CKA_UNWRAP             0x00000107
#define CKA_SIGN               0x00000108
#define CKA_SIGN_RECOVER       0x00000109
#define CKA_VERIFY             0x0000010A
#define CKA_VERIFY_RECOVER     0x0000010B
#define CKA_DERIVE             0x0000010C
#define CKA_START_DATE         0x00000110
#define CKA_END_DATE           0x00000111
#define CKA_MODULUS            0x00000120
#define CKA_MODULUS_BITS       0x00000121
#define CKA_PUBLIC_EXPONENT    0x00000122
#define CKA_PRIVATE_EXPONENT   0x00000123
```



```
#define CKA_PRIME_1             0x00000124
#define CKA_PRIME_2             0x00000125
#define CKA_EXPONENT_1          0x00000126
#define CKA_EXPONENT_2          0x00000127
#define CKA_COEFFICIENT         0x00000128
#define CKA_PRIME               0x00000130
#define CKA_SUBPRIME            0x00000131
#define CKA_BASE                0x00000132
#define CKA_VALUE_BITS          0x00000160
#define CKA_VALUE_LEN           0x00000161
#define CKA_EXTRACTABLE         0x00000162
#define CKA_LOCAL               0x00000163
#define CKA_NEVER_EXTRACTABLE   0x00000164
#define CKA_ALWAYS_SENSITIVE    0x00000165
#define CKA_MODIFIABLE          0x00000170
#define CKA_VENDOR_DEFINED      0x80000000
```

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_ULONG ulValueLen;
} CK_ATTRIBUTE;
```

The fields of the structure have the following meanings:

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

If an attribute has no value, then *pValue* = NULL_PTR, and *ulValueLen* = 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 Data types for mechanisms

Cryptoki supports the following types for describing mechanisms and parameters to them:

♦ CK_MECHANISM_TYPE

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

```
typedef CK_ULONG CK_MECHANISM_TYPE;
```

For Cryptoki v2.0, the following mechanism types are defined:

```
#define CKM_RSA_PKCS_KEY_PAIR_GEN      0x00000000
#define CKM_RSA_PKCS                   0x00000001
#define CKM_RSA_9796                   0x00000002
#define CKM_RSA_X_509                  0x00000003
#define CKM_MD2_RSA_PKCS               0x00000004
#define CKM_MD5_RSA_PKCS               0x00000005
#define CKM_SHA1_RSA_PKCS              0x00000006
#define CKM_DSA_KEY_PAIR_GEN           0x00000010
#define CKM_DSA                        0x00000011
#define CKM_DSA_SHA1                   0x00000012
#define CKM_DH_PKCS_KEY_PAIR_GEN       0x00000020
#define CKM_DH_PKCS_DERIVE             0x00000021
#define CKM_RC2_KEY_GEN                0x00000100
#define CKM_RC2_ECB                    0x00000101
#define CKM_RC2_CBC                    0x00000102
#define CKM_RC2_MAC                    0x00000103
#define CKM_RC2_MAC_GENERAL            0x00000104
#define CKM_RC2_CBC_PAD                0x00000105
#define CKM_RC4_KEY_GEN                0x00000110
#define CKM_RC4                        0x00000111
#define CKM_DES_KEY_GEN                0x00000120
#define CKM_DES_ECB                    0x00000121
#define CKM_DES_CBC                    0x00000122
```



```
#define CKM_DES_MAC                     0x00000123
#define CKM_DES_MAC_GENERAL             0x00000124
#define CKM_DES_CBC_PAD                 0x00000125
#define CKM_DES2_KEY_GEN                0x00000130
#define CKM_DES3_KEY_GEN                0x00000131
#define CKM_DES3_ECB                    0x00000132
#define CKM_DES3_CBC                    0x00000133
#define CKM_DES3_MAC                    0x00000134
#define CKM_DES3_MAC_GENERAL            0x00000135
#define CKM_DES3_CBC_PAD                0x00000136
#define CKM_CDMF_KEY_GEN                0x00000140
#define CKM_CDMF_ECB                    0x00000141
#define CKM_CDMF_CBC                    0x00000142
#define CKM_CDMF_MAC                    0x00000143
#define CKM_CDMF_MAC_GENERAL            0x00000144
#define CKM_CDMF_CBC_PAD                0x00000145
#define CKM_MD2                         0x00000200
#define CKM_MD2_HMAC                    0x00000201
#define CKM_MD2_HMAC_GENERAL            0x00000202
#define CKM_MD5                         0x00000210
#define CKM_MD5_HMAC                    0x00000211
#define CKM_MD5_HMAC_GENERAL            0x00000212
#define CKM_SHA_1                       0x00000220
#define CKM_SHA_1_HMAC                  0x00000221
#define CKM_SHA_1_HMAC_GENERAL          0x00000222
#define CKM_CAST_KEY_GEN                0x00000300
#define CKM_CAST_ECB                    0x00000301
#define CKM_CAST_CBC                    0x00000302
#define CKM_CAST_MAC                    0x00000303
#define CKM_CAST_MAC_GENERAL            0x00000304
#define CKM_CAST_CBC_PAD                0x00000305
#define CKM_CAST3_KEY_GEN               0x00000310
#define CKM_CAST3_ECB                   0x00000311
#define CKM_CAST3_CBC                   0x00000312
#define CKM_CAST3_MAC                   0x00000313
#define CKM_CAST3_MAC_GENERAL           0x00000314
#define CKM_CAST3_CBC_PAD               0x00000315
#define CKM_CAST5_KEY_GEN               0x00000320
#define CKM_CAST5_ECB                   0x00000321
#define CKM_CAST5_CBC                   0x00000322
#define CKM_CAST5_MAC                   0x00000323
#define CKM_CAST5_MAC_GENERAL           0x00000324
#define CKM_CAST5_CBC_PAD               0x00000325
#define CKM_RC5_KEY_GEN                 0x00000330
#define CKM_RC5_ECB                     0x00000331
#define CKM_RC5_CBC                     0x00000332
#define CKM_RC5_MAC                     0x00000333
#define CKM_RC5_MAC_GENERAL             0x00000334
#define CKM_RC5_CBC_PAD                 0x00000335
#define CKM_IDEA_KEY_GEN                0x00000340
#define CKM_IDEA_ECB                    0x00000341
#define CKM_IDEA_CBC                    0x00000342
#define CKM_IDEA_MAC                    0x00000343
#define CKM_IDEA_MAC_GENERAL            0x00000344
#define CKM_IDEA_CBC_PAD                0x00000345
#define CKM_GENERIC_SECRET_KEY_GEN      0x00000350
#define CKM_CONCATENATE_BASE_AND_KEY    0x00000360
#define CKM_CONCATENATE_BASE_AND_DATA   0x00000362
#define CKM_CONCATENATE_DATA_AND_BASE   0x00000363
#define CKM_XOR_BASE_AND_DATA           0x00000364
#define CKM_EXTRACT_KEY_FROM_KEY        0x00000365
```



```
#define CKM_SSL3_PRE_MASTER_KEY_GEN    0x00000370
#define CKM_SSL3_MASTER_KEY_DERIVE     0x00000371
#define CKM_SSL3_KEY_AND_MAC_DERIVE    0x00000372
#define CKM_SSL3_MD5_MAC               0x00000380
#define CKM_SSL3_SHA1_MAC              0x00000381
#define CKM_MD5_KEY_DERIVATION         0x00000390
#define CKM_MD2_KEY_DERIVATION         0x00000391
#define CKM_SHA1_KEY_DERIVATION        0x00000392
#define CKM_PBE_MD2_DES_CBC            0x000003A0
#define CKM_PBE_MD5_DES_CBC            0x000003A1
#define CKM_PBE_MD5_CAST_CBC           0x000003A2
#define CKM_PBE_MD5_CAST3_CBC          0x000003A3
#define CKM_PBE_MD5_CAST5_CBC          0x000003A4
#define CKM_KEY_WRAP_LYNKS             0x00000400
#define CKM_KEY_WRAP_SET_OAEP          0x00000401
#define CKM_SKIPJACK_KEY_GEN           0x00001000
#define CKM_SKIPJACK_ECB64             0x00001001
#define CKM_SKIPJACK_CBC64             0x00001002
#define CKM_SKIPJACK_OFB64             0x00001003
#define CKM_SKIPJACK_CFB64             0x00001004
#define CKM_SKIPJACK_CFB32             0x00001005
#define CKM_SKIPJACK_CFB16             0x00001006
#define CKM_SKIPJACK_CFB8              0x00001007
#define CKM_SKIPJACK_WRAP              0x00001008
#define CKM_SKIPJACK_PRIVATE_WRAP      0x00001009
#define CKM_SKIPJACK_RELAYX            0x0000100a
#define CKM_KEA_KEY_PAIR_GEN           0x00001010
#define CKM_KEA_KEY_DERIVE             0x00001011
#define CKM_FORTEZZA_TIMESTAMP         0x00001020
#define CKM_BATON_KEY_GEN              0x00001030
#define CKM_BATON_ECB128               0x00001031
#define CKM_BATON_ECB96                0x00001032
#define CKM_BATON_CBC128               0x00001033
#define CKM_BATON_COUNTER              0x00001034
#define CKM_BATON_SHUFFLE              0x00001035
#define CKM_BATON_WRAP                 0x00001036
#define CKM_ECDSA_KEY_PAIR_GEN         0x00001040
#define CKM_ECDSA                      0x00001041
#define CKM_ECDSA_SHA1                 0x00001042
#define CKM_MAYFLY_KEY_PAIR_GEN        0x00001050
#define CKM_MAYFLY_KEY_DERIVE          0x00001051
#define CKM_JUNIPER_KEY_GEN            0x00001060
#define CKM_JUNIPER_ECB128             0x00001061
#define CKM_JUNIPER_CBC128             0x00001062
#define CKM_JUNIPER_COUNTER            0x00001063
#define CKM_JUNIPER_SHUFFLE            0x00001064
#define CKM_JUNIPER_WRAP               0x00001065
#define CKM_FASTHASH                   0x00001070
#define CKM_VENDOR_DEFINED             0x80000000
```

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_ULONG ulParameterLen;
} 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 a "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

For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.

The following table defines the *flags* parameter:

Table 7-4, Mechanism Information Flags

Bit Flag	Mask	Meaning
CKF_HW	0x00000001	TRUE if the mechanism is performed by the device; FALSE if the mechanism is performed in software
CKF_ENCRYPT	0x00000100	TRUE if the mechanism can be used with **C_EncryptInit**
CKF_DECRYPT	0x00000200	TRUE if the mechanism can be used with **C_DecryptInit**
CKF_DIGEST	0x00000400	TRUE if the mechanism can be used with **C_DigestInit**
CKF_SIGN	0x00000800	TRUE if the mechanism can be used with **C_SignInit**
CKF_SIGN_RECOVER	0x00001000	TRUE if the mechanism can be used with **C_SignRecoverInit**
CKF_VERIFY	0x00002000	TRUE if the mechanism can be used with **C_VerifyInit**
CKF_VERIFY_RECOVER	0x00004000	TRUE if the mechanism can be used with **C_VerifyRecoverInit**
CKF_GENERATE	0x00008000	TRUE if the mechanism can be used with **C_GenerateKey**
CKF_GENERATE_KEY_PAIR	0x00010000	TRUE if the mechanism can be used with **C_GenerateKeyPair**
CKF_WRAP	0x00020000	TRUE if the mechanism can be used with **C_WrapKey**
CKF_UNWRAP	0x00040000	TRUE if the mechanism can be used with **C_UnwrapKey**
CKF_DERIVE	0x00080000	TRUE if the mechanism can be used with **C_DeriveKey**
CKF_EXTENSION	0x80000000	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.

7.6 Function types

Cryptoki represents information about functions with the following data types:

♦ CK_ENTRY

CK_ENTRY is not really a type. Rather, it is a string used provided to a C compiler in a given environment to produce an entry into Cryptoki (*i.e.*, a Cryptoki function). It is implementation-

dependent. For a Win32 Cryptoki .dll, it might be "`__declspec( dllexport )` ". For a Win16 Cryptoki .dll, it might be "`_export _far _pascal` ". For a Unix library, it might be "".

♦ CK_RV

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

```
typedef CK_ULONG CK_RV;
```

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

```
#define CKR_OK                                0x00000000
#define CKR_CANCEL                            0x00000001
#define CKR_HOST_MEMORY                       0x00000002
#define CKR_SLOT_ID_INVALID                   0x00000003
#define CKR_GENERAL_ERROR                     0x00000005
#define CKR_FUNCTION_FAILED                   0x00000006
#define CKR_ATTRIBUTE_READ_ONLY               0x00000010
#define CKR_ATTRIBUTE_SENSITIVE               0x00000011
#define CKR_ATTRIBUTE_TYPE_INVALID            0x00000012
#define CKR_ATTRIBUTE_VALUE_INVALID           0x00000013
#define CKR_DATA_INVALID                      0x00000020
#define CKR_DATA_LEN_RANGE                    0x00000021
#define CKR_DEVICE_ERROR                      0x00000030
#define CKR_DEVICE_MEMORY                     0x00000031
#define CKR_DEVICE_REMOVED                    0x00000032
#define CKR_ENCRYPTED_DATA_INVALID            0x00000040
#define CKR_ENCRYPTED_DATA_LEN_RANGE          0x00000041
#define CKR_FUNCTION_CANCELED                 0x00000050
#define CKR_FUNCTION_NOT_PARALLEL             0x00000051
#define CKR_FUNCTION_PARALLEL                 0x00000052
#define CKR_FUNCTION_NOT_SUPPORTED            0x00000054
#define CKR_KEY_HANDLE_INVALID                0x00000060
#define CKR_KEY_SIZE_RANGE                    0x00000062
#define CKR_KEY_TYPE_INCONSISTENT             0x00000063
#define CKR_KEY_NOT_NEEDED                    0x00000064
#define CKR_KEY_CHANGED                       0x00000065
#define CKR_KEY_NEEDED                        0x00000066
#define CKR_KEY_INDIGESTIBLE                  0x00000067
#define CKR_KEY_FUNCTION_NOT_PERMITTED        0x00000068
#define CKR_KEY_NOT_WRAPPABLE                 0x00000069
#define CKR_KEY_UNEXTRACTABLE                 0x0000006A
#define CKR_MECHANISM_INVALID                 0x00000070
#define CKR_MECHANISM_PARAM_INVALID           0x00000071
#define CKR_OBJECT_HANDLE_INVALID             0x00000082
#define CKR_OPERATION_ACTIVE                  0x00000090
#define CKR_OPERATION_NOT_INITIALIZED         0x00000091
#define CKR_PIN_INCORRECT                     0x000000A0
#define CKR_PIN_INVALID                       0x000000A1
#define CKR_PIN_LEN_RANGE                     0x000000A2
#define CKR_SESSION_CLOSED                    0x000000B0
#define CKR_SESSION_COUNT                     0x000000B1
#define CKR_SESSION_EXCLUSIVE_EXISTS          0x000000B2
#define CKR_SESSION_HANDLE_INVALID            0x000000B3
#define CKR_SESSION_PARALLEL_NOT_SUPPORTED    0x000000B4
#define CKR_SESSION_READ_ONLY                 0x000000B5
#define CKR_SESSION_EXISTS                    0x000000B6
#define CKR_SESSION_READ_ONLY_EXISTS          0x000000B7
#define CKR_SESSION_READ_WRITE_SO_EXISTS      0x000000B8
```



```
#define CKR_SIGNATURE_INVALID                  0x000000C0
#define CKR_SIGNATURE_LEN_RANGE                0x000000C1
#define CKR_TEMPLATE_INCOMPLETE                0x000000D0
#define CKR_TEMPLATE_INCONSISTENT              0x000000D1
#define CKR_TOKEN_NOT_PRESENT                  0x000000E0
#define CKR_TOKEN_NOT_RECOGNIZED               0x000000E1
#define CKR_TOKEN_WRITE_PROTECTED              0x000000E2
#define CKR_UNWRAPPING_KEY_HANDLE_INVALID      0x000000F0
#define CKR_UNWRAPPING_KEY_SIZE_RANGE          0x000000F1
#define CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT   0x000000F2
#define CKR_USER_ALREADY_LOGGED_IN             0x00000100
#define CKR_USER_NOT_LOGGED_IN                 0x00000101
#define CKR_USER_PIN_NOT_INITIALIZED           0x00000102
#define CKR_USER_TYPE_INVALID                  0x00000103
#define CKR_WRAPPED_KEY_INVALID                0x00000110
#define CKR_WRAPPED_KEY_LEN_RANGE              0x00000112
#define CKR_WRAPPING_KEY_HANDLE_INVALID        0x00000113
#define CKR_WRAPPING_KEY_SIZE_RANGE            0x00000114
#define CKR_WRAPPING_KEY_TYPE_INCONSISTENT     0x00000115
#define CKR_RANDOM_SEED_NOT_SUPPORTED          0x00000120
#define CKR_RANDOM_NO_RNG                      0x00000121
#define CKR_INSERTION_CALLBACK_NOT_SUPPORTED   0x00000141
#define CKR_BUFFER_TOO_SMALL                   0x00000150
#define CKR_SAVED_STATE_INVALID                0x00000160
#define CKR_INFORMATION_SENSITIVE              0x00000170
#define CKR_STATE_UNSAVEABLE                   0x00000180
#define CKR_VENDOR_DEFINED                     0x80000000
```

Section 9.1 defines the meaning 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 values through the PKCS process.

♦ CK_NOTIFY

CK_NOTIFY is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is implementation-dependent, but it is typically defined as follows, where CK_PTR is the C string used to create function pointers (*e.g.*, "*"):

```
typedef CK_RV (CK_ENTRY CK_PTR CK_NOTIFY)(
   CK_SESSION_HANDLE hSession,
   CK_NOTIFICATION event,
   CK_VOID_PTR pApplication
);
```

The arguments to a notification callback function have the following meanings:

hSession	The handle of the session performing the callback
event	The type of notification callback
pApplication	An application-defined value. This is the same value as was passed to **C_OpenSession** to open the session performing the callback

Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the Cryptoki API (there are 67 such functions in Cryptoki v2.0; see Section 9 for detailed

information about each of them), Cryptoki defines a type **CK_C_XXX**, which is a pointer to a function of **C_XXX**'s type.

♦ CK_FUNCTION_LIST

CK_FUNCTION_LIST is a structure which contains a Cryptoki version and a function pointer to each function in the Cryptoki API. It is defined as follows:

```
typedef struct CK_FUNCTION_LIST {
    CK_VERSION version;
    CK_C_Initialize C_Initialize;
    CK_C_Finalize C_Finalize;
    CK_C_ GetInfo C_GetInfo;
    CK_C_GetFunctionList C_GetFunctionList;
    CK_C_GetSlotList C_GetSlotList;
    CK_C_GetSlotInfo C_GetSlotInfo;
    CK_C_GetTokenInfo C_GetTokenInfo;
    CK_C_GetMechanismList C_GetMechanismList;
    CK_C_GetMechanismInfo C_GetMechanismInfo;
    CK_C_InitToken C_InitToken;
    CK_C_InitPIN C_InitPIN;
    CK_C_SetPIN C_SetPIN;
    CK_C_OpenSession C_OpenSession;
    CK_C_CloseSession C_CloseSession;
    CK_C_CloseAllSessions C_CloseAllSessions;
    CK_C_GetSessionInfo C_GetSessionInfo;
    CK_C_GetOperationState C_GetOperation  State;
    CK_C_SetOperationState C_SetOperationState;
    CK_C_Login C_Login;
    CK_C_Logout C_Logout;
    CK_C_CreateObject C_CreateObject;
    CK_C_CopyObject C_CopyObject;
    CK_C_DestroyObject C_DestroyObject;
    CK_C_GetObjectSize C_GetObjectSize;
    CK_C_GetAttributeValue C_GetAttributeValue;
    CK_C_SetAttributeValue C_SetAttributeValue;
    CK_C_FindObjectsInit C_FindObjectsInit;
    CK_C_FindObjects C_FindObjects;
    CK_C_FindObjectsFinal C_FindObjectsFinal;
    CK_C_EncryptInit C_EncryptInit;
    CK_C_Encrypt C_Encrypt;
    CK_C_Encry ptUpdate C_EncryptUpdate;
    CK_C_EncryptFinal C_EncryptFinal;
    CK_C_DecryptInit C_DecryptInit;
    CK_C_Decrypt C_Decrypt;
    CK_C_DecryptUpdate C_DecryptUpdate;
    CK_C_DecryptFinal C_DecryptFinal;
    CK_C_DigestInit C_DigestInit;
    CK_C_Digest C_Digest;
    CK_C_DigestUpdate C_DigestUpdate;
    CK_C_DigestKey C_DigestKey;
    CK_C_DigestFinal C_DigestFinal;
    CK_C_SignInit C_SignInit;
    CK_C_Sign C_Sign;
    CK_C_SignUpdate C_SignUpdate;
    CK_C_SignFinal C_SignFinal;
    CK_C_SignRecoverInit C_SignRecoverInit;
    CK_C_SignRecover C_Sig  nRecover;
    CK_C_VerifyInit C_VerifyInit;
```



```
    CK_C_Verify C_Verify;
    CK_C_VerifyUpdate C_VerifyUpdate;
    CK_C_VerifyFinal C_VerifyFinal;
    CK_C_VerifyRecoverInit C_VerifyRecoverInit;
    CK_C_VerifyRecover C_VerifyRecover;
    CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
    CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
    CK_C_SignEncryptUpdate C_SignEncryptUpdate;
    CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
    CK_C_GenerateKey C_GenerateKey;
    CK_C_GenerateKeyPair C_GenerateKeyPair;
    CK_C_WrapKey C_WrapKey;
    CK_C_UnwrapKey C_UnwrapKey;
    CK_C_DeriveKey C_DeriveKey;
    CK_C_SeedRandom C_SeedRandom;
    CK_C_GenerateRandom C_GenerateRandom;
    CK_C_GetFunctionStatus C_GetFunctionStatus;
    CK_C_CancelFunction C_CancelFunction;
} CK_FUNCTION_LIST;
```

Each Cryptoki library has a **CK_FUNCTION_LIST** structure, and a pointer to it may be obtained by the **C_GetFunctionList** function (see Section 9.2). The value that this pointer points to can be used by an application to quickly and easily find out which Cryptoki functions the library supports (and where the code for those functions is located). If a Cryptoki library does not support a particular Cryptoki function, then the entry for that function in the library's **CK_FUNCTION_LIST** structure should be NULL_PTR.

♦ CK_FUNCTION_LIST_PTR

CK_FUNCTION_LIST_PTR points to a **CK_FUNCTION_LIST**. It is implementation-dependent.

♦ CK_FUNCTION_LIST_PTR_PTR

CK_FUNCTION_LIST_PTR_PTR points to a **CK_FUNCTION_LIST_PTR**. It is implementation-dependent.

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:

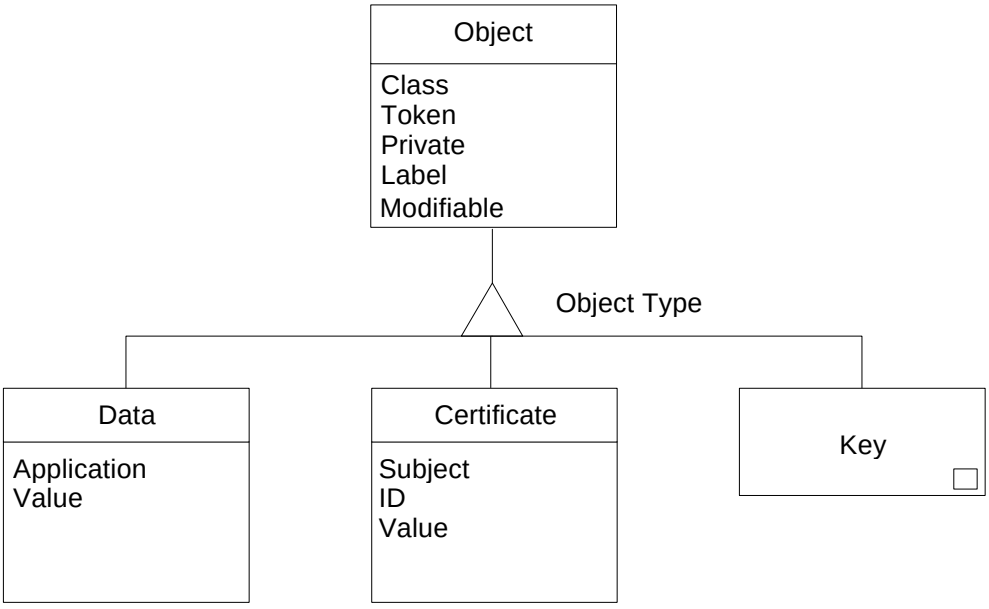

Figure 8-1, Cryptoki Object Hierarchy

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 some 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 9 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 unsigned integer of arbitrary size, most-significant byte first (*e.g.*, the integer 32768 is represented as the 2-byte string 0x80 0x00)
Local string	a string of **CK_CHAR**s (see Table 4-3)

A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the same values for all their attributes.

With the exception of RSA private key objects (see Section 0), each type of object possesses a completely well-defined set of attributes. For example, an X.509 certificate (see Section 8.3.1) has precisely the following attributes: **CKA_CLASS**, **CKA_TOKEN**, **CKA_PRIVATE**, **CKA_MODIFIABLE**, **CKA_LABEL**, **CKA_CERTIFICATE_TYPE**, **CKA_SUBJECT**, **CKA_ID**, **CKA_ISSUER**, **CKA_SERIAL_NUMBER**, **CKA_VALUE**. Some of these attributes possess default values, and need not be specified when creating an object; some of these default values may even be the empty string (“”). Nonetheless, the object possesses these attributes.

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; FALSE if object is a session object (default FALSE)
CKA_PRIVATE	CK_BBOOL	TRUE if object is a private object; FALSE if object is a public object (default FALSE)
CKA_MODIFIABLE	CK_BBOOL	TRUE if object can be modified (default TRUE)
CKA_LABEL	Local string	Description of the object (default empty)

[1]Must be specified when object is created

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

Cryptoki v2.0 supports the following values for **CKA_CLASS** (i.e., the following classes (types) of objects): **CKO_DATA**, **CKO_CERTIFICATE**, **CKO_PUBLIC_KEY**, **CKO_PRIVATE_KEY**, and **CKO_SECRET_KEY**.

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

The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only. It may or may not be the case that an unmodifiable object can be deleted.

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 using Cryptoki are listed. Objects may well carry other useful information on a token which is not visible to the application via Cryptoki.

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 **CKA_APPLICATION** and **CKA_VALUE** may be modified after the object is created.

The **CKA_APPLICATION** attribute may be provided by applications to indicate ownership of 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 containing attributes 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, sizeof(true)},
   {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 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.

The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an 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). However, 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 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, although this will not be enforced by Cryptoki.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_VALUE, certificate, sizeof(certificate)}
};
```

8.4 Key objects

The following figure illustrates the details of key objects:

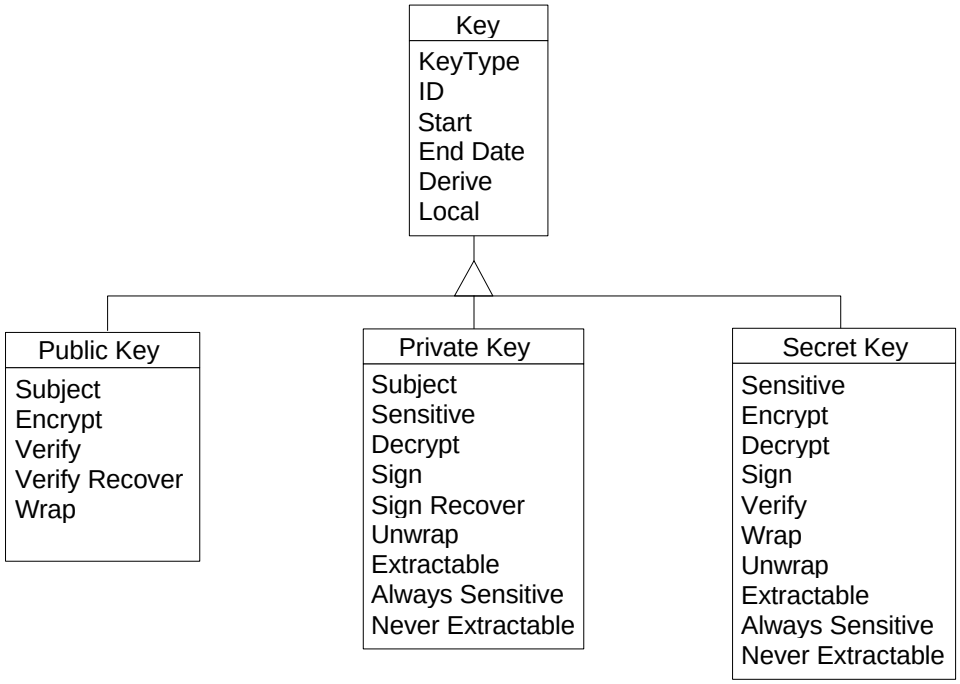

Figure 8-2, Key Object Detail

Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret keys. The following common footnotes apply to all the tables describing attributes of keys:

Table 8-5, Common footnotes for key attribute tables

[1] Must be specified when object is created.

[2] Must not be specified when object is created.

[3] Must be specified when object is generated.

[4] Must not be speficied when object is generated.

[5] Must be specified when object is unwrapped.

[6] Must not be specified when object is unwrapped.

[7] Cannot be revealed if object has its **CKA_SENSITIVE** attribute set to TRUE or its **CKA_EXTRACTABLE** attribute set to FALSE.

[8] May be modified after object is created.

[9] Default is up to the token. The application can specify an explicit value in the template, and Cryptoki will reject it if it cannot be supported by the library or token.

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 8-6, Common Key Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE[1,3,5]	CK_KEY_TYPE	Type of key
CKA_ID[8]	Byte array	Key identifier for key (default empty)
CKA_START_DATE[8]	CK_DATE	Start date for the key (default empty)
CKA_END_DATE[8]	CK_DATE	End date for the key (default empty)
CKA_DERIVE[8]	CK_BBOOL	TRUE if key supports key derivation (default FALSE)
CKA_LOCAL[2,4,6]	CK_BBOOL	TRUE if key was generated locally (*i.e.*, on token)

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 Certificate objects for further commentary.)

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

Note that the **CKA_START_DATE** and **CKA_END_DATE** attributes 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; doing this is up to the application.

The **CKA_DERIVE** attribute has the value TRUE if and only if it is possible to derive other keys from the key.

8.5 Public key objects

Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. This version of Cryptoki recognizes six types of public keys: RSA, DSA, ECDSA, Diffie-Hellman, KEA, and MAYFLY. The following table defines the attributes common to all public keys, in addition to the common attributes listed in Table 8-1 and Table 8-6:

Table 8-7, Common Public Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[8]	Byte array	DER encoding of the key subject name (default empty)
CKA_ENCRYPT[8]	CK_BBOOL	TRUE if key supports encryption[9]
CKA_VERIFY[8]	CK_BBOOL	TRUE if key supports verification[9]
CKA_VERIFY_RECOVER[8]	CK_BBOOL	TRUE if key supports verification where the data is recovered from the signature[9]
CKA_WRAP[8]	CK_BBOOL	TRUE if key supports wrapping[9]

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, Cryptoki does not enforce this, and 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 8-6, and Table 8-7:

Table 8-8, RSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS[1,4,6]	Big integer	Modulus *n*
CKA_MODULUS_BITS[2,3,6]	CK_ULONG	Length in bits of modulus *n*
CKA_PUBLIC_EXPONENT[1,3,6]	Big integer	Public exponent *e*

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {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 8-6, and Table 8-7:

Table 8-9, DSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3,6]	Big integer	Base *g*
CKA_VALUE[1,4,6]	Big integer	Public value *y*

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


8.5.3 ECDSA public key objects

ECDSA public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_ECDSA**) hold ECDSA public keys. The following table defines the ECDSA public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-7:

Table 8-10, ECDSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6]	Big integer	Public value *W*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "ECDSA parameters".

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_  ECDSA;
CK_CHAR label[] = "A n ECDSA public key object";
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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.5.4 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 8-6, and Table 8-7:

Table 8-11, Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p*
CKA_BASE[1,3,6]	Big integer	Base *g*
CKA_VALUE[1,4,6]	Big integer	Public value *y*

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 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.5.5 KEA public key objects

KEA public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_KEA**) hold KEA public keys. The following table defines the KEA public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-7:

Table 8-12, KEA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6]	Big integer	Public value *y*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "KEA parameters".

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_  KEA;
CK_CHAR label[] = "A  KEA public key object";
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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
```



```
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

8.5.6 MAYFLY public key objects

MAYFLY public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_MAYFLY**) hold MAYFLY public keys. The following table defines the MAYFLY public key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-7:

Table 8-13, MAYFLY Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,3,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,3,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,3,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6]	Big integer	Public value *W*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "MAYFLY parameters".

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_  MAYFLY ;
CK_CHAR label[] = "A  MAYFLY public key object";
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, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

8.6 Private key objects

Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. This version of Cryptoki recognizes six types of private key: RSA, DSA, ECDSA, Diffie-Hellman, KEA, and MAYFLY. The following table defines the attributes common to all private keys, in addition to the common attributes listed in Table 8-1 and Table 8-6:

Table 8-14, Common Private Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[8]	Byte array	DER encoding of certificate subject name (default empty)
CKA_SENSITIVE[8]	CK_BBOOL	TRUE if key is sensitive[9]
CKA_DECRYPT[8]	CK_BBOOL	TRUE if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	TRUE if key supports signatures where the signature is an appendix to the data[9]
CKA_SIGN_RECOVER[8]	CK_BBOOL	TRUE if key supports signatures where the data can be recovered from the signature[9]
CKA_UNWRAP[8]	CK_BBOOL	TRUE if key supports unwrapping[9]
CKA_EXTRACTABLE[8]	CK_BBOOL	TRUE if key is extractable[9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	TRUE if key has *always* had the CKA_SENSITIVE attribute set to TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	TRUE if key has *never* had the CKA_EXTRACTABLE attribute set to TRUE

After an object is created, the **CKA_SENSITIVE** attribute may only be set to TRUE. Similarly, after an object is created, the **CKA_EXTRACTABLE** attribute may only be set to FALSE.

If the **CKA_SENSITIVE** attribute is TRUE, or if the **CKA_EXTRACTABLE** attribute is false, then certain attributes of the private key cannot be revealed off the token. Which attributes these are is specified for each type of private key in the attribute table in the section describing that type of key.

If the **CKA_EXTRACTABLE** attribute is false, then the key cannot be wrapped.

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, this is not enforced by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

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 8-6, and Table 8-14:

Table 8-15, RSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS[1,4,6]	Big integer	Modulus *n*
CKA_PUBLIC_EXPONENT[4,6]	Big integer	Public exponent *e*
CKA_PRIVATE_EXPONENT[1,4,6,7]	Big integer	Private exponent *d*
CKA_PRIME_1[4,6,7]	Big integer	Prime *p*
CKA_PRIME_2[4,6,7]	Big integer	Prime *q*
CKA_EXPONENT_1[4,6,7]	Big integer	Private exponent *d* modulo *p*-1
CKA_EXPONENT_2[4,6,7]	Big integer	Private exponent *d* modulo *q*-1
CKA_COEFFICIENT[4,6,7]	Big integer	CRT coefficient q^{-1} mod *p*

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

Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above attributes, which can assist in performing rapid RSA computations. Other tokens might store only the **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values.

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 8-15 it keeps track of. Later, if an application asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed from these three values). However, a Cryptoki implementation may or may not actually do this extra computation. The only attributes from Table 8-15 that a Cryptoki implementation is *required* to be able to return values for are **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT**.

If an RSA private key object is created on a token, and more attributes from Table 8-15 are supplied to the object creation call than are supported by the token, the extra attributes are likely to be thrown away. If an attempt is made to create an RSA private key object on a token with insufficient attributes for that particular token, then the object creation call fails.

Note that when generating an RSA private key, there is no **CKA_MODULUS_BITS** attribute specified. This is because RSA private keys are only generated as part of an RSA key *pair*, and the **CKA_MODULUS_BITS** attribute for the pair is specified in the template for the RSA public key.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_DECRYPT, &true, sizeof(true)},
   {CKA_SIGN, &true, sizeof(true)},
   {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 8-6, and Table 8-14:

Table 8-16, DSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*

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.

Note that when generating a DSA private key, the DSA parameters are *not* specified in the key's template. This is because DSA private keys are only generated as part of a DSA key *pair*, and the DSA parameters for the pair are specified in the template for the DSA public key.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.6.3 ECDSA private key objects

ECDSA private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_ECDSA**) hold ECDSA private keys. The following table defines the ECDSA private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-14:

Table 8-17, ECDSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6,7]	Big integer	Private value *w*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "ECDSA parameters".

Note that when generating an ECDSA private key, the ECDSA parameters are *not* specified in the key's template. This is because ECDSA private keys are only generated as part of an ECDSA key *pair*, and the ECDSA parameters for the pair are specified in the template for the ECDSA public key.

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_  ECDSA;
CK_CHAR label[] = "A n ECDSA 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_ DERIVE , &true, sizeof(true)},
   {CKA_PRIME, prime, sizeof(prime)},
   {CKA_SUBPRIME, subprime, sizeof(subprime)},
   {CKA_BASE, base, sizeof(base)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.6.4 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 8-6, and Table 8-14:

Table 8-18, Diffie-Hellman Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p*
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*
CKA_VALUE_BITS[2,6]	CK_ULONG	Length in bits of private value *x*

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.

Note that when generating an Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in the key's template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-Hellman key *pair*, and the Diffie-Hellman parameters for the pair are specified in the template for the Diffie-Hellman public key.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
```



```
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

8.6.5 KEA private key objects

KEA private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_KEA**) hold KEA private keys. The following table defines the KEA private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-14:

Table 8-19, KEA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "KEA parameters".

Note that when generating a KEA private key, the KEA parameters are *not* specified in the key's template. This is because KEA private keys are only generated as part of a KEA key *pair*, and the KEA parameters for the pair are specified in the template for the KEA public key.

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_KEA;
CK_CHAR label[] = "A KEA 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,  sizeof(true) },
    {CKA_LABEL, label, sizeof(label)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
```



```
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

8.6.6 MAYFLY private key objects

MAYFLY private key objects (object class **CKO_PRIVATE_KEY,** key type **CKK_MAYFLY**) hold MAYFLY private keys. The following table defines the MAYFLY private key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-14:

Table 8-20, MAYFLY Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits)
CKA_BASE[1,4,6]	Big integer	Base *g* (512 to 1024 bits, in steps of 64 bits)
CKA_VALUE[1,4,6,7]	Big integer	Private value *w*

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "MAYFLY parameters".

Note that when generating a MAYFLY private key, the MAYFLY parameters are *not* specified in the key's template. This is because MAYFLY private keys are only generated as part of a MAYFLY key *pair*, and the MAYFLY parameters for the pair are specified in the template for the MAYFLY public key.

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_  MAYFLY ;
CK_CHAR label[] = "A  MAYFLY 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, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_ DERIVE , &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {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 the following types of secret key: generic, RC2, RC4, RC5, DES, DES2, DES3, CAST, CAST3, CAST5, IDEA, SKIPJACK, BATON, JUNIPER, and CDMF. The following table defines the attributes common to all secret keys, in addition to the common attributes listed in Table 8-1 and Table 8-6:

Table 8-21, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE[8]	CK_BBOOL	TRUE if object is sensitive (default FALSE)
CKA_ENCRYPT[8]	CK_BBOOL	TRUE if key supports encryption[9]
CKA_DECRYPT[8]	CK_BBOOL	TRUE if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	TRUE if key supports signatures (*i.e.*, authentication codes) where the signature is an appendix to the data[9]
CKA_VERIFY[8]	CK_BBOOL	TRUE if key supports verification (*i.e.*, of authentication codes) where the signature is an appendix to the data[9]
CKA_WRAP[8]	CK_BBOOL	TRUE if key supports wrapping[9]
CKA_UNWRAP[8]	CK_BBOOL	TRUE if key supports unwrapping[9]
CKA_EXTRACTABLE[8]	CK_BBOOL	TRUE if key is extractable[9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	TRUE if key has *always* had the CKA_SENSITIVE attribute set to TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	TRUE if key has *never* had the CKA_EXTRACTABLE attribute set to TRUE

After an object is created, the **CKA_SENSITIVE** attribute may only be set to TRUE. Similarly, after an object is created, the **CKA_EXTRACTABLE** attribute may only be set to FALSE.

If the **CKA_SENSITIVE** attribute is TRUE, or if the **CKA_EXTRACTABLE** attribute is false, then certain attributes of the private key cannot be revealed in plaintext outside the token. These attributes are specified for each type of private key in the attribute table in the section describing that type of key.

If the **CKA_EXTRACTABLE** attribute is false, then the key cannot be wrapped.

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. These 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 8-6, and Table 8-21:

Table 8-22, Generic Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (arbitrary length)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_DERIVE, &true, sizeof(true)},
   {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 8-6, and Table 8-21:

Table 8-23, RC2 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (1 to 128 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {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 8-6, and Table 8-21:

Table 8-24, RC4 Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (1 to 256 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.4 RC5 secret key objects

RC5 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_RC5**) hold RC5 keys. The following table defines the RC5 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-25, RC4 Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (0 to 255 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

The following is a sample template for creating an RC5 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC5;
CK_CHAR label[] = “An RC5 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
```



```
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.5 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 8-6, and Table 8-21:

Table 8-26, DES Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 8 bytes long)

DES keys must always have their parity bits properly set, as described in FIPS PUB 46-2. Attempting to create or unwrap a DES key with incorrect parity will return an error.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.6 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 8-6, and Table 8-21:

Table 8-27, DES2 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 16 bytes long)

DES2 keys must always have their parity bits properly set, as described in FIPS PUB 46-2 (*i.e.*, each of the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or unwrap a DES2 key with incorrect parity will return an error.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.7 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 8-6, and Table 8-21:

Table 8-28, DES3 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 24 bytes long)

DES3 keys must always have their parity bits properly set, as described in FIPS PUB 46-2 (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or unwrap a DES3 key with incorrect parity will return an error.

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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```


8.7.8 CAST secret key objects

CAST secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_CAST**) hold CAST keys. The following table defines the CAST secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-29, CAST Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (1 to 8 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST;
CK_CHAR label[] = "A CAST 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.9 CAST3 secret key objects

CAST3 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_CAST3**) hold CAST3 keys. The following table defines the CAST3 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-30, CAST3 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (1 to 8 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

The following is a sample template for creating an CAST3 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST3;
CK_CHAR label[] = "A CAST3 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.10 CAST5 secret key objects

CAST5 secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_CAST5**) hold CAST5 keys. The following table defines the CAST5 secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-31, CAST5 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (1 to 16 bytes)
CKA_VALUE_LEN [2,3,6]	CK_ULONG	Length in bytes of key value

The following is a sample template for creating an CAST5 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST5;
CK_CHAR label[] = "A CAST5 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.11 IDEA secret key objects

IDEA secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_IDEA**) hold IDEA keys. The following table defines the IDEA secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-32, IDEA Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE [1,4,6,7]	Byte array	Key value (always 16 bytes long)

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_IDEA;
CK_CHAR label[] = "An IDEA 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.12 CDMF secret key objects

CDMF secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_CDMF**) hold single-length CDMF keys. The following table defines the CDMF secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-33, CDMF Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 8 bytes long)

CDMF keys must always have their parity bits properly set in exactly the same fashion described for DES keys in FIPS PUB 46-2. Attempting to create or unwrap a CDMF key with incorrect parity will return an error.

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CDMF;
CK_CHAR label[] = "A CDMF 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, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.13 SKIPJACK secret key objects

SKIPJACK secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_SKIPJACK**) holds a single-length MEK or a TEK. The following table defines the SKIPJACK secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-34, SKIPJACK Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 10 bytes long)

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ SKIPJACK ;
CK_CHAR label[] = "A SKIPJACK MEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SKIPJACK;
CK_CHAR label[] = "A SKIPJACK   TEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.14 BATON secret key objects

BATON secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_BATON**) hold single-length BATON keys. The following table defines the BATON secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, and Table 8-21:

Table 8-35, BATON Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 20 bytes long)

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ BATON ;
CK_CHAR label[] = "A BATON MEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
```



```
   {CKA_VALUE, value, sizeof(value)}
};
```

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BATON;
CK_CHAR label[] = "A BATON TEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true,  sizeof(true) },
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

8.7.15 JUNIPER secret key objects

JUNIPER secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_JUNIPER**) hold single-length JUNIPER keys. The following table defines the JUNIPER secret key object attributes, in addition to the common attributes listed in Table 8-1, Table 8-6, Table 8-21:

Table 8-36, JUNIPER Secret Key Object

Attribute	Data type	Meaning
CKA_VALUE[1,4,6,7]	Byte array	Key value (always 20 bytes long)

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_CHAR label[] = "A JUNIPER   MEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_LABEL, label, sizeof(label)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VALUE, value, sizeof(value)}
};
```

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_CHAR label[] = "A JUNIPER TEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &class, sizeof(class)},
```



```
      {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
      {CKA_TOKEN, &true, sizeof(true)},
      {CKA_LABEL, label, sizeof(label)},
      {CKA_ENCRYPT, &true, sizeof(true)},
      {CKA_WRAP, &true, sizeof(true)},
      {CKA_VALUE, value, sizeof(value)}
   };
```


9. Functions

Cryptoki's functions are organized into the following categories:

- general-purpose functions (4 functions)
- slot and token management functions (8 functions)
- session management functions (8 functions)
- object management functions (9 functions)
- encryption functions (4 functions)
- decryption functions (4 functions)
- message digesting functions (5 functions)
- signing and MACing functions (6 functions)
- functions for verifying signatures and MACs (6 functions)
- dual-purpose cryptographic functions (4 functions)
- key management functions (5 functions)
- random number generation functions (2 functions)
- parallel function management functions (2 functions)

In addition to these 67 functions in the Cryptoki v2.0 API proper, Cryptoki makes use of application-supplied callback functions to notify an application of certain events.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function should have a "stub" in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED. If a Cryptoki API function is unsupported, its pointer in the library's **CK_FUNCTION_LIST** structure (as obtained by **C_GetFunctionList)** should be NULL_PTR (see Section 7.6).

9.1 Function return values

9.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

- CKR_GENERAL_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.

- CKR_HOST_MEMORY: The computer that the Cryptoki library is running on has insufficient memory to perform the requested function. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.

- CKR_FUNCTION_FAILED: The requested function could not be performed, but detailed information about why not is not available in this error return. If the failed function uses a session, it is possible that the **CK_SESSION_INFO** that can be obtained by calling **C_GetSessionInfo** holds useful information about what happened in its *ulDeviceError* field. In any event, although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when CKR_GENERAL_ERROR is returned.

- CKR_OK: The function executed successfully.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_GENERAL_ERROR or CKR_HOST_MEMORY would be an appropriate error return, then CKR_GENERAL_ERROR should be returned.

Because the above values can be returned by *any* Cryptoki function, they will never explicitly be mentioned as possible returns when the Cryptoki functions are described. All other return values are more specific to particular functions, and will be listed with all relevant functions.

9.1.2 Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, **C_GetTokenInfo**, **C_GetMechanismList**, **C_GetMechanismInfo**, **C_InitToken**, **C_OpenSession**, and **C_CloseAllSessions**) can return the following values:

- CKR_SESSION_HANDLE_INVALID: The specified session handle was invalid *at the time that the function was invoked*. Note that this can happen if the session's token is removed before the function invokation, since removing a token closes all sessions with it.

- CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.

- CKR_SESSION_CLOSED: The session was closed *during the execution of the function*.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_SESSION_HANDLE_INVALID or CKR_DEVICE_REMOVED would be an appropriate error return, then CKR_SESSION_HANDLE_INVALID should be returned.

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed before a function invokation and a token being removed during a function execution.

9.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a token (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, or **C_GetSlotInfo**) can return any of the following values:

- CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested function.
- CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot.
- CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was invoked*.
- CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_DEVICE_MEMORY or CKR_DEVICE_ERROR would be an appropriate error return, then CKR_DEVICE_MEMORY should be returned.

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed before a function invokation and a token being removed during a function execution.

9.1.4 All the other Cryptoki function return values

The other Cryptoki function returns follow. Except as mentioned in the descriptions of particular error codes, there are in general no particular priorities among the errors listed below, *i.e.*, if more than one error code might apply to a function's execution, the function may return any applicable error code.

- CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not be set, or which may not be modified.
- CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or unextractable.
- CKR_ATTRIBUTE_TYPE_INVALID: An invalid attribute type was specified in a template.
- CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for an attribute in a template.
- CKR_BUFFER_TOO_SMALL: The output of the function does not fit in the supplied buffer.
- CKR_CANCEL: This is a value for an application callback to return. When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a CKN_SURRENDER callback. If the callback returns the value CKR_CANCEL, then the function aborts (see CKR_FUNCTION_CANCELED).

- CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This error only applies to the **CKM_RSA_X_509** mechanism, when plaintext is supplied that has the same number of bytes as the RSA modulus and is numerically at least as large as the modulus. This return value has lower priority than CKR_DATA_LEN_RANGE.

- CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length. Depending on the operation's mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than CKR_DATA_INVALID.

- CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been determined to be invalid ciphertext. This return value has lower priority than CKR_ENCRYPTED_DATA_LEN_RANGE.

- CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.

- CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This can happen to a function executing in parallel with an application, if the application calls **C_CancelFunction**; it can also happen to a function executing in serial with an application, if the function makes a **CKN_SURRENDER** application callback, and the callback returns CKR_CANCEL (see CKR_CANCEL).

- CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the specified session.

- CKR_FUNCTION_NOT_SUPPORTED: The requested function is not supported by this Cryptoki library. Even unsupported functions in the Cryptoki API should have a "stub" in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED.

- CKR_FUNCTION_PARALLEL: There is currently a function executing in parallel in the specified session. CKR_FUNCTION_PARALLEL is also returned whenever a Cryptoki function call is made that executes in parallel.

- CKR_INFORMATION_SENSITIVE: The information requested could not be obtained because the token considers it sensitive, and is not able or willing to reveal it.

- CKR_INSERTION_CALLBACK_NOT_SUPPORTED: The specified slot does not support setting an application callback for token insertion.

- CKR_KEY_CHANGED: One of the keys specified in a **C_SetOperationState** operation is not the same key that was being used in the original saved session.

- CKR_KEY_FUNCTION_NOT_PERMITTED: An attempt has been made to use a key for a cryptographic purpose that the key's attributes are not set to allow it to do. For example, to use a key for performing encryption, that key must have its **CKA_ENCRYPT** attribute set to TRUE (the fact that the key must have a **CKA_ENCRYPT** attribute implies that the key cannot be a private key). This return value has lower priority than CKR_KEY_TYPE_INCONSISTENT.

- CKR_KEY_HANDLE_INVALID: The specified key handle is not valid. It may be the case that the specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a valid key handle.

- CKR_KEY_INDIGESTIBLE: The value of the specified key cannot be digested for some reason (perhaps the key isn't a secret key, or perhaps the token simply can't digest this kind of key).

- CKR_KEY_NEEDED: The **C_SetOperationState** operation cannot be carried out because it needs to be supplied with a key that was being used in the original saved session.

- CKR_KEY_NOT_NEEDED: An extraneous key was supplied to **C_SetOperationState**. For example, an attempt was made to restore a session that had been performing a message digesting operation, and an encryption key was supplied.

- CKR_KEY_NOT_WRAPPABLE: Although the specified private or secret key does not have its CKA_UNEXTRACTABLE attribute set to TRUE, Cryptoki (or the token) is unable to wrap the key as requested (possibly the token simply won't support it).

- CKR_KEY_SIZE_RANGE: Although the requested keyed cryptographic operation could in principal be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key 's size is outside the range of key sizes that it can handle.

- CKR_KEY_TYPE_INCONSISTENT: The specified key is not the correct type of key to use with the specified mechanism. This return value has a higher priority than CKR_KEY_FUNCTION_NOT_PERMITTED.

- CKR_KEY_UNEXTRACTABLE: The specified private or secret key can't be wrapped because its CKA_UNEXTRACTABLE attribute is set to TRUE.

- CKR_MECHANISM_INVALID: An invalid mechanism was specified to the cryptographic operation.

- CKR_MECHANISM_PARAM_INVALID: Invalid parameters were supplied to the mechanism specified to the cryptographic operation.

- CKR_OBJECT_HANDLE_INVALID: The specified object handle is not valid. We reiterate here that 0 is never a valid object handle.

- CKR_OPERATION_ACTIVE: There is already an active operation (or combination of active operations) which prevents Cryptoki from activating the specified operation. For example, an active object-searching operation would prevent Cryptoki from activating an encryption operation with **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent Cryptoki from activating a signature operation. Or, on a token which doesn't support dual cryptographic operations (see the description of the CKF_DUAL_CRYPTO_OPERATIONS flag in the **CK_TOKEN_INFO** structure), an active signature operation would prevent Cryptoki from activating an encryption operation.

- CKR_OPERATION_NOT_INITIALIZED: There is no active operation of an appropriate type in the specified session. For example, an application cannot call **C_Encrypt** in a session without having called **C_EncryptInit** first to activate an encryption operation.

- CKR_PIN_INCORRECT: The specified PIN is wrong, and does not match the PIN stored on the token. More generally, the attempt to authenticate the user has failed.

- CKR_PIN_INVALID: The specified PIN has invalid characters in it. This return code only applies to functions which attempt to set a PIN.

- CKR_PIN_LEN_RANGE: The specified PIN is too long or too short. This return code only applies to functions which attempt to set a PIN.

- CKR_RANDOM_NO_RNG: The specified token doesn't have a random number generator.

- CKR_RANDOM_SEED_NOT_SUPPORTED: The token's random number generator does not accept seeding from an application.

- CKR_SAVED_STATE_INVALID: The supplied saved cryptographic operations state is invalid, and so it cannot be restored to the specified session.

- CKR_SESSION_COUNT: The attempt to open a session failed, either because the token has too many sessions already open, or because the token has too many read/write sessions already open.

- CKR_SESSION_EXCLUSIVE_EXISTS: The attempt to open a session failed because there already exists an exclusive session.

- CKR_SESSION_EXISTS: A session with the token is already open.

- CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel sessions.

- CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action because it is a read-only session. This return value has higher priority than CKR_TOKEN_WRITE_PROTECTED.

- CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot be logged in.

- CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a read-only session cannot be opened.

- CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid, solely on the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.

- CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower priority than CKR_SIGNATURE_LEN_RANGE.

- CKR_SLOT_ID_INVALID: The specified slot ID is not valid.

- CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be saved for some reason (possibly the token is simply unable to save the current state). This return value has lower priority than CKR_FUNCTION_PARALLEL and CKR_OPERATION_NOT_INITIALIZED.

- CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes.

- CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting attributes.

- CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in the slot.

- CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.

- CKR_UNWRAPPING_KEY_HANDLE_INVALID: The key handle specified to be used to unwrap another key is not valid.

- CKR_UNWRAPPING_KEY_SIZE_RANGE: Although the requested unwrapping operation could in principal be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the range of key sizes that it can handle.

- CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: The type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.

- CKR_USER_ALREADY_LOGGED_IN: The session cannot be logged in, because it is already logged in.

- CKR_USER_NOT_LOGGED_IN: The desired action cannot be performed because the appropriate user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out unless it is logged in. Another example is that a private object cannot be created on a token unless the session attempting to create it is logged in as the normal user. A final example is that cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.

- CKR_USER_PIN_NOT_INITIALIZED: The normal user's PIN has not been initialized with **C_InitPIN**.

- CKR_USER_TYPE_INVALID: An invalid value was specified as a **CK_USER_TYPE**. Valid types are **CKU_SO** and **CKU_USER**.

- CKR_WRAPPED_KEY_INVALID: The wrapped key is not valid. This return value has lower priority than CKR_WRAPPED_KEY_LEN_RANGE.

- CKR_WRAPPED_KEY_LEN_RANGE: The provided wrapped key can be seen to be invalid, solely on the basis of its length. This return value has higher priority than CKR_WRAPPED_KEY_INVALID.

- CKR_WRAPPING_KEY_HANDLE_INVALID: The key handle specified to be used to wrap another key is not valid.

- CKR_WRAPPING_KEY_SIZE_RANGE: Although the requested wrapping operation could in principal be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range of key sizes that it can handle.

- CKR_WRAPPING_KEY_TYPE_INCONSISTENT: The type of the key specified to wrap another key is not consistent with the mechanism specified for wrapping.

9.1.5 More on relative priorities of Cryptoki errors

In general, error codes from Section 9.1.1 take precedence over error codes from Section 9.1.2, which take precedence over error codes from Section 9.1.3, which take precedence over error codes from Section 9.1.4. One minor implication of this is that functions that use a session handle never return the error code CKR_TOKEN_NOT_PRESENT (they return CKR_SESSION_HANDLE_INVALID instead). Other than these precedences, if more than one error code might apply to a Cryptoki call, any of the applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned.

9.2 Conventions for functions which return output in a variable-length buffer

A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism. The amount of output returned by these functions is returned in a variable-length application-supplied buffer. An example of a function of this sort is **C_Encrypt**, which takes some plaintext as an argument, and outputs a buffer full of ciphertext.

These functions have some common calling conventions, which we describe here. Two of the arguments to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the length of the output produced (say *pulBufLen*). There are two ways for an application to call such a function:

1. If *pBuf* is NULL_PTR, then all that the function does is return (in **pulBufLen*) a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may somewhat exceed the precise number of bytes needed, but should not exceed it by a large amount. CKR_OK is returned by the function.

2. If *pBuf* is not NULL_PTR, then **pulBufLen* must contain the size in bytes of the buffer pointed to by *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the function, then that cryptographic output is placed there, and CKR_OK is returned by the function. If the buffer is not large enough, then CKR_BUFFER_TOO_SMALL is returned. In either case, **pulBufLen* is set to hold the *exact* number of bytes needed to hold the cryptographic output produced from the input to the function.

All functions which use the above convention will explicitly say so.

Cryptographic functions which return output in a variable-length buffer should always return as much output as can be computed from what has been passed in to them thus far. As an example, consider a session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C_DecryptUpdate** function. The blocksize of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at least 8 bytes. Hence the call to **C_DecryptUpdate** should return 0 bytes of plaintext. If a single additional byte of ciphertext is subsequently supplied by **C_DecryptUpdate**, the call to **C_DecryptUpdate** should return 8 bytes of plaintext (one full DES block).

9.3 Disclaimer concerning sample code

For the remainder of Section 9, we enumerate the various functions defined in Cryptoki. Most functions will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will frequently be somewhat incomplete. In particular, sample code will generally ignore possible error returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

9.4 General-purpose functions

Cryptoki provides the following general-purpose functions. These functions do *not* run in parallel with the application.

♦ C_Initialize

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

C_Initialize initializes the Cryptoki library. **C_Initialize** should be the first Cryptoki call made by an application, except for calls to **C_GetFunctionList**. What this function actually does is implementation-dependent: for example, it may cause Cryptoki to initialize its internal memory buffers, or any other resources it requires; or it may perform no action. The *pReserved* parameter is reserved for future versions; for this version, it should be set to NULL_PTR.

If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize** should (eventually) be succeeded by a single call to **C_Finalize**.

Return values: none other than the "universal" return values.

Example: see **C_GetInfo**.

♦ C_Finalize

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

C_Finalize is called to indicate that an application is finished with the Cryptoki library. It should be the last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for this version, it should be set to NULL_PTR.

If several applications are using Cryptoki, each one should call **C_Finalize**. Every call to **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application makes calls to other Cryptoki functions.

Return values: none other than the "universal" return values.

Example: see **C_GetInfo**.

♦ 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: none other than the "universal" return values.

Example:

```
CK_INFO info;
CK_RV rv;

rv = C_Initialize(NULL_PTR);
assert(rv == CKR_OK);

rv = C_GetInfo(&info);
assert(rv == CKR_OK);
if(info.version.major == 2) {
   /* Do lots of interesting cryptographic things with the token */
   .
   .
   .
}

rv = C_Finalize(NULL_PTR);
assert(rv == CKR_OK);
```

♦ C_GetFunctionList

```
CK_RV CK_ENTRY C_GetFunctionList(
     CK_FUNCTION_LIST_PTR_PTR ppFunctionList
);
```

C_GetFunctionList obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList* points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which contains function pointers for all the Cryptoki API routines in the library. *The pointer obtained may point into memory which is owned by the Cryptoki library*, and which may or may not be writable. In any case, no attempt should be made to write to this memory.

C_GetFunctionList is the only Cryptoki function which an application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

Return values: none other than the "universal" return values.

Example:

```
CK_FUNCTION_LIST_PTR pFunctionList;
CK_C_Initialize pC_Initialize;
CK_RV rv;

/* It's OK to call C_GetFunctionList before calling C_Initialize */
rv = C_GetFunctionList(&pFunctionList);
assert(rv == CKR_OK);
pC_Initialize = pFunctionList -> C_Initialize;

/* Call the C_Initialize function in the library */
rv = (*pC_Initialize)(NULL_PTR);
```

9.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management. These functions do *not* run in parallel with the application.

♦ C_GetSlotList

```
CK_RV CK_ENTRY C_GetSlotList(
     CK_BBOOL tokenPresent,
     CK_SLOT_ID_PTR pSlotList,
     CK_ULONG_PTR pulCount
);
```

C_GetSlotList is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list obtained includes only those slots with a token present (TRUE), or all slots (FALSE); *pulCount* points to the location that receives the number of slots.

There are two ways for an application to call **C_GetSlotList**:

1. If *pSlotList* is NULL_PTR, then all that **C_GetSlotList** does is return (in **pulCount*) the number of slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on entry to **C_GetSlotList** has no meaning in this case, and the call returns the value CKR_OK.

2. If *pSlotList* is not NULL_PTR, then **pulCount* must contain the size (in terms of **CK_SLOT_ID** elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots, then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetSlotList** returns the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number of slots.

Because **C_GetSlotList** does not allocate any space of its own, an application will often call **C_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots with a token present, then the number of such slots can change between when the application asks for how many such slots there are, and when the application asks for the slots themselves). However, this is by no means required.

Return values: CKR_BUFFER_TOO_SMALL.

Example:

```
CK_ULONG ulSlotCount, ulSlotWithTokenCount;
CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;
CK_RV rv;

/* Get list of all slots */
rv = C_GetSlotList(FALSE, NULL_PTR, &ulSlotCount);
if (rv == CKR_OK) {
   pSlotList =
      (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
   rv = C_GetSlotList(FALSE, pSlotList, &ulSlotCount);
   if (rv == CKR_OK) {
      /* Now use that list of all slots */
      .
      .
      .
   }

   free(pSlotList);
}

/* Get list of all slots with a token present */
pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
ulSlotWithTokenCount = 0;
while (1) {
   rv = C_GetSlotList(
      TRUE, pSlotWithTokenList, ulSlotWithTokenCount);
   if (rv != CKR_BUFFER_TOO_SMALL)
      break;
   pSlotWithTokenList = realloc(
      pSlotWithTokenList,
      ulSlotWithTokenList*sizeof(CK_SLOT_ID));
}

if (rv == CKR_OK) {
   /* Now use that list of all slots with a token present */
   .
   .
   .
}

free(pSlotWithTokenList);
```

♦ 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_DEVICE_ERROR, CKR_SLOT_ID_INVALID.

Example: see **C_GetTokenInfo.**

♦ 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_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED.

Example:

```
    CK_ULONG ulCount;
    CK_SLOT_ID_PTR pSlotList;
    CK_SLOT_INFO slotInfo;
    CK_TOKEN_INFO tokenInfo;
    CK_RV rv;

    rv = C_GetSlotList(FALSE, NULL_PTR, &ulCount);
    if ((rv == CKR_OK) && (ulCount > 0)) {
       pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
       rv = C_GetSlotList(FALSE, pSlotList, &ulCount);
       assert(rv == CKR_OK);

       /* Get slot information for first slot */
       rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
       assert(rv == CKR_OK);

       /* Get token information for first slot */
       rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
       if (rv == CKR_TOKEN_NOT_PRESENT) {
          .
          .
          .
       }
       .
       .
       .
       free(pSlotList);
    }
```

♦ C_GetMechanismList

```
CK_RV CK_ENTRY C_GetMechanismList(
     CK_SLOT_ID slotID,
     CK_MECHANISM_TYPE_PTR pMechanism  List,
     CK_ULONG_PTR pulCount
);
```

C_GetMechanismList is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

There are two ways for an application to call **C_GetMechanismList**:

1. If *pMechanismList* is NULL_PTR, then all that **C_GetMechanismList** does is return (in **pulCount*) the number of mechanisms, without actually returning a list of mechanisms. The contents of **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the value CKR_OK.

2. If *pMechanismList* is not NULL_PTR, then **pulCount* must contain the size (in terms of **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large enough to hold the list of mechanisms, then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetMechanismList** returns the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number of mechanisms.

Because **C_GetMechanismList** does not allocate any space of its own, an application will often call **C_GetMechanismList** twice. However, this is by no means required.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED.

Example:

```
CK_SLOT_ID slotID;
CK_ULONG ulCount;
CK_MECHANISM_TYPE_PTR pMechanismList;
CK_RV rv;

.
.
.
rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);
if ((rv == CKR_OK) && (ulCount > 0)) {
   pMechanismList =
      (CK_MECHANISM_TYPE_PTR)
         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));
   rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);
   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; *pInfo* points to the location that receives the mechanism information.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_MECHANISM_INVALID, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED.

Example:

```
CK_SLOT_ID slotID;
CK_MECHANISM_INFO info;
CK_RV rv;

.
.
.
/* Get information about the CKM_MD2 mechanism for this token */
rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
if (rv == CKR_OK) {
   if (info.flags & CKF_DIGEST) {
      .
      .
      .
   }
}
```

♦ C_InitToken

```
CK_RV CK_ENTRY C_InitToken(
     CK_SLOT_ID slotID,
     CK_CHAR_PTR pPin,
     CK_ULONG ulPinLen,
     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; *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (must be padded with blank characters).

When a token is initialized, all objects that can be destroyed are destroyed (*i.e.*, all except for "indestructible" objects such as keys built into 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.

If the token has a "protected authentication path", as indicated by the CKR_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin* parameter to **C_InitToken** should be NULL_PTR. During the execution of **C_InitToken**, the SO's PIN will be entered through the protected authentication path.

If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_InitToken** can be used to initialize the token.

A token cannot be initialized if Cryptoki detects that an application has an open session with it; when a call to **C_InitToken** is made under such circumstances, the call fails with error CKR_SESSION_EXISTS. It may happen that some other application *does* have an open session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other applications using the token. If this is the case, then what happens as a result of the **C_InitToken** call is undefined.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INCORRECT, CKR_SESSION_EXISTS, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_TOKEN_WRITE_PROTECTED.

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_ULONG ulPinLen
);
```

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

C_InitPIN can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any other state fails with error CKR_USER_NOT_LOGGED_IN.

If the token has a "protected authentication path", as indicated by the CKR_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To initialize the normal user's PIN on a token with such a protected authentication path, the *pPin* parameter to **C_InitPIN** should be NULL_PTR. During the execution of **C_InitPIN**, the SO will enter the new PIN through the protected authentication path.

If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_InitPIN** can be used to initialize the normal user's token access.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY, CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

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_ULONG ulOldLen,
     CK_CHAR_PTR pNewPin,
     CK_ULONG ulNewLen
);
```

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

C_SetPIN can only be called in the "R/W SO Functions" state or "R/W User Functions" state. An attempt to call it from a session in any other state fails with error CKR_SESSION_READ_ONLY.

If the token has a "protected authentication path", as indicated by the CKR_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the

application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To modify the current user's PIN on a token with such a protected authentication path, the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be NULL_PTR. During the execution of **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication path. It is not specified how the PINpad should be used to enter *two* PINs; this varies.

If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_SetPIN** can be used to modify the current user's PIN.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INCORRECT, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

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) {
   .
   .
   .
}
```

9.6 Session management functions

Cryptoki provides the following functions for session management. These functions do *not* run in parallel with the application.

A typical application might perform the following series of steps to make use of a token:

1. Select a token.

2. Make one or more calls to **C_OpenSession** to obtain sessions with the token.

3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a shared login state, **C_Login** only needs to be called for one session.

4. Perform cryptographic operations using the sessions with the token.

5. Call **C_CloseSession** once for each session that the application has with the token.

An application should not normally call **C_CloseAllSessions** or **C_Logout** to close its sessions with a token. This is because these functions can affect the sessions "owned" by other applications (see the discussion in Section 0 for more information). Therefore, an application should call these functions only under exceptional circumstances, unless the application somehow "knows" that no other applications have sessions open with the token.

An application may have concurrent sessions with more than one token. It is also possible for a token to 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_NOTIFY Notify,
     CK_SESSION_HANDLE_PTR phSession
);
```

C_OpenSession has two distinct functions: it can set up an application callback so that an application will be notified when a token is inserted into a particular slot, or it can open a session between an application and a token in a particular slot. *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 (see Section 9.17); *phSession* points to the location that receives the handle for the new session.

To set up a token insertion callback (instead of actually opening a session), the **CKF_INSERTION_CALLBACK** bit in the *flags* parameter should be set. As a result of setting up this callback, when a token is inserted into the specified slot, the application-supplied callback *Notify* will be called with parameters `(0, CKN_TOKEN_INSERTION, pApplication)` . If a token is already present when **C_OpenSession** is called, then *Notify* will be called immediately (conceivably even before **C_OpenSession** returns).

When **C_OpenSession** is called to set up a token insertion callback, the return code is either CKR_INSERTION_CALLBACK_NOT_SUPPORTED (if the token doesn't support insertion callbacks) or CKR_OK (if the token does support insertion callbacks).

When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or more bit flags defined in the **CK_SESSION_INFO** data type. For example, if no bits are set in the *flags* parameter, then **C_OpenSession** attempts to open a shared, read-only session, with certain cryptographic functions being performed in parallel with the application. Any or all of the **CKF_EXCLUSIVE_SESSION**, **CKF_RW_SESSION**, and **CKF_SERIAL_SESSION** bits can be set in the *flags* parameter to modify the type of session requested.

If an exclusive session is requested (by setting the **CKF_EXCLUSIVE_SESSION** bit), but is not available (because there is already a session open), **C_OpenSession** returns CKR_SESSION_EXISTS. If a parallel session is requested (by not setting the **CKR_SERIAL_SESSION** bit), but is not supported on this token, then **C_OpenSession** returns CKR_PARALLEL_NOT_SUPPORTED. These two error returns have equal priorities.

In a parallel session, cryptographic functions may return control to the application before completing (the return value CKR_FUNCTION_PARALLEL indicates that this condition applies). The application may then call **C_GetFunctionStatus** to obtain an updated status of the function's execution, which will continue to be CKR_FUNCTION_PARALLEL until the function completes, and CKR_OK or some other return value 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 before it completes.

Note that even in a parallel session, there is no guarantee that a particular function will execute in parallel. Therefore, an application should always check cryptographic functions' return codes to see whether the function is running in parallel, or whether it ran in serial [and is already finished].

If an application calls another function (cryptographic or otherwise) before one that is executing in parallel in the same session 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 periodically surrender control to the application by calling *Notify* with a **CKN_SURRENDER** 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. A function in a parallel session will never surrender control back to the application via a **CKN_SURRENDER** application callback, even if that particular function is actually executing in serial with the application.

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". An attempt to open a session which does not succeed because there are too many existing sessions of some type should return CKR_SESSION_COUNT.

If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only sessions may be opened with it.

If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code CKR_SESSION_READ_WRITE_SO_EXISTS (see Section 5.5.8).

The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not wish to support callbacks, it should pass a value of NULL_PTR as the *Notify* parameter. See Section 9.17 for more information about application callbacks.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_INSERTION_CALLBACK_NOT_SUPPORTED, CKR_SESSION_COUNT, CKR_SESSION_EXISTS, CKR_SESSION_EXCLUSIVE_EXISTS, CKR_SESSION_PARALLEL_NOT_SUPPORTED, CKR_SESSION_READ_WRITE_SO_EXISTS, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_TOKEN_WRITE_PROTECTED.

Example: see **C_CloseSession**.

♦ 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, all session objects created by the session are destroyed automatically, even if the application has other sessions "using" the objects (see Sections 5.5.5-5.5.8 for more details). If a function is running in parallel with the session, it is canceled.

Depending on the token, when the last open session any application has with the token is closed, the token may be "ejected" from its reader (if this capability exists).

Despite the fact this **C_CloseSession** is supposed to close a session, the return value CKR_SESSION_CLOSED is an *error* return. It indicates the (probably somewhat unlikely) event that while this function call was executing, another call was made to **C_CloseSession** to close this particular session, and that call finished executing first. Such uses of sessions are a bad idea, and Cryptoki makes little promise of what will occur in general if an application indulges in this sort of behavior.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

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

.
.
.
application = 17;
MyNotify = &EncryptionSessionCallback;
rv = C_OpenSession(
   slotID, CKF_RW_SESSION,(CK_VOID_PTR) &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 an application has with a token. *slotID* specifies the token's slot.

Because an application may have access to sessions "owned" by another application (see Section 0), this function should only be called under special circumstances. In general, an application should close all its sessions one at a time with **C_CloseSession**, rather than calling **C_CloseAllSessions**.

Depending on the token, when the last open session any application has with the token is closed, the token may be "ejected" from its reader (if this capability exists).

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

Example:

```
CK_SLOT_ID slotID;
CK_RV rv;

.
.
.
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 a session. *hSession* is the session's handle; *pInfo* points to the location that receives the session information.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

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

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


♦ C_GetOperationState

```
CK_RV CK_ENTRY C_GetOperationState(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pOperationState,
     CK_ULONG_PTR pulOperationStateLen
);
```

C_GetOperationState obtains the cryptographic operations state of a session, encoded as a string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

Although the saved state output by **C_GetOperationState** is not really produced by a "cryptographic mechanism", **C_GetOperationState** nonetheless uses the convention described in Section 9.2 on producing output.

Precisely what the "cryptographic operations state" this function saves is varies from token to token; however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic activities of a session.

Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The application now wants to "save the state" of this digest operation, so that it can continue it later. In this particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken together, these three pieces of information suffice to continue the current hashing operation at a later time.

Consider next a session which is performing an encryption operation with DES (a block cipher with a block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_RC2_CBC** mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already been produced and output. In this case, the cryptographic operations state of the session most likely consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some administrative data indicating that this saved state comes from a session which was performing DES encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState** for more information on whether or not the key is present in the saved state).

If a session is performing two cryptographic operations simultaneously (see Section 9.13), then the cryptographic operations state of the session will contain all the necessary information to restore both operations.

A session which is in the middle of executing a Cryptoki function cannot have its cryptographic operations state saved. An attempt to do so returns the error CKR_FUNCTION_PARALLEL.

An attempt to save the cryptographic operations state of a session which does not currently have some active saveable cryptographic operation(s) (encryption, decryption, digesting, signing

without message recovery, verification without message recovery, or some legal combination of two of these) should fail with the error CKR_OPERATION_NOT_INITIALIZED.

An attempt to save the cryptographic operations state of a session which is performing an appropriate cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain necessary state information and/or key information can't leave the token, for example) should fail with the error CKR_STATE_UNSAVEABLE.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_STATE_UNSAVEABLE.

Example: see **C_SetOperationState**.

♦ C_SetOperationState

```
CK_RV CK_ENTRY C_SetOperationState(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pOperationState,
     CK_ULONG ulOperationStateLen,
     CK_OBJECT_HANDLE hEncryptionKey,
     CK_OBJECT_HANDLE hAuthenticationKey
);
```

C_SetOperationState restores the cryptographic operations state of a session from a string of bytes obtained with **C_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the location holding the saved state; *ulOperationStateLen* holds the length of the saved state; *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption operation in the restored session (or 0 if no encryption or decryption key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state).

The state need not have been obtained from the same session (the "source session") as it is being restored to (the "destination session"). However, the source session and destination session should have a common session state (*e.g.*, CKS_RW_USER_FUNCTIONS), and should be with a common token. There is also no guarantee that cryptographic operations state may be carried across logins, or across different Cryptoki implementations.

If **C_SetOperationState** is supplied with alleged saved cryptographic operations state which it can determine is not valid saved state (or is cryptographic operations state from a session with a different session state, or is cryptographic operations state from a different token), it fails with the error CKR_SAVED_STATE_INVALID.

Saved state obtained from calls to **C_GetOperationState** may or may not contain information about keys in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing encryption or decryption operation, and the key in use for the operation is not

saved in the state, then it must be supplied to **C_SetOperationState** in the *hEncryptionKey* argument. If it is not, then **C_SetOperationState** will fail and return the error CKR_KEY_NEEDED. If the key in use for the operation *is* saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not required.

Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification operation, and the key in use for the operation is not saved in the state, then it must be supplied to **C_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C_SetOperationState** will fail with the error CKR_KEY_NEEDED. If the key in use for the operation *is* saved in the state, then it *can* be supplied in the *hAuthenticationKey* argument, but this is not required.

If an *irrelevant* key is supplied to **C_SetOperationState** call (*e.g.*, a nonzero key handle is submitted in the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an ongoing encryption or decryption operation, then **C_SetOperationState** fails with the error CKR_KEY_NOT_NEEDED.

If a key is supplied as an argument to **C_SetOperationState**, and **C_SetOperationState** can somehow detect that this key was not the key being used in the source session for the supplied cryptographic operations state (it may be able to detect this if the key or a hash of the key is present in the saved state, for example), then **C_SetOperationState** fails with the error CKR_KEY_CHANGED.

An application can look at the CKF_RESTORE_KEY_NOT_NEEDED flag in the flags field of the **CK_TOKEN_INFO** field for a token to determine whether or not it needs to supply key handles to **C_SetOperationState** calls. If this flag is TRUE, then a call to **C_SetOperationState** *never* needs a key handle to be supplied to it. If this flag is FALSE, then at least some of the time, **C_SetOperationState** requires a key handle, and so the application should probably *always* pass in any relevant key handles when restoring cryptographic operations state to a session.

C_SetOperationState can successfully restore cryptographic operations state to a session even if that session has active cryptographic or object search operations when **C_SetOperationState** is called (the ongoing operations are abruptly cancelled).

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_MECHANISM digestMechanism;
CK_ULONG ulStateLen;
CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};
CK_BYTE data2[] = {0x02, 0x04, 0x08};
CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};
CK_BYTE pDigest[20];
CK_ULONG ulDigestLen;
CK_RV rv;

.
.
.
/* Initialize hash operation */
```



```
    rv = C_DigestInit(hSession, &digestMechanism);
    assert(rv == CKR_OK);

    /* Start hashing */
    rv = C_DigestUpdate(hSession, data1, sizeof(data1));
    assert(rv == CKR_OK);

    /* Find out how big the state might be */
    rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);
    assert(rv == CKR_OK);

    /* Allocate some memory and then get the state */
    pState = (CK_BYTE_PTR) malloc(ulStateLen);
    rv = C_GetOperationState(hSession, pState, &ulStateLen);

    /* Continue hashing */
    rv = C_DigestUpdate(hSession, data2, sizeof(data2));
    assert(rv == CKR_OK);

    /* Restore state.  No key handles needed */
    rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);
    assert(rv == CKR_OK);

    /* Continue hashing from where we saved state */
    rv = C_DigestUpdate(hSession, data3, sizeof(data3));
    assert(rv == CKR_OK);

    /* Conclude hashing operation */
    ulDigestLen = sizeof(pDigest);
    rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
    if (rv == CKR_OK) {
       /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
       .
       .
       .
    }
```

♦ C_Login

```
CK_RV CK_ENTRY C_Login(
     CK_SESSION_HANDLE hSession,
     CK_USER_TY PE userType,
     CK_CHAR_PTR pPin,
     CK_ULONG ulPinLen
);
```

C_Login logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to the user's PIN; *ulPinLen* is the length of the PIN.

Depending on the user type, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state.

If the token has a "protected authentication path", as indicated by the **CKR_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a

PIN on a PINpad on the token itself, or on the slot device. Or the user might not even use a PIN—authentication could be achieved by some fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin* parameter to **C_Login** should be NULL_PTR. When **C_Login** returns, whatever authentication method supported by the token will have been performed; a return value of CKR_OK means that the user was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was denied access.

If there are any active cryptographic or object finding operations in a session, and then **C_Login** is successfully executed, it may or may not be the case that those operations are still active. Therefore, before logging in, any active operations should be finished.

If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the SO into a session (see Section 5.5.8). An attempt to do this will result in the error code CKR_SESSION_READ_ONLY_EXISTS.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INCORRECT, CKR_SESSION_READ_ONLY_EXISTS, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TYPE_INVALID.

Example: see **C_Logout**.

♦ 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, if the call succeeds, each of the application's sessions will enter either the "R/W Public Session" state or the "R/O Public Session" state.

When **C_Logout** successfully executes, any of the application's handles to private objects become invalid (even if a user is later logged back into the token, those handles remain invalid). In addition, all private session objects are destroyed.

If there are any active cryptographic or object finding operations in a session, and then **C_Logout** is successfully executed, it may or may not be the case that those operations are still active. Therefore, before logging out, any active operations should be finished.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

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) {
   .
   .
   .
   rv == C_Logout(hSession);
   if (rv == CKR_OK) {
      .
      .
      .
   }
}
```

9.7 Object management functions

Cryptoki provides the following functions for managing objects. These functions do *not* run in parallel with the application. Additional functions provided specifically for managing key objects are described in Section 9.14.

♦ C_CreateObject

```
CK_RV CK_ENTRY C_CreateObject(
     CK_SESSION_HANDLE hSession,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_ULONG ulCount,
     CK_OBJECT_HANDLE_PTR phObject
);
```

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

If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to FALSE.

Only session object can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Return values: CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

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, sizeof(true)},
   {CKA_APPLICATION, application, sizeof(application)},
   {CKA_VALUE, dataValue, sizeof(dataValue)}
};
CK_ATTRIBUTE certificateTemplate[] = {
   {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
   {CKA_TOKEN, &true, sizeof(true)},
   {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, sizeof(true)},
   {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 an 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_ULONG ulCount,
     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; *ulCount* is the number of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of the object.

The template may specify new values for any attributes of the object that can ordinarily be modified (*e.g.*, in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from TRUE to FALSE, but not the other way around. If this change is made, the new key's **CKA_NEVER_EXTRACTABLE** attribute will have the value FALSE. Similarly, the template may specify that the new key's **CKA_SENSITIVE** attribute be TRUE; the new key will have the same value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). 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). If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Return values: CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OBJECT_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey, 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, sizeof(false)},
   {CKA_ID, id, sizeof(id)},
   {CKA_VALUE, keyValue, sizeof(keyValue)}
};
CK_ATTRIBUTE copyTemplate[] = {
   {CKA_TOKEN, &true, sizeof(true)}
```



```
    };
    CK_RV rv;

    .
    .
    .
    /* Create a DES secret key session object */
    rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
    if (rv == CKR_OK) {
       /* Create a copy which is a token object */
       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 unless the normal user is logged in.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OBJECT_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

Example: see **C_GetObjectSize**.

♦ C_GetObjectSize

```
CK_RV CK_ENTRY C_GetObjectSize(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE hObject,
     CK_ULONG_PTR pulSize
);
```

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

Cryptoki does not specify what the meaning of an object's size is. Intuitively, it is some measure of how much token memory the object takes up. If an application deletes (say) a private object of size S, it might be reasonable to assume that the *ulFreePrivateMemory* field of the token's **CK_TOKEN_INFO** structure increases by approximately S.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

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, sizeof(true)},
       {CKA_APPLICATION, application, sizeof(application)},
       {CKA_VALUE, value, sizeof(value)}
    };
    CK_ULONG ulSize;
    CK_RV rv;

    .
    .
    .
    rv = C_CreateObject(hSession, &template, 4, &hObject);
    if (rv == CKR_OK) {
       rv = C_GetObjectSize(hSession, hObject, &ulSize);
       if (rv != CKR_INFORMATION_SENSITIVE) {
          .
          .
          .
       }

       rv = 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_ULONG ulCount
);
```

C_GetAttributeValue obtains 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 obtained, and receives the attribute values; *ulCount* is the number of attributes in the template.

For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C_GetAttributeValue** performs the following algorithm:

1. If the specified attribute (*i.e.*, the attribute specified by the *type* field) for the object cannot be revealed because the object is sensitive or nonextractable, then the *ulValueLen* field in that triple is modified to hold the value -1 (*i.e.*, when it is cast to a CK_LONG, it holds -1).

2. Otherwise, if the specified attribute for the object is invalid (the object does not possess such an attribute), then the *ulValueLen* field in that triple is modified to hold the value -1.

3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold the exact length of the specified attribute for the object.

4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the *ulValueLen* field is modified to hold the exact length of the attribute.

5. Otherwise, the *ulValueLen* field is modified to hold the value -1.

If case 1 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes, then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the requested attributes will CKR_OK be returned.

Return values: CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OBJECT_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

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].ulValueLen);
   template[0].pValue = pModulus;
   /* template[0].ulValueLen was set by C_GetAttributeValue */

   pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
   template[1].pValue = pExponent;
   /* template[1].ulValueLen was set by C_GetAttributeValue */

   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_ULONG ulCount
);
```

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; *ulCount* is the number of attributes in the template.

The template specifies new values for any attributes of the object that can be modified. If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

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

Return values: CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OBJECT_HANDLE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED.

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_ULONG ulCount
);
```

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; *ulCount* 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 *ulCount* to 0.

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

The object search operation will only find objects that the session can view. For example, an object search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the search template specifies that the search is for private objects).

If a search operation is active, and objects are created or destroyed which fit the search template for the active search operation, then those objects may or may not be found by the search operation. Note that this means that, under these circumstances, the search operation may return invalid object handles.

Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can return CKR_OK and initialize a search operation which will match no objects.

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

Example: see **C_FindObjectsFinal**.

♦ C_FindObjects

```
CK_RV CK_ENTRY C_FindObjects(
     CK_SESSION_HANDLE hSession,
     CK_OBJECT_HANDLE_PTR phObject,
     CK_ULONG ulMaxObjectCount,
     CK_ULONG_PTR pulObjectCount
);
```

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; *ulMaxObjectCount* is the maximum number of

object handles to be returned; *pulObjectCount* 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 *pulObjectCount* points to receives the value 0.

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

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_FindObjectsFinal**.

♦ C_FindObjectsFinal

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

C_FindObjectsFinal terminates a search for token and session objects. *hSession* is the session's handle.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_ULONG ulObjectCount;
CK_RV rv;

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

rv = C_FindObjectsFinal(hSession);
assert(rv == CKR_OK);
```


9.8 Encryption functions

Cryptoki provides the following functions for encrypting data. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ 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; *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 can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application must call **C_EncryptInit** again.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

♦ C_Encrypt

```
CK_RV CK_ENTRY C_Encrypt(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_ULONG ulDataLen,
     CK_BYTE_PTR pE ncryptedData,
     CK_ULONG_PTR pulEncryptedDataLen
);
```

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

C_Encrypt uses the convention described in Section 9.2 on producing output.

The encryption operation must have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input data must consist of an integral number of blocks). If these constraints are not satisfied, then **C_Encrypt** will fail with return code CKR_DATA_LEN_RANGE.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to the same location.

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

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_EncryptFinal** for an example of similar functions.

♦ C_EncryptUpdate

```
CK_RV CK_ENTRY C_EncryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_ULONG ulPartLen,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG_PTR pulEncryptedPartLen
);
```

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

C_EncryptUpdate uses the convention described in Section 9.2 on producing output.

The encryption operation must have been initialized with **C_EncryptInit**. This function may be called any number of times in succession. A call to **C_EncryptUpdate** which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current encryption operation.

The encryption operation must have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to the same location.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_EncryptFinal.**

♦ C_EncryptFinal

```
CK_RV CK_ENTRY C_EncryptFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pLastEncryptedPart,
     CK_ULONG_PTR pulLastEncryptedPartLen
);
```

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; *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

C_EncryptFinal uses the convention described in Section 9.2 on producing output.

The encryption operation must have been initialized with **C_EncryptInit**. A call to **C_EncryptFinal** always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism's input data must consist of an integral number of blocks. If these constraints are not satisfied, then **C_EncryptFinal** will fail with return code CKR_DATA_LEN_RANGE.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define PLAINTEXT_BUF_SZ 200
#define CIPHERTEXT_BUF_SZ 256

CK_ULONG firstPieceLen, secondPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
   CKM_DES_CBC_PAD, iv, sizeof(iv)
};
```



```
CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulEncryptedData1Len;
CK_ULONG ulEncryptedData2Len;
CK_ULONG ulEncryptedData3Len;
CK_RV rv;

.
.
.
firstPieceLen = 90;
secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
rv = C_EncryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
   /* Encrypt first piece */
   ulEncryptedData1Len = sizeof(encryptedData);
   rv = C_EncryptUpdate(
      hSession,
      &data[0], firstPieceLen,
      &encryptedData[0], &ulEncryptedData1Len);
   if (rv != CKR_OK) {
      .
      .
      .
   }

   /* Encrypt second piece */
   ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
   rv = C_EncryptUpdate(
      hSession,
      &data[firstPieceLen], secondPieceLen,
      &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
   if (rv != CKR_OK) {
      .
      .
      .
   }

   /* Get last little encrypted bit */
   ulEncryptedData3Len =
      sizeof(encryptedData)
         -ulEncryptedData1Len-ulEncryptedData2Len;
   rv = C_EncryptFinal(
      hSession,
      &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
      &ulEncryptedData3Len);
   if (rv != CKR_OK) {
      .
      .
      .
   }
}
```


9.9 Decryption functions

Cryptoki provides the following functions for decrypting data. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ 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; *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 can either call **C_Decrypt** to decrypt data in a single part; or call **C_DecryptUpdate** zero or more times, followed by **C_DecryptFinal**, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to **C_Decrypt** or **C_DecryptFinal** *to actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the application must call **C_DecryptInit** again

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_DecryptFinal**.

♦ C_Decrypt

```
CK_RV CK_ENTRY C_Decrypt(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedData,
     CK_ULONG ulEncryptedDataLen,
     CK_BYTE_PTR pData,
     CK_ULONG_PTR pulDataLen
);
```

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

C_Decrypt uses the convention described in Section 9.2 on producing output.

The decryption operation must have been initialized with **C_DecryptInit**. A call to **C_Decrypt** always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to the same location.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

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

Return values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DecryptFinal** for an example of similar functions.

♦ C_DecryptUpdate

```
CK_RV CK_ENTRY C_DecryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG ulEncryptedPartLen,
     CK_BYTE_PTR pPart,
     CK_ULONG_PTR pulPartLen
);
```

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; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

C_DecryptUpdate uses the convention described in Section 9.2 on producing output.

The decryption operation must have been initialized with **C_DecryptInit**. This function may be called any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to the same location.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,

CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: See **C_DecryptFinal**.

♦ C_DecryptFinal

```
CK_RV CK_ENTRY C_DecryptFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pLastPart,
     CK_ULONG_PTR pulLastPartLen
);
```

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; *pulLastPartLen* points to the location that holds the length of the last recovered data part.

C_DecryptFinal uses the convention described in Section 9.2 on producing output.

The decryption operation must have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal** always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define CIPHERTEXT_BUF_SZ 256
#define PLAINTEXT_BUF_SZ 256

CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
   CKM_DES_CBC_PAD, iv, sizeof(iv)
};
CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulData1Len, ulData2Len, ulData3Len;
CK_RV rv;

.
.
```



```
    .
    firstEncryptedPieceLen = 90;
    secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ-firstEncryptedPieceLen;
    rv = C_DecryptInit(hSession, &mechanism, hKey);
    if (rv == CKR_OK) {
       /* Decrypt first piece */
       ulData1Len = sizeof(data);
       rv = C_DecryptUpdate(
          hSession,
          &encryptedData[0], firstEncryptedPieceLen,
          &data[0], &ulData1Len);
       if (rv != CKR_OK) {
          .
          .
          .
       }

       /* Decrypt second piece */
       ulData2Len = sizeof(data)-ulData1Len;
       rv = C_DecryptUpdate(
          hSession,
          &encryptedData[firstEncryptedPieceLen],
          secondEncryptedPieceLen,
          &data[ulData1Len], &ulData2Len);
       if (rv != CKR_OK) {
          .
          .
          .
       }

       /* Get last little decrypted bit */
       ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
       rv = C_DecryptFinal(
          hSession,
          &data[ulData1Len+ulData2Len], &ulData3Len);
       if (rv != CKR_OK) {
          .
          .
          .
       }
    }
```

9.10 Message digesting functions

Cryptoki provides the following functions for digesting data. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ 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; *pMechanism* points to the digesting mechanism.

After calling **C_DigestInit**, the application can either call **C_Digest** to digest data in a single part; or call **C_DigestUpdate** zero or more times, followed by **C_DigestFinal**, to digest data in multiple parts. The message-digesting operation is active until the application uses a call to **C_Digest** or **C_DigestFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application must call **C_DigestInit** again.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_DigestFinal**.

♦ C_Digest

```
CK_RV CK_ENTRY C_Digest(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_ULONG ulDataLen,
     CK_BYTE_PTR pDigest,
     CK_ULONG_PTR pulDigestLen
);
```

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

C_Digest uses the convention described in Section 9.2 on producing output.

The digest operation must have been initialized with **C_DigestInit**. A call to **C_Digest** always terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message digest.

The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the same location.

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

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DigestFinal** for an example of similar functions.

♦ C_DigestUpdate

```
CK_RV CK_ENTRY C_DigestUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_ULONG ulPartLen
);
```

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; *ulPartLen* is the length of the data part.

The message-digesting operation must have been initialized with **C_DigestInit**. Calls to this function and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate** which results in an error terminates the current digest operation.

Return values: CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DigestFinal**.

♦ C_DigestKey

```
CK_RV CK_ENTRY C_DigestKey(
     CK_SESSION_HANDLE hSe  ssion,
     CK_OBJECT_HANDLE hKey
);
```

C_DigestKey continues a multiple-part message-digesting operation by digesting the value of a secret key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

The message-digesting operation must have been initialized with **C_DigestInit**. Calls to this function and **C_DigestUpdate** may be interspersed any number of times in any order.

If the value of the supplied key cannot be digested purely for some reason related to its length, **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_INDIGESTIBLE, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DigestFinal**.

♦ C_DigestFinal

```
CK_RV CK_ENTRY C_DigestFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pDigest,
     CK_ULONG_PTR pulDigestLen
);
```

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; *pulDigestLen* points to the location that holds the length of the message digest.

C_DigestFinal uses the convention described in Section 9.2 on producing output.

The digest operation must have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message digest.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_MECHANISM mechanism = {
   CKM_MD5, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE digest[16];
CK_ULONG ulDigestLen;
CK_RV rv;

.
.
.
rv = C_DigestInit(hSession, &mechanism);
if (rv != CKR_OK) {
   .
   .
   .
}

rv = C_DigestUpdate(hSession, data, sizeof(data));
if (rv != CKR_OK) {
   .
   .
   .
}

rv = C_DigestKey(hSession, hKey);
if (rv != CKR_OK) {
   .
   .
```



```
    .
}

ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
.
.
.
```

9.11 Signing and MACing functions

Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations also encompass message authentication codes). All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ 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; *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 can either call **C_Sign** to sign in a single part; or call **C_SignUpdate** one or more times, followed by **C_SignFinal,** to sign data in multiple parts. The signature operation is active until the application uses a call to **C_Sign** or **C_SignFinal** *to actually obtain* the signature. To process additional data (in single or multiple parts), the application must call **C_SignInit** again.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_SignFinal**.

♦ C_Sign

```
CK_RV CK_ENTRY C_Sign(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_ULONG ulDataLen,
     CK_BYTE_PTR pSignature,
     CK_ULONG_PTR pulSignatureLen
);
```

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; *ulDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

C_Sign uses the convention described in Section 9.2 on producing output.

The signing operation must have been initialized with **C_SignInit**. A call to **C_Sign** always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

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

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_SignFinal** for an example of similar functions.

♦ C_SignUpdate

```
CK_RV CK_ENTRY C_SignUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_ULONG ulPartLen
);
```

C_SignUpdate continues a multiple-part signature operation, processing another data part. *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* 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. A call to **C_SignUpdate** which results in an error terminates the current signature operation.

Return values: CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL,

CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_SignFinal**.

♦ C_SignFinal

```
CK_RV CK_ENTRY C_SignFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_ULONG_PTR pulSignatureLen
);
```

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; *pulSignatureLen* points to the location that holds the length of the signature.

C_SignFinal uses the convention described in Section 9.2 on producing output.

The signing operation must have been initialized with **C_SignInit**. A call to **C_SignFinal** always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

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_ULONG ulMacLen;
CK_RV rv;

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



```
        .
    }
```

♦ 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; *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 uses a call to **C_SignRecover** *to actually obtain* the signature. To process additional data in a single part, the application must call **C_SignRecoverInit** again.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_ACTIVE, CKR_USER_NOT_LOGGED_IN.

Example: see **C_SignRecover**.

♦ C_SignRecover

```
CK_RV CK_ENTRY C_SignRecover(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_ULONG ulDataLen,
     CK_BYTE_PTR pSignature,
     CK_ULONG_PTR pulSignatureLen
);
```

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; *uLDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

C_SignRecover uses the convention described in Section 9.2 on producing output.

The signing operation must have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover** always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_ACTIVE.

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_ULONG ulSignatureLen;
CK_RV rv;

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


9.12 Functions for verifying signatures and MACs

Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki, these operations also encompass message authentication codes). All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ 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; *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 can either call **C_Verify** to verify a signature on data in a single part; or call **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.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_VerifyFinal**.

♦ C_Verify

```
CK_RV CK_ENTRY C_Verify(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pData,
     CK_ULONG ulDataLen,
     CK_BYTE_PTR pSignature,
     CK_ULONG ulSignatureLen
);
```


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; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

The verification operation must have been initialized with **C_VerifyInit**. A call to **C_Verify** always terminates the active verification operation.

A successful call to **C_Verify** should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active signing operation is terminated.

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

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

Example: see **C_VerifyFinal** for an example of similar functions.

♦ C_VerifyUpdate

```
CK_RV CK_ENTRY C_VerifyUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPa rt,
     CK_ULONG ulPartLen
);
```

C_VerifyUpdate continues a multiple-part verification operation, processing another data part. *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* 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. A call to **C_VerifyUpdate** which results in an error terminates the current verification operation.

Return values: CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_VerifyFinal**.

♦ C_VerifyFinal

```
CK_RV CK_ENTRY C_VerifyFinal(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_ULONG ulSignatureLen
);
```

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

The verification operation must have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always terminates the active verification operation.

A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying operation is terminated.

Return values: CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE.

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; *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 uses a call to **C_VerifyRecover** *to actually obtain* the recovered message.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_VerifyRecover**.

♦ C_VerifyRecover

```
CK_RV CK_ENTRY C_VerifyRecover(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSignature,
     CK_ULONG ulSignatureLen,
     CK_BYTE_PTR pData,
     CK_ULONG_PTR pulDataLen
);
```

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; *ulSignatureLen* is the length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen* points to the location that holds the length of the recovered data.

C_VerifyRecover uses the convention described in Section 9.2 on producing output.

The verification operation must have been initialized with **C_VerifyRecoverInit**. A call to **C_VerifyRecover** always terminates the active verification operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the recovered data.

A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never return CKR_BUFFER_TOO_SMALL.

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

Example:

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

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

9.13 Dual-function cryptographic functions

Cryptoki provides the following functions to perform two cryptographic operations "simultaneously" within a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and from a token. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ C_DigestEncryptUpdate

```
CK_RV CK_ENTRY C_DigestEncryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_ULONG ulPartLen,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG_PTR pulEncryptedPartLen
);
```

C_DigestEncryptUpdate continues multiple-part digest and encryption operations, processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part; *pulEncryptedPart* points to the location that holds the length of the encrypted data part.

C_DigestEncryptUpdate uses the convention described in Section 9.2 on producing output.

Digest and encryption operations must both be active (they must have been initialized with **C_DigestInit** and **C_EncryptInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to **C_DigestKey**, however).

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM digestMechanism = {
   CKM_MD5, NULL_PTR, 0
};
CK_MECHANISM encryptionMechanism = {
   CKM_DES_ECB, iv, sizeof(iv)
};
CK_BYTE encryptedData[BUF_SZ];
CK_ULONG ulEncryptedDataLen;
CK_BYTE digest[16];
CK_ULONG ulDigestLen;
CK_BYTE data[(2*BUF_SZ)+8];
CK_RV rv;
int i;

.
.
.
memset(iv, 0, sizeof(iv));
memset(data, 'A', ((2*BUF_SZ)+5));
```



```
rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
if (rv != CKR_OK) {
   .
   .
   .
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK) {
   .
   .
   .
}

ulEncryptedDataLen = sizeof(encryptedData);
rv = C_DigestEncryptUpdate(
   hSession,
   &data[0], BUF_SZ,
   encryptedData, &ulEncryptedDataLen);
.
.
.
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_DigestEncryptUpdate(
   hSession,
   &data[BUF_SZ], BUF_SZ,
   encryptedData, &ulEncryptedDataLen);
.
.
.

/*
 * The last portion of the buffer needs to be handled with
 * separate calls to deal with padding issues in ECB mode
 */

/* First, complete the digest on the buffer */
rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
.
.
.
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
.
.
.

/* Then, pad last part with 3 0x00 bytes, and complete encryption */
for(i=0;i<3;i++)
   data[((BUF_SZ*2)+5)+i] = 0x00;

/* Now, get second-to-last piece of ciphertext */
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_EncryptUpdate(
   hSession,
   &data[BUF_SZ*2], 8,
   encryptedData, &ulEncryptedDataLen);
.
.
.

/* Get last piece of ciphertext (should have length 0, here) */
```



```
    ulEncryptedDataLen = sizeof(encryptedData);
    rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
    .
    .
    .
```

♦ C_DecryptDigestUpdate

```
CK_RV CK_ENTRY C_DecryptDigestUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG ulEncryptedPartLen,
     CK_BYTE_PTR pPart,
     CK_ULONG_PTR pulPartLen
);
```

C_DecryptDigestUpdate continues a multiple-part combined decryption and digest operation, processing another data part. *hSession* is the session's handle; *pEncryptedData* points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the location that receives the recovered data; *pulDataLen* points to the location that holds the length of the recovered data.

C_DecryptDigestUpdate uses the convention described in Section 9.2 on producing output.

Decryption and digesting operations must both be active (they must have been initialized with **C_DecryptInit** and **C_DigestInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to **C_DigestKey**, however).

Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using **C_DigestEncryptUpdate**, the "inverse function" of **C_DecryptDigestUpdate**. This is because when **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the input passed to the active digesting operation is the *output of* the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and digest the original plaintext thereby obtained.

After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2 bytes of plaintext are *not* passed on to be digested.

A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called, the last 2 bytes of plaintext get passed into the active digesting operation.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting operation. *Extreme caution is warranted when using a padded decryption mechanism with **C_DecryptDigestUpdate**.*

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM decryptionMechanism = {
   CKM_DES_ECB, iv, sizeof(iv)
};
CK_MECHANISM digestMechanism = {
   CKM_MD5, NULL_PTR, 0
};
CK_BYTE encryptedData[(2*BUF_SZ)+8];
CK_BYTE digest[16];
CK_ULONG ulDigestLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;

.
.
.
memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
if (rv != CKR_OK) {
   .
   .
   .
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK){
   .
   .
   .
}

ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
   hSession,
   &encryptedData[0], BUF_SZ,
   data, &ulDataLen);
```



```
.
.
.
ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
   hSession,
   &encryptedData[BUF_SZ], BUF_SZ,
   data, &uldataLen);
.
.
.

/*
 * The last portion of the buffer needs to be handled with
 * separate calls to deal with padding issues in ECB mode
 */

/* First, complete the decryption of the buffer */
ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate(
   hSession,
   &encryptedData[BUF_SZ*2], 8,
   data, &ulLastUpdateSize);
.
.
.
/* Get last piece of plaintext (should have length 0, here) */
ulDataLen = sizeof(data)-ulLastUpdateSize;
rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
if (rv != CKR_OK) {
   .
   .
   .
}

/* Digest last bit of plaintext */
rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
if (rv != CKR_OK) {
   .
   .
   .
}
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
if (rv != CKR_OK) {
   .
   .
   .
}
```


♦ C_SignEncryptUpdate

```
CK_RV CK_ENTRY C_SignEncryptUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pPart,
     CK_ULONG ulPartLen,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG_PTR pulEncryptedPartLen
);
```

C_SignEncryptUpdate continues a multiple-part combined signature and 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 digested and encrypted data part; and *pusEncryptedPart* points to the location that holds the length of the encrypted data part.

C_SignEncryptUpdate uses the convention described in Section 9.2 on producing output.

Signature and encryption operations must both be active (they must have been initialized with **C_SigntInit** and **C_EncryptInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM signMechanism = {
   CKM_DES_MAC, NULL_PTR, 0
};
CK_MECHANISM encryptionMechanism = {
   CKM_DES_ECB, iv, sizeof(iv)
};
CK_BYTE encryptedData[BUF_SZ];
CK_ULONG ulEncryptedDataLen;
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[(2*BUF_SZ)+8];
CK_RV rv;
int i;

.
.
.
memset(iv, 0, sizeof(iv));
memset(data, 'A', ((2*BUF_SZ)+5));
rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
if (rv != CKR_OK) {
```



```
      .
      .
      .
   }
   rv = C_SignInit(hSession, &signMechanism, hMacKey);
   if (rv != CKR_OK) {
      .
      .
      .
   }

   ulEncryptedDataLen = sizeof(encryptedData);
   rv = C_SignEncryptUpdate(
      hSession,
      &data[0], BUF_SZ,
      encryptedData, &ulEncryptedDataLen);
   .
   .
   .
   ulEncryptedDataLen = sizeof(encryptedData);
   rv = C_SignEncryptUpdate(
      hSession,
      &data[BUF_SZ], BUF_SZ,
      encryptedData, &ulEncryptedDataLen);
   .
   .
   .

   /*
    * The last portion of the buffer needs to be handled with
    * separate calls to deal with padding issues in ECB mode
    */

   /* First, complete the signature on the buffer */
   rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
   .
   .
   .
   ulMacLen = sizeof(MAC);
   rv = C_DigestFinal(hSession, MAC, &ulMacLen);
   .
   .
   .

   /* Then pad last part with 3 0x00 bytes, and complete encryption */
   for(i=0;i<3;i++)
      data[((BUF_SZ*2)+5)+i] = 0x00;

   /* Now, get second-to-last piece of ciphertext */
   ulEncryptedDataLen = sizeof(encryptedData);
   rv = C_EncryptUpdate(
      hSession,
      &data[BUF_SZ*2], 8,
      encryptedData, &ulEncryptedDataLen);
   .
   .
   .

   /* Get last piece of ciphertext (should have length 0, here) */
   ulEncryptedDataLen = sizeof(encryptedData);
   rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
```


.
.
.

♦ C_DecryptVerifyUpdate

```
CK_RV CK_ENTRY C_DecryptVerifyUpdate(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pEncryptedPart,
     CK_ULONG ulEncryptedPartLen,
     CK_BYTE_PTR pPart,
     CK_ULONG_PTR pulPartLen
);
```

C_DecryptVerifyUpdate continues a multiple-part combined decryption and verification operation, processing another data part. *hSession* is the session's handle; *pEncryptedData* points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the location that receives the recovered data; and *pulDataLen* points to the location that holds the length of the recovered data.

C_DecryptVerifyUpdate uses the convention described in Section 9.2 on producing output.

Decryption and signature operations must both be active (they must have been initialized with **C_DecryptInit** and **C_VerifyInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using **C_SignEncryptUpdate**, the "inverse function" of **C_DecryptVerifyUpdate**. This is because when **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed to the active verifying operation is the *output of* the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and verify a signature on the original plaintext thereby obtained.

After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2 bytes of plaintext are *not* passed on to the verification mechanism.

A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature on *the first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before

C_VerifyFinal is called, the last 2 bytes of plaintext get passed into the active verification operation.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active verification operation. *Extreme caution is warranted when using a padded decryption mechanism with **C_DecryptVerifyUpdate**.*

Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM decryptionMechanism = {
   CKM_DES_ECB, iv, sizeof(iv)
};
CK_MECHANISM verifyMechanism = {
   CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE encryptedData[(2*BUF_SZ)+8];
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;

.
.
.
memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
if (rv != CKR_OK) {
   .
   .
   .
}
rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
if (rv != CKR_OK){
   .
   .
   .
}

ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
   hSession,
   &encryptedData[0], BUF_SZ,
   data, &ulDataLen);
.
```



```
.
.
ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
   hSession,
   &encryptedData[BUF_SZ], BUF_SZ,
   data, &uldataLen);
.
.
.

/*
 * The last portion of the buffer needs to be handled with
 * separate calls to deal with padding issues in ECB mode
 */

/* First, complete the decryption of the buffer */
ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate(
   hSession,
   &encryptedData[BUF_SZ*2], 8,
   data, &ulLastUpdateSize);
.
.
.
/* Get last little piece of plaintext.  Should have length 0 */
ulDataLen = sizeof(data)-ulLastUpdateSize;
rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
if (rv != CKR_OK) {
   .
   .
   .
}

/* Send last bit of plaintext to verification operation */
rv = C_VerifyUpdate(hSession, &data[BUF_SZ*2], 5);
if (rv != CKR_OK) {
   .
   .
   .
}
rv = C_VerifyFinal(hSession, MAC, ulMacLen);
if (rv == CKR_SIGNATURE_INVALID) {
   .
   .
   .
}
```

9.14 Key management functions

Cryptoki provides the following functions for key management. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ C_GenerateKey

```
CK_RV CK_ENTRY C_GenerateKey(
     CK_SESSION_HANDLE hSession,
     CK_MECHANISM_PTR pMechanism,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_ULONG ulCount,
     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; *ulCount* is the number of attributes in the template; *phKey* points to the location that receives the handle of the new key.

Since the type of key to be generated is implicit in the key generation mechanism, the template does not need to supply a key type. If it does supply a key type which is inconsistent with the key generation mechanism, **C_GenerateKey** fails and returns the error code CKR_TEMPLATE_INCONSISTENT.

The key object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to TRUE.

Return values: CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

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_ULONG ulPublicKeyAttributeCount,
     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
     CK_ULONG ulPrivateKeyAttributeCount,
     CK_OBJECT_HANDLE_PTR phPublicKey,
     CK_OBJECT_HANDLE_PTR phPrivateKey
);
```

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

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code CKR_TEMPLATE_INCONSISTENT.

The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL** attributes set to TRUE.

Return values: CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
   CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_ULONG modulusBits = 768;
CK_BYTE publicExponent[] = { 3 };
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
```



```
      {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
   };
   CK_ATTRIBUTE privateKeyTemplate[] = {
      {CKA_TOKEN, &true, sizeof(true)},
      {CKA_PRIVATE, &true, sizeof(true)},
      {CKA_SUBJECT, subject, sizeof(subject)},
      {CKA_ID, id, sizeof(id)},
      {CKA_SENSITIVE, &true, sizeof(true)},
      {CKA_DECRYPT, &true, sizeof(true)},
      {CKA_SIGN, &true, sizeof(true)},
      {CKA_UNWRAP, &true, sizeof(true)}
   };
   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_ULONG_PTR pulWrappedKeyLen
);
```

C_WrapKey wraps (*i.e.*, encrypts) a private or secret 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 *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

C_WrapKey uses the convention described in Section 9.2 on producing output.

The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping, must be TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped must also be TRUE.

If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its **CKA_EXTRACTABLE** attribute set to TRUE, then **C_WrapKey** fails with error code CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism solely because of its length, then **C_WrapKey** fails with error code CKR_KEY_SIZE_RANGE.

C_WrapKey can a priori be used in the following situations:

- To wrap any secret key with an RSA public key.
- To wrap any secret key with any other secret key which is not a SKIPJACK, BATON, or JUNIPER key.
- To wrap a SKIPJACK, BATON, or JUNIPER key with another SKIPJACK, BATON, or JUNIPER key (the two keys need not be the same type of key).
- To wrap an RSA, Diffie-Hellman, or DSA private key with any secret key which is not a SKIPJACK, BATON, or JUNIPER key.
- To wrap a KEA or DSA private key with a SKIPJACK key.

Of course, tokens vary in which specified

Return Values: CKR_BUFFER_TOO_SMALL, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_HANDLE_INVALID, CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE, CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hWrappingKey, hKey;
CK_MECHANISM mechanism = {
   CKM_DES3_ECB, NULL_PTR, 0
};
CK_BYTE wrappedKey[8];
CK_ULONG ulWrappedKeyLen;
CK_RV rv;

.
.
.
ulWrappedKeyLen = sizeof(wrappedKey);
rv = C_WrapKey(
   hSession, &mechanism,
   hWrappingKey, hKey,
   wrappedKey, &ulWrappedKeyLen);
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_ULONG ulWrappedKeyLen,
     CK_ATTRIBUTE_PTR pTemplate,
     CK_ULONG ulAttributeCount,
     CK_OBJECT_HANDLE_PTR phKey
);
```

C_UnwrapKey unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or secret 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; *ulWrappedKeyLen* is the length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number of attributes in the template; *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 new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, and the **CKA_EXTRACTABLE** attribute set to TRUE. If the template for the new key has the **CKA_EXTRACTABLE** attribute set to FALSE, **C_UnwrapKey** fails with the error CKR_TEMPLATE_INCONSISTENT.

When **C_UnwrapKey** is used to unwrap a key with the **CKM_KEY_WRAP_SET_OAEP** mechanism (see Section 10.32.1), additional "extra data" is decrypted at the same time that the key is unwrapped. The return of this data follows the convention in Section 9.2 on producing output. If the extra data is not returned from a call to **C_UnwrapKey** (either because the call was only to find out how large the extra data is, or because the buffer provided for the extra data was too small), then **C_UnwrapKey** will not create a new key, either.

The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to FALSE.

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

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, sizeof(true)},
      {CKA_DECRYPT, &true, sizeof(true)}
   };
   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_ULONG ulAttributeCount,
     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; *ulAttributeCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

The values of the **CK_SENSITIVE**, **CK_ALWAYS_SENSITIVE**, **CK_EXTRACTABLE**, and **CK_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold for the newly-derived key. See the description of each particular key-derivation mechanism in Section 10 for any constraints of this type.

The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to FALSE.

Return values: CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALUE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_HANDLE_INVALID, CKR_KEY_TYPE_INCONSISTENT, CKR_KEY_SIZE_RANGE,

CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

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, sizeof(true)}
};
CK_ATTRIBUTE template[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_DECRYPT, &true, sizeof(true)}
};
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) {
      /* Put other guy's public value in otherPublicValue */
      .
      .
      .
      rv = C_DeriveKey(
         hSession, &mechanism,
            hPrivateKey, template, 4, &hKey);
      if (rv == CKR_OK) {
```



```
            .
            .
            .
        }
    }
}
```

9.15 Random number generation functions

Cryptoki provides the following functions for generating random numbers. All these functions may run in parallel with the application if the session was opened with the **CKF_SERIAL_SESSION** flag set to FALSE (check the return code of the function call to see if the function is running in parallel).

♦ C_SeedRandom

```
CK_RV CK_ENTRY C_SeedRandom(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pSeed,
     CK_ULONG ulSeedLen
);
```

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 *ulSeedLen* is the length in bytes of the seed material.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_ACTIVE, CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

The return code CKR_RANDOM_NO_RNG has a higher priority than the return code CKR_RANDOM_SEED_NOT_SUPPORTED. That is, if the token doesn't have a random number generator, then **C_SeedRandom** will return the value CKR_RANDOM_NO_RNG.

Example: see **C_GenerateRandom**.

♦ C_GenerateRandom

```
CK_RV CK_ENTRY C_GenerateRandom(
     CK_SESSION_HANDLE hSession,
     CK_BYTE_PTR pRandomData,
     CK_ULONG ulRandomLen
);
```

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

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example:

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

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

9.16 Parallel function management functions

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

If there is currently a function running in parallel in the specified session, **C_GetFunctionStatus** returns CK_FUNCTION_PARALLEL. If the most recently-executed Cryptoki function other than **C_GetFunctionStatus** that was called in the specified session was not executed in parallel (or if *no* Cryptoki function other than **C_GetFunctionState** has been called in the specified session), then **C_GetFunctionStatus** returns CK_FUNCTION_NOT_PARALLEL. Otherwise, **C_GetFunctionState** returns the return value of whatever the last parallel function executed in the specified session was.

Typically, an application might call this function repeatedly when a function is executing in parallel. Eventually, once the function has finished its execution, the return value of **C_GetFunctionStatus** will no longer be CKR_FUNCTION_PARALLEL; instead, it will be the

return code of the function. Because of the way **C_GetFunctionState**'s behavior is defined above, repeated calls to **C_GetFunctionStatus** will all yield the same return code of the function (until some other Cryptoki function is called in the specified session).

Note that the application will also receive a **CKN_COMPLETE** notification callback when the function completes its parallel execution, assuming that the session the function is running in was opened with callbacks.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_NOT_PARALLEL, CKR_FUNCTION_PARALLEL, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

In addition to the return values listed above, once the function executing in parallel is finished executing, calls to **C_GetFunctionStatus** will return whatever the error return of the parallel function was.

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.

Note that **C_CancelFunction** cannot be used to cancel a function which is not running in parallel. For example, consider an application which consists of two threads, one of which is executing a (slow) **C_GenerateKeyPair** in session 1, which is a serial session. If the other thread attempts to cancel the **C_GenerateKeyPair** call with **C_CancelFunction**, the **C_CancelFunction** call may block until the **C_GenerateKeyPair** call is done, and then return the value CKR_FUNCTION_NOT_PARALLEL.

Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_NOT_PARALLEL, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
   CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_ULONG modulusBits = 768;
CK_BYTE publicExponent[] = {...};
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
```



```
   {CKA_WRAP, &true, sizeof(true)},
   {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
   {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
   {CKA_TOKEN, &true, sizeof(true)},
   {CKA_PRIVATE, &true, sizeof(true)},
   {CKA_SUBJECT, subject, sizeof(subject)},
   {CKA_ID, id, sizeof(id)},
   {CKA_SENSITIVE, &true, sizeof(true)},
   {CKA_DECRYPT, &true, sizeof(true)},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_UNWRAP, &true, sizeof(true)}
};
CK_RV rv;

.
.
.
rv = C_GenerateKeyPair(
   hSession, &mechanism,
   publicKeyTemplate, 5,
   privateKeyTemplate, 8,
   &hPublicKey, &hPrivateKey);
while (rv == CKR_FUNCTION_PARALLEL) {
   /* Check if user wants to cancel function */
   if (kbhit()) {
      if (getch() == 27) { /* If user hit ESCape key */
         rv = C_CancelFunction(hSession);
         .
         .
         .
      }
   }

   /* Perform other tasks or delay */
   .
   .
   .
   rv = C_GetFunctionStatus(hSession);
}
```

9.17 Callback functions

Cryptoki uses function pointers of type **CK_NOTIFY** to notify the application of certain events. There are four different types of application callbacks.

9.17.1 Token insertion callbacks

An application can use **C_OpenSession** to set up a token insertion callback function (assuming insertion callbacks are supported for that slot). When a token is inserted into the specified slot, the application callback function that was supplied to **C_OpenSession** is called with the arguments `(0, CKN_TOKEN_INSERTION, pApplication)` , where `pApplication` was supplied to **C_OpenSession**. Token insertion callbacks should return the value CKR_OK.

9.17.2 Token removal callbacks

When a token is removed from its slot, each open session which had a callback function specified when it was opened receives a callback. Each session's callback is called with the arguments `(hSession, CKN_DEVICE_REMOVED, pApplication)` , where `hSession` is the session's handle (although when the callback occurs, the session has just been closed because of the token removal) and `pApplication` was supplied to **C_OpenSession**. It is not necessarily the case that all slots/tokens will support token removal callbacks. Token removal callbacks should return the value CKR_OK.

9.17.3 Parallel function completion callbacks

When a function executing in parallel finishes execution, the callback for the session that function was running in (if there is such a callback) is executed with arguments `(hSession, CKN_COMPLETE, pApplication)` , where `hSession` is the session's handle and `pApplication` was supplied to **C_OpenSession**. Parallel function completion callbacks should return the value CKR_OK.

9.17.4 Serial function surrender callbacks

Functions executing in serial sessions can periodically surrender control to the application who called them, if the session they are executing in has a callback function. They do this by calling their session's callback with arguments `(hSession, CKN_SURRENDER, pApplication)` , where `hSession` is the session's handle and `pApplication` was supplied to **C_OpenSession**. Serial function surrender callbacks should return either the value CKR_OK (to indicate that Cryptoki should continue executing the function) or the value CKR_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation.

Note that this type of callback is somewhat different from the other three types of callbacks, because it doesn't require a spontaneous generation of a thread or process to execute the callback.

10. Mechanisms

A mechanism specifies precisely how a certain cryptographic process is to be performed.

The following table shows which Cryptoki mechanisms are supported by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token which supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation). For example, even if a token is able to create RSA digital signatures with the **CKM_RSA_PKCS** mechanism, it may or may not be the case that the same token can also perform RSA encryption.

Table 10-1, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR[1]	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓[2]	✓[2]	✓			✓	
CKM_RSA_9796		✓[2]	✓				
CKM_RSA_X_509	✓[2]	✓[2]	✓			✓	
CKM_MD2_RSA_PKCS	✓	✓					
CKM_MD5_RSA_PKCS	✓	✓					
CKM_SHA1_RSA_PKCS	✓	✓					
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA		✓[2]					
CKM_DSA_SHA1		✓					
CKM_FORTEZZA_TIMESTAMP		✓[2]					
CKM_ECDSA_KEY_PAIR_GEN					✓		
CKM_ECDSA		✓[2]					
CKM_ECDSA_SHA1		✓					
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_KEA_KEY_PAIR_GEN					✓		
CKM_KEA_KEY_DERIVE							✓
CKM_MAYFLY_KEY_PAIR_GEN					✓		
CKM_MAYFLY_KEY_DERIVE							✓
CKM_GENERIC_SECRET_KEY_GEN					✓		
CKM_RC2_KEY_GEN					✓		
CKM_RC2_ECB	✓					✓	
CKM_RC2_CBC	✓					✓	
CKM_RC2_CBC_PAD	✓					✓	
CKM_RC2_MAC_GENERAL		✓					
CKM_RC2_MAC		✓					
CKM_RC4_KEY_GEN					✓		
CKM_RC4	✓						
CKM_RC5_KEY_GEN					✓		
CKM_RC5_ECB	✓					✓	
CKM_RC5_CBC	✓					✓	
CKM_RC5_CBC_PAD	✓					✓	
CKM_RC5_MAC_GENERAL		✓					
CKM_RC5_MAC		✓					
CKM_DES_KEY_GEN					✓		
CKM_DES_ECB	✓					✓	
CKM_DES_CBC	✓					✓	
CKM_DES_CBC_PAD	✓					✓	
CKM_DES_MAC_GENERAL		✓					
CKM_DES_MAC		✓					
CKM_DES2_KEY_GEN					✓		
CKM_DES3_KEY_GEN					✓		
CKM_DES3_ECB	✓					✓	

CKM_DES3_CBC	✓					✓	
CKM_DES3_CBC_PAD	✓					✓	
CKM_DES3_MAC_GENERAL		✓					
CKM_DES3_MAC		✓					
CKM_CAST_KEY_GEN					✓		
CKM_CAST_ECB	✓					✓	
CKM_CAST_CBC	✓					✓	
CKM_CAST_CBC_PAD	✓					✓	
CKM_CAST_MAC_GENERAL		✓					
CKM_CAST_MAC		✓					
CKM_CAST3_KEY_GEN					✓		
CKM_CAST3_ECB	✓					✓	
CKM_CAST3_CBC	✓					✓	
CKM_CAST3_CBC_PAD	✓					✓	
CKM_CAST3_MAC_GENERAL		✓					
CKM_CAST3_MAC		✓					
CKM_CAST5_KEY_GEN					✓		
CKM_CAST5_ECB	✓					✓	
CKM_CAST5_CBC	✓					✓	
CKM_CAST5_CBC_PAD	✓					✓	
CKM_CAST5_MAC_GENERAL		✓					
CKM_CAST5_MAC		✓					
CKM_IDEA_KEY_GEN					✓		
CKM_IDEA_ECB	✓					✓	
CKM_IDEA_CBC	✓					✓	
CKM_IDEA_CBC_PAD	✓					✓	
CKM_IDEA_MAC_GENERAL		✓					
CKM_IDEA_MAC		✓					
CKM_CDMF_KEY_GEN					✓		
CKM_CDMF_ECB	✓					✓	
CKM_CDMF_CBC	✓					✓	
CKM_CDMF_CBC_PAD	✓					✓	
CKM_CDMF_MAC_GENERAL		✓					
CKM_CDMF_MAC		✓					
CKM_SKIPJACK_KEY_GEN					✓		
CKM_SKIPJACK_ECB64	✓						
CKM_SKIPJACK_CBC64	✓						
CKM_SKIPJACK_OFB64	✓						
CKM_SKIPJACK_CFB64	✓						
CKM_SKIPJACK_CFB32	✓						
CKM_SKIPJACK_CFB16	✓						
CKM_SKIPJACK_CFB8	✓						
CKM_SKIPJACK_WRAP						✓	
CKM_SKIPJACK_PRIVATE_WRAP						✓	
CKM_SKIPJACK_RELAYX						✓[3]	
CKM_BATON_KEY_GEN					✓		
CKM_BATON_ECB128	✓						
CKM_BATON_ECB96	✓						
CKM_BATON_CBC128	✓						
CKM_BATON_COUNTER	✓						

CKM_BATON_SHUFFLE	✓						
CKM_BATON_WRAP						✓	
CKM_JUNIPER_KEY_GEN					✓		
CKM_JUNIPER_ECB128	✓						
CKM_JUNIPER_CBC128	✓						
CKM_JUNIPER_COUNTER	✓						
CKM_JUNIPER_SHUFFLE	✓						
CKM_JUNIPER_WRAP						✓	
CKM_MD2				✓			
CKM_MD2_HMAC_GENERAL		✓					
CKM_MD2_HMAC		✓					
CKM_MD2_KEY_DERIVATION							✓
CKM_MD5				✓			
CKM_MD5_HMAC_GENERAL		✓					
CKM_MD5_HMAC		✓					
CKM_MD5_KEY_DERIVATION							✓
CKM_SHA_1				✓			
CKM_SHA_1_HMAC_GENERAL		✓					
CKM_SHA_1_HMAC		✓					
CKM_SHA1_KEY_DERIVATION							✓
CKM_FASTHASH				✓			
CKM_PBE_MD2_DES_CBC					✓		
CKM_PBE_MD5_DES_CBC					✓		
CKM_PBE_MD5_CAST_CBC					✓		
CKM_PBE_MD5_CAST3_CBC					✓		
CKM_PBE_MD5_CAST5_CBC					✓		
CKM_KEY_WRAP_SET_OAEP						✓	
CKM_KEY_WRAP_LYNKS						✓	
CKM_SSL3_PRE_MASTER_KEY_GEN					✓		
CKM_SSL3_MASTER_KEY_DERIVE							✓
CKM_SSL3_KEY_AND_MAC_DERIVE							✓
CKM_SSL3_MD5_MAC		✓					
CKM_SSL3_SHA1_MAC		✓					
CKM_CONCATENATE_BASE_AND_KEY							✓
CKM_CONCATENATE_BASE_AND_DATA							✓
CKM_CONCATENATE_DATA_AND_BASE							✓
CKM_XOR_BASE_AND_DATA							✓
CKM_EXTRACT_KEY_FROM_KEY							✓

[1] SR = SignRecover, VR = VerifyRecover.

[2] Single-part operations only.

[3] Mechanism can only be used for wrapping, not unwrapping.

The remainder of Section 10 will present in detail the mechanisms supported by Cryptoki v2.0 and the parameters which are supplied to them.

In general, if a mechanism makes no mention of the *ulMinKeyLen* and *ulMaxKeyLen* fields of the CK_MECHANISM_INFO structure, then those fields have no meaning for that particular mechanism.

10.1 RSA mechanisms

10.1.1 PKCS #1 RSA key pair generation

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

It does not have a parameter. This 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_PUBLIC_EXPONENT** attributes of the template for the public key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT** (see Section 8.6.1). 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 can be used with the following mechanisms: PKCS #1 RSA; ISO/IEC 9796 RSA; X.509 (raw) RSA; PKCS #1 RSA with MD2; PKCS #1 RSA with MD5; PKCS #1 RSA with SHA-1; and OAEP key wrapping for SET.

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

10.1.2 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.

This mechanism does not have a parameter. The mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. 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_CLASS** and

CKA_VALUE (and **CKA_VALUE_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

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 `02`
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 `01`
C_VerifyRecover	RSA public key	*k*	≤ *k*-11	block type `01`
C_WrapKey	RSA public key	≤ *k*-11	*k*	block type `02`
C_UnwrapKey	RSA private key	*k*	≤ *k*-11	block type `02`

[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.

10.1.3 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; it is 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.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. 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	$\leq \lfloor k/2 \rfloor$	*k*
C_SignRecover	RSA private key	$\leq \lfloor k/2 \rfloor$	*k*
C_Verify[1]	RSA public key	$\leq \lfloor k/2 \rfloor$, *k*[2]	N/A
C_VerifyRecover	RSA public key	*k*	$\leq \lfloor k/2 \rfloor$

[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.

10.1.4 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. All these operations are based on 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.

This mechanism does not have a parameter. any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. 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, and supply them when unwrapping the key.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0 bytes. In effect, to encrypt the sequence of plaintext bytes $b_1\ b_2\ \ldots\ b_n$ ($n \leq k$), Cryptoki forms $P=2^{n-1}b_1+2^{n-2}b_2+\ldots+b_n$. This number must be less than the RSA modulus. The *k*-byte ciphertext (*k* is the length in bytes of the RSA modulus) is produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a *k*-byte ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly *k* bytes. If the resulting plaintext is to be used to produce an unwrapped key, then however many bytes are specified in the template for the length of the key are taken *from the end* of this sequence of bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns CKR_DATA_INVALID (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and CKR_ENCRYPTED_DATA_INVALID (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

Constraints on key types and the length of input and output data are summarized in the following table. 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	$\leq k$	k
C_Decrypt[1]	RSA private key	k	k
C_Sign[1]	RSA private key	$\leq k$	k
C_SignRecover	RSA private key	$\leq k$	k
C_Verify[1]	RSA public key	$\leq k$, k[2]	N / A
C_VerifyRecover	RSA public key	k	k
C_WrapKey	RSA public key	$\leq k$	k
C_UnwrapKey	RSA private key	k	$\leq k$ (specified in template)

[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.

10.1.5 PKCS #1 RSA signature with MD2, MD5, or SHA-1

The PKCS #1 RSA signature with MD2 mechanism, denoted **CKM_MD2_RSA_PKCS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier md2WithRSAEncryption.

Similarly, the PKCS #1 RSA signature with MD5 mechanism, denoted **CKM_MD5_RSA_PKCS**, performs the same operations described in PKCS #1 with the object identifier md5WithRSAEncryption. The PKCS #1 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS**, performs the same operations, except that it uses the hash function SHA-1, instead of MD2 or MD5.

None of these mechanisms has a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For the PKCS #1 RSA signature with MD2 and PKCS #1 RSA signature with MD5 mechanisms, *k* must be at least 27; for the PKCS #1 RSA signature with SHA-1 mechanism, *k* must be at least 31.

Table 10-5, PKCS #1 RSA Signatures with MD2, MD5, or SHA-1: Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Sign	RSA private key	any	*k*	block type `01`
C_Verify	RSA public key	any, *k*[2]	N / A	block type `01`

[2] Data length, signature length.

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

10.2 DSA mechanisms

10.2.1 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.

This 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.

10.2.2 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:

Table 10-6, 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.

10.2.3 DSA with SHA-1

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA1**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186. This mechanism computes the entire DSA specification, including the hashing with SHA-1.

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.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 10-7, DSA with SHA-1: Key And Data Length Constraints

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

[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.

10.2.4 FORTEZZA timestamp

The FORTEZZA timestamp mechanism, denoted **CKM_FORTEZZA_TIMESTAMP**, is a mechanism for single-part signatures and verification. The signatures it produces and verifies are DSA digital signatures over the provided hash value and the current time.

It has no parameters.

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 10-8, FORTEZZA timestamp: 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.

10.3 ECDSA mechanisms

10.3.1 ECDSA key pair generation

The ECDSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation mechanism based on the Elliptic Curve Digital Signature Algorithm defined in IEEE P1363.

This mechanism does not have a parameter.

The mechanism generates ECDSA 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 ECDSA 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 ECDSA 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 ECDSA prime sizes, in bits.

10.3.2 ECDSA

The ECDSA mechanism, denoted **CKM_ECDSA**, is a mechanism for single-part signatures and verification based on the Elliptic Curve Digital Signature Algorithm defined in IEEE P1363. (This mechanism corresponds only to the part of ECDSA that processes the 20-byte hash value; it does not compute the hash value.)

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

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table :

Table 10-9, ECDSA: Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Sign[1]	ECDSA private key	20	40
C_Verify[1]	ECDSA 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 ECDSA prime sizes, in bits.

10.3.3 ECDSA with SHA-1

The ECDSA with SHA-1 mechanism, denoted **CKM_ECDSA_SHA1**, is a mechanism for single- and multiple-part signatures and verification based on the Elliptic Curve Digital Signature Algorithm defined in IEEE P1363. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1.

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

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 10-10, ECDSA with SHA-1: Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Sign	ECDSA private key	any	40
C_Verify	ECDSA public key	any, 40[2]	N / A

[2] Data length, signature length.

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

10.4 Diffie-Hellman mechanisms

10.4.1 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** (and the **CKA_VALUE_BITS** attribute, if it is not already provided in the template) 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.

10.4.2 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 as a Cryptoki "Big integer" (*i.e.*, a sequence of bytes, most-significant byte first).

This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other party. 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_KEY_TYPE** attribute 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.

The derived key inherits the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The values of the **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes may be overridden in the template for the derived key, however. Of course, if the base key has the **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_SENSITIVE** attribute set to FALSE; similarly, if the base key

has the **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_EXTRACTABLE** attribute set to TRUE.

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.

10.5 KEA mechanism parameters

♦ CK_KEA_DERIVE_PARAMS

CK_KEA_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_KEA_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_ KEA_DERIVE _PARAMS {
    CK_BBOOL  isSender;
    CK_ULONG  ulRandomLen;
    CK_BYTE_PTR  pRandomA;
    CK_BYTE_PTR  pRandomB;
    CK_ULONG  ulPublicDataLen;
    CK_BYTE_PTR  pPublicData;
} CK_KEA_DERIVE;
```

The fields of the structure have the following meanings:

isSender	Option for generating the key (called a TEK). The value is TRUE if the sender (originator) generates the TEK, FALSE if the recipient is regenerating the TEK.
ulRandomLen	size of random Ra and Rb, in bytes
pRandomA	pointer to Ra data
pRandomB	pointer to Rb data
ulPublicDataLen	other party's KEA public key size
pPublicData	pointer to other party's KEA public key value

♦ CK_KEA_DERIVE_PARAMS_PTR

CK_KEA_DERIVE_PARAMS_PTR points to a **CK_KEA_DERIVE_PARAMS** structure. It is implementation-dependent.

10.6 KEA mechanisms

10.6.1 KEA key pair generation

The KEA key pair generation mechanism, denoted **CKM_KEA_KEY_PAIR_GEN**, is a key pair generation mechanism

It does not have a parameter.

The mechanism generates KEA 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 KEA 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 KEA 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 KEA prime sizes, in bits.

10.6.2 KEA key derivation

The KEA key derivation mechanism, denoted **CKM_KEA_DERIVE**, is a mechanism for key derivation based on KEA, the Key Exchange Algorithm.

It has a parameter, a **CK_KEA_DERIVE_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute 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.

The derived key inherits the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The values of the **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes may be overridden in the template for the derived key, however. Of course, if the base key has the **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_SENSITIVE** attribute set to FALSE; similarly, if the base key has the **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_EXTRACTABLE** attribute set to TRUE.

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

10.7 MAYFLY mechanism parameters

♦ CK_MAYFLY_DERIVE_PARAMS

CK_MAYFLY_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_MAYFLY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_MAYFLY_DERIVE_PARAMS {
    CK_BBOOL  isSender;
    CK_ULONG ulRandomLen;
    CK_BYTE_PTR  pRandomA;
    CK_BYTE_PTR  pRandomB;
    CK_ULONG  ulPublicDataLen;
    CK_BYTE_PTR  pPublicData;
} CK_MAYFLY_DERIVE;
```

The fields of the structure have the following meanings:

isSender	Option for generating the key (called a TEK). The value is TRUE if the sender (originator) generates the TEK, FALSE if the recipient is regenerating the TEK.
ulRandomLen	size of random Ra and Rb, in bytes
pRandomA	pointer to Ra data
pRandomB	pointer to Rb data
ulPublicDataLen	other party's MAYFLY public key size
pPublicData	pointer to other party's MAYFLY public key value

♦ CK_MAYFLY_DERIVE_PARAMS_PTR

CK_MAYFLY_DERIVE_PARAMS_PTR points to a **CK_MAYFLY_DERIVE_PARAMS** structure. It is implementation-dependent.

10.8 MAYFLY mechanisms

10.8.1 MAYFLY key pair generation

The MAYFLY key pair generation mechanism, called **CKM_KEA_KEY_PAIR_GEN**, is a key pair generation mechanism for the MAYFLY key exchange key pair.

It does not have a parameter.

The mechanism generates MAYFLY 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 MAYFLY 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 MAYFLY 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 MAYFLY prime sizes, in bits.

10.8.2 MAYFLY key derivation

The MAYFLY key derivation mechanism, denoted **CKM_MAYFLY_DERIVE**, is a mechanism for key derivation based on MAYFLY.

It has a parameter, a **CK_MAYFLY_DERIVE_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute 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.

The derived key inherits the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The values of the **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes may be overridden in the template for the derived key, however. Of course, if the base key has the **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_SENSITIVE** attribute set to FALSE; similarly, if the base key has the **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the template may not specify that the derived key should have the **CKA_EXTRACTABLE** attribute set to TRUE.

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

10.9 Generic secret key mechanisms

10.9.1 Generic secret key generation

The generic secret key generation mechanism, denoted **CKM_GENERIC_SECRET_KEY_GEN**, is used to generate generic secret keys. The generated keys take on any attributes provided in the template passed to the **C_GenerateKey** call, and the **CKA_VALUE_LEN** attribute specifies the length of the key to be generated.

It does not have a parameter.

The template supplied must specify a value for the **CKA_VALUE_LEN** attribute. If the template specifies an object type and a class, they must have the following values:

CK_OBJECT_CLASS = **CKO_SECRET_KEY**;

CK_KEY_TYPE = **CKK_GENERIC_SECRET**;

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

10.10 Wrapping/unwrapping private keys (RSA, Diffie-Hellman, and DSA)

Cryptoki v2.0 allows the use of secret keys for wrapping and unwrapping RSA private keys, Diffie-Hellman private keys, and DSA private keys. For wrapping, a private key is BER-encoded according to PKCS #8's PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the secret key. The object identifiers for the needed algorithm identifiers are as follows:

```
rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }

dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1}

DSA OBJECT IDENTIFIER ::= { iso(1) identifier-organization(3)
          oiw(14) secsig(3) algorithm(2) 12 }
```

where

```
pkcs-1 OBJECT IDENTIFIER ::= { iso(1) member-body(2) US(840)
          rsadsi(113549) pkcs(1) 1 }

pkcs-3 OBJECT IDENTIFIER ::= { iso(1) member-body(2) US(840)
          rsadsi(113549) pkcs(1) 3 }
```

These object identifiers have the following parameters, respectively:

```
NULL

DHParameter ::= SEQUENCE {
    prime INTEGER,  -- p
    base INTEGER ,  -- g
    privateValueLength INTEGER OPTIONAL
}

DSAParameters ::= SEQUENCE {
    modulusLength INTEGER,  -- length of p in bits
    prime1 INTEGER,  -- modulus p
    prime2 INTEGER,  -- modulus q
    base INTEGER  -- base g
}
```

Within the PrivateKeyInfo type:

- RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type requires particular values for all the attributes specific to Cryptoki's RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key's **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,

CKA_PRIME_1, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT2**, and **CKA_COEFFICIENT** values, it cannot create an RSAPrivateKey BER-encoding of the key, and so it cannot prepare it for wrapping.

- Diffie-Hellman private keys are encoded by expressing the private value as a sequence of bytes, most-significant byte first, and then BER-encoding that sequence of bytes as an OCTET STRING.

- DSA private keys are encoded by expressing the private value as a sequence of of bytes, most-significant byte first, and then BER-encoding that sequence of bytes as an OCTET STRING.

Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes can be encrypted with the secret key. This encryption must be done in CBC mode with PKCS padding.

Unwrapping a wrapped private key undoes the above procedure. The CBC-encrypted ciphertext is decrypted, and the PKCS padding is removed. The data thereby obtained are parsed as a PrivateKeyInfo type, and the wrapped key is produced. An error will result if the original wrapped key does not decrypt properly, or if the decrypted unpadded data does not parse properly, or its type does not match the key type specified in the template for the new key. The unwrapping mechanism contributes only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will take their default values.

10.11 The RC2 cipher

RC2 is a proprietary block cipher which is trademarked by RSA Data Security. It has a variable keysize and an additional parameter, the “effective number of bits in the RC2 search space”, which can take on values in the range 1-1024, inclusive.

10.12 RC2 mechanism parameters

♦ CK_RC2_PARAMS

CK_RC2_PARAMS provides the parameters to the **CKM_RC2_ECB** and **CKM_RC2_MAC** mechanisms. It holds the effective number of bits in the RC2 search space. It is defined as follows:

```
typdef CK_ULONG CK_RC2_PARAMS;
```

♦ CK_RC2_PARAMS_PTR

CK_RC2_PARAMS_PTR points to a **CK_RC2_PARAMS** 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** and **CKM_RC2_CBC_PAD** mechanisms. It is defined as follows:

```
typedef struct CK_RC2_CBC_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_BYTE iv[8];
} CK_RC2_CBC_PARAMS;
```

The fields of the structure have the following meanings:

ulEffectiveBits	the effective number of bits in the RC2 search space
iv	the initialization vector (IV) for cipher block chaining mode

♦ CK_RC2_CBC_PARAMS_PTR

CK_RC2_CBC_PARAMS_PTR points to a **CK_RC2_CBC_PARAMS** structure. It is implementation-dependent.

♦ CK_RC2_MAC_GENERAL_PARAMS

CK_RC2_MAC_GENERAL_PARAMS is a structure that provides the parameters to the **CKM_RC2_MAC_GENERAL** mechanism. It is defined as follows:

```
typedef struct CK_RC2_MAC_GENERAL_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_ULONG ulMacLength;
} CK_RC2_MAC_GENERAL_PARAMS;
```

The fields of the structure have the following meanings:

ulEffectiveBits	the effective number of bits in the RC2 searc h space
ulMacLength	length of the MAC produced, in bytes

♦ CK_RC2_MAC_GENERAL_PARAMS_PTR

CK_RC2_MAC_GENERAL_PARAMS_PTR points to a **CK_RC2_MAC_GENERAL_PARAMS** structure. It is implementation-dependent.

10.13 RC2 mechanisms

10.13.1 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.

10.13.2 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 **CK_RC2_PARAMS**, which indicates the effective number of bits in the RC2 search space.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. 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_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. 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:

Table 10-11, 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	multiple of 8	determined by type of key being unwrapped or CKA_VALUE_LEN	

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

10.13.3 RC2-CBC

RC2-CBC, denoted **CKM_RC2_CBC**, 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 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.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. 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_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. 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:

Table 10-12, 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
C_WrapKey	RC2	any	input length rounded up to multiple of 8	
C_UnwrapKey	RC2	multiple of 8	determined by type of key being unwrapped or CKA_VALUE_LEN	

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

10.13.4 RC2-CBC with PKCS padding

RC2-CBC with PKCS padding, denoted **CKM_RC2_CBC_PAD**, 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; cipher-block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.

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.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA_VALUE_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, and DSA private keys (see Section 0 for details). The entries in table Table 10-13 for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 10-13, RC2-CBC with PKCS padding: Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Encrypt	RC2	any	input length rounded up to multiple of 8
C_Decrypt	RC2	multiple of 8	between 1 and 8 bytes shorter than input length
C_WrapKey	RC2	any	input length rounded up to multiple of 8
C_UnwrapKey	RC2	multiple of 8	between 1 and 8 bytes shorter than input length

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

10.13.5 General-length RC2-MAC

General-length RC2-MAC, denoted **CKM_RC2_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures 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 **CK_RC2_MAC_GENERAL_PARAMS** structure, which specifies the effective number of bits in the RC2 search space and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC2 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 10-14, General-length RC2-MAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	RC2	any	0-8, as specified in parameters
C_Verify	RC2	any	0-8, as specified in parameters

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

10.13.6 RC2-MAC

RC2-MAC, denoted by **CKM_RC2_MAC**, is a special case of the general-length RC2-MAC mechanism (see Section 10.13.5). Instead of taking a **CK_RC2_MAC_GENERAL_PARAMS** parameter, it takes a **CK_RC2_PARAMS** parameter, which only contains the effective number of bits in the RC2 search space. RC2-MAC always produces and verifies 4-byte MACs.

Constraints on key types and the length of data are summarized in the following table:

Table 10-15, 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.

10.14 RC4 mechanisms

10.14.1 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 generates RC4 keys with a particular length in bytes, as specified in the CK**A_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 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.

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

10.14.2 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.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 10-16, 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 The RC5 cipher

RC5 is a parametrizable block cipher for which RSA Data Security has applied for a patent. It has a variable wordsize, a variable keysize, and a variable number of rounds. The blocksize of RC5 is always equal to twice its wordsize.

10.16 RC5 mechanism parameters

♦ CK_RC5_PARAMS

CK_RC5_PARAMS provides the parameters to the **CKM_RC5_ECB** and **CKM_RC5_MAC** mechanisms. It is defined as follows:

```
typdef struct CK_RC5_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
} CK_RC5_PARAMS;
```

The fields of the structure have the following meanings:

ulWordsize	wordsize of RC5 cipher in bytes
ulRounds	number of rounds of RC5 encipherment

♦ CK_RC5_PARAMS_PTR

CK_RC5_PARAMS_PTR points to a **CK_RC5_PARAMS** structure. It is implementation-dependent.

♦ CK_RC5_CBC_PARAMS

CK_RC5_CBC_PARAMS is a structure that provides the parameters to the **CKM_RC5_CBC** and **CKM_RC5_CBC_PAD** mechanisms. It is defined as follows:

```
typedef struct CK_RC5_CBC_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_BYTE_PTR pIv;
    CK_ULONG ulIvLen;
} CK_RC5_CBC_PARAMS;
```

The fields of the structure have the following meanings:

ulWordsize	wordsize of RC5 cipher in bytes
ulRounds	number of rounds of RC5 encipherment
pIv	pointer to initialization vector (IV) for CBC encryption
ulIvLen	length of initialization vector (must be same as blocksize)

♦ CK_RC5_CBC_PARAMS_PTR

CK_RC5_CBC_PARAMS_PTR points to a **CK_RC5_CBC_PARAMS** structure. It is implementation-dependent.

♦ CK_RC5_MAC_GENERAL_PARAMS

CK_RC5_MAC_GENERAL_PARAMS is a structure that provides the parameters to the **CKM_RC5_MAC_GENERAL** mechanism. It is defined as follows:

```
typedef struct CK_RC5_MAC_GENERAL_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_ULONG ulMacLength;
} CK_RC5_MAC_GENERAL_PARAMS;
```

The fields of the structure have the following meanings:

ulWordsize	wordsize of RC5 cipher in bytes
ulRounds	number of rounds of RC5 encipherment
ulMacLength	length of the MAC produced, in bytes

♦ CK_RC5_MAC_GENERAL_PARAMS_PTR

CK_RC5_MAC_GENERAL_PARAMS_PTR points to a **CK_RC5_MAC_GENERAL_PARAMS** structure. It is implementation-dependent.

10.17 RC5 mechanisms

10.17.1 RC5 key generation

The RC5 key generation mechanism, denoted **CKM_RC5_KEY_GEN**, is a key generation mechanism for RSA Data Security's block cipher RC5.

It does not have a parameter. RC5 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 RC5 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 RC5 key sizes, in bytes.

10.17.2 RC5-ECB

RC5-ECB, denoted **CKM_RC5_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Data Security's block cipher RC5 and electronic codebook mode as defined in FIPS PUB 81.

It has a parameter, a **CK_RC5_PARAMS**, which indicates the wordsize and number of rounds of encryption to use.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with null bytes so that the resulting length is a multiple of the cipher blocksize (twice the wordsize). 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_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 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:

Table 10-17, RC5-ECB: Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC5	multiple of blocksize	same as input length	no final part
C_Decrypt	RC5	multiple of blocksize	same as input length	no final part
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	RC5	multiple of blocksize	determined by type of key being unwrapped or CKA_VALUE_LEN	

10.17.3 RC5-CBC

RC5-CBC, denoted **CKM_RC5_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Data Security's block cipher RC5 and cipher-block chaining mode as defined in FIPS PUB 81.

It has a parameter, a **CK_RC5_CBC_PARAMS** structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. 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_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. 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:

Table 10-18, RC5-CBC: Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	RC5	multiple of blocksize	same as input length	no final part
C_Decrypt	RC5	multiple of blocksize	same as input length	no final part
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	RC5	multiple of blocksize	determined by type of key being unwrapped or CKA_VALUE_LEN	

10.17.4 RC5-CBC with PKCS padding

RC5-CBC with PKCS padding, denoted **CKM_RC5_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Data Security's block cipher RC5; cipher-block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a **CK_RC5_CBC_PARAMS** structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA_VALUE_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, and DSA private keys (see Section 0 for details). The entries in table Table 10-19 for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 10-19, RC5-CBC with PKCS padding: Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Encrypt	RC5	any	input length rounded up to multiple of blocksize
C_Decrypt	RC5	multiple of blocksize	between 1 and blocksize bytes shorter than input length
C_WrapKey	RC5	any	input length rounded up to multiple of blocksize
C_UnwrapKey	RC5	multiple of blocksize	between 1 and blocksize bytes shorter than input length

10.17.5 General-length RC5-MAC

General-length RC5-MAC, denoted **CKM_RC5_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on RSA Data Security's block cipher RC5 and data authentication as defined in FIPS PUB 113.

It has a parameter, a **CK_RC5_MAC_GENERAL_PARAMS** structure, which specifies the wordsize and number of rounds of encryption to use and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC5 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 10-20, General-length RC2-MAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	RC2	any	0-blocksize, as specified in parameters
C_Verify	RC2	any	0-blocksize, as specified in parameters

10.17.6 RC5-MAC

RC5-MAC, denoted by **CKM_RC5_MAC**, is a special case of the general-length RC5-MAC mechanism (see Section 10.17.5). Instead of taking a **CK_RC5_MAC_GENERAL_PARAMS** parameter, it takes a **CK_RC5_PARAMS** parameter. RC5-MAC always produces and verifies MACs half as large as the RC5 blocksize.

Constraints on key types and the length of data are summarized in the following table:

Table 10-21, RC5-MAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	RC5	any	RC5 wordsize = ⌊blocksize/2⌋
C_Verify	RC5	any	RC5 wordsize = ⌊blocksize/2⌋

10.18 General block cipher mechanism parameters

♦ CK_MAC_GENERAL_PARAMS

CK_MAC_GENERAL_PARAMS provides the parameters to the general-length MACing mechanisms of the DES, DES3 (triple-DES), CAST, CAST3, CAST5, IDEA, and CDMF ciphers. It holds the length of the MAC that these mechanisms will produce. It is defined as follows:

```
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

♦ CK_MAC_GENERAL_PARAMS_PTR

CK_MAC_GENERAL_PARAMS_PTR points to a **CK_MAC_GENERAL_PARAMS**. It is implementation-dependent.

10.19 General block cipher mechanisms

For brevity's sake, the mechanisms for the DES, DES3 (triple-DES), CAST, CAST3, CAST5, IDEA, and CDMF block ciphers will be described together here. Each of these ciphers has the following mechanisms, which will be described in a templatized form:

10.19.1 General block cipher key generation

Cipher <NAME> has a key generation mechanism, "<NAME> key generation", denoted **CKM_<NAME>_KEY_GEN.**

This mechanism 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 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.

When DES keys or CDMF keys are generated, their parity bits are set properly, as specified in FIPS PUB 46-2. Similarly, when a triple-DES key is generated, each of the DES keys comprising it has its parity bits set properly.

When DES or CDMF keys are generated, it is token-dependent whether or not it is possible for "weak" or "semi-weak" keys to be generated. Similarly, when triple-DES keys are generated, it is token dependent whether or not it is possible for any of the component DES keys to be "weak" or "semi-weak" keys.

When CAST, CAST3, or CAST5 keys are generated, the template for the secret key must specify a **CKA_VALUE_LEN** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST5 ciphers have variable key sizes, and so for the the key generation mechanisms for these ciphers, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

10.19.2 General block cipher ECB

Cipher <NAME> has an electronic codebook mechanism, "<NAME>-ECB", denoted **CKM_<NAME>_ECB**. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>.

It does not have a parameter. It can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with null bytes so that the resulting length is a multiple of <NAME>'s blocksize. 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_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. 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:

Table 10-22, General block cipher ECB: Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_Decrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	<NAME>	any	determined by type of key being unwrapped or CKA_VALUE_LEN	

10.19.3 General block cipher CBC

Cipher <NAME> has a cipher-block chaining mode, "<NAME>-CBC", denoted **CKM_<NAME>_CBC**. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as <NAME>'s blocksize.

Constraints on key types and the length of data are summarized in the following table:

Table 10-23, General block cipher CBC: Key And Data Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_Decrypt	<NAME>	multiple of blocksize	same as input length	no final part
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize	
C_UnwrapKey	<NAME>	any	determined by type of key being unwrapped or CKA_VALUE_LEN	

10.19.4 General block cipher CBC with PKCS padding

Cipher <NAME> has a cipher-block chaining mode with PKCS padding, "<NAME>-CBC with PKCS padding", denoted **CKM_<NAME>_CBC_PAD**. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>. All ciphertext is padded with PKCS padding.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as <NAME>'s blocksize.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA_VALUE_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, and DSA private keys (see Section 0 for details). The entries in table Table 10-24 for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 10-24, General block cipher CBC with PKCS padding: Key And Data Length Constraints

Function	Key type	Input length	Output length
C_Encrypt	<NAME>	any	input length rounded up to multiple of blocksize
C_Decrypt	<NAME>	multiple of blocksize	between 1 and blocksize bytes shorter than input length
C_WrapKey	<NAME>	any	input length rounded up to multiple of blocksize
C_UnwrapKey	<NAME>	multiple of blocksize	between 1 and blocksize bytes shorter than input length

10.19.5 General-length general block cipher MAC

Cipher <NAME> has a general-length MACing mode, "General-length <NAME>-MAC", denoted **CKM_<NAME>_MAC_GENERAL**. It is a mechanism for single- and multiple-part signatures and verification.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the size of the output.

The output bytes from this mechanism are taken from the start of the final cipher block produced in the MACing process.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 10-25, General-length general block cipher MAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	<NAME>	any	0-blocksize, depending on parameters
C_Verify	<NAME>	any	0-blocksize, depending on parameters

10.19.6 General block cipher MAC

Cipher <NAME> has a MACing mechanism, "<NAME>-MAC", denoted **CKM_<NAME>_MAC**. This mechanism is a special case of the **CKM_<NAME>_MAC_GENERAL** mechanism described in Section 10.19.5. It always produces an output of size half as large as <NAME>'s blocksize.

This mechanism has no parameters.

Constraints on key types and the length of data are summarized in the following table:

Table 10-26, General block cipher MAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	<NAME>	any	⌊blocksize/2⌋
C_Verify	<NAME>	any	⌊blocksize/2⌋

10.20 Double-length DES mechanisms

10.20.1 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. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in FIPS PUB 46-2.

The new key has its **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes set to the new key. Other attributes supported by the double-length 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.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys: **CKM_DES_ECB**, **CKM_DES_CBC**, **CKM_DES_CBC_PAD**, **CKM_DES_MAC_GENERAL**, and **CKM_DES_MAC** (these mechanisms are described in templatized form in Section 10.19). Triple-DES encryption with a double-length DES key consists of three steps: encryption with the first DES key; decryption with the second DES key; and encryption with the first DES key.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be “weak” or “semi-weak” keys.

10.21 SKIPJACK mechanism parameters

♦ CK_SKIPJACK_PRIVATE_WRAP_PARAMS

CK_SKIPJACK_PRIVATE_WRAP_PARAMS is a structure that provides the parameters to the **CKM_SKIPJACK_PRIVATE_WRAP** mechanism. It is defined as follows:

```
typedef struct CK_ SKIPJACK_PRIVATE_WRAP _PARAMS {
    CK_ULONG ulPasswordLen;
    CK_BYTE_PTR  pPassword;
    CK_ULONG  ulPublicDataLen;
    CK_BYTE_PTR  pPublicData;
    CK_ULONG  ulPandGLen;
    CK_ULONG  ulQLen;
    CK_ULONG  ulRandomLen;
    CK_BYTE_PTR  pRandomA;
```



```
    CK_BYTE_PTR  pPrimeP;
    CK_BYTE_PTR  pBaseG;
    CK_BYTE_PTR  pSubprime Q;
} CK_ SKIPJACK_PRIVATE_WRAP  _PARAMS;
```

The fields of the structure have the following meanings:

ulPasswordLen	length of the password
pPassword	pointer to the buffer which contains the user-supplied password
ulPublicDataLen	other party's key exchange public key size
pPublicData	pointer to other party's key exchange public key value
ulPandGLen	length of prime and base values
ulQLen	length of subprime value
ulRandomLen	size of random Ra, in bytes
pRandomA	pointer to Ra data
pPrimeP	pointer to Prime, p, value
pBaseG	pointer to Base, g, value
pSubprimeQ	pointer to Subprime, q, value

♦ CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR

CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR points to a **CK_PRIVATE_WRAP_PARAMS** structure. It is implementation-dependent.

♦ CK_SKIPJACK_RELAYX_PARAMS

CK_SKIPJACK_RELAYX_PARAMS is a structure that provides the parameters to the **CKM_SKIPJACK_RELAYX** mechanism. It is defined as follows:

```
typedef struct CK_ SKIPJACK_ RELAY X_PARAMS {
    CK_ULONG ul OldWrappedXLen;
    CK_BYTE_PTR  pOldWrappedX;
    CK_ULONG ulOldPasswordLen;
    CK_BYTE_PTR  pOldPassword;
    CK_ULONG  ulOldPublicDataLen;
    CK_BYTE_PTR  pOldPublicData;
    CK_ULONG  ulOldRandomLen;
    CK_BYTE_PTR  pOldRandomA;
    CK_ULONG ulNewPasswordLen;
    CK_BYTE_PTR  pNewPassword;
    CK_ULONG  ulNewPublicDataLen;
```



```
    CK_BYTE_PTR  pNewPublicData;
    CK_ULONG  ulNewRandomLen;
    CK_BYTE_PTR  pNewRandomA;
} CK_SKIPJACK_RELAYX_PARAMS;
```

The fields of the structure have the following meanings:

ulOldWrappedXLen	length of old wrapped key in bytes
pOldWrappedX	pointer to old wrapper key
ulOldPasswordLen	length of the old password
pOldPassword	pointer to the buffer which contains the old user-supplied password
ulOldPublicDataLen	old key exchange public key size
pOldPublicData	pointer to old key exchange public key value
ulOldRandomLen	size of old random Ra in bytes
pOldRandomA	pointer to old Ra data
ulNewPasswordLen	length of the new password
pNewPassword	pointer to the buffer which contains the new user-supplied password
ulNewPublicDataLen	new key exchange public key size
pNewPublicData	pointer to new key exchange public key value
ulNewRandomLen	size of new random Ra in bytes
pNewRandomA	pointer to new Ra data

♦ CK_SKIPJACK_RELAYX_PARAMS_PTR

CK_SKIPJACK_RELAYX_PARAMS_PTR points to a **CK_SKIPJACK_RELAYX_PARAMS** structure. It is implementation-dependent.

10.22 SKIPJACK mechanisms

10.22.1 SKIPJACK key generation

The SKIPJACK key generation mechanism, denoted **CKM_SKIPJACK_KEY_GEN**, is a key generation mechanism for SKIPJACK. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key.

10.22.2 SKIPJACK-ECB64

SKIPJACK-ECB64, denoted **CKM_SKIPJACK_ECB64**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit electronic codebook mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-27, SKIPJACK-ECB64: Data and Length Constraints

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

10.22.3 SKIPJACK-CBC64

SKIPJACK-CBC64, denoted **CKM_SKIPJACK_CBC64**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher-block chaining mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-28, SKIPJACK-CBC64: Data and Length Constraints

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

10.22.4 SKIPJACK-OFB64

SKIPJACK-OFB64, denoted **CKM_SKIPJACK_OFB64**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit output feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-29, SKIPJACK-OFB64: Data and Length Constraints

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

10.22.5 SKIPJACK-CFB64

SKIPJACK-CFB64, denoted **CKM_SKIPJACK_CFB64**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-30, SKIPJACK-CFB64: Data and Length Constraints

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

10.22.6 SKIPJACK-CFB32

SKIPJACK-CFB32, denoted **CKM_SKIPJACK_CFB32**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 32-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-31, SKIPJACK-CFB32: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

10.22.7 SKIPJACK-CFB16

SKIPJACK-CFB16, denoted **CKM_SKIPJACK_CFB16**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 16-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-32, SKIPJACK-CFB16: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

10.22.8 SKIPJACK-CFB8

SKIPJACK-CFB8, denoted **CKM_SKIPJACK_CFB8**, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 8-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-33, SKIPJACK-CFB8: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	SKIPJACK	multiple of 4	same as input length	no final part
C_Decrypt	SKIPJACK	multiple of 4	same as input length	no final part

10.22.9 SKIPJACK-WRAP

The SKIPJACK-WRAP mechanism, denoted **CKM_SKIPJACK_WRAP**, is used to wrap and unwrap a secret key (MEK). It can wrap or unwrap SKIPJACK, BATON, and JUNIPER keys.

It does not have a parameter.

10.22.10 SKIPJACK-PRIVATE-WRAP

The SKIPJACK-PRIVATE-WRAP mechanism, denoted **CKM_SKIPJACK_PRIVATE_WRAP**, is used to wrap and unwrap a private key. It can wrap KEA and DSA private keys.

It has a parameter, a **CK_SKIPJACK_PRIVATE_WRAP_PARAMS** structure

10.22.11 SKIPJACK-RELAYX

The SKIPJACK-RELAYX mechanism, denoted **CKM_SKIPJACK_RELAYX**, is used with the **C_WrapKey** function to "change the wrapping" on a private key which was wrapped with the SKIPJACK-PRIVATE-WRAP mechanism (see Section 10.22.10).

It has a parameter, a **CK_SKIPJACK_RELAYX_PARAMS** structure.

Although the SKIPJACK-RELAYX mechanism is used with **C_WrapKey**, it differs from other key-wrapping mechanisms. Other key-wrapping mechanisms take a key handle as one of the arguments to **C_WrapKey**; however, for the SKIPJACK_RELAYX mechanism, the [always invalid] value 0 should be passed as the key handle for **C_WrapKey**, and the already-wrapped key is passed in as part of the **CK_SKIPJACK_RELAYX_PARAMS** structure.

10.23 BATON mechanisms

10.23.1 BATON key generation

The BATON key generation mechanism, denoted **CKM_BATON_KEY_GEN**, is a key generation mechanism for BATON. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

This mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key.

10.23.2 BATON-ECB128

BATON-ECB128, denoted **CKM_BATON_ECB128**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-34, BATON-ECB128: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

10.23.3 BATON-ECB96

BATON-ECB96, denoted **CKM_BATON_ECB96**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 96-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-35, BATON-ECB96: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 12	same as input length	no final part
C_Decrypt	BATON	multiple of 12	same as input length	no final part

10.23.4 BATON-CBC128

BATON-CBC128, denoted **CKM_BATON_CBC128**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit cipher-block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-36, BATON-CBC128: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

10.23.5 BATON-COUNTER

BATON-COUNTER, denoted **CKM_BATON_COUNTER**, is a mechanism for single- and multiple-part encryption and decryption with BATON in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-37, BATON-COUNTER: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

10.23.6 BATON-SHUFFLE

BATON-SHUFFLE, denoted **CKM_BATON_SHUFFLE**, is a mechanism for single- and multiple-part encryption and decryption with BATON in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table :

Table 10-38, BATON-SHUFFLE: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	BATON	multiple of 16	same as input length	no final part
C_Decrypt	BATON	multiple of 16	same as input length	no final part

10.23.7 BATON WRAP

The BATON wrap and unwrap mechanism, denoted **CKM_BATON_WRAP**, is a function used to wrap and unwrap a secret key (MEK). It can wrap and unwrap SKIPJACK, BATON, and JUNIPER keys.

It has no parameters.

When used to unwrap a key, this mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to it.

10.24 JUNIPER mechanisms

10.24.1 JUNIPER key generation

The JUNIPER key generation mechanism, denoted **CKM_JUNIPER_KEY_GEN**, is a key generation mechanism for JUNIPER. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key.

10.24.2 JUNIPER-ECB128

JUNIPER-ECB128, denoted **CKM_JUNIPER_ECB128**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 10-39, JUNIPER-ECB128: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

10.24.3 JUNIPER-CBC128

JUNIPER-CBC128, denoted **CKM_JUNIPER_CBC128**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit cipher-block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 10-40, JUNIPER-CBC128: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

10.24.4 JUNIPER-COUNTER

JUNIPER COUNTER, denoted **CKM_JUNIPER_COUNTER**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 10-41, JUNIPER-COUNTER: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

10.24.5 JUNIPER-SHUFFLE

JUNIPER-SHUFFLE, denoted **CKM_JUNIPER_SHUFFLE**, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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

Table 10-42, JUNIPER-SHUFFLE: Data and Length Constraints

Function	Key type	Input length	Output length	Comments
C_Encrypt	JUNIPER	multiple of 16	same as input length	no final part
C_Decrypt	JUNIPER	multiple of 16	same as input length	no final part

10.24.6 JUNIPER WRAP

The JUNIPER wrap and unwrap mechanism, denoted **CKM_JUNIPER_WRAP**, is a function used to wrap and unwrap an MEK. It can wrap or unwrap SKIPJACK, BATON, and JUNIPER keys.

It has no parameters.

When used to unwrap a key, this mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to it.

10.25 MD2 mechanisms

10.25.1 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:

Table 10-43, MD2: Data Length Constraints

Function	Data length	Digest length
C_Digest	any	16

10.25.2 General-length MD2-HMAC

The general-length MD2-HMAC mechanism, denoted **CKM_MD2_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD2 hash function. The keys it uses are generic secret keys.

It has a parameter, a **CKA_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-16 (the output size of MD2 is 16 bytes).

Signatures produced by this mechanism will be taken from the start of the full 16-byte HMAC output.

Table 10-44, General-length MD2-HMAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-16, depending on parameters
C_Verify	generic secret	any	0-16, depending on parameters

10.25.3 MD2-HMAC

The MD2-HMAC mechanism, denoted **CKM_MD2_HMAC**, is a special case of the general-length MD2-HMAC mechanism in Section 10.25.2.

It has no parameter, and always produces an output of length 16.

10.25.4 MD2 key derivation

MD2 key derivation, denoted **CKM_MD2_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD2.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 16 bytes (the output size of MD2).
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 16 bytes, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

10.26 MD5 mechanisms

10.26.1 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. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 10-45, MD5: Data Length Constraints

Function	Data length	Digest length
C_Digest	any	16

10.26.2 General-length MD5-HMAC

The general-length MD5-HMAC mechanism, denoted **CKM_MD5_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD5 hash function. The keys it uses are generic secret keys.

It has a parameter, a **CKA_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-16 (the output size of MD5 is 16 bytes). Signatures produced by this mechanism will be taken from the start of the full 16-byte HMAC output.

Table 10-46, General-length MD5-HMAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-16, depending on parameters
C_Verify	generic secret	any	0-16, depending on parameters

10.26.3 MD5-HMAC

The MD5-HMAC mechanism, denoted **CKM_MD5_HMAC**, is a special case of the general-length MD5-HMAC mechanism in Section 10.26.2.

It has no parameter, and always produces an output of length 16.

10.26.4 MD5 key derivation

MD5 key derivation, denoted **CKM_MD5_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD5.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 16 bytes (the output size of MD5).

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 16 bytes, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

10.27 SHA-1 mechanisms

10.27.1 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.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 10-47, SHA-1: Data Length Constraints

Function	Input length	Digest length
C_Digest	any	20

10.27.2 General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys.

It has a parameter, a **CKA_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 0-20 (the output size of SHA-1 is 20 bytes). Signatures produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 10-48, General-length SHA-1-HMAC: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	0-20, depending on parameters
C_Verify	generic secret	any	0-20, depending on parameters

10.27.3 SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC**, is a special case of the general-length SHA-1-HMAC mechanism in Section 10.27.2.

It has no parameter, and always produces an output of length 20.

10.27.4 SHA-1 key derivation

SHA-1 key derivation, denoted **CKM_SHA1_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 20 bytes, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

10.28 FASTHASH mechanisms

10.28.1 FASTHASH

The FASTHASH mechanism, denoted **CKM_FASTHASH**, is a mechanism for message digesting, following the U. S. government's algorithm.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table :

Table 10-49, FASTHASH: Data Length Constraints

Function	Input length	Digest length
C_Digest	any	40

10.29 Password-based encryption mechanism parameters

♦ CK_PBE_PARAMS

CK_PBE_PARAMS is a structure which provides all of the necessary information required by the CKM_PBE mechanisms (see PKCS#5 for information on the PBE generation mechanisms). It is defined as follows:

```
typedef struct CK_PBE_PARAMS {
    CK_CHAR_PTR pInitVector;
    CK_CHAR_PTR pPassword;
    CK_ULONG ulPasswordLen;
    CK_CHAR_PTR pSalt;
    CK_ULONG ulSaltLen;
    CK_ULONG ulIteration;
} CK_PBE_PARAMS;
```

The fields of the structure have the following meanings:

pInitVector	pointer to the location that receives the 8-byte initialization vector (IV);
pPassword	points to the password to be used in the PBE key generation;
ulPasswordLen	length in bytes of the password information;
pSalt	points to the salt to be used in the PBE key generation;
usSaltLen	length in bytes of the salt information;
usIteration	number of iterations required for the generation.

♦ CK_PBE_PARAMS_PTR

CK_PBE_PARAMS_PTR points to a **CK_PBE_PARAMS** structure. It is implementation-dependent.

10.30 Password-based encryption mechanisms

10.30.1 MD2-PBE for DES-CBC

MD2-PBE for DES-CBC, denoted **CKM_PBE_MD2_DES_CBC**, is a mechanism used for generating a DES secret key and an initialization vector by using a password and a salt value and the MD2 digest algorithm. This functionality is defined in PKCS#5.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

10.30.2 MD5-PBE for DES-CBC

MD5-PBE for DES-CBC, denoted **CKM_PBE_MD5_DES_CBC**, is a mechanism used for generating a DES secret key and an initialization vector by using a password and a salt value and the MD5 digest algorithm. This functionality is defined in PKCS#5.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

10.30.3 MD5-PBE for CAST-CBC

MD5-PBE for CAST-CBC, denoted **CKM_PBE_MD5_CAST_CBC**, is a mechanism used for generating a CAST secret key and an initialization vector by using a password and a salt value and the MD5 digest algorithm. This functionality is essentially that defined in PKCS#5.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The CAST key generated by this mechanism is 8 bytes long.

10.30.4 MD5-PBE for CAST3-CBC

MD5-PBE for CAST3-CBC, denoted **CKM_PBE_MD5_CAST3_CBC**, is a mechanism used for generating a CAST3 secret key and an initialization vector by using a password and a salt value and the MD5 digest algorithm. This functionality is essentially that defined in PKCS#5.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The CAST3 key generated by this mechanism is 8 bytes long.

10.30.5 MD5-PBE for CAST5-CBC

MD5-PBE for CAST5-CBC, denoted **CKM_PBE_MD5_CAST5_CBC**, is a mechanism used for generating a CAST5 secret key and an initialization vector by using a password and a salt value and the MD5 digest algorithm. This functionality is essentially that defined in PKCS#5.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The CAST5 key generated by this mechanism is 8 bytes long.

10.31 SET mechanism parameters

♦ CK_KEY_WRAP_SET_OAEP_PARAMS

CK_KEY_WRAP_SET_OAEP_PARAMS is a structure that provides the parameters to the **CKM_KEY_WRAP_SET_OAEP** mechanism. It is defined as follows:

```
typedef struct CK_KEY_WRAP_SET_OAEP_PARAMS {
    CK_BYTE bBC;
    CK_BYTE_PTR pX;
    CK_ULONG ulXLen;
} CK_KEY_WRAP_SET_OAEP_PARAMS;
```

The fields of the structure have the following meanings:

bBC	block contents byte
pX	extra data
ulXLen	length in bytes of extra data

♦ CK_KEY_WRAP_SET_OAEP_PARAMS_PTR

CK_KEY_WRAP_SET_OAEP_PARAMS_PTR points to a **CK_KEY_WRAP_SET_OAEP_PARAMS** structure. It is implementation-dependent.

10.32 SET mechanisms

10.32.1 OAEP key wrapping for SET

The OAEP key wrapping for SET mechanism, denoted **CKM_KEY_WRAP_SET_OAEP**, is a mechanism for wrapping and unwrapping DES keys (and possibly some extra data) with RSA keys. This mechanism is defined in the SET protocol specifications.

It takes a parameter, a **CK_KEY_WRAP_SET_OAEP_PARAMS** structure. This structure holds the "Block Contents" byte of the data, as well as any extra data. If no extra data is present, that is indicated by the *ulXLen* field having the value 0.

When this mechanism is used to unwrap a key, the extra data is returned following the convention described in Section 9.2 on producing output. If the inputs to **C_UnwrapKey** are such that the extra data is not returned (*e.g.*, the buffer supplied in the **CK_KEY_WRAP_SET_OAEP_PARAMS** structure is NULL_PTR), then the unwrapped key object will not be created, either.

Note that when this mechanism is used to unwrap a key, the *bBC* and *pX* fields of the parameter supplied to the mechanism may be modified.

If an application uses **C_UnwrapKey** with **CKM_KEY_WRAP_SET_OAEP**, it is general preferable to simply allocate a 128-byte buffer for the extra data (the extra data is never larger than 128 bytes), rather than calling **C_UnwrapKey** twice. Each call of **C_UnwrapKey** with **CKM_KEY_WRAP_SET_OAEP** requires an RSA decryption operation to be performed, and this overhead can be avoided by this means.

10.33 LYNKS mechanisms

10.33.1 LYNKS key wrapping

The LYNKS key wrapping mechanism, denoted **CKM_WRAP_LYNKS**, is a mechanism for wrapping and unwrapping secret keys with DES keys. It can wrap any 8-byte secret key, and it produces a 10-byte wrapped key, containing a cryptographic checksum.

It does not have a parameter.

When unwrapping a key with this mechanism, if the cryptographic checksum does not check out properly, an error is returned. In addition, if a DES key or CDMF key is unwrapped with this mechanism, the parity bits on the wrapped key must be set appropriately; if they are not set properly, an error is returned.

10.34 SSL mechanism parameters

♦ CK_SSL3_RANDOM_DATA

CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the **CKM_SSL3_MASTER_KEY_DERIVE** and the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:


```
typedef struct CK_SSL3_RANDOM_DATA {
    CK_BYTE_PTR pClientRandom;
    CK_ULONG ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CK_ULONG ulServerRandomLen;
} CK_SSL3_RANDOM_DATA;
```

The fields of the structure have the following meanings:

pClientRandom	pointer to the client's random data. (see SSL 3.0 for details)
ulClientRandomLen	length in bytes of the client's random data
pServerRandom	pointer to the server's random data. (see SSL 3.0 for details)
ulServerRandomLen	length in bytes of the server's random data

♦ CK_SSL3_MASTER_KEY_DERIVE_PARAMS

CK_SSL3_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_VERSION_PTR pVersion;
} CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

RandomInfo	client's and server's random data information.
pVersion	pointer to a **CK_VERSION** structure which receives the SSL protocol version information (see SSL 3.0 for details)

♦ CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR

CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR points to a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure. It is implementation-dependent.

♦ CK_SSL3_KEY_MAT_OUT

CK_SSL3_KEY_MAT_OUT is a structure that contains the resulting key handles after performing a C_DeriveKey function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:


```
typedef struct CK_SSL3_KEY_MAT_OUT {
    CK_OBJECT_HANDLE hClientMacSecret;
    CK_OBJECT_HANDLE hServerMacSecret;
    CK_OBJECT_HANDLE hClientKey;
    CK_OBJECT_HANDLE hServerKey;
    CK_BYTE_PTR pIVClient;
    CK_BYTE_PTR pIVSe rver;
} CK_SSL3_KEY_MAT_OUT;
```

The fields of the structure have the following meanings:

hClientMacSecret	key handle for the resulting Client MAC Secret key
hServerMacSecret	key handle for the resulting Server MAC Secret key
hClientKey	key handle for the resulting Client Secret key
hServerKey	key handle for the resulting Server Secret key
pIVClient	pointer to a location which receives the initialization vector (IV) created for the client, if any (see SSL 3.0 for details)
pIVServer	pointer to a location which receives the initialization vector (IV) created for the server, if any (see SSL 3.0 for details)

♦ CK_SSL3_KEY_MAT_OUT_PTR

CK_SSL3_KEY_MAT_OUT_PTR points to a **CK_SSL3_KEY_MAT_OUT** structure. It is implementation-dependent.

♦ CK_SSL3_KEY_MAT_PARAMS

CK_SSL3_KEY_MAT_PARAMS is a structure that provides the parameters to the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_SSL3_KEY_MAT_PARAMS {
    CK_ULONG ulMacSizeInBits;
    CK_ULONG ulKeySizeInBits;
    CK_ULONG ulIVSizeInBits;
    CK_BBOOL bIsExport;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_SSL3_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

ulMacSizeInBits	establishes the length (in bits) of the MACing keys agreed upon during the protocol handshake phase (see SSL 3.0 for details)

ulKeySizeInBits	establishes the length (in bits) of the secret keys agreed upon during the protocol handshake phase (see SSL 3.0 for details)
ulIVSizeInBits	establishes the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0 (see SSL 3.0 for details)
bIsExport	a boolean value which indicates whether the keys have to be derived for an export version of the protocol (see SSL 3.0 for details)
RandomInfo	client's and server's random data information.
pReturnedKeyMaterial	points to a **CK_SSL3_KEY_MAT_OUT** structures which receives the handles for the keys generated, as well as the IVs when required (see SSL 3.0 for details)

♦ CK_SSL3_KEY_MAT_PARAMS_PTR

CK_SSL3_KEY_MAT_PARAMS_PTR points to a **CK_SSL3_KEY_MAT_PARAMS** structure. It is implementation-dependent.

10.35 SSL mechanisms

10.35.1 Pre_master key generation

Pre_master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL version 3.0.

It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

The mechanism contributes to the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

10.35.2 Master key derivation

Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the value of the client version found in the "pre_master" key as well as a handle to the derived "master_secret" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 10.34.

The mechanism contributes to the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either TRUE or FALSE. If omitted, these attributes each take on some default value.

- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure both indicate 48 bytes.

10.35.3 Key and MAC derivation

Key, MAC and IV derivation in SSL 3.0, denoted **CKM_SSL3_KEY_AND_MAC_DERIVE**, is a mechanism is used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the initialization vectors (IVs) created.

It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 10.34.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing, verification (they are used for MACing), and derivation operations.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will by modified by the **C_DeriveKey** call; in particular, the four key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys. In addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For other mechanisms, the **C_DeriveKey** function returns a single key handle as a result of a successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

10.35.4 MD5 MACing in SSL 3.0

MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 10-50, MD5 MACing in SSL 3.0: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

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

10.35.5 SHA-1 MACing in SSL 3.0

SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 10-51, SHA-1 MACing in SSL 3.0: Key And Data Length Constraints

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

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

10.36 Parameters for miscellaneous simple key derivation mechanisms

♦ CK_KEY_DERIVATION_STRING_DATA

CK_KEY_DERIVATION_STRING_DATA is a structure that holds a pointer to a byte string and the byte string's length. It provides the parameters for the **CKM_CONCATENATE_BASE_AND_DATA**, **CKM_CONCATENATE_DATA_AND_BASE**, and **CKM_XOR_BASE_AND_DATA** mechanisms. It is defined as follows:

```
typedef struct CK_KEY_DERIVATION_STRING_DATA {
    CK_BYTE_PTR pData;
    CK_ULONG ulLen;
} CK_KEY_DERIVATION_STRING_DATA;
```

The fields of the structure have the following meanings:

pData pointer to the byte string

ulLen length of the byte string

♦ CK_KEY_DERIVATION_STRING_DATA_PTR

CK_KEY_DERIVATION_STRING_DATA_PTR points to a **CK_KEY_DERIVATION_STRING_DATA** structure. It is implementation-dependent.

♦ CK_EXTRACT_PARAMS

CK_KEY_EXTRACT_PARAMS provides the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:

```
typedef CK_ULONG CK_EXTRACT_PARAMS;
```

♦ CK_EXTRACT_PARAMS_PTR

CK_EXTRACT_PARAMS_PTR points to a **CK_EXTRACT_PARAMS**. It is implemenation-dependent.

10.37 Miscellaneous simple key derivation mechanisms

10.37.1 Concatenation of a base key and another key

This mechanism, denoted **CKM_CONCATENATE_BASE_AND_KEY**, derives a secret key from the concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys specified are concatenated together in a buffer.

This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value information which is appended to the end of the base key's value information (the base key is the key whose handle is supplied as an argument to **C_DeriveKey**).

For example, if the value of the base key is `0x01234567,` and the value of the other key is `0x89ABCDEF` , then the value of the derived key will be taken from a buffer containing the string `0x0123456789ABCDEF` .

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the values of the two original keys.

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the two original keys' values, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If either of the two original keys has its **CKA_SENSITIVE** attribute set to TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.

- Similarly, if either of the two original keys has its **CKA_EXTRACTABLE** attribute set to FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.

- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to TRUE if and only if both of the original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to TRUE.

- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to TRUE if and only if both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to TRUE.

10.37.2 Concatenation of a base key and data

This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by concatenating data onto the end of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be appended to the base key to derive another key.

For example, if the value of the base key is `0x01234567`, and the value of the data is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x0123456789ABCDEF`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the value of the original key and the data.

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the original key's value and the data, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.

- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.

- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE.

- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE.

10.37.3 Concatenation of data and a base key

This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by prependting data to the start of a specified secret key.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the length and value of the data which will be prepended to the base key to derive another key.

For example, if the value of the base key is `0x01234567`, and the value of the data is `0x89ABCDEF`, then the value of the derived key will be taken from a buffer containing the string `0x89ABCDEF01234567`.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the original key.

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the data and the original key's value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.

- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.

- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE.

- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE.

10.37.4 XORing of a key and data

XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle and some data.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the data with which to XOR the original key's value.

For example, if the value of the base key is `0x01234567`, and the value of the data is `0x89ABCDEF` , then the value of the derived key will be taken from a buffer containing the string `0x88888888` .

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.

- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.

- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by taking the shorter of the data and the original key's value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.

- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.

- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE.

- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE.

10.37.5 Extraction of one key from another key

Extraction of one key from another key, denoted **CKM_EXTRACT_KEY_FROM_KEY**, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a **CK_EXTRACT_PARAMS**, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value `0x329F84A9` . We will derive a 2-byte secret key from this key, starting at bit position 21 (*i.e.*, the value of the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism is 21).

1. We write the key's value in binary: `0011 0010 1001 1111 1000 0100 1010 1001` . We regard this binary string as holding the 32 bits of the key, labelled as b_0, b_1, …, b_{31}.

2. We then extract 16 consecutive bits (*i.e.*, 2 bytes) from this binary string, starting at bit b_{21}. We obtain the binary string `1001 0101 0010 0110` .

3. The value of the new key is thus `0x9526`.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key's value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

- If no length or key type is provided in the template, then an error will be returned.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to TRUE.

11. Cryptoki tips and reminders

In this section, we clarify, review, and/or emphasize a few odds and ends about how Cryptoki works.

11.1 Sessions

In Cryptoki, there are several different types of operations which can be "active" in a session. An active operation is essentially one which takes more than one Cryptoki function call to perform. The types of active operations are object searching; encryption; decryption; message-digesting; signature with appendix; signature with recovery; verification with appendix; and verification with recovery.

A given session can have 0, 1, or 2 operations active at a time. It can only have 2 operations active simultaneously if the token supports this; moreover, those two operations must be one of the four following pairs of operations: digesting and encryption; decryption and digesting; signing and encryption; decryption and verification.

If an application attempts to initialize an operation (make it active), but this cannot be accomplished because of some other active operation(s), the application receives the error value CKR_OPERATION_ACTIVE. This error value can also be received if the application attempts to perform any of various operations which do not become "active", but which require cryptographic processing, such as using the token's random number generator, or generating/wrapping/unwrapping/deriving a key.

In general, different threads/processes of an application should not share sessions, unless they are extremely careful not to make function calls at the same time. Sharing sessions can easily lead to trouble.

11.2 Objects, attributes, and templates

In Cryptoki, every object (with the possible exception of RSA private keys) always possesses *all* possible attributes for an object of its type. This means, for example, that a Diffie-Hellman private key object *always* possesses a **CKA_VALUE_BITS** attribute, *even if that attribute wasn't specified when the key was generated* (in such a case, the proper value for the attribute is computed during the key generation process).

In general, a Cryptoki function which requires a template for an object needs the template to specify any atributes that are not specified elsewhere (explicitly or implicitly). If a template specifies a particular atttribute more than once, the function can return CKR_TEMPLATE_INVALID; or it can choose a particular value of the attribute from among those specified, and use that value. In any event, object attributes are single-valued.

11.3 Signing with recovery

Signing with recovery is a general alternative to ordinary digital signatures which is supported by certain mechanisms. Recall that for ordinary digital signatures, a signature of a message is computed as some function of the message and the signer's private key; this signature can then be used (together with the message and the signer's public key) as input to the verification process, which yields a simple "signature valid / signature invalid" decision.

Signing with recovery also creates a signature from a message and the signer's private key. However, to verify this signature, no message is required as input. Only the signature and the signer's public key are inputs to the verification process, and this process outputs either "signature invalid" or the original message (if the signature was valid).

Consider a simple example with the **CKM_RSA_C_509** mechanism. Here, a message is a byte string which we will consider to be a number modulo *n* (the signer's RSA modulus). When this mechanism is used for ordinary digital signatures (signatures with appendix), a signature is computed by raising the message to the signer's private exponent modulo *n*. To verify this signature, a verifier raises the signature to the signer's public exponent modulo *n*, and accepts the signature as valus if and only if the result matches the original message.

If **CKM_RSA_C_509** is used to create signatures with recovery, the signatures are produced in exactly the same fashion. For this particular mechanism, *any* number modulo *n* is a valid signature. To recover the message from a signature, the signature is raised to the signer's public exponent modulo *n*.

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 and profiles

Mechanism	Application		
	Privacy-Enhanced Mail	**Government Authentication-only**	**Cellular Digital Packet Data**
CKM_RSA_PKCS_KEY_PAIR_GEN	✓		
CKM_RSA_PKCS	✓		
CKM_RSA_9796			
CKM_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 it 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 APIs

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
- X/Open GCS-API - Generic Cryptographic Service API, Draft 2, February 14, 1995

B.1 FORTEZZA CIPG, Rev. 1.52

This document defines an API to the Fortezza PCMCIA Crypto Card. It is at a level similar to Cryptoki. The following table lists the FORTEZZA CIPG functions, together with the equivalent Cryptoki functions:

Table B-1, FORTEZZA CIPG vs. Cryptoki

FORTEZZA CIPG	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	C_GetTokenInfo
CI_Hash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal
CI_Initialize	C_Initialize
CI_InitializeHash	C_DigestInit

FORTEZZA CIPG	Equivalent Cryptoki
CI_InstallX	C_UnwrapKey
CI_LoadCertificate	C_CreateObject
CI_LoadDSAParameters	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_Lock	Implicit in session management
CI_Open	C_OpenSession
CI_RelayX	C_WrapKey
CI_Reset	C_CloseAllSessions
CI_Restore	Implicit in session management
CI_Save	Implicit in session management
CI_Select	C_OpenSession
CI_SetConfiguration	No equivalent
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_Terminate	C_CloseAllSessions
CI_Timestamp	C_SignInit, C_Sign
CI_Unlock	Implicit in session management
CI_UnwrapKey	C_UnwrapKey
CI_VerifySignature	C_VerifyInit, C_Verify
CI_VerifyTimestamp	C_VerifyInit, C_Verify
CI_WrapKey	C_WrapKey
CI_Zeroize	C_InitToken

B.2 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.

Table B-2, GCS-API vs. Cryptoki

GCS-API	Cryptoki implementation

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	
initialise_random_number_generator	C_SeedRandom
install_algorithm	
de_install_algorithm	
disable_algorithm	
enable_algorithm	
set_defaults	

